USENIX Association

Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium

August 9–11, 2023
Anaheim, CA, USA

© 2023 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-37-3

Artifact Evaluation Committee

Artifact Evaluation Committee Co-Chairs
Cristiano Giuffrida, *Vrije Universiteit Amsterdam*
Anjo Vahldiek-Oberwagner, *Intel Labs*

Artifact Evaluation Publication Chair
Alexios Voulimeneas, *Delft University of Technology*

Artifact Evaluation Committee
Shubham Agarwal, *CISPA Helmholtz Center for Information Security*
Nikolaos Alexopoulos, *Technical University of Darmstadt*
Amit Seal Ami, *College of William & Mary*
Daniel Arp, *University College London*
Erin Avllazagaj, *University of Maryland College Park*
Jessy Ayala, *University of California, Irvine*
Alessandro Baccarini, *University at Buffalo*
David Balash, *The George Washington University*
Nils Bars, *CISPA Helmholtz Center for Information Security*
Raounak Benabidallah, *CEA LIST, Université Paris-Saclay*
Shay Berkovich, *BlackBerry*
Lukas Bernhard, *Ruhr University Bochum*
Yohan Beugin, *University of Wisconsin—Madison*
Jakob Bleier, *Technische Universität Wien*
Amel Bourdoucen, *Aalto University*
Clemens-Alexander Brust, *DLR Institute of Data Science*
Jiahao Cao, *Tsinghua University*
Marco Casagrande, *EURECOM*
Stefanos Chaliasos, *Imperial College London*
Guangke Chen, *ShanghaiTech University*
Weiteng Chen, *Microsoft Research*
Pascal Cotret, *ENSTA Bretagne*
Daniel De Almeida Braga, *Université de Rennes, CNRS, IRISA*
Giulio De Pasquale, *King's College London*
Luca Degani, *University of Trento*
Nurullah Demir, *Karlsruhe Institute of Technology*
Omkar Dilip Dhawal, *Indian Institute of Technology Madras*
Sayanton Dibbo, *Dartmouth College*
Aneet Kumar Dutta, *CISPA Helmholtz Center for Information Security*
Alessandro Erba, *CISPA Helmholtz Center for Information Security*
Yusi Feng, *University of Chinese Academy of Science*
Christof Ferreira Torres, *ETH Zurich*
Lucas Franceschino, *Inria*
Chuanpu Fu, *Tsinghua University*
Alexandre Gonzalvez, *Université de Rennes, CNRS, IRISA*
Sindhu Reddy Kalathur Gopal, *University of Wyoming*
Sanket Goutam, *Stony Brook University*
Vishal Gupta, *EPFL*
HyungSeok Han, *Georgia Institute of Technology*
Muhammad Haseeb, *New York University*
Ningyu He, *Peking University*
Tiago Heinrich, *Federal University of Paraná*
Son Ho, *Inria*
Hailong Hu, *University of Luxembourg*
Shengtuo Hu, *Meta*
Yang Hu, *The University of Texas at Austin*
Hai Huang, *CISPA Helmholtz Center for Information Security*
Qiqing Huang, *University at Buffalo*
Raphael Isemann, *Vrije Universiteit Amsterdam*
Fabian Ising, *FH Münster University of Applied Sciences*
Charlie Jacomme, *Inria*
Jafar Haadi Jafarian, *University of Colorado Denver*
Yilin Ji, *Karlsruhe Institute of Technology*
Evan Johnson, *University of California, San Diego*
Kaushal Kafle, *College of William & Mary*
Imtiaz Karim, *Purdue University*
Soheil Khodayari, *CISPA Helmholtz Center for Information Security*
Hyungsub Kim, *Purdue University*
Soomin Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
Rachel King, *University of Wisconsin—Madison*
John Kressel, *University of Manchester*
Anunay Kulshrestha, *Princeton University*
Guilhem Lacombe, *CEA LIST*
Hieu Le, *University of California, Irvine*
Hugo Lefeuvre, *The University of Manchester*
Caihua Li, *Yale University*
Yuan Li, *Zhejiang University*
Xu Lin, *University of Illinois Chicago*
Zhibo Liu, *Hong Kong University of Science and Technology*
Eleonora Losiouk, *University of Padua*
Mulong Luo, *Cornell University*
Ning Luo, *Yale University*
Charles Babu M, *CEA LIST, Université Paris-Saclay*
Eman Maali, *Imperial College London*
Yanmao Man, *ByteDance, Inc*
Grégoire Menguy, *CEA LIST, Université Paris-Saclay*
Vladislav Mladenov, *Ruhr University Bochum*
Shouvick Mondal, *Concordia University*
Kazi Samin Mubasshir, *Purdue University*
Abdun Nihaal, *Indian Institute of Technology Madras*
Yu Nong, *Washington State University*
Eric Pauley, *University of Wisconsin—Madison*
Imranur Rahman, *North Carolina State University*
Mirza Masfiqur Rahman, *Purdue University*
Dilip Ravindran
Jenni Reuben, *Totalförsvarets forskningsinstitut (FOI)*
Joe Rowell, *Royal Holloway, University of London*
Abhinaya S.B., *North Carolina State University*
Nuno Sabino, *Carnegie Mellon University, Portugal*
Saiful Islam Salim, *University of Arizona*
Solmaz Salimi, *Sharif University of Technology*
Harshad Sathaye, *Northeastern University*
Tobias Scharnowski, *Ruhr University Bochum*
Nico Schiller, *Ruhr University Bochum*
Till Schlüter, *CISPA Helmholtz Center for Information Security*
Basavesh Ammanaghatta Shivakumar, *Max Planck Institute for Security and Privacy (MPI-SP)*
Pradyumna Shome, *Georgia Institute of Technology*
Sudheesh Singanamalla, *University of Washington*
Sachin Kumar Singh, *University of Utah*
Johnny So, *Stony Brook University*
Salwa Souaf, *CEA LIST, Université Paris-Saclay*

Avinash Sudhodanan, *Meta*
Xi Tan, *University at Buffalo*
Zahra Tarkhani, *Microsoft and University of Cambridge*
Erik Tews, *University of Twente*
Rodothea Myrsini Tsoupidi, *KTH Royal Institute of Technology*
Sai Venkata Krishnan V, *Indian Institute of Technology Madras*
Emanuele Vannacci, *Vrije Universiteit Amsterdam*
Dawei Wang, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences*
Ningfei Wang, *University of California, Irvine*
Shu Wang, *George Mason University*
Wenxi Wang, *The University of Texas at Austin*
Yuke Wang, *University of California, Santa Barbara*
Alexander Warnecke, *Technische Universität Braunschweig*
Feng Wei, *University at Buffalo*
Shijia Wei, *The University of Texas at Austin*
Ruoyu Wu, *Purdue University*
Dongwei Xiao, *Hong Kong University of Science and Technology*
Jiacen Xu, *University of California, Irvine*
Peisen Yao, *Zhejiang University*
Yuanyuan Yuan, *The Hong Kong University of Science and Technology*
Insu Yun, *Korea Advanced Institute of Science and Technology (KAIST)*
Samee Zahur, *Google*
Menghao Zhang, *Tsinghua University and Kuaishou Technology*
Ruiyi Zhang, *CISPA Helmholtz Center for Information Security*
Shaohu Zhang, *North Carolina State University*
Yiming Zhang, *The Hong Kong Polytechnic University*
Zhiyu Zhang, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences*
Zhiyuan Zhang, *University of Adelaide*
Xia Zhou, *Zhejiang University*
Xin'an Zhou, *University of California, Riverside*
Xugui Zhou, *University of Virginia*
Huadi Zhu, *University of Texas at Arlington*
Weidong Zhu, *University of Florida*
Maximilian Zinkus, *Johns Hopkins University*

Message from the USENIX Security ’23 Artifact Evaluation Committee Co-Chairs

On behalf of USENIX, we want to welcome you to the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium. Over a year ago, we started to work with everyone to run an artifact evaluation (AE) in conjunction with the main conference. We are proud of what our community has accomplished together. Learning from the last years, we increased the artifact evaluation committee to 127 members. We mainly recruited the members via a self-nomination process leading to PhD students, post-docs, or early career researchers from around the world to apply. One significant change was the introduction of a publication chair since the expected workload to validate the numerous artifact appendices would have been too large.

Artifact Evaluation in the security community is still relatively new and has only occurred for the 4th time since 2020. We aligned the artifact evaluation with the paper submission cycles such that the artifact evaluation follows after the final paper deadline of each of the 3 main paper submission cycles. Authors may submit their artifacts for review and select which badges they would like to be evaluated against. Each evaluation cycle took 4–5 weeks to complete, in which the evaluators worked with the author-provided artifacts and discussed issues with the authors in a single-blind manner. In each cycle, evaluators were assigned 1 to 2 artifacts to judge against the badge criterion.

After three AE cycles, we concluded with 140 artifacts receiving at least one of the three badges. As a result, out of 422 USENIX Security ’23 papers, 31% went through artifact evaluation and successfully received a badge, a decrease of 13% from 2022. In total, 138 Artifacts Available, 120 Artifacts Functional, and 96 Results Reproduced badges were awarded by the Artifact Evaluation Committee (AEC). The Artifacts Available badge ensures that the artifact is publicly available; the Artifacts Functional badge ensures that the artifact is complete, documented, and exercisable; the Results Reproduced badge ensures that the major claims of the paper have been reproduced by an evaluator independently using the author-provided artifacts.

We are tremendously grateful to the AEC. The evaluators spent countless hours improving and communicating with the authors leading to more than 2,750 comments in HotCRP and close to 350 reviews. Without this effort, this proceedings volume of 140 artifact appendices wouldn’t have been possible.

To further promote artifact evaluation in the community and highlight great artifacts, the AEC selected 4 distinguished artifact awards and 5 distinguished artifact reviewer awards. The former recognizes artifacts that should act as lighthouses to anyone seeking guidance on how to build a great artifact. The latter is to recognize the efforts of reviewers who went above and beyond to review and help authors improve their artifacts.

New this year was updated documentation for authors and reviewers as well as the artifact appendix template. In addition, we allowed out-of-cycle submissions of artifacts. That is, authors were able to submit artifacts in the same cycle their paper was accepted or any of the following cycles. This resulted in a slow start to the artifact submissions in the summer cycle and doubled with every cycle leading to almost 80 artifact submissions in the final winter cycle.

We want to express our immeasurable gratitude to the community without whom these proceedings would not be possible.

Cristiano Giuffrida, *Vrije Universiteit Amsterdam*
Anjo Vahldiek-Oberwagner, *Intel Labs*
USENIX Security ’23 Artifact Evaluation Committee Co-Chairs

Alexios Voulimeneas, *Delft University of Technology*
USENIX Security ’23 Artifact Evaluation Publication Chair

Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium

August 9–11, 2023
Anaheim, CA, USA

Wednesday, August 9

Inferring User Details

Auditory Eyesight: Demystifying μs-Precision Keystroke Tracking Attacks on Unconstrained Keyboard Inputs 1
Yazhou Tu, Liqun Shan, and Md Imran Hossen, *University of Louisiana at Lafayette;* Sara Rampazzi and Kevin Butler, *University of Florida;* Xiali Hei, *University of Louisiana at Lafayette*

Adversarial ML beyond ML

Squint Hard Enough: Attacking Perceptual Hashing with Adversarial Machine Learning 7
Jonathan Prokos, *Johns Hopkins University;* Neil Fendley, *Johns Hopkins University Applied Physics Laboratory;* Matthew Green, *Johns Hopkins University;* Roei Schuster, *Vector Institute;* Eran Tromer, *Tel Aviv University and Columbia University;* Tushar Jois and Yinzhi Cao, *Johns Hopkins University*

Private Set Operations

Linear Private Set Union from Multi-Query Reverse Private Membership Test 9
Cong Zhang, *State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences;* Yu Chen, *School of Cyber Science and Technology, Shandong University; State Key Laboratory of Cryptology; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University;* Weiran Liu, *Alibaba Group;* Min Zhang, *School of Cyber Science and Technology, Shandong University; State Key Laboratory of Cryptology; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University;* Dongdai Lin, *State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences*

Logs and Auditing

Improving Logging to Reduce Permission Over-Granting Mistakes 13
Bingyu Shen, Tianyi Shan, and Yuanyuan Zhou, *University of California, San Diego*

Fighting the Robots

BotScreen: Trust Everybody, but Cut the Aimbots Yourself 15
Minyeop Choi, *KAIST;* Gihyuk Ko, *Cyber Security Research Center at KAIST and Carnegie Mellon University;* Sang Kil Cha, *KAIST and Cyber Security Research Center at KAIST*

Traffic Analysis

HorusEye: A Realtime IoT Malicious Traffic Detection Framework using Programmable Switches 19
Yutao Dong, *Tsinghua Shenzhen International Graduate School, Shenzhen, China; Peng Cheng Laboratory, Shenzhen, China;* Qing Li, *Peng Cheng Laboratory, Shenzhen, China;* Kaidong Wu and Ruoyu Li, *Tsinghua Shenzhen International Graduate School, Shenzhen, China; Peng Cheng Laboratory, Shenzhen, China;* Dan Zhao, *Peng Cheng Laboratory, Shenzhen, China;* Gareth Tyson, *Hong Kong University of Science and Technology* (*GZ*), *Guangzhou, China;* Junkun Peng, Yong Jiang, and Shutao Xia, *Tsinghua Shenzhen International Graduate School, Shenzhen, China; Peng Cheng Laboratory, Shenzhen, China;* Mingwei Xu, *Tsinghua University, Beijing, China*

Adversarial Patches and Images

Towards Targeted Obfuscation of Adversarial Unsafe Images using Reconstruction and Counterfactual Super Region Attribution Explainability 23
Mazal Bethany, Andrew Seong, Samuel Henrique Silva, Nicole Beebe, Nishant Vishwamitra, and Peyman Najafirad, *The University of Texas at San Antonio*

Decentralized Finance

Is Your Wallet Snitching On You? An Analysis on the Privacy Implications of Web3 25
Christof Ferreira Torres, Fiona Willi, and Shweta Shinde, *ETH Zurich*

Memory

Capstone: A Capability-based Foundation for Trustless Secure Memory Access 29
Jason Zhijingcheng Yu, *National University of Singapore;* Conrad Watt, *University of Cambridge;* Aditya Badole, Trevor E. Carlson, and Prateek Saxena, *National University of Singapore*

FloatZone: Accelerating Memory Error Detection using the Floating Point Unit 31
Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos, *Vrije Universiteit Amsterdam*

PUMM: Preventing Use-After-Free Using Execution Unit Partitioning 33
Carter Yagemann, *The Ohio State University;* Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee, *Georgia Institute of Technology*

Security in Digital Realities

Unique Identification of 50,000+ Virtual Reality Users from Head & Hand Motion Data 37
Vivek Nair and Wenbo Guo, *UC Berkeley;* Justus Mattern, *RWTH Aachen;* Rui Wang and James F. O'Brien, *UC Berkeley;* Louis Rosenberg, *Unanimous AI;* Dawn Song, *UC Berkeley*

Erebus: Access Control for Augmented Reality Systems 41
Yoonsang Kim, Sanket Goutam, Amir Rahmati, and Arie Kaufman, *Stony Brook University*

Privacy Policies, Labels, Etc.

PoliGraph: Automated Privacy Policy Analysis using Knowledge Graphs 43
Hao Cui, Rahmadi Trimananda, Athina Markopoulou, and Scott Jordan, *University of California, Irvine*

Calpric: Inclusive and Fine-grain Labeling of Privacy Policies with Crowdsourcing and Active Learning 47
Wenjun Qiu, David Lie, and Lisa Austin, *University of Toronto*

Lalaine: Measuring and Characterizing Non-Compliance of Apple Privacy Labels 49
Yue Xiao, Zhengyi Li, and Yue Qin, *Indiana University Bloomington;* Xiaolong Bai, *Orion Security Lab, Alibaba Group;* Jiale Guan, Xiaojing Liao, and Luyi Xing, *Indiana University Bloomington*

ML Applications to Malware

Evading Provenance-Based ML Detectors with Adversarial System Actions 53
Kunal Mukherjee, Joshua Wiedemeier, Tianhao Wang, James Wei, Feng Chen, Muhyun Kim, Murat Kantarcioglu, and Kangkook Jee, *The University of Texas at Dallas*

Secure Messaging

TreeSync: Authenticated Group Management for Messaging Layer Security 57
Théophile Wallez, *Inria Paris;* Jonathan Protzenko, *Microsoft Research;* Benjamin Beurdouche, *Mozilla;* Karthikeyan Bhargavan, *Inria Paris*

Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations 59
Cas Cremers, *CISPA Helmholtz Center for Information Security;* Charlie Jacomme, *Inria Paris;* Aurora Naska, *CISPA Helmholtz Center for Information Security*

x-Fuzz

MorFuzz: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation 63
Jinyan Xu and Yiyuan Liu, *Zhejiang University;* Sirui He, *City University of Hong Kong;* Haoran Lin and Yajin Zhou, *Zhejiang University;* Cong Wang, *City University of Hong Kong*

FishFuzz: Catch Deeper Bugs by Throwing Larger Nets 65
Han Zheng, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Science; School of Computer and Communication Sciences, EPFL; Zhongguancun Laboratory;* Jiayuan Zhang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Science; School of Computer and Communication, Lanzhou University of Technology;* Yuhang Huang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Science;* Zezhong Ren, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Science; Zhongguancun Laboratory;* He Wang, *School of Cyber Engineering, Xidian University;* Chunjie Cao, *School of Cyberspace Security, Hainan University;* Yuqing Zhang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Science; Zhongguancun Laboratory; School of Cyberspace Security, Hainan University; School of Cyber Engineering, Xidian University;* Flavio Toffalini and Mathias Payer, *School of Computer and Communication Sciences, EPFL*

PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems 67
Wen Li, Jinyang Ruan, and Guangbei Yi, *Washington State University;* Long Cheng, *Clemson University;* Xiapu Luo, *The Hong Kong Polytechnic University;* Haipeng Cai, *Washington State University*

Programs, Code, and Binaries

Not All Data are Created Equal: Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks 71
Salman Ahmed, *IBM Research;* Hans Liljestrand, *University of Waterloo;* Hani Jamjoom, *IBM Research;* Matthew Hicks, *Virginia Tech;* N. Asokan, *University of Waterloo;* Danfeng (Daphne) Yao, *Virginia Tech*

Safer: Efficient and Error-Tolerant Binary Instrumentation 73
Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R. Sekar, *Stony Brook University*

Reassembly is Hard: A Reflection on Challenges and Strategies 77
Hyungseok Kim, *KAIST and The Affiliated Institute of ETRI;* Soomin Kim and Junoh Lee, *KAIST;* Kangkook Jee, *University of Texas at Dallas;* Sang Kil Cha, *KAIST*

IoT Security Expectations and Barriers

Are Consumers Willing to Pay for Security and Privacy of IoT Devices? 81
Pardis Emami-Naeini, *Duke University;* Janarth Dheenadhayalan, Yuvraj Agarwal, and Lorrie Faith Cranor, *Carnegie Mellon University*

Differential Privacy

PrivateFL: Accurate, Differentially Private Federated Learning via Personalized Data Transformation 85
Yuchen Yang, Bo Hui, and Haolin Yuan, *The Johns Hopkins University;* Neil Gong, *Duke University;* Yinzhi Cao, *The Johns Hopkins University*

Poisoning

Meta-Sift: How to Sift Out a Clean Subset in the Presence of Data Poisoning? 89
Yi Zeng, *Virginia Tech and SONY AI;* Minzhou Pan, Himanshu Jahagirdar, and Ming Jin, *Virginia Tech;* Lingjuan Lyu, *SONY AI;* Ruoxi Jia, *Virginia Tech*

Towards A Proactive ML Approach for Detecting Backdoor Poison Samples 93
Xiangyu Qi, Tinghao Xie, Jiachen T. Wang, Tong Wu, Saeed Mahloujifar, and Prateek Mittal, *Princeton University*

Every Vote Counts: Ranking-Based Training of Federated Learning to Resist Poisoning Attacks 97
Hamid Mozaffari, Virat Shejwalkar, and Amir Houmansadr, *University of Massachusetts Amherst*

Smart Contracts

Smart Learning to Find Dumb Contracts 101
Tamer Abdelaziz, *National University of Singapore;* Aquinas Hobor, *University College London*

x-Fuzz and Fuzz-x

Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge 103
Nils Bars, Moritz Schloegel, Tobias Scharnowski, and Nico Schiller, *Ruhr-Universität Bochum;* Thorsten Holz, *CISPA Helmholtz Center for Information Security*

FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler 105
Junjie Wang, *College of Intelligence and Computing, Tianjin University;* Zhiyi Zhang, *CodeSafe Team, Qi An Xin Group Corp.;* Shuang Liu, *College of Intelligence and Computing, Tianjin University;* Xiaoning Du, *Monash University;* Junjie Chen, *College of Intelligence and Computing, Tianjin University*

autofz: Automated Fuzzer Composition at Runtime 109
Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim, *Georgia Institute of Technology*

CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing 113
Dawei Wang, Ying Li, and Zhiyu Zhang, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China; School of Cyber Security, University of Chinese Academy of Sciences, China;* Kai Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China; School of Cyber Security, University of Chinese Academy of Sciences, China; Beijing Academy of Artificial Intelligence, China*

Cache Attacks

The Gates of Time: Improving Cache Attacks with Transient Execution 117
Daniel Katzman, *Tel Aviv University;* William Kosasih, *The University of Adelaide;* Chitchanok Chuengsatiansup, *The University of Melbourne;* Eyal Ronen, *Tel Aviv University;* Yuval Yarom, *The University of Adelaide*

CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software 121
Yuanyuan Yuan, Zhibo Liu, and Shuai Wang, *The Hong Kong University of Science and Technology*

Authentication

Security and Privacy Failures in Popular 2FA Apps 123
Conor Gilsenan, *UC Berkeley / ICSI;* Fuzail Shakir and Noura Alomar, *UC Berkeley;* Serge Egelman, *UC Berkeley / ICSI*

Multi-Factor Key Derivation Function (MFKDF) for Fast, Flexible, Secure, & Practical Key Management 125
Vivek Nair and Dawn Song, *University of California, Berkeley*

Generative AI

Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants 129
Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt, *New York University*

Deep Thoughts on Deep Learning

Aegis: Mitigating Targeted Bit-flip Attacks against Deep Neural Networks 133
Jialai Wang, *Tsinghua University;* Ziyuan Zhang, *Beijing University of Posts and Telecommunications;* Meiqi Wang, *Tsinghua University;* Han Qiu, *Tsinghua University and Zhongguancun Laboratory;* Tianwei Zhang, *Nanyang Technological University;* Qi Li, *Tsinghua University and Zhongguancun Laboratory;* Zongpeng Li, *Tsinghua University and Hangzhou Dianzi University;* Tao Wei, *Ant Group;* Chao Zhang, *Tsinghua University and Zhongguancun Laboratory*

IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks 135
Neophytos Christou, Di Jin, and Vaggelis Atlidakis, *Brown University;* Baishakhi Ray, *Columbia University;* Vasileios P. Kemerlis, *Brown University*

Thursday, August 10

Smart? Assistants

Spying through Your Voice Assistants: Realistic Voice Command Fingerprinting 139
Dilawer Ahmed, Aafaq Sabir, and Anupam Das, *North Carolina State University*

Security-Adjacent Worker Perspectives

Pushed by Accident: A Mixed-Methods Study on Strategies of Handling Secret Information in Source Code Repositories 143
Alexander Krause, *CISPA Helmholtz Center for Information Security;* Jan H. Klemmer and Nicolas Huaman, *Leibniz University Hannover;* Dominik Wermke, *CISPA Helmholtz Center for Information Security;* Yasemin Acar, *Paderborn University, George Washington University;* Sascha Fahl, *CISPA Helmholtz Center for Information Security*

Censorship and Internet Freedom

DeResistor: Toward Detection-Resistant Probing for Evasion of Internet Censorship 145
Abderrahmen Amich and Birhanu Eshete, *University of Michigan, Dearborn;* Vinod Yegneswaran, *SRI International;* Nguyen Phong Hoang, *University of Chicago*

How the Great Firewall of China Detects and Blocks Fully Encrypted Traffic 149
Mingshi Wu, *GFW Report;* Jackson Sippe, *University of Colorado Boulder;* Danesh Sivakumar and Jack Burg, *University of Maryland;* Peter Anderson, *Independent researcher;* Xiaokang Wang, *V2Ray Project;* Kevin Bock, *University of Maryland;* Amir Houmansadr, *University of Massachusetts Amherst;* Dave Levin, *University of Maryland;* Eric Wustrow, *University of Colorado Boulder*

Integrity

ARI: Attestation of Real-time Mission Execution Integrity 153
Jinwen Wang, Yujie Wang, and Ao Li, *Washington University in St. Louis;* Yang Xiao, *University of Kentucky;* Ruide Zhang, Wenjing Lou, and Y. Thomas Hou, *Virginia Polytechnic Institute and State University;* Ning Zhang, *Washington University in St. Louis*

XCheck: Verifying Integrity of 3D Printed Patient-Specific Devices via Computing Tomography 157
Zhiyuan Yu, Yuanhaur Chang, Shixuan Zhai, Nicholas Deily, and Tao Ju, *Washington University in St. Louis;* XiaoFeng Wang, *Indiana University Bloomington;* Uday Jammalamadaka, *Rice University;* Ning Zhang, *Washington University in St. Louis*

Fuzzing Firmware and Drivers

Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs 161
Tobias Scharnowski and Simon Wörner, *CISPA Helmholtz Center for Information Security;* Felix Buchmann, *Ruhr University Bochum;* Nils Bars, Moritz Schloegel, and Thorsten Holz, *CISPA Helmholtz Center for Information Security*

Forming Faster Firmware Fuzzers 167
Lukas Seidel, *Qwiet AI;* Dominik Maier, *TU Berlin;* Marius Muench, *VU Amsterdam and University of Birmingham*

Vehicles and Security

Understand Users' Privacy Perception and Decision of V2X Communication in Connected Autonomous Vehicles 171
Zekun Cai and Aiping Xiong, *The Pennsylvania State University*

DNS Security

NRDelegationAttack: Complexity DDoS attack on DNS Recursive Resolvers 175
Yehuda Afek and Anat Bremler-Barr, *Tel-Aviv University;* Shani Stajnrod, *Reichman University*

Ethereum Security

ACon^2: Adaptive Conformal Consensus for Provable Blockchain Oracles 181
Sangdon Park, *Georgia Institute of Technology;* Osbert Bastani, *University of Pennsylvania;* Taesoo Kim, *Georgia Institute of Technology*

Supply Chains and Third-Party Code

SandDriller: A Fully-Automated Approach for Testing Language-Based JavaScript Sandboxes 185
Abdullah AlHamdan and Cristian-Alexandru Staicu, *CISPA Helmholtz Center for Information Security*

Cellular Networks

Instructions Unclear: Undefined Behaviour in Cellular Network Specifications . . . 189
Daniel Klischies, *Ruhr University Bochum;* Moritz Schloegel and Tobias Scharnowski, *CISPA Helmholtz Center for Information Security;* Mikhail Bogodukhov, *Independent;* David Rupprecht, *Radix Security;* Veelasha Moonsamy, *Ruhr University Bochum*

MobileAtlas: Geographically Decoupled Measurements in Cellular Networks for Security and Privacy Research . . . 193
Gabriel K. Gegenhuber, *University of Vienna;* Wilfried Mayer, *SBA Research;* Edgar Weippl, *University of Vienna;* Adrian Dabrowski, *CISPA Helmholtz Center for Information Security*

BaseComp: A Comparative Analysis for Integrity Protection in Cellular Baseband Software . . . 195
Eunsoo Kim, Min Woo Baek, and CheolJun Park, *KAIST;* Dongkwan Kim, *Samsung SDS;* Yongdae Kim and Insu Yun, *KAIST*

Entomology

Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs . . . 199
Jianhao Xu, *Nanjing University;* Kangjie Lu, *University of Minnesota;* Zhengjie Du, Zhu Ding, and Linke Li, *Nanjing University;* Qiushi Wu, *University of Minnesota;* Mathias Payer, *EPFL;* Bing Mao, *Nanjing University*

A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs . . . 201
Gertjan Franken, Tom Van Goethem, Lieven Desmet, and Wouter Joosen, *imec-DistriNet, KU Leuven*

Remote Code Execution from SSTI in the Sandbox: Automatically Detecting and Exploiting Template Escape Bugs . . . 207
Yudi Zhao, Yuan Zhang, and Min Yang, *Fudan University*

Adversarial Examples

SMACK: Semantically Meaningful Adversarial Audio Attack . . . 209
Zhiyuan Yu, Yuanhaur Chang, and Ning Zhang, *Washington University in St. Louis;* Chaowei Xiao, *Arizona State University*

URET: Universal Robustness Evaluation Toolkit (for Evasion) . . . 213
Kevin Eykholt, Taesung Lee, Douglas Schales, Jiyong Jang, and Ian Molloy, *IBM Research;* Masha Zorin, *University of Cambridge*

Private Record Access

Authenticated private information retrieval . . . 217
Simone Colombo, *EPFL;* Kirill Nikitin, *Cornell Tech;* Henry Corrigan-Gibbs, *MIT;* David J. Wu, *UT Austin;* Bryan Ford, *EPFL*

GigaDORAM: Breaking the Billion Address Barrier . . . 221
Brett Falk, *University of Pennsylvania;* Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang, *University of California, Los Angeles*

One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval . . . 225
Alexandra Henzinger, Matthew M. Hong, and Henry Corrigan-Gibbs, *MIT;* Sarah Meiklejohn, *Google;* Vinod Vaikuntanathan, *MIT*

Duoram: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation . . . 229
Adithya Vadapalli, *University of Waterloo;* Ryan Henry, *University of Calgary;* Ian Goldberg, *University of Waterloo*

It's All Fun and Games Until...

A Peek into the Metaverse: Detecting 3D Model Clones in Mobile Games . . . 233
Chaoshun Zuo, Chao Wang, and Zhiqiang Lin, *The Ohio State University*

Enclaves and Serverless Computing

EnigMap: External-Memory Oblivious Map for Secure Enclaves . . . 235
Afonso Tinoco, Sixiang Gao, and Elaine Shi, *CMU*

Controlled Data Races in Enclaves: Attacks and Detection 241
Sanchuan Chen, *Fordham University;* Zhiqiang Lin, *The Ohio State University;* Yinqian Zhang, *Southern University of Science and Technology*

Guarding Serverless Applications with Kalium 245
Deepak Sirone Jegan, *University of Wisconsin-Madison;* Liang Wang, *Princeton University;* Siddhant Bhagat, *Microsoft;* Michael Swift, *University of Wisconsin-Madison*

OSes and Security

PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel 249
Zicheng Wang, *Nanjing University;* Yueqi Chen, *University of Colorado Boulder;* Qingkai Zeng, *Nanjing University*

Mitigating Security Risks in Linux with KLAUS: A Method for Evaluating Patch Correctness 253
Yuhang Wu and Zhenpeng Lin, *Northwestern University;* Yueqi Chen, *University of Colorado Boulder;* Dang K Le, *Northwestern University;* Dongliang Mu, *Huazhong University of Science and Technology;* Xinyu Xing, *Northwestern University*

Intrusion Detection

ARGUS: Context-Based Detection of Stealthy IoT Infiltration Attacks 257
Phillip Rieger, Marco Chilese, Reham Mohamed, Markus Miettinen, Hossein Fereidooni, and Ahmad-Reza Sadeghi, *Technical University of Darmstadt*

xNIDS: Explaining Deep Learning-based Network Intrusion Detection Systems for Active Intrusion Responses . . . 261
Feng Wei, *University at Buffalo;* Hongda Li, *Palo Alto Networks;* Ziming Zhao and Hongxin Hu, *University at Buffalo*

Privacy Preserving Crypto Blocks

Curve Trees: Practical and Transparent Zero-Knowledge Accumulators 263
Matteo Campanelli, *Protocol Labs;* Mathias Hall-Andersen and Simon Holmgaard Kamp, *Aarhus University, Denmark*

VeriZexe: Decentralized Private Computation with Universal Setup 265
Alex Luoyuan Xiong, *Espresso Systems, National University of Singapore;* Binyi Chen and Zhenfei Zhang, *Espresso Systems;* Benedikt Bünz, *Espresso Systems, Stanford University;* Ben Fisch, *Espresso Systems, Yale University;* Fernando Krell and Philippe Camacho, *Espresso Systems*

Warm and Fuzzing

Systematic Assessment of Fuzzers using Mutation Analysis 267
Philipp Görz, Björn Mathis, and Keno Hassler, *CISPA Helmholtz Center for Information Security;* Emre Güler, *Ruhr-Universität Bochum;* Thorsten Holz and Andreas Zeller, *CISPA Helmholtz Center for Information Security;* Rahul Gopinath, *University of Sydney*

Keeping Computations Confidential

HECO: Fully Homomorphic Encryption Compiler 271
Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi, *ETH Zurich*

A Verified Confidential Computing as a Service Framework for Privacy Preservation 275
Hongbo Chen and Haobin Hiroki Chen, *Indiana University Bloomington;* Mingshen Sun, *Independent Researcher;* Kang Li and Zhaofeng Chen, *CertiK;* XiaoFeng Wang, *Indiana University Bloomington*

Towards Robust Learning

Precise and Generalized Robustness Certification for Neural Networks 279
Yuanyuan Yuan, *The Hong Kong University of Science and Technology and ETH Zurich;* Shuai Wang, *The Hong Kong University of Science and Technology;* Zhendong Su, *ETH Zurich*

HOLMES: Efficient Distribution Testing for Secure Collaborative Learning 281
Ian Chang and Katerina Sotiraki, *UC Berkeley;* Weikeng Chen, *UC Berkeley & DZK Labs;* Murat Kantarcioglu, *University of Texas at Dallas & UC Berkeley;* Raluca Popa, *UC Berkeley*

Network Cryptographic Protocols

Exploring the Unknown DTLS Universe: Analysis of the DTLS Server Ecosystem on the Internet 285
Nurullah Erinola and Marcel Maehren, *Ruhr University Bochum;* Robert Merget, *Technology Innovation Institute;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum*

We *Really* Need to Talk About Session Tickets: A Large-Scale Analysis of Cryptographic Dangers with TLS Session Tickets 289
Sven Hebrok, *Paderborn University;* Simon Nachtigall, *Paderborn University and achelos GmbH;* Marcel Maehren and Nurullah Erinola, *Ruhr University Bochum;* Robert Merget, *Technology Innovation Institute and Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum*

Warmer and Fuzzers

DAFL: Directed Grey-box Fuzzing guided by Data Dependency 293
Tae Eun Kim, *KAIST;* Jaeseung Choi, *Sogang University;* Kihong Heo and Sang Kil Cha, *KAIST*

Friday, August 11

Kernel Analysis

BoKASAN: Binary-only Kernel Address Sanitizer for Effective Kernel Fuzzing 297
Mingi Cho, Dohyeon An, Hoyong Jin, and Taekyoung Kwon, *Yonsei University*

FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules 299
Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele, *Boston University*

Uncontained: Uncovering Container Confusion in the Linux Kernel 303
Jakob Koschel, *Vrije Universiteit Amsterdam;* Pietro Borrello and Daniele Cono D'Elia, *Sapienza University of Rome;* Herbert Bos and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

It's Academic

Educators' Perspectives of Using (or Not Using) Online Exam Proctoring 307
David G. Balash, Elena Korkes, Miles Grant, and Adam J. Aviv, *The George Washington University;* Rahel A. Fainchtein and Micah Sherr, *Georgetown University*

No more Reviewer #2: Subverting Automatic Paper-Reviewer Assignment using Adversarial Learning 309
Thorsten Eisenhofer, *Ruhr University Bochum;* Erwin Quiring, *Ruhr University Bochum and International Computer Science Institute (ISCI) Berkeley;* Jonas Möller, *Technische Universität Berlin;* Doreen Riepel, *Ruhr University Bochum;* Thorsten Holz, *CISPA Helmholtz Center for Information Security;* Konrad Rieck, *Technische Universität Berlin*

Distributed Secure Computations

WaterBear: Practical Asynchronous BFT Matching Security Guarantees of Partially Synchronous BFT 313
Haibin Zhang, *Beijing Institute of Technology;* Sisi Duan, *Tsinghua University, Zhongguancun Laboratory;* Boxin Zhao, *Zhongguancun Laboratory;* Liehuang Zhu, *Beijing Institute of Technology*

Practical Asynchronous High-threshold Distributed Key Generation and Distributed Polynomial Sampling 315
Sourav Das, *University of Illinois at Urbana-Champaign;* Zhuolun Xiang, *Aptos;* Lefteris Kokoris-Kogias, *IST Austria and Mysten Labs;* Ling Ren, *University of Illinois at Urbana-Champaign*

TVA: A multi-party computation system for secure and expressive time series analytics 317
Muhammad Faisal, *Boston University;* Jerry Zhang, *University of California San Diego;* John Liagouris, Vasiliki Kalavri, and Mayank Varia, *Boston University*

Mobile Security and Privacy

Powering Privacy: On the Energy Demand and Feasibility of Anonymity Networks on Smartphones 321
Daniel Hugenroth and Alastair R. Beresford, *University of Cambridge*

The OK Is Not Enough: A Large Scale Study of Consent Dialogs in Smartphone Applications 325
Simon Koch, *TU Braunschweig;* Benjamin Altpeter, *Datenanfragen.de e.V.;* Martin Johns, *TU Braunschweig*

Web Security

Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js **327**
Mikhail Shcherbakov and Musard Balliu, *KTH Royal Institute of Technology;* Cristian-Alexandru Staicu, *CISPA Helmholtz Center for Information Security*

Cookie Crumbles: Breaking and Fixing Web Session Integrity **331**
Marco Squarcina, *TU Wien;* Pedro Adão, *Instituto Superior Técnico, ULisboa, Instituto de Telecomunicações;* Lorenzo Veronese and Matteo Maffei, *TU Wien*

Minimalist: Semi-automated Debloating of PHP Web Applications through Static Analysis **335**
Rasoul Jahanshahi, *Boston University;* Babak Amin Azad and Nick Nikiforakis, *Stony Brook University;* Manuel Egele, *Boston University*

Routing and VPNs

How Effective is Multiple-Vantage-Point Domain Control Validation? **337**
Grace H. Cimaszewski, Henry Birge-Lee, Liang Wang, Jennifer Rexford, and Prateek Mittal, *Princeton University*

Bypassing Tunnels: Leaking VPN Client Traffic by Abusing Routing Tables **339**
Nian Xue, *New York University;* Yashaswi Malla, Zihang Xia, and Christina Pöpper, *New York University Abu Dhabi;* Mathy Vanhoef, *imec-DistriNet, KU Leuven*

Embedded Systems and Firmware

Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation **343**
Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, Adam Doupé, Tiffany Bao, Yan Shoshitaishvili, and Ruoyu Wang, *Arizona State University*

ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation **347**
Adam Caulfield, *Rochester Institute of Technology;* Norrathep Rattanavipanon, *Prince of Songkla University, Phuket Campus;* Ivan De Oliveira Nunes, *Rochester Institute of Technology*

The Impostor Among US(B): Off-Path Injection Attacks on USB Communications **351**
Robert Dumitru, *The University of Adelaide and Defence Science and Technology Group;* Daniel Genkin, *Georgia Tech;* Andrew Wabnitz, *Defence Science and Technology Group;* Yuval Yarom, *The University of Adelaide*

Attacks on Cryptography

A comprehensive, formal and automated analysis of the EDHOC protocol **355**
Charlie Jacomme, *Inria Paris;* Elise Klein, Steve Kremer, and Maïwenn Racouchot, *Inria Nancy and Université de Lorraine*

Automated Analysis of Protocols that use Authenticated Encryption: How Subtle AEAD Differences can impact Protocol Security **357**
Cas Cremers, *CISPA Helmholtz Center for Information Security;* Alexander Dax, *CISPA Helmholtz Center for Information Security and Saarland University;* Charlie Jacomme, *Inria Paris;* Mang Zhao, *CISPA Helmholtz Center for Information Security and Saarland University*

Cloud Insecurity

Credit Karma: Understanding Security Implications of Exposed Cloud Services through Automated Capability Inference **361**
Xueqiang Wang, *University of Central Florida;* Yuqiong Sun, *Meta;* Susanta Nanda, *ServiceNow;* XiaoFeng Wang, *Indiana University Bloomington*

Remote Direct Memory Introspection **363**
Hongyi Liu, Jiarong Xing, and Yibo Huang, *Rice University;* Danyang Zhuo, *Duke University;* Srinivas Devadas, *Massachusetts Institute of Technology;* Ang Chen, *Rice University*

More Web and Mobile Security

SQIRL: Grey-Box Detection of SQL Injection Vulnerabilities Using Reinforcement Learning **365**
Salim Al Wahaibi, Myles Foley, and Sergio Maffeis, *Imperial College London*

Hiding in Plain Sight: An Empirical Study of Web Application Abuse in Malware 369
Mingxuan Yao, *Georgia Institute of Technology;* Jonathan Fuller, *United States Military Academy;* Ranjita Pai Kasturi, Saumya Agarwal, Amit Kumar Sikder, and Brendan Saltaformaggio, *Georgia Institute of Technology*

Networks and Security

Device Tracking via Linux's New TCP Source Port Selection Algorithm 373
Moshe Kol, Amit Klein, and Yossi Gilad, *Hebrew University of Jerusalem*

An Efficient Design of Intelligent Network Data Plane 377
Guangmeng Zhou, *Tsinghua University;* Zhuotao Liu, *Tsinghua University and Zhongguancun Laboratory;* Chuanpu Fu, *Tsinghua University;* Qi Li and Ke Xu, *Tsinghua University and Zhongguancun Laboratory*

Arming and Disarming ARM

SpectrEM: Exploiting Electromagnetic Emanations During Transient Execution 379
Jesse De Meulemeester, Antoon Purnal, Lennert Wouters, Arthur Beckers, and Ingrid Verbauwhede, *COSIC, KU Leuven*

ARMore: Pushing Love Back Into Binaries 381
Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer, *EPFL*

Cryptography for Privacy

Prime Match: A Privacy-Preserving Inventory Matching System 385
Antigoni Polychroniadou, *J.P. Morgan;* Gilad Asharov, *Bar-Ilan University;* Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso, *J.P. Morgan*

Eos: Efficient Private Delegation of zkSNARK Provers 387
Alessandro Chiesa, *UC Berkeley and EPFL;* Ryan Lehmkuhl, *MIT;* Pratyush Mishra, *Aleo and University of Pennsylvania;* Yinuo Zhang, *UC Berkeley*

TAP: Transparent and Privacy-Preserving Data Services 389
Daniel Reijsbergen and Aung Maw, *Singapore University of Technology and Design;* Zheng Yang, *Southwest University;* Tien Tuan Anh Dinh and Jianying Zhou, *Singapore University of Technology and Design*

Vulnerabilities and Threat Detection

Trojan Source: Invisible Vulnerabilities 393
Nicholas Boucher, *University of Cambridge;* Ross Anderson, *University of Cambridge and University of Edinburgh*

Cheesecloth: Zero-Knowledge Proofs of Real World Vulnerabilities 395
Santiago Cuéllar, Bill Harris, James Parker, and Stuart Pernsteiner, *Galois, Inc.;* Eran Tromer, *Columbia University*

VulChecker: Graph-based Vulnerability Localization in Source Code 397
Yisroel Mirsky, *Ben-Gurion University of the Negev;* George Macon, *Georgia Tech Research Institute;* Michael Brown, *Georgia Institute of Technology;* Carter Yagemann, *Ohio State University;* Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee, *Georgia Institute of Technology*

Automated Analysis of Deployed Systems

Automated Security Analysis of Exposure Notification Systems 399
Kevin Morio and Ilkan Esiyok, *CISPA Helmholtz Center for Information Security;* Dennis Jackson, *Mozilla;* Robert Künnemann, *CISPA Helmholtz Center for Information Security*

Formal Analysis of SPDM: Security Protocol and Data Model version 1.2 401
Cas Cremers, Alexander Dax, and Aurora Naska, *CISPA Helmholtz Center for Information Security*

One Size Does Not Fit All: Uncovering and Exploiting Cross Platform Discrepant APIs in WeChat 405
Chao Wang, Yue Zhang, and Zhiqiang Lin, *The Ohio State University*

Side Channel Attacks

NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems 407
Zixuan Wang, *UC San Diego;* Mohammadkazem Taram, *Purdue University and UC San Diego;* Daniel Moghimi, *UT Austin and UC San Diego;* Steven Swanson, Dean Tullsen, and Jishen Zhao, *UC San Diego*

Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software **411**
Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth, *University of Lübeck*

Side-Channel Attacks on Optane Persistent Memory **413**
Sihang Liu, *University of Virginia;* Suraaj Kanniwadi, *Cornell University;* Martin Schwarzl, Andreas Kogler, and Daniel Gruss, *Graz University of Technology;* Samira Khan, *University of Virginia*

Transportation and Infrastructure

ICSPatch: Automated Vulnerability Localization and Non-Intrusive Hotpatching in Industrial Control Systems using Data Dependence Graphs **417**
Prashant Hari Narayan Rajput, *NYU Tandon School of Engineering;* Constantine Doumanidis and Michail Maniatakos, *New York University Abu Dhabi*

Language-Based Security

Argus: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions **421**
Siddharth Muralee, *Purdue University;* Igibek Koishybayev, Aleksandr Nahapetyan, Greg Tystahl, and Brad Reaves, *North Carolina State University;* Antonio Bianchi, *Purdue University;* William Enck and Alexandros Kapravelos, *North Carolina State University;* Aravind Machiry, *Purdue University*

McFIL: Model Counting Functionality-Inherent Leakage **423**
Maximilian Zinkus, Yinzhi Cao, and Matthew D. Green, *Johns Hopkins University*

Browsers

Isolated and Exhausted: Attacking Operating Systems via Site Isolation in the Browser **425**
Matthias Gierlings, Marcus Brinkmann, and Jörg Schwenk, *Ruhr University Bochum*

Pool-Party: Exploiting Browser Resource Pools for Web Tracking **433**
Peter Snyder, *Brave Software;* Soroush Karami, *University of Illinois at Chicago;* Arthur Edelstein, *Brave Software;* Benjamin Livshits, *Imperial College London;* Hamed Haddadi, *Brave Software and Imperial College of London*

Speculation Doesn’t Pay

Ultimate SLH: Taking Speculative Load Hardening to the Next Level **449**
Zhiyuan Zhang, *The University of Adelaide;* Gilles Barthe, *MPI-SP and IMDEA Software Institute;* Chitchanok Chuengsatiansup, *The University of Melbourne;* Peter Schwabe, *MPI-SP and Radboud University;* Yuval Yarom, *The University of Adelaide*

Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions **453**
Jana Hofmann, *Azure Research, Microsoft;* Emanuele Vannacci, *Vrije Universiteit Amsterdam;* Cédric Fournet, Boris Köpf, and Oleksii Oleksenko, *Azure Research, Microsoft*

ProSpeCT: Provably Secure Speculation for the Constant-Time Policy **457**
Lesly-Ann Daniel, Marton Bognar, and Job Noorman, *imec-DistriNet, KU Leuven;* Sébastien Bardin, *CEA, LIST, Université Paris Saclay;* Tamara Rezk, *INRIA, Université Côte d’Azur, Sophia Antipolis;* Frank Piessens, *imec-DistriNet, KU Leuven*

More Hardware Side Channels

(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels **459**
Ruiyi Zhang, *CISPA Helmholtz Center for Information Security;* Taehyun Kim, *Independent;* Daniel Weber and Michael Schwarz, *CISPA Helmholtz Center for Information Security*

Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels **461**
Andreas Kogler, Jonas Juffinger, and Lukas Giner, *Graz University of Technology;* Lukas Gerlach, *CISPA Helmholtz Center for Information Security;* Martin Schwarzl, *Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security;* Daniel Gruss and Stefan Mangard, *Graz University of Technology*

Inception: Exposing New Attack Surfaces with Training in Transient Execution **465**
Daniël Trujillo, Johannes Wikner, and Kaveh Razavi, *ETH Zurich*

BunnyHop: Exploiting the Instruction Prefetcher 469
Zhiyuan Zhang, Mingtian Tao, and Sioli O'Connell, *The University of Adelaide;* Chitchanok Chuengsatiansup, *The University of Melbourne;* Daniel Genkin, *Georgia Tech;* Yuval Yarom, *The University of Adelaide*

Deeper Thoughts on Deep Learning

Decompiling x86 Deep Neural Network Executables 473
Zhibo Liu, Yuanyuan Yuan, and Shuai Wang, *The Hong Kong University of Science and Technology;* Xiaofei Xie, *Singapore Management University;* Lei Ma, *University of Alberta*

Attacks on Deployed Cryptosystems

Every Signature is Broken: On the Insecurity of Microsoft Office's OOXML Signatures 477
Simon Rohlmann, Vladislav Mladenov, Christian Mainka, Daniel Hirschberger, and Jörg Schwenk, *Ruhr University Bochum*

Security Analysis of MongoDB Queryable Encryption 483
Zichen Gui, Kenneth G. Paterson, and Tianxin Tang, *ETH Zurich*

Attacking, Defending, and Analyzing

AutoFR: Automated Filter Rule Generation for Adblocking 487
Hieu Le, Salma Elmalaki, and Athina Markopoulou, *University of California, Irvine;* Zubair Shafiq, *University of California, Davis*

USENIX'23 Artifact Appendix
Auditory Eyesight: Demystifying μs-Precision Keystroke Tracking Attacks on Unconstrained Keyboard Inputs

Yazhou Tu, Liqun Shan, Md Imran Hossen, Sara Rampazzi[†], Kevin Butler[†], Xiali Hei
University of Louisiana at Lafayette
[†]University of Florida

A Artifact Appendix

A.1 Abstract

The artifact contains the dataset and benchmark results of tracking users' keystroke sounds to recover unconstrained keyboard inputs. It contains test cases with code to reproduce the attack results in different scenarios.

A.2 Description & Requirements

We run experiments on a computer with dual Xeon v3 E2683 processors and 32-GB RAM.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact can be accessed via the following GitHub link
https://github.com/auditoryeye/auditoryeye_artifact/releases/tag/20230809

A.2.3 Hardware dependencies

We recommend using computers with at least 12-core processors and 32-GB RAM.

A.2.4 Software dependencies

The code was written and tested in Matlab R2020b. We recommend using R2020b or newer versions of Matlab with the parallel computing toolbox.

A.2.5 Benchmarks

None.

A.3 Set-up

We organize the contents in test cases. There are 11 folders in the repository. Each folder contains the data, benchmark results, and code. In each folder, there is a main.m file. The artifact can be evaluated after navigating to one of the folders and opening the main.m file in Matlab.

A.3.1 Installation

The artifact is ready to run after downloading the contents from its GitHub repository:
https://github.com/auditoryeye/auditoryeye_artifact/tree/20230809

A.3.2 Basic Test

Navigate to the 01_proofofconcept_multiround_apple_keys folder. Open main.m file in Matlab and run each line in order. The results will be saved to the 01_proofofconcept_multiround_apple_keys/recording01_keys_interpolated folder. The results will be illustrated in the prompted windows. Step-by-step instructions to start the parallel pool and run the basic test are available at the artifact's GitHub repository.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Auditory Eyesight can distinguish compactly spaced keys by localizing keystroke sound signals in the differential range of microseconds.*
This is proven by: 1) experiments (E1, E2) described in paper's Section 4 whose results are illustrated in the paper's Figure 6, Figure 7, Figure 8, Figure 10, Figure 12, Figure 26, Figure 27, Figure 28, Figure 29, Figure 30, Figure 31, Table 1, and Table 2;
2) experiments (E9, E10) described in paper's Section 7 whose results are illustrated in the paper's Figure 22 and Figure 23.

(C2): *Auditory Eyesight can reveal unconstrained user inputs. This is proven by the experiment (E3) described in the paper's Section 5, whose results are summarized in the paper's Table 3.*

(C3): *Auditory Eyesight works with different angles, distances, and certain non-line-of-sight scenarios. This is proven by: 1) experiments (E4, E5, E6, E7, E8) described in paper's Section 7 whose results are summarized in the paper's Table 6.*
2) experiments (E9, E10) described in paper's Section 7 whose results are illustrated in the paper's Figure 22 and Figure 23.
3) experiment (E11) described in paper's Section 6.2 whose results are summarized in the paper's Figure 21 and Table 5.

A.4.2 Experiments

(E1): *[30 human-minutes + 10 compute-minutes + 2GB disk]: Proof-of-concept attacks to localize 598 keystrokes on the Apple Magic keyboard from a 0.5-m attack distance.*

How to:
1. Navigate to the 01_proofofconcept_multiround_apple_keys folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 01_proofofconcept_multiround_apple_keys/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.

Preparation:
Start the Parallel Pools in Matlab.

Execution:
Run each line in the main.m file.

Results:
We describe the results and the corresponding contents as follows:

- **E1.01**
  Command:
  run('YZProcessing05_statistics_remoutlier.m')
  Reproduced Result:
  Paper's Figure 6 top

- **E1.02**
  Command:
  run('Statistics101_round1.m');
  Reproduced Result:
  Paper's Figure 7

- **E1.03**
  Command:
  run('Statistics201_figure_2d_1round.m');
  Reproduced Result:
  Paper's Figure 26

- **E1.04**
  Command:
  run('YZProcessing07_2ndroundstatistics.m')
  Reproduced Result:
  Paper's Figure 6 middle

- **E1.05**
  Command:
  run('Statistics101_round2.m');
  Reproduced Result:
  Paper's Figure 8

- **E1.06**
  Command:
  run('Statistics201_figure_2d_2round.m');
  Reproduced Result:
  Paper's Figure 27

- **E1.07**
  Command:
  run('YZProcessing09_3rdroundstatistics')
  Reproduced Result:
  Paper's Figure 6 bottom

- **E1.08**
  Command:
  run('Statistics101_round3.m');
  Reproduced Result:
  Paper's Figure 10

- **E1.09**
  Command:
  run('Statistics101_round5');
  Reproduced Result:
  Paper's Table 2

- **E1.10**
  Command:
  run('Statistics201_figure_2d_5round');
  Reproduced Result:
  Paper's Figure 28

- **E1.11**
  Command:
  run('Statistics_accuracy_calculation');
  Reproduced Result:
  Paper's Table 1

(E2): *[30 human-minutes + 10 compute-minutes + 2GB*

disk]: Proof-of-concept attacks to localize 595 keystrokes on the Razor keyboard from a 0.5-m attack distance.
How to:
1. Navigate to the 02_proofofconcept_multiround_razor_keys folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 02_proofofconcept_multiround_razor_keys/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run each line in the main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E2.01**
  Command:
  run('YZProcessing05_statistics_remoutlier.m')
  Reproduced Result:
  Paper's Figure 12 top

- **E2.02**
  Command:
  run('Statistics201_figure_2d_1round.m');
  Reproduced Result:
  Paper's Figure 29

- **E2.03**
  Command:
  run('YZProcessing07_2ndroundstatistics.m')
  Reproduced Result:
  Paper's Figure 12 middle

- **E2.04**
  Command:
  run('Statistics201_figure_2d_2round.m');
  Reproduced Result:
  Paper's Figure 30

- **E2.05**
  Command:
  run('YZProcessing09_3rdroundstatistics')
  Reproduced Result:
  Paper's Figure 12 bottom

- **E2.06**
  Command:
  run('Statistics201_figure_2d_5round');
  Reproduced Result:
  Paper's Figure 31

- **E2.07**
  Command:
  run('Statistics101_round5');
  Reproduced Result:
  Paper's Table 2

- **E2.08**
  Command:
  run('Statistics_accuracy_calculation');
  Reproduced Result:
  Paper's Table 1

(E3): *[30 human-minutes + 20 compute-minutes + 10GB disk]: Attacks on unconstrained user inputs*
How to:
1. Navigate to the 03_userstudy_1 folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 02_proofofconcept_multiround_razor_keys/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E3.1**
  Command:
  run('Statistics_accuracy_calculation');
  Reproduced Result:
  Paper's Table 3

(E4): *[15 human-minutes + 5 compute-minutes + 2GB disk]: Test case with different angle*
How to:
1. Navigate to the 04_additiontestcase_angle01 folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 04_additiontestcase_angle01/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:

Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E4.1**
  Command:
  run('Statistics_accuracy_calculation');
  Reproduced Result:
  Paper's Table 6, Test Case 1

(E5): *[15 human-minutes + 5 compute-minutes + 2GB disk]: 2nd test case with different angle*
How to:
1. Navigate to the 04_additiontestcase_angle02 folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 04_additiontest-case_angle02/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E5.1**
  Command:
  run('Statistics_accuracy_calculation');
  Reproduced Result:
  Paper's Table 6, Test Case 2

(E6): *[15 human-minutes + 5 compute-minutes + 2GB disk]: 3rd test case with different angle*
How to:
1. Navigate to the 04_additiontestcase_angle03 folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 04_additiontest-case_angle03/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E6.1**
  Command:
  run('Statistics_accuracy_calculation');
  Reproduced Result:
  Paper's Table 6, Test Case 3

(E7): *[15 human-minutes + 5 compute-minutes + 2GB disk]: 4th test case with different angle and 3 microphones*
How to:
1. Navigate to the 04_additiontestcase_angle04_3mics folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 04_additiontest-case_angle04_3mics/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E7.1**
  Command:
  run('Statistics_accuracy_calculation');
  Reproduced Result:
  Paper's Table 6, Test Case 4

(E8): *[15 human-minutes + 5 compute-minutes + 2GB disk]: 5th test case with different angle and 3 microphones*
How to:
1. Navigate to the 04_additiontestcase_angle05_3mics folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 04_additiontest-case_angle05_3mics/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E8.1**
  Command:

run('Statistics_accuracy_calculation');
Reproduced Result:
Paper's Table 6, Test Case 5

(E9): *[15 human-minutes + 5 compute-minutes + 2GB disk]: test case with 1-m distance*
How to:
1. Navigate to the 04_additiontestcase_distance1m folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 04_additiontestcase_distance1m/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E9.1**
  Command:
  run('Statistics101_round3.m');
  Reproduced Result:
  Paper's Figure 22 top

(E10): *[15 human-minutes + 5 compute-minutes + 2GB disk]: test case with 2-m distance*
How to:
1. Navigate to the 04_additiontestcase_distance2m folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 04_additiontestcase_distance2m/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E10.1**
  Command:
  run('YZProcessing05_statistics_remoutlier');
  Reproduced Result:
  Paper's Figure 23 top

- **E10.2**
  Command:
  run('YZProcessing07_2ndroundstatistics');
  Reproduced Result:
  Paper's Figure 23 middle

- **E10.3**
  Command:
  run('YZProcessing09_3rdroundstatistics');
  Reproduced Result:
  Paper's Figure 23 bottom

- **E10.4**
  Command:
  run('Statistics101_round3.m');
  Reproduced Result:
  Paper's Figure 22 bottom

(E11): *[20 human-minutes + 5 compute-minutes + 2GB disk]: non-line-of-sight test case using a laptop*
How to:
1. Navigate to the 04_additiontestcase_nloslaptop folder.
2. Read the 01_readme.txt file.
3. Open main.m file in Matlab and run each line in order to calculate the results. The results will be saved into the 04_additiontestcase_nloslaptop/recording01_keys_interpolated folder.
4. The results will be illustrated in the prompted windows and the console.
Preparation:
Start the Parallel Pools in Matlab.
Execution:
Run main.m file.
Results:
We describe the results and the corresponding contents as follows:

- **E11.1**
  Command:
  run('YZProcessing05_statistics_remoutlier');
  Reproduced Result:
  Paper's Figure 21 top

- **E11.2**
  Command:
  run('YZProcessing07_2ndroundstatistics');
  Reproduced Result:
  Paper's Figure 21 middle

- **E11.3**

Command:
run('YZProcessing13_5throundstatistics.m');
Reproduced Result:
Paper's Figure 21 bottom

- **E11.4**
  Command:
  run('Statistics_accuracy_calculation');
  Reproduced Result:
  Paper's Table 5

A.5 Notes on Reusability

There is a lack of an existing reference study of acoustic side-channel keystroke attacks with publicly available datasets. To address this issue, we publish the dataset, benchmark results, and the software of Auditory Eyesight to reproduce the results.

This artifact is suitable for various research purposes. The dataset can be used to benchmark different acoustic-channel keyboard attack methods in the future. Future works can investigate integrating additional signal processing or other extracted features to improve the attack performance.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Squint Hard Enough: Attacking Perceptual Hashing with Machine Learning

Jonathan Prokos[1*], Neil Fendley[2], Matthew Green[1], Roei Schuster[3], Eran Tromer[4], Tushar M. Jois[1], and Yinzhi Cao[1]

[1]Johns Hopkins University, jprokos4@gmail.com, {jois, mgreen, yzcao}@cs.jhu.edu
[2]Johns Hopkins University Applied Physics Laboratory, fendley@jhu.edu
[3]Vector Institute, roei@vectorinstitute.ai
[4]Tel Aviv University and Columbia University, et2555@columbia.edu

A Artifact Appendix

A.1 Abstract

To the reviewers we provide our entire codebase for running our attacks described from the paper. This is a private GitHub repository which we have packaged into a tarball with a README.md with more information into configuration. This codebase is not released to the public due to ethical reason discussed in A.2.1.

A.2 Description & Requirements

The attack has been tested on a 64-bit Windows machine and on a 2017 Intel Macbook Pro. This limitation is due to the PhotoDNA binary which is architecture specific. Our attacks on PDQ have been tested to work on Linux as well.

To reproduce our major results, you can run our attack as described in the README.md with the appropriate parameters or left blank for default parameters.

A.2.1 Security, privacy, and ethical concerns

One ethical concern is with the protection of the PhotoDNAx64.dll and PhotoDNAx64.so files since these are not publicly released. Additionally, our attack could be modified to use on real world systems and therefore should not be publicly available at the risk of aiding in the actions of a malicious actor[1].

A.2.2 How to access

While we do not provide public access to our codebase as discussed in A.2.1, we do provide perceptualhashing.lol which contains more details regarding our codebase.

A.2.3 Hardware dependencies

The only dependency is to have a machine compatible with the corresponding .dll or .so file for PhotoDNA. This should work on a 64-bit Windows or Intel Mac environment. The rest of our attack (including that on PDQ) works with or without gpu[2].

A.2.4 Software dependencies

The main dependency is pytorch. Additional dependencies listed in requirements.txt.

A.2.5 Benchmarks

Our attack requires the ImageNet 2012 Validation dataset. Instructions to obtain are detailed in the README.md file.

A.3 Set-up

You may install all requirements utilizing our provided setup.py file, simply run *pip install -e .* to do so. For GPU support see the provided README.md for more instruction. Additionally, the ILSVRC2012 Validation Images Dataset tarball and two development kits must be placed in *src/data/imagenet*.

A.3.1 Installation

To install any dependencies run *pip install -e src/*. Then to test functionality you may run *python src/hashattack/hashing/pyphotodna.py* and *python src/hashattack/hashing/pdq_hash_test.py*.

[*]Currently affiliated with Two Six Technologies, LLC (Arlington, VA).
[1]Discussed in more detail in §1 of our paper.

[2]Note that in our evaluation of the PDQ algorithm, we utilize a 72-core machine to achieve our 3-hour runtime. See §5.1 in our paper for more info.

A.3.2 Basic Test

The above mentioned files *pyphotodna.py* & *pdq_hash_test.py* will test the ability to produce a hash with either algorithm. These tests will output a 2-d tensor of integers. To perform a more thorough test with our system you may run *python src/hashattack/hash_atk.py --no-write -cm 10* and *python src/hashattack/fuzzy_collisions_avoidance.py --no-write* which you can kill once it begins looping.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We are able to achieve a successful attack on both PhotoDNA and PDQ utilizing our threat model described in §3. This includes producing a Second-Preimage and Collision Avoidance image at the calculated baseline threshold[3].

A.4.2 Experiments

(E1): *Targeted-Second-Preimage Attack [10 human-minutes + 4 compute-hours per run*[4] *+ 20GB disk]:*
How to: To run the experiment run *python src/hashattack/hash_atk.py* using both the *-ha pdna* and *-ha pdq* flag.
Preparation: Install the pip package using *pip install -e src/* and download the ImageNet dataset as described above.
Execution: Run the provided code using the appropriate flags as mentioned in the results section 5.3. Once all 20 images have converged using both algorithms, you may view the results from the produced tensorboard.
Results: The results will be displayed using tensorboard.

(E2): *Detection Avoidance Attack [10 human-minutes + 1 compute-hour]:*
How to: To run the experiment run *python src/hashattack/fuzzy_collisions_avoidance.py* using both the *-ha pdna* and *-ha pdq* flag.
Preparation: Install the pip package using *pip install -e src/* and download the ImageNet dataset as described above.
Execution: Run the provided code using default parameters. This will output the attack progression as an image over three set distance values.
Results: The results can be viewed from the terminal. The final output image will be saved to disk.

[3] Code used to calculate this baseline provided as well.
[4] Attacks on PDQ require significantly longer to run and should be carried out on a high core cpu machine. Discussed further in §5.1 & §5.3.1.

A.5 Notes on Reusability

This artifact may be used for additional study on a wide range of perceptual hash functions, but providing this code publicly could facilitate malicious activity.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Linear Private Set Union from Multi-Query Reverse Private Membership Test

Cong Zhang[1,2], Yu Chen[3,4,5] (✉), Weiran Liu[6], Min Zhang[3,4,5] and Dongdai Lin[1,2]

[1]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China*
{zhangcong,ddlin}@iie.ac.cn
[3]*School of Cyber Science and Technology, Shandong University, Qingdao 266237, China*
[4]*State Key Laboratory of Cryptology, P.O. Box 5159, Beijing 100878, China*
[5]*Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao 266237, China*
yuchen.prc@gmail.com, zm_min@mail.sdu.edu.cn
[6]*Alibaba Group*
weiran.lwr@alibaba-inc.com

A Artifact Appendix

A.1 Abstract

We introduce our open-source project mpc4j, an efficient and easy-to-use Secure Multi-Party Computation (MPC) library mainly written in Java. Package psu in mpc4j-s2pc-pso of mpc4j contains the implementations, along with configurations needed to replicate our experiments from Section 6. In particular, our artifact supports running and comparing Private Set Union (PSU) protocols with element set sizes up to 2^{20} on machines having 128GB memory. We also provide guidelines for installing dependencies and compiling native libraries needed by mpc4j on different platforms, including x86_64 MacBook, MacBook with M1 chip, Ubuntu 20.04, and CentOS 8. The project is licensed under Apache License 2.0. The source code is available online at https://github.com/alibaba-edu/mpc4j. The stable version for the artifact evaluation is available at https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.4.

In this artifact appendix, we first introduce the minimal hardware and software requirements to get performance reports shown in our paper using mpc4j. Then, we introduce how to install and run mpc4j on different platforms. We note that there are some performance gaps between different platforms, and having complete comparisons for different protocols is very challenging. Aside from that, mpc4j still tries to provide a library for having relatively unified comparisons. We welcome suggestions and performance reports on other platforms with future reproducibility.

A.2 Description & Requirements

We introduce our open-source project mpc4j (Multi-Party Computation for Java), an efficient and easy-to-use Secure Multi-Party Computation (MPC) library mainly written in Java. mpc4j aims to provide an academic library for researchers to study and develop MPC and related protocols in a unified manner. As mpc4j tries to provide state-of-the-art MPC implementations, researchers could leverage the library to have quick and unified comparisons between the proposed and existing protocols.

Package psu in mpc4j-s2pc-pso of mpc4j contains the implementations, along with configurations needed to replicate our experiments from Section 6. Existing Private Set Union (PSU) implementations are under different MPC frameworks and different experimental settings. After carefully studying existing open-source codes, we fully re-implement exisiting PSU protocols and their underlying basic protocols using Java. Evaluators can test PSU protocols on mpc4j by simply using different configuration files. All experiment results shown in Section 6 of our paper are obtained by running mpc4j.

Evaluators can compile and run mpc4j on different 64-bit platforms. We provide guidelines for installing dependencies and compiling native libraries needed by mpc4j on different platforms, including x86_64 MacBook, MacBook with M1 chip, Ubuntu 20.04, and CentOS 8. Note that successfully running all PSU experiments with large element size (i.e., $n = 2^{20}$) requires 128GB RAM. We run our experiments on a single Intel Core i9-9900K with 3.6GHz and 128GB RAM. We note that there are some performance gaps between different platforms. We welcome suggestions and performance reports on other platforms with future reproducibility.

In the full version of our paper, we further provide experiment results on two PSU applications, namely IP blacklist aggregation and Private ID. The related source code has been merged into version v1.0.5[1].

[1] https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.5

A.2.1 How to access

mpc4j is available online on GitHub at https://github.com/alibaba-edu/mpc4j. Evaluators can visit the stable version v1.0.4 (https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.4) to reproduce the experiment results shown in the paper.

A.2.2 Hardware dependencies

mpc4j currently support 64-bit macOS, Ubuntu, and CentOS systems. Evaluators may meet errors when compiling mpc4j on a 32-bit or less system. The reason is that mpc4j uses some 64-bit Single instruction, multiple data (SIMD) operations.

A.2.3 Software dependencies

mpc4j leverages native C/C++ codes to speed up cryptographic operations. The native codes and Java codes are interacted by the Java Native Interface (JNI) technique.

We separate native C/C++ codes into two modules, namely mpc4j-native-tool and mpc4j-native-fhe. mpc4j-native-tool contains native codes for basic cryptographic operations, while mpc4j-native-fhe contains native codes for Fully Homomorphic Encryption (FHE) using SEAL[2]. All basic cryptographic operations in mpc4j-native-tool have alternative pure-Java implementations in mpc4j with the same functionalities and the same data representations. Note that if evaluators only run mpc4j for PSU, there is no need to install SEAL and compile mpc4j-native-fhe. mpc4j-native-tool relies on the following C/C++ libraries:

- GMP (https://gmplib.org/): An efficient library for operations with arbitrary precision integers, rationals, and floating-point numbers.
- NTL (https://libntl.org/): A high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers and for vectors, matrices, and polynomials over the integers and over finite fields, developed by Victor Shoup (https://shoup.net/). Note that one can further introduce GF2X (https://gitlab.inria.fr/gf2x/gf2x) for more efficient operations in a Galois Field. However, since the installation procedure for GF2X is rather complicated, we use NTL by default.
- MCL (https://github.com/herumi/mcl): A portable and fast pairing-based cryptography library. MCL also includes fast Elliptic Curve implementations, especially the optimized implementation for the elliptic curve secp256k1.
- libsodium (https://doc.libsodium.org): A modern, easy-to-use software library for encryption, decryption, signatures, password hashing, and more. libsodium includes efficient implementations for the elliptic curve Curve25519 with APIs for X25519 and Ed25519.
- OpenSSL (https://www.openssl.org/): a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication. OpenSSL includes many efficient cryptographic primitive implementations.

[2] https://github.com/microsoft/SEAL

A.3 Set-up

A.3.1 Installation

Installing mpc4j-native-tool might be a bit complicated for ones who are not that familiar with Unix-like systems, since the procedures differ across platforms. The documentation (README.md) in package mpc4j-native-tool provides instructions for installing mpc4j-native-tool on macOS (x86_64 / aarch64), Ubuntu, and CentOS, respectively.

A.3.2 Basic Test

We develop mpc4j using Intellij IDEA (https://www.jetbrains.com/idea/) and CLion (https://www.jetbrains.com/clion/). After successfully compiling mpc4j-native-tool (Please see readme.md in these modules for more details on how to compile them), evaluators only need the community version of Intellij IDEA to run all basic tests.

Evaluators need to configure IDEA with the following procedures so that IDEA can link to the complied mpc4j-native-tool native libraries.

1. Open "Run->Edit Configurations..."
2. Open "Edit Configuration templates..."
3. Select "JUnit".
4. Add the following command into "VM Options": -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH.

After that, evaluators can run tests of any submodule by pressing the green arrows showing on the left of the source code in test packages. See Section **Demonstration** of readme.md in mpc4j on details for running the tests.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): In our paper, we claimed that we fully re-implement state-of-the-art PSU protocols and their underlying basic protocols using Java. This can be verified by running basic tests in psu (See Section A.3.2 for details), or running experiments with different configuration files (See Section A.4.2 for details).

(C2): In our paper, we claimed that although there is some performance gap, most basic operations in Java and C/C++ have similar performances. This can be verified by running all efficient tests in mpc4j-common-tool (test classes with names end with "EfficiencyTest"). For example, try running "PrpEfficiencyTest" in the package edu.alibaba.mpc4j.common.tool.crypto.prp of the submodule mpc4j-common-tool, evaluators can see the performance comparisons between using AES provided by Java and by AES-NI invoked with JNI.

A.4.2 Experiments

(E1): *[Generate jar file] [5 human-minutes + 5 compute-minutes]: Generate mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar containing the main function entry.*
How to: On the charms bar of IDEA, evaluators can find a button with name "Maven". Press that botton, double-click "mpc4j -> Lifecycle -> package", IDEA would automatically compile and generate mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar containing the main function entry.
Preparation: Evaluators need to successfully running basic tests before generating the jar file.
Execution: Just double-click "mpc4j -> Lifecycle -> package".
Results: The generated file would be located in "mpc4j/mpc4j-s2pc-pso/target".

(E2): *[(optimal) Config network settings] [5 human-minutes + 1 compute-minute]: Config network settings using tc.*
How to: Open a terminal, and execute the following command: "tc qdisc add dev lo root netem rate 10Mbit latency 80ms". Then, the local network is configured as 10Mbit bandwidth with 80ms latency. Evaluators can try other network settings with other parameters, e.g., 100Mbit/80ms, 1Gbit/40ms, 10Gbit/0.02ms.
Preparation: None
Results: Execute "sudo tc qdisc show dev lo" to see if the network is configured correctly.

(E3): *[Run experiments] [10 human-minutes + 5 compute-hour]: Run experiments using different configuration files.*
How to: Open two terminals, one for the PSU server and one for the PSU client. Switch to the dictionary where mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar located (Evaluators can also copy the generated jar file to other dictionaries). For the server's terminal, execute "java -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH -Djava.util.concurrent.ForkJoinPool.common.parallelism=8 -jar mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar CONFIG_SERVER_FILE.txt". For the client's terminal, execute "java -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH -Djava.util.concurrent.ForkJoinPool.common.parallelism=8 -jar mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar CONFIG_CLIENT_FILE.txt". The corresponding server/client configuration files are in "mpc4j-s2pc-pso/conf/psu". Note that evaluators must first run server and then run client.
Preparation: None.
Note: It would take a long time to run if the network has limited bandwidth, long latency, and/or a large set size. See the performance results of our paper to estimate the total running time. Evaluators may find that the setup of SKE-PSU time is quite different from the result presented in Table 3 of our paper. This is because in the paper, we assume Boolean multiplication triples are pre-computed offline and stored locally in a temporary file. Therefore, the setup phase only contains loading Boolean multiplication triples into the memory. In our artifact, we dynamically generate Boolean multiplication triples in the setup phase using silent Oblivious Transfer techniques. In the full version of the paper, we provide the triple generation costs for SKE-PSU, which would be similar to the costs in the setup phase evaluators obtained using the artifact.
Results: Java would run the experiments and generate the performance reports under the current dictionary.

A.5 Notes on Reusability

Evaluators can check and modify server/client configuration files to change IP addresses, port numbers, the element byte length used for PSU. We also provide other configuration examples (marked with #) for specific PSU protocols.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

Improving Logging to Reduce Permission Over-Granting Mistakes

Bingyu Shen, Tianyi Shan, Yuanyuan Zhou
UC San Diego

A Artifact Appendix

A.1 Abstract

This artifact includes the source code of our static analysis tool to improve the access-control logging, as well as the software binaries that we used to conduct evaluation. By executing the tool on the compiled software LLVM bitcode, it produces the logging locations and the list of variables that should be included in the logging.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Tthe stable URL evaluated in the artifact evaluation. `https://github.com/byshen/seclog_ae/releases/tag/v1.0`.

Please pull the latest version if there are any issues or updates. `git@github.com:byshen/seclog_ae.git`.

Rename the folder to `AceInstrument` after clone it. `git clone git@github.com:byshen/seclog_ae.git AceInstrument`

A.2.3 Hardware dependencies

- Please ensure enough memory for compiling LLVM. 16GB memory is enough during our evaluation.

A.2.4 Software dependencies

- Ubuntu 18.04;
- LLVM 9.0.0 for compiling the static analysis source code;
- wllvm for extracting the software binary's LLVM bitcode;
- Python `pandas` for processing the analysis output.

A.2.5 Benchmarks

We provide several testing programs in `dir_bcfiles` directory. For the ten server programs, we provide instructions to compile them in `compile-software.md`. We also provide the compiled bitcode files here. You can download and directly unzip it into `dir_bcfiles` directory.

A.3 Set-up

You should run the following the following command on a Linux platform. We used Ubuntu 18.04 in our experiment.

A.3.1 Installation

```
./build_llvm.sh
```

A.3.2 Basic Test

We provide several tests for quick testing.

1. Modify `BUILD_DIR` to `/home/USER/llvm-9.0.0.obj` and `APP_DIR` to `/home/USER/llvm-9.0.0.src/lib/Transforms/AceInstrument` in `scripts/opt_exec.sh`.
2. Simply run `./scripts/opt_exec.sh test_releatParam` , the output is in `output/test_releatParam.output`.
3. To get a csv format from out put, run the following. `cd output python3 output_parser.py -i test_releatParam`

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *SecLog can suggest logging locations and logging variables for the software. This is proven by the experiment (E1).*

A.4.2 Experiments

(E1): [Logging Enhancement]:
How to: The experiment will simply execute the static analysis tool on the software's llvm bitcode to analyze the logging locations and logging variables.

Preparation: Using the provided bitcode files, or compile from source following the instructions in `compile-software.md` .

Execution: Under the cloned repository.

```
./scripts/opt_exec.sh softwarename
```

Results: The results are in `output/softwarename` Use the script to convert it to a csv format. `python3 output_parser.py -i softwarename`

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: BotScreen: Trust Everybody, but Cut the Aimbots Yourself

Minyeop Choi[1], Gihyuk Ko[2,3], and Sang Kil Cha[1,2]

[1]*KAIST* [2]*Cyber Security Research Center at KAIST* [3]*Carnegie Mellon University*

{okas832,gihyuk.ko,sangkilc}@kaist.ac.kr

A Artifact Appendix

A.1 Abstract

BotScreen is a client-side distributed system for detecting aimbots in FPS games. In its operation, BotScreen is deployed in each client's machine and pre-processes incoming stream of FPS game data in a trusted manner (i.e., in SGX). Then, BotScreen uses a pre-trained deep learning model (SGRU) to detect aimbots in the game. This artifact includes the source code of BotScreen, the (anonymized) dataset of gameplay logs we collected for training and validation of BotScreen, and scripts for reproducing results in the paper. In the following sections, we provide step-wise instructions in order to reproduce the results in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Not applicable.

A.2.2 How to access

The source code of BotScreen is accessible through GitHub at `https://github.com/SoftSec-KAIST/BotScreen/tree/8ad88322f6abbcff6de1974103b275940a839028`. We also provide pre-processed dataset as well as pre-trained weights of the SGRU models used in our experiments at `https://doi.org/10.5281/zenodo.8058051`.

A.2.3 Hardware dependencies

To reproduce the results in our paper locally, a machine with at least one PyTorch-supported GPU is required. Specifically, we recommend using a GPU that has more than 11GB of VRAM. In our experiments, we used a machine equipped with 4 Intel Xeon Silver 4214 CPUs and 4 NVIDIA RTX 2080 Ti (11GB VRAM) GPU cards for training SGRU models, and used a machine equipped with an AMD Ryzen 9 5950X CPU and one NVIDIA RTX 3090 Ti (24GB VRAM) for testing the trained models.

A.2.4 Software dependencies

BotScreen is designed to run on a Linux machine, and we tested it on Ubuntu 20.04 and Ubuntu 22.04. Also, BotScreen is written in Python 3 (3.10), and it depends on packages such as `torch`, `numpy`, `pandas`, and more. Please refer to the provided `requirements.txt` for a full list of required Python packages.

A.2.5 Benchmarks

We make our anonymized pre-processed dataset, pre-trained SGRU models and pre-evaluated data available through Zenodo: `https://doi.org/10.5281/zenodo.8058051`.

A.3 Set-up

A.3.1 Installation

Our implementation of BotScreen depends on several Python packages. The required Python packages can be installed through `pip`, via running the following command:

```
$ pip3 install -r requirements.txt
```

Please also note that our scripts run via GNU Makefile. In the provided `makefile`, the default configurations for training and evaluation parameters are set.

A.3.2 Basic Test

First, download the dataset from `https://doi.org/10.5281/zenodo.8058051` and place the `data_processed` folder into the root of the source code.

Next, one can train SGRU models using downloaded dataset by running the following:

```
$ make train
```

The above command will train a total of 7 SGRU models, where each model is trained according to the 7-fold cross-validation split of the dataset. Specifically, it will produce the following files as output in the `trained_models` directory: `config.json`, and `gru_k0.pt`-`gru_k6.pt`. `config.json` saves the model parameters in JSON format, and `gru_k0-6.pt` contains the

trained weights of each SGRU model from 7-fold cross validation.

Once the SGRU models are trained, one can evaluate their effectiveness by running the following:

```
$ make eval
```

The above command will produce the following files as output in the `trained_models` directory: `eval_k0-eval_k6`. These files are pickle files containing (true label, predicted label) tuples, generated to speed up further evaluation.

A.4 Evaluation workflow

A.4.1 Set up trained SRGU models

There are two ways to obtain the trained SRGU models. One is to train new models from our dataset, and the other is to use the pre-trained model that we provide through Zenodo.

`[10 human-mins + 70 compute-hrs + 15GB disk]`
(Option 1) Train from our dataset as follows,

```
$ unzip BotScreen_data.zip
$ mv BotScreen_data/data_processed ./
$ make train
$ make eval
```

`[10 human-mins + 15GB disk]`
(Option 2) Use the provided pre-trained model. This can be done by copying `data_processed` and `trained_models` from provided artifact into the root of the source code.

```
$ unzip BotScreen_data.zip
$ mv BotScreen_data/trained_models ./
```

A.4.2 Major Claims

(C1): BotScreen can detect aimbot properly. This is proven by Experiment (E1) described in Section 5.2.1 of our paper.

(C2): BotScreen can perform better than previously suggested methods. This is proven by Experiment (E2) described in Section 5.4 of our paper.

(C3): Differences in observations does not impact largely to BotScreen. This is proven by Experiment (E3) described in Section 5.5 of our paper.

A.4.3 Experiments

(E1): `[5 human-minutes]`
The experiment will show the accuracy of the detection model.
How to: Run `make experiments/exp_bench` and see result in `bench.tsv`.
Preparation: Trained model and evaluation data in A.4.1 are needed.
Execution: `$ make experiments/exp_bench`
Results: Check the report file in `bench/bench.tsv` Each line in `bench.tsv` shows the result of model's performance from each split.

(E2): `[15 human-minutes + 30 compute-minutes]`
The experiment will compare the performance between previous tools and BotScreen.
How to: Run experiments in `comp_study` and `experiments/exp_bench`.
Preparation: Dataset, trained model and evaluation data in A.4.1 are needed.
Execution: Execute whole experiments is as follows,

```
$ make experiments/exp_bench
$ make comp_study/th_vara
$ make comp_study/th_acca
$ make comp_study/th_kill
$ make comp_study/ks_acca
$ make comp_study/os_cac
$ make comp_study/os_lac
$ make comp_study/os_smac
$ make comp_study/history
```

Results: Each experiment will print the evaluation result.

```
$ make experiments/exp_bench
```

Botscreen:
best_acc: 0.9764, best_prec: 0.9685, auc_roc: 0.9712
TP: 63, TN: 185, FP: 1, FN: 5

Experiment `comp_study/history` will produces history based detection result of each player in tsv file per methods.

```
$ make comp_study/history
$ ls *.tsv
```

history_botscreen.tsv history_os_LAC.tsv
history_th_Kill.tsv history_ks_AccA.tsv
history_os_SMAC.tsv history_th_VacA.tsv
history_os_CAC.tsv history_th_AccA.tsv

(E3): `[5 human-minutes + 20 compute-minutes]`
The experiment will show the differences in detection results between players.
How to: Run `make experiments/stat_obs`, `make experiments/exp_obs` and see the statistic of differences of observation bewteen clients and see effects of observation rate to accuracy.
Preparation: Dataset, trained model and evaluation data in A.4.1 are needed.
Execution: Run as follows,

```
$ make experiments/stat_obs
$ make experiments/exp_obs
```

Results: After running `experiments/stat_obs`, saved statistic data and visualized results of each game are stored in the `data_loss` directory.

```
$ make experiments/stat_obs
$ ls data_loss/exp_1/
```
figures game_1 game_2 ...

Next, by running `experiments/exp_obs`, you will get a figure in `figures/fig_07_obs.pdf` which is Figure 8 in the paper.

A.5 Notes on Reusability

A.5.1 How to train and evaluate under different parameters

If you want to train and evaluate BotScreen with different model parameters, you can try it by changing the corresponding parameter values defined in `makefile` such as number of hidden units, number of layers, and more. We refer to our paper for a detailed explanation on what each parameter means.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: <HorusEye: A Realtime IoT Malicious Traffic Detection Framework using Programmable Switches>

Yutao Dong[1,2], Qing Li *[2*], Kaidong Wu[1,2], Ruoyu Li[1,2], Dan Zhao[2], Gareth Tyson[3], Junkun Peng[1,2], Yong Jiang[1,2], Shutao Xia[1,2], and Mingwei Xu[4]

[1]Tsinghua Shenzhen International Graduate School, Shenzhen, China
[2]Peng Cheng Laboratory, Shenzhen, China
[3]Hong Kong University of Science and Technology (GZ), Guangzhou, China
[4]Tsinghua University, Beijing, China

A Artifact Appendix

A.1 Abstract

In this artifact, we provide datasets and prototype related to our paper. Specifically, We use Python to implement iForest training and rule generation. We use P4 programming language to deploy Gulliver Tunnel on a H3C S9830-32H-H data center switch with an Intel Tofino switch ASIC, and test hardware performance. We use PyTorch to implement Magnifier and use TensorRT to implement quantization operations. We deploy Magnifier on a GeForce RTX 2080 SUPER. The artifact can reproduce all experimental results reported in the main body of the paper.

Our source code is available at `https://github.com/vicTorKd/HorusEye`.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

N/A.

A.2.2 How to access

We host our source code on GitHub at `https://github.com/vicTorKd/HorusEye`. Specifically, we use this commit for the artifact evaluation: `https://github.com/vicTorKd/HorusEye/releases/tag/v1.0.1`.

A.2.3 Hardware dependencies

CPU, GeForce RTX 2080 SUPER GPU (for Magnifier), H3C S9830-32H-H data center switch with an Intel Tofino switch ASIC (for Gulliver Tunnel hardware performance, optional).

*Corresponding author: Qing Li (liq@pcl.ac.cn)

A.2.4 Software dependencies

The artifact is based on Python, PyTorch, TensorRT, sklearn and other Python packages. All packages can be easily installed with pip; we provide a list of required packages in iot.yaml

A.2.5 Data Set

The extracted data set (used in the article experiment) can be downloaded at link (The compressed file needs to be extracted under the DataSets folder to get HorusEye/DataSets/Pcap).

Also, we can download the original Pcap file at link and re-do the feature extraction. For burst level feature extraction, in pcap_process packet, python files should be executed in the following order (You need to manually change the datasets path in .py files and more detail can be found in README):

1. cd pcap_process
2. python3 pcap2csv_attack.py
3. python3 csv_process_attack.py
4. python3 extract_flow_size.py

For flow level feature extraction:

1. cd HorusEye
2. python3 FE.py.

A.2.6 Models

Our model is placed in AE.py under the model folder, which implements our Magnifier model. In the repository, we also include our baseline model Kitsune.

A.3 Set-up

This section should include all the installation and configuration steps required to prepare the environment to be used for the evaluation of your artifact.

A.3.1 Installation

1. conda install -c anaconda conda-env (anaconda or miniconda is needed)
2. Clone the source code from https://github.com/vicTorKd/HorusEye.
3. conda env create -f iot.yaml
4. If the TensorRT installation fails during the above installation process, you need to install it manually (TensorRT-8.2.1.8).
5. Download extracted datasets.

A.3.2 Basic Test

After downloading the data and ensuring that the data path is correct, you can run the main function in the iForest_detect.py file, which is a simple demo of the rule generation algorithm in Gulliver Tunnel (no GPU required). After ensuring that TensorRT is installed and you have a GPU, you can run the control_plane.py file to get all the experimental results except hardware performance. If there is a programmable switch, you can compile iot_dect_waterflow8.p4 to the switch, and check the hardware performance through p4i.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *HorusEye notably outperforms the Kitsune model in most attacks on both our and public datasets, especially in low false positive scenarios. Additionally, HorusEye and Magnifier achieve comparable detection performance, while HorusEye is even slightly better at detecting most anomalies than Magnifier. This is proven by the experiment (E1) and (E2) described in section 6.7 whose results are illustrated in Table 5 and Table 10.*

(C2): *Rule generation algorithm can convert hundreds of iTrees into whitelists. Then, the whitelists can offload 76% of normal traffic on our dataset, i.e., the throughput gains are 4.13x. This is proven by the experiments (E1) in sections 6.4 and 6.9 whose results are illustrated in Table 1 and Figure 8(a).*

(C3): *Magnifier (fp32) has significantly better packet throughput than Kitsune. Additionally, quantizing Magnifier from 32-bit float (fp32) to 8-bit int (int8) brings about a 2.5x throughput gain while the TPR only slightly drops from 0.675 to 0.636 on our dataset. This is proven by the experiments (E1) and (E3) in sections 6.9 whose results are illustrated in Figure 8.*

(C4): *Gulliver Tunnel is fairly robust to three common black-box attacks: injection attack, low-rate attack and poison attack. This is proven by the experiments (E4) in section 6.8 whose results are reported in Figure 7.*

(C5): *Gulliver Tunnel deployed on the switch can reach 100Gbps and occupies very little TCAM and SRAM. This is proven by the experiments (E5) in section 6.6 whose results are reported in Table 3 and Table 4.*

A.4.2 Experiments

(E1): *[Experiment on our dataset] [30 human-minutes + 30 compute minutes + 20GB disk]: evaluate the detection performance and throughput of the model on our dataset.*
Preparation: *After cloning the source code, configure the conda environment according to "iot.yaml", download the extracted dataset zip and extract it to the root directory of the source code project, and use the model files packaged in the source code project to evaluate it directly without training.*
Execution: *To evaluate HorusEye (Magnifier (fp32) + Gulliver Tunnel), run "python control_plane.py --train False --experiment A --horuseye True" in the root of the source code project. To evaluate Magnifier (fp32) only, run "python control_plane.py --train False --experiment A --horuseye False". You can run "python control_plane.py --help" for more detailed instructions.*
Results: *For the evaluation of HorusEye, the detection performance of HorusEye, the throughput of Magnifier (fp32) and the throughput gain of Gulliver Tunnel will be displayed in the terminal and the detection performance results will also be saved in the csv file located at ". /result/HorusEye/record_attack.csv". For the evaluation of Magnifier (fp32), the detection performance and the throughput of Magnifier (fp32) will be displayed in the terminal and the detection performance results will also be saved in the csv file located at ". /result/Magnifier/record_attack.csv".*

(E2): *[Experiment on public dataset] [30 human-minutes + 30 compute-minutes + 20GB disk]: evaluate the detection performance of the model on public dataset.*
Preparation: *Same as that of (E1).*
Execution: *To evaluate HorusEye (Magnifier (fp32) + Gulliver Tunnel), run "python control_plane.py train False experiment B horuseye True" in the root of the source code project. To evaluate Magnifier (fp32) only, run "python control_plane.py train False experiment B horuseye False". You can run "python control_plane.py help" for more detailed instructions.*
Results: *For the evaluation of HorusEye, the detection performance of HorusEye will be displayed in the terminal and further saved in the csv file located at ".*

/result/Open-Source/HorusEye/record_attack.csv". For the evaluation of Magnifier (fp32), the detection performance of Magnifier (fp32) will be displayed in the terminal and further saved in the csv file located at ". /result/Open-Source/Magnifier/record_attack.csv".

(E3): *[Experiment with int8 model] [30 human-minutes + 20 computeminutes + 20GB disk]: evaluate the detection performance and throughput of the int8 model after quantizing on our dataset.*

Preparation: *In addition to the same as that of (E1), an additional configuration of TensorRT-8.2.1.8 is required.*

Execution: *To evaluate HorusEye (Magnifier (int8) + Gulliver Tunnel), run "python control_plane.py train False experiment C horuseye True" in the root of the source code project. To evaluate Magnifier (int8) only, run "python control_plane.py train False experiment horuseye False". You can run "python control_plane.py help" for more detailed instructions.*

Results: *For the evaluation of HorusEye, the detection performance of HorusEye and the throughput of Magnifier (int8) will be displayed in the terminal and the detection performance results will also be saved in the csv file located at ". /result/HorusEye/record_attack.csv". For the evaluation of Magnifier (int8), the detection performance and the throughput of Magnifier(int8) will be displayed in the terminal and the detection performance results will also be saved in the csv file located at ". /result/Magnifier/record_attack.csv".*

(E4): *[Experiment on robustness] [30 human-minutes + 10 compute-minutes + 20GB disk]: evaluate the detection performance (robustness) of Gulliver Tunnel under three common black-box attacks.*

Preparation: *Same as that of (E1).*

Execution: *Run "python control_plane.py train False experiment D horuseye False" in the root of the source code project. Note that after running this experiment, the parameters of Gulliver Tunel will be modified, and retraining is required when re-running other experiments. You can retrain by setting "train" to True, as in "python control_plane.py train True experiment A horuseye True". You can run "python control_plane.py help" for more detailed instructions.*

Results: *The detection performance of Gulliver Tunnel will be displayed in the terminal and the detection performance results of each type of black-box attacks will be saved in the csv file located at "./result/df_robust_result_robust_type.csv".*

(E5): *[Hardware performance] [30 human-minutes + 10 compute-minutes + 20GB disk]: evaluate the hardware performance of Gulliver Tunnel after deployment on the programmable switch. (optional)*

Preparation: *An Intel Tofino switch ASIC and a traffic generator (e.g., SPIRENT N11U).*

Execution: *(1) Use winscp to log in to the switch, and put p4 file into the switch, e.g., /mnt/onl/data/bf-sde-9.1.0/pkgsrc/p4-examples/p4_16_programs/iot/iot_dect_waterflow8.p4. (2) Use ssh to log in switch, cd $SDE/pkgsrc/p4-build. (3) ./configure –prefix=$SDE_INSTALL –with-tofino –with-bf-runtime P4_NAME=iot_dect P4_PATH=$SDE/pkgsrc/p4-examples/p4_16_programs/iot/iot_dect_waterflow8.p4 P4_VERSION=p4-16 P4C=p4c. (4) make. (5) make install. (6) Use a traffic generator to test forwarding.*

Results: *The resource occupation performance of Gulliver Tunnel will be displayed in $SDE/pkgsrc/p4-build/tofino/iot_dect/pipe/log. The single port forwarding performance can be shown in Traffic Analyzer.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926.

USENIX'23 Artifact Appendix: Towards Targeted Obfuscation of Adversarial Unsafe Images using Reconstruction and Counterfactual Super Region Attribution Explainability

Mazal Bethany, Andrew Seong, Samuel Henrique Silva,
Nicole Beebe, Nishant Vishwamitra, Peyman Najafirad
The University of Texas at San Antonio

A Artifact Appendix

A.1 Abstract

We release the image reconstruction and explainability-based image obfuscation code that was used in our paper's experiments.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We do not release the datasets used in our paper due to various privacy and ethical reasons.

A.2.2 How to access

Stable Reference: https://github.com/SecureAIAutonomyLab/uGuard/tree/dbd98a38611af486d992b36024f78a96f99d43cc

A.2.3 Hardware dependencies

We ran our experiments on a desktop system with an Nvidia 1080 ti GPU, and 64 GB RAM. CUDA compatible GPU's are required for our project.

A.2.4 Software dependencies

The project was designed to be run in a conda environment using python. An extensive list of software dependencies is contained within the the environment.yml file on the project repository.

A.2.5 Benchmarks

We do not release datasets or model weights, though our code is extendable to other datasets.

A.3 Set-up

A.3.1 Installation

To set up the system, users should first install conda to their system, clone the code repository, navigate to the repository, being building the environment using "conda env create -f environment.yml" and then activate the conda environment using "conda activate uGuard".

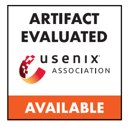
A.3.2 Basic Test

To run the code, users can navigate to the scripts directory. Users would need to add their own datasets to the datasets directory and edit the scripts so that they point to the correct datasets, save paths, etc.

To test that all packages are correctly installed, users can simply run the scripts. If the only errors received are related to missing files due to missing folders or model weights, this indicates that the environment is functioning correctly.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: Is Your Wallet Snitching On You? An Analysis on the Privacy Implications of Web3

Christof Ferreira Torres
ETH Zurich

Fiona Willi
ETH Zurich

Shweta Shinde
ETH Zurich

A Artifact Appendix

This work explores the privacy implications that Web3 technologies such as decentralized applications and wallets have on users. To this end, we build a framework that measures exposure of wallet information. First, we study whether information about installed wallets is being used to track users online. We analyze Tranco's top 100K websites and find evidence that 1,325 websites run scripts to probe whether users have wallets installed in their browser. Second, we measure whether decentralized applications and wallets leak the user's unique wallet address to third-parties. We intercept the traffic of decentralized applications and wallets and find over 2000 leaks across 211 applications and more than 300 leaks across 13 wallets. This appendix details how to access our artifact (implementation of framework and our dataset) and to reproduce our results.

A.1 Abstract

Our artifact consists of source code, datasets, and scripts to generate the results of our paper. We aim for Artifacts Available, Artifacts Functional, and Results Reproduced badges. In more detail, we open-source the implementation of our framework via GitHub. We also provide our dataset of collected site snapshots on the Top 100K websites, DApps and wallet extensions, which can be utilized to reproduce the figures and tables included in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

At its core, our framework visits websites and interacts with wallet extensions automatically while recording any outgoing traffic such as HTTP requests, WebSocket requests and cookies. Hence, there may be security issues if the user provides a malicious website or wallet extension to interact with. We advise testing our framework on websites and wallet extensions that the user trusts. However, as part of our reproducibility experiments, our framework tries to crawl some of the top websites provided by Tranco. Hence, it might be that the users visit illegal websites or websites with adult content, depending on which country they reside. In terms of privacy, websites may fingerprint or track the utilization of our framework.

A.2.2 How to access

Code: The code of our artifact is available via the following GitHub repository: `https://github.com/christoftorres/Web3-Privacy/commit/d5884c73dba5783ea3dc419433680596ea90e882`. The repository provides a detailed `README.md` file on how to set up our framework and how to use it to reproduce our results. For artifact evaluation, please checkout the branch "artifact-review".
Data: The GitHub repository contains mainly the code. Most of the data that is necessary to reproduce our results needs to be downloaded via Zenodo: `https://zenodo.org/record/8071006`.

A.2.3 Hardware dependencies

Our framework has been evaluated using an Apple MacBook Pro with an Apple M1 Pro chip containing 10 cores and 32 GiB of memory. However, we also tested our framework on a machine with a 12th Gen Intel(R) Core(TM) i9-12900K containing 16 cores. We recommend using something similar. Going as low as 16 GiB of memory and 30 GiB of storage should work as well.

A.2.4 Software dependencies

Our framework has been tested on MacOS Monterey version 12.6 and on 64 bit Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-67-generic). The framework leverages Node.js, Python, and MongoDB.

A.2.5 Benchmarks

This artifact already provides all the necessary data (e.g., Tranco top 100K websites, blocklists, etc.) that is required to test its functionality and reproduce the results from our paper.

A.3 Set-up

For more details and easy to use copy and paste commands, we refer to the README.md of https://github.com/christoftorres/Web3-Privacy.

A.3.1 Installation

1. Git clone our repository: https://github.com/christoftorres/Web3-Privacy. The rest of the instructions assume you are in the project directory using a terminal window.
2. For artifact reviewers: "git checkout artifact-review"
3. Install Python3 and its dependencies:
   (a) apt-get update -q && apt-get install -y wget curl unzip software-properties-common python3-distutils python3-pip python3-apt python3-dev
   (b) python3 –version
   (c) pip3 install -r requirements.txt
4. Install Node.js and its dependencies:
   (a) curl -sL https://deb.nodesource.com/setup_18.x | bash -
   (b) apt-get update -q && apt-get install -y nodejs
   (c) node –version && npm –version
   (d) cd framework/tracker-radar-collector && npm install
   (e) cd framework/request-interceptor && npm install
5. Install MongoDB:
   (a) wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | apt-key add && echo "deb [arch=amd64,arm64] https://repo.mongodb.org/apt/ubuntu bionic/mongodb-org/4.4 multiverse" | tee /etc/apt/sources.list.d/mongodb-org-4.4.list && apt-get update && apt-get install -y mongodb-org

A.3.2 Basic Test

We can perform the following two basic tests to test whether our framework is installed properly:

1. **Test Web3-based browser fingerprinting detection:**
   (a) cd framework/tracker-radar-collector
   (b) npm run crawl – -u "https://www.nytimes.com" -o ./data/ -f -v -d "requests,targets,apis,screenshots"
   (c) cat data/www.nytimes.com_89db.json | grep ethereum -C 10
   (d) The terminal should display "window.ethereum" along with other JavaScript properties.
   (e) In case nytimes.com does not return any results, it might be that they updated their script. In this case you can try "https://xhamster.com", however, be aware that this is a website with adult content.
2. **Test wallet address leakage detection:**
   (a) cd framework/request-interceptor
   (b) node run –interactive -u https://notional.finance/portfolio –debug verbose -w metamask-chrome-10.22.2 -t 30
   (c) cat notional.finance.json | grep 7e4abd63a7c8314cc28d388303472353d884f292
   (d) The terminal should display several entries which highlight that the wallet address is being leaked by the DApp to third-parties.

A.4 Evaluation workflow

Disclaimer. The web is constantly changing, websites may remove or add scripts from one day to the other. Hence, it might be that some websites that were found to be probing user's wallets in the past, might not be doing so anymore. Moreover, our framework is only as good as the components that is uses (e.g., TRC, Puppeteer, etc.). Thus, if Puppeteer is not able to intercept a request or if TRC is not able to load a website properly, then our framework might not be able to detect JavaScript calls to wallet APIs or detect leaks.

A.4.1 Major Claims

(C1): *There are at least 10 websites among Tranco's top 1K websites that probe whether their users have a wallet extension installed in their browser. This is proven by the experiment (E1) described in Section 4.2 whose results are reported in Table 3.*

(C2): *While most websites only probe for the window.ethereum object, there are also websites that probe for different combinations of wallet APIs. This is proven by the experiment (E1) described in Section 4.2 whose results are reported in Table 4.*

(C3): *Wallet extension probing is being performed mostly by websites categorized as adult content. This is proven by the experiment (E1) described in Section 4.2 whose results are reported in Table 5.*

(C4): *The top 10 third-party scripts that probe for wallet APIs also collect other information that is required to perform browser fingerprinting. This is proven by the experiment (E1) described in Section 4.2 whose results are reported in Table 6.*

(C5): *The combination of five popular blocklists results in 56% of the third-party scripts being blocked. This is proven by the experiment (E1) described in Section 4.2 whose results are depicted in Figure 5.*

(C6): *We detect more leaks than Winter et al. [66] due to the fact that we also analyze HTTP POST requests and WebSocket requests. This is proven by the experiment (E2) described in Section 4.3.1 whose results are reported in Table 7.*

(C7): *Infura is the most widespread third-party towards where wallet addresses are leaked. This is proven by the experiment (E3) described in Section 4.3.1 whose results are reported in Table 8.*

(C8): *Exchanges leak the most often the user's wallet address to third-parties. This is proven by the experiment (E3) described in Section 4.3.1 whose results are reported in Table 9.*

(C9): *13 out of 100 wallet extensions leak the user's wallet address to third-parties. This is proven by the experiment (E4) described in Section 4.3.2 whose results are reported in Table 10.*

A.4.2 Experiments

(E1): *[Analyze Web3-Based Browser Fingerprinting] [5 human-minutes + 5 compute-minutes]: This experiment analyses the data that was gathered through our crawl on the top 100K websites in November 2022 and parsed via our browser fingerprinting detection script.*

How to: *Performing the entire crawl from scratch on the top 100K websites would take very long and result in different results as the web keeps on changing. Therefore, we provide a dump of our MongoDB collection which already contains the data processed by our "detect_fingerprinting.py" script. The dump can be imported to analyze our findings. However, for reproducibility purposes we also provide a raw snapshot of all the requests and JavaScript calls that were collected via our crawl in November 2022.*

Preparation: *Download the browser fingerprinting datasets using "wget https://zenodo.org/record/8071006/files/browser-fingerprinting-datasets.zip && unzip browser-fingerprinting-datasets.zip && mv datasets browser-fingerprinting/ && rm browser-fingerprinting-datasets.zip" and the browser fingerprinting results using "wget https://zenodo.org/record/8071006/files/browser-fingerprinting-results.zip && unzip browser-fingerprinting-results.zip && mv results browser-fingerprinting/ && rm browser-fingerprinting-results.zip". Change the working directory using "cd browser-fingerprinting/results". Import the MongoDB dump by first creating a temporary directory using "mkdir db". Afterwards, run MongoDB locally using the temporary directory: "mongod –dbpath db" and import the collection using "mongoimport –uri="mongodb://localhost:27017/web3_privacy" –collection fingerprinting_results –type json –file fingerprinting_results.json.*

Execution: *After having imported the MongoDB dump and making sure that MongoDB is running, we can run the analysis script by first chaining our working directory using "cd browser-fingerprinting/analysis" and running the analysis script using "python3 analyze_detected_fingerprinting.py".*

Results: *The terminal will display Tables 3, 4, 5, and 6, which should be equivalent to the tables included in the paper. Moreover, the script will also output in the same directory as the analysis script a PDF file named "blocklists.pdf" which should be equivalent to Figure 5 in the paper. Please note, in order to be able to plot the file "blocklists.pdf" you are required to have LaTeX installed on your system.*

(E2): *[Analyze Wallet Address Leakage] [5 human-minutes + 5 compute-minutes]: This experiment analyzes the requests collected via our interceptor on the 66 DApps by Winter et al. and compares them to the results of Winter et al.*

How to: *Performing the entire crawl from scratch on 66 websites would take very long and result in different results as the web keeps on changing. Therefore, we provide a snapshot of all the requests that we intercepted during our crawl.*

Preparation: *Download the wallet address leakage datasets using "wget https://zenodo.org/record/8071006/files/wallet-address-leakage-datasets.zip && unzip wallet-address-leakage-datasets.zip && mv datasets wallet-address-leakage/ && rm wallet-address-leakage-datasets.zip" and the wallet address leakage results using "wget https://zenodo.org/record/8071006/files/wallet-address-leakage-results.zip && unzip wallet-address-leakage-results.zip && mv results wallet-address-leakage/ && rm wallet-address-leakage-results.zip". Change the working directory using "cd wallet-address-leakage/analysis".*

Execution: *Run the comparison script using "python3 find-leaks-and-scripts-winter-et-al.py ../results/whats_in_your_wallet/crawl ../datasets/whats_in_your_wallet".*

Results: *The terminal will display at the end Table 7, which should be equivalent to Table 7 included in the paper.*

(E3): *[Analyze Wallet Address Leakage] [5 human-minutes + 60 compute-minutes]: This experiment analyzes the requests collected via our interceptor on the DAppRadar.com dataset.*

How to: *Performing the entire crawl from scratch on DAppRadar.com dataset would take very long and result in different results as the web keeps on changing. Therefore, we provide a snapshot of all the requests that we intercepted during our crawl.*

Preparation: *Change the working directory using "cd wallet-address-leakage/analysis".*

Execution: *Run the analysis script using "python3 find-leaks-and-scripts-dapps.py".*

Results: *The terminal will display at the end Tables 8 and 9, which should be equivalent to Tables 8 and 9 included in the paper.*

(E4): *[Analyze Wallet Address Leakage] [5 human-minutes + 5 compute-minutes]: This experiment analyzes the requests collected via our interceptor on 100 popular wallet extensions.*

How to: *Performing the entire crawl from scratch on the wallet extensions dataset would take very long as it requires a large amount of manual interaction with each wallet extension. Therefore, we provide a snapshot of all the requests that we intercepted during our crawl.*

Preparation: *Change the working directory using "cd wallet-address-leakage/analysis".*

Execution: *Run the analysis script using "python3 find-leaks-and-scripts-wallet-extensions.py".*

Results: *The terminal will display at the end Table 10, which should be equivalent to Table 10 included in the paper.*

A.5 Notes on Reusability

The folder "browser-fingerprinting/datasets/tranco" contains the list of websites that have been crawled during our study. Researchers can reuse this list to try to reproduce the results or perform followup studies. The folder "browser-fingerprinting/results/crawl" contains snapshots of all the JavaScript calls and requests that we collected during our study. Researchers can reuse these snapshots to compare to reproduce our results and compare their own results to ours. Researchers can easily extend our framework to analyze other wallet APIs by modifying the files "walletSimulator.js" and "walletSimulatorWithAntiBotDetection.js" contained in "framework/tracker-radar-collector/helpers".

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artefact Appendix: CAPSTONE: A Capability-based Foundation for Trustless Secure Memory Access

Jason Zhijingcheng Yu
National University of Singapore

Conrad Watt
University of Cambridge

Aditya Badole
National University of Singapore

Trevor E. Carlson
National University of Singapore

Prateek Saxena
National University of Singapore

A Artefact Appendix

A.1 Abstract

This artefact includes the following components:

- Functional prototypes of CAPSTONE. More specifically, those include the emulator CAPSTONEEmu, the compiler CAPSTONECC, and the library CAPSTONELib, along with sample source codes for the case studies discussed in the paper that are runnable with the aforementioned tools. This part resides under the `functional` subfolder.
- The GEM5 model used for evaluating CAPSTONE. This includes the source code and the scripts for building both the model and the benchmarks as well as for running the experiments presented in the paper. This part resides under the `gem5` subfolder.

All the artefact components have been made publicly available in the source format. To improve portability, reduce the impact on the artefact user's own system, and ease the process of using the artefact itself, we provide the option of building and running the artefact inside Docker containers.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Building and running this artefact are not expected to cause security or privacy risks to the artefact user. Nor is it expected to raise ethical concerns.

A.2.2 How to access

The artefact is available on Github at https://github.com/jasonyu1996/capstone (revision hash: 9b5319c). Note that this Github repository includes submodules. To download all the included components, make sure that you supply `--recurse-submodules` when you clone it, or run

```
git submodule update --init --recursive
```

afterwards.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

The platform to use this artefact on is expected to have Bash installed and support running x86-64 Docker containers (Docker version 20 or later recommended).

A.2.5 Benchmarks

For copyright reasons, we have not included SPEC CPU 2017, the benchmark suite used for the evaluation experiments with the GEM5 model. The user needs to supply the benchmark suite by themselves if they wish to run those experiments.

A.3 Set-up

We have included detailed instructions in the `README.md` files in the Github repository. Below we only reproduce the brief steps.

A.3.1 Installation

Functional prototypes Change the working directory to `functional`. Build the Docker image with

```
./build
```

GEM5 model Change the working directory to `gem5`. Build the GEM5 model for Capstone and the baseline model with

```
./run-docker build
```

Note that the above command will pull `corank/gem5-dev` if the Docker image does not exist locally. You can pull it manually with

```
docker pull corank/gem5-dev
```

or alternatively, build it on your own machine

```
cd docker-build
docker build . -t corank/gem5-dev
```

To build SPEC CPU 2017, place it under `./spec` and apply a patch before running the build script

```
(cd spec && patch -p1 \
< ../tests/capstone/speckle/spec17.patch)
./run-docker build-spec
```

A.3.2 Basic test

Functional prototype Test with

```
./run compiler/samples/dummy.c
```

You should be able to see the output which starts with

```
18: GPR 1 = Value 0
19: halted
```

followed by runtime statistics.

GEM5 model Run

```
./run-docker run-hello
```

The output should include

```
Hello gem5!
```

A.4 Evaluation Workflow

The evaluation workflow applies to the GEM5 model only.

A.4.1 Major Claims

(C1): In comparison to the baseline RISC-V model, the GEM5 model for CAPSTONE exhibits overhead that ranges from 0 to 50% across SPEC CPU 2017 workloads (as shown in Figure 3 in the paper).

A.4.2 Experiments

(E1): estimated 30 compute-hours (when running workloads in parallel):
How to: Please follow the following steps to run this experiment.
Preparation: Build both the GEM5 model and the benchmark suite SPEC CPU 2017 following the steps described in Section A.3.1.
Execution: Run SPEC CPU 2017 with the GEM5 model for CAPSTONE first

```
./run-docker run-capstone --multiproc
```

followed by the baseline RISC-V model

```
./run-docker run-baseline --multiproc
```

Note that the `--multiproc` flag can be omitted, but that will result in the experiments being run on a single CPU core, which would be slow and hence not recommended.
Results: The logs are available in `./outputs`. To parse the logs and produce the data shown in Figure 3 in the paper,

```
./run-docker collect-results
```

which prints to the standard output the parsed results in the LaTeX table format.

A.5 Notes on Reusability

The behaviours of the compiler CAPSTONECC can be adjusted through command line flags. Please read the source code `compiler/src/main.rs` or `README.md` for details.

For the GEM5-based evaluation, it is possible to change the number of fast-forwarded instructions, and the number of instructions to simulate after fast-forwarding. This is achieved by adjusting the variables `GEM5_SKIP` and `GEM5_LIM` in scripts `docker-scripts/run-capstone` and `docker-scripts/run-baseline`. Similarly, the size of the node cache can be set through the variable `GEM5_NCACHE`. To print more data, set `GEM5_FLAGS` to `--debug-flags=...` with the debug flags defined in GEM5.

A.6 Version

Based on the LaTeX template for Artefact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artefact can be found at https://secartefacts.github.io/usenixsec2023/.

FloatZone: Accelerating Memory Error Detection using the Floating Point Unit

Floris Gorter* f.c.gorter@vu.nl
Enrico Barberis* e.barberis@vu.nl
Raphael Isemann r.isemann@vu.nl
Erik van der Kouwe vdkouwe@cs.vu.nl
Cristiano Giuffrida giuffrida@cs.vu.nl
Herbert Bos herbertb@cs.vu.nl

Vrije Universiteit Amsterdam
* Equal contribution joint first authors

A Artifact Appendix

A.1 Abstract

In this artifact we provide the means to reproduce our main results. Specifically, we show that our memory sanitizer, FloatZone, can detect memory errors, and that FloatZone's performance is higher than traditional comparison-based solutions. We have validated the artifact using an Intel i9-13900K CPU running Ubuntu 22.04 with a stock v5.15 Linux kernel. Our source code is available at: `github.com/vusec/floatzone`.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We require the evaluators to obtain the SPEC CPU benchmarking suites themselves, since we cannot distribute the licensed software. As a memory sanitizer, FloatZone poses no risks to the security of the target machine.

A.2.2 How to access

The files for the artifact evaluation are available at: `https://github.com/vusec/floatzone/releases/tag/ae-final`.

A.2.3 Hardware dependencies

While FloatZone has no strict hardware requirements (we assume x86-64), we highly recommend using a modern Intel CPU, since FloatZone's performance depends on the throughput of the floating point unit. We have ran benchmarking experiments on various CPUs (see Figure 6 for more information).

A.2.4 Software dependencies

Some packages from the Ubuntu package manager are required to be installed to accomodate for the build process of FloatZone (e.g., for building LLVM). These are described in the Set-up section.

A.2.5 Benchmarks

For this artifact we benchmark using the SPEC CPU2006 benchmarking suite.

A.3 Set-up

We recommend using a bare-metal desktop system with 32GB of RAM, running Ubuntu 22.04, glibc 2.35, and a stock v5.15 Linux kernel.

A.3.1 Installation

1. Obtain the artifact source:

```
git clone \
https://github.com/vusec/floatzone.git \
 --recurse-submodules
cd floatzone
```

2. Install some standard dependencies:

```
sudo apt install ninja-build cmake gcc-9 \
autoconf2.69 bison build-essential flex \
texinfo libtool zlib1g-dev
```

3. Configure the FloatZone environment by editing the `env.sh` file and modifying the `FLOATZONE_TOP` variable to reflect the working directory of the system, and then run:

```
source env.sh
```

4. Install the FloatZone infrastructure by running:

```
./install.sh
```

NOTE: installing LLVM can take up a lot of RAM when using multiple cores. If the compilation process crashes, use the `ninja -j <cores>` parameter inside `install.sh` to use less cores.

A.3.2 Basic Test

To test the functionality of FloatZone, we provide a test case in the `example` directory. Run `make` to obtain three versions of the `buggy` binary: uninstrumented, instrumented by FloatZone, and instrumented by ASan. The program contains a buffer of size 16, and the command line argument is used as an index in this array. Confirm that executing:

```
./buggy_floatzone_run_base 16
```

results in an error report containing a faulting address, while using index 15 does not. See the README on GitHub for the exact expected output format.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *FloatZone can detect spatial and temporal memory errors bounded by its security guarantees (as described in Section 5). This is proven by experiment E1.*

(C2): *FloatZone provides high performance in terms of runtime and memory overhead (see Sections 7.3 and 7.4). This is proven by experiment E2.*

A.4.2 Experiments

(E1): *[1 human-hour]: Confirming memory error detection.*

How to: *The Juliet Test Suite can be used to confirm that FloatZone detects memory errors. This suite contains test cases for spatial and temporal memory errors.*

Preparation: *Make sure that SEGFAULTS are reported: in the `runtime` directory, edit `wrap.c` and ensure that `CATCH_SEGFAULT` is set to 1. Run `make` inside this directory to ensure the shared object file is up-to-date. No further preparation is required if the `env.sh` and `install.sh` scripts have been used. If interested, FloatZoneExt (with partial overflow detection capabilities, see Section 5 and Figure 5) can be tested by modifying the FLOATZONE_MODE variable to also contain the term '`just_size`' in `env.sh`.*

Execution:

```
python3 run.py run juliet \
 floatzone_O0 --build --cwe 121 122 \
 124 126 127 415 416
```

Results: *FloatZone and FloatZoneExt can detect most of the spatial and temporal memory errors present in the Juliet Test Suite. The expected results are reported in Table 1 and Section 7.2.*

(E2): *[15 human-minutes + 5 compute-hours]: Confirming runtime and memory performance*

How to: Run the SPEC CPU2006 benchmarking suite instrumented by FloatZone and ASan, and observe the performance overhead.

Preparation: SPEC CPU2006 needs to be available on the system and the `FLOATZONE_SPEC06` variable in `env.sh` needs to point to the directory where it is installed. For the artifact evaluators, if they cannot obtain SPEC CPU2006, we can provide access to a machine ready to run SPEC. In order to run SPEC CPU and its benchmarks, we make use of a public infrastructure under the `infra` directory. The infra also makes sure the SPEC binaries run pinned to core 0. Make sure that the necessary python packages are installed:

```
pip3 install psutil terminaltables
```

Then, since some of the SPEC binaries contain false positives (see Table 3), in the `runtime` directory, edit `wrap.c` and ensure that `SURVIVE_EXCEPTIONS` is set to 1. Run `make` inside this directory to ensure the shared object file is up-to-date. As can be seen in the `wrap.c` source file, this only ensures that exceptions do not abort, and the program continues executing where it left off.

Execution: We make use of the `run.py` script to run SPEC CPU2006 along with the intended instrumentations. Execute the following command, which runs SPEC CPU2006 for three runs: the baseline, one with ASan, and one with FloatZone, and hence takes multiple hours:

```
python3 run.py run spec2006 default_O2 \
asan_O2 floatzone_O2 --build \
--parallel=proc --parallelmax=1
```

Results: To obtain the results from the SPEC CPU2006 runs, we again make use of the `run.py` script. Find the corresponding output folder in the `results` directory that matches the start timestamp (e.g.: `results/run.2023-06-19.13-56-59`). Then execute the following command, replacing the directory with the one just obtained:

```
python3 run.py report spec2006 \
results/run.2023-06-19.13-56-59 \
--aggregate geomean --field runtime:median \
maxrss:median
```

The output of this command can then be used to calculate the runtime and memory overheads for each individual binary, as well as for the geomean. As reported in Table 4: if ran on the i9-13900K machine, the expected runtime overhead for FloatZone is 36.4%, and 77.8% for ASan, while the memory overhead is expected to be 182% and 237%, for FloatZone and ASan, respectively.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: PUMM: Preventing Use-After-Free Using Execution Unit Partitioning

Carter Yagemann
The Ohio State University*

Simon P. Chung
Georgia Institute of Technology

Brendan Saltaformaggio
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

A Artifact Appendix

A.1 Abstract

The artifact is a code repository (with supporting documentation) for PUMM, a runtime Linux defense that prevents use-after-free vulnerabilities from being exploitable. PUMM consists of an offline profiling phase that generates a security profile for an online monitor that wraps libc's memory management functions (e.g., malloc, free, etc.).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact should not pose any inherent security, privacy, or ethical concerns. No preexisting data is read or transmitted and no preexisting defenses are disabled or bypassed. If the user decides to install software containing use-after-free vulnerabilities for the purposes of verifying PUMM's security claims, they do so at their own risk. For security and ethical reasons, the artifact does not include any vulnerable programs.

A.2.2 How to access

The artifact is a code repository and documentation that can be accessed on Github: https://github.com/carter-yagemann/PUMM/tree/91e58cd5d929e25d0b83fdfd0ec3c5517e2a32e7.

A.2.3 Hardware dependencies

PUMM requires a *baremetal* Linux computer (virtual machines are not supported) running an Intel Core processor (e.g., i3, i5, i7, etc.).

A.2.4 Software dependencies

The preferred environment for running PUMM is Debian Buster, however PUMM should work with any recent version of Debian or Ubuntu. PUMM has several software dependencies, including cmake, gawk, Graph-Tool, and Linux Perf. Users should follow the Setup section in the README.

A.2.5 Benchmarks

The primary benchmark used in the paper is a collection of open source programs with known use-after-free vulnerabilities and publicly reported steps for triggering these issues. Reference IDs are provided in Table 1 of the paper and supporting artifacts can be located by looking up the CVE in the National Vulnerability Database[1] or the issue number in the affected program's bug tracking website.

Additionally, performance and memory overheads are measured using the SPEC CPU 2006 standard benchmark. FFmalloc and MarkUs are used as baseline defenses for comparison.

A.3 Set-up

A.3.1 Installation

Users should follow the Setup section of the README.

A.3.2 Basic Test

Users should follow the Usage section of the README, which will cover all core components of PUMM, using /bin/ls as the target program to be protected:

1. Collecting runtime traces of the target program.
2. Generating a security profile for the target program.
3. Using PUMM's runtime monitor (and generated profile) to protect the target program during execution.

The README provides example terminal outputs for each step for users to verify success. Notice that because the basic test does not involve launching an exploit, the expected outcome of using the online portion of PUMM is no observable change to the target program's behavior.

*Work done while at the Georgia Institute of Technology.

[1] https://nvd.nist.gov/

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PUMM is able to prevent 40 real-world use-after-free exploits targeting 26 popular open source programs. This is proven by experiment (E1) described in Section 4.1 of the paper and illustrated in Table 2.

(C2): PUMM prevents the same vulnerabilities as FFmalloc and MarkUs while aborting in fewer cases. This is proven by experiment (E2) described in Section 4.1 of the paper and illustrated in Table 3.

(C3): PUMM incurs 2.74% less performance overhead than FFmalloc and 52.0% less memory overhead than MarkUs on the SPEC CPU 2006 standard benchmark. This is proven in experiment (E3) described in Section 4.2 of the paper and illustrated in Figures 4 and 5.

(C4): PUMM is able to prevent use-after-free exploitation in all but 4 cases out of 3,000 synthetically generated vulnerabilities. This is proven by experiment (E4) described in Section 4.3 of the paper and illustrated in Figures 8, 9, and 10.

A.4.2 Experiments

(E1): [5 human-days + 3 compute-days + 300GB storage]: Collected vulnerable programs and triggering inputs, record traces, generate profiles, and verify that the vulnerability is triggered without PUMM's defense and then prevented with PUMM.

Preparation: For each vulnerability in Figure 2 of the paper, the affected software must be downloaded and compiled. A triggering input for the program must also be downloaded and verified to be working (e.g., by causing a segmentation fault).

Execution: PUMM should be used to generate a security profile for the target program, using the recorded traces.

Results: Running the target program with the triggering input and PUMM's online monitor activated should prevent the use-after-free from being exploitable. Notice that the target program may still crash, so verification requires debugging.

(E2): [4 human-hours + 30 compute-minutes + 1GB storage]: Collected vulnerable programs and triggering inputs should be executed with PUMM, MarkUs, and FFmalloc to observe whether the vulnerability is exploitable.

Preparation: Download and compile FFmalloc and MarkUs. Dataset preparation is covered in E1.

Execution: The vulnerable program should be executed with the triggering input for each defense.

Results: Running the target program with the triggering input and PUMM's online monitor activated should prevent the use-after-free from being exploitable. Notice that the target program may still crash, so verification requires debugging.

(E3): [2 human-hours + 2 compute-hours + 50GB storage]: Compare the runtime and memory overheads of PUMM, FFmalloc, and MarkUs using the SPEC CPU 2006 benchmark.

Preparation: Download and compile SPEC CPU 2006.

Execution: Run the benchmark with no evaluated defenses enabled to establish a baseline. Then rerun the benchmark with PUMM, FFmalloc, and MarkUs enabled and calculate overheads.

Results: PUMM should have a lower average runtime and storage overhead than FFmalloc and MarkUs.

(E4): [1 human-day + 2 compute-hours + 2GB storage]: Synthetically generate use-after-free vulnerabilities using the monkey scripts provided in PUMM's code repository and verify whether PUMM prevents their exploitation.

Preparation: Compile the vulnerable programs described in E1.

Execution: For each vulnerable program, run the monkey script provided in the Scripts directory of the code repository, with and without PUMM's runtime monitor enabled.

Results: Without PUMM, a subset of the vulnerabilities generated with the monkey script should yield observable behaviors like segmentation faults. With PUMM, there should be fewer of these adverse behaviors.

A.5 Notes on Reusability

The scripts directory of the code repository contains scripts for using and evaluating PUMM. The scripts and their intended purposes are as follows:

build.sh : Build script to help compile PUMM. See the README in the code repository for full build steps.

dump-vdso.py : Helper script invoked by trace.sh to copy the system's vDSO object. Users should not need to call this script directly.

hook-debug.sh : Runs a target program with PUMM's protection activated. Uses a debug build of PUMM's runtime hooks with extra verbosity.

hook-monkey.sh : Runs a target program with PUMM's protection activated and additional code to synthetically inject use-after-free vulnerabilities for evaluation. Users should use monkey.sh instead of calling this script directly.

hook.sh : Runs a target program with PUMM's protection activated. Uses the optimized production build of PUMM's runtime hooks.

monkey-debug.sh : Runs a target program with PUMM's protection activated and synthetically injects a use-after-free vulnerability for evaluation. Whereas monkey.sh is the primary script for batch evaluation, this script allows the user to specify a specific seed to make it easier to investigate a particular trial. A typical workflow is to

use monkey.sh first and then use monkey-debug.sh to investigate interesting cases.

monkey.gdb : A helper script used by monkey.sh to automate GDB. Users should not need to call this script directly.

monkey.sh : Evaluation script for producing the results in Subsection 4.3 of the conference paper. Specifically, this script runs the target program several times with PUMM's protection activated, but also tries to randomly inject synthetic use-after-free vulnerabilities. The purpose of doing this is to evaluate PUMM at a larger scale than what is feasible using only real-world vulnerabilities.

procmap.sh : Helper script for extracting memory layout information from Perf recordings. Users should not need to use this script directly.

profile-name.sh : Helper script for determining the profile name for a given target program. This script is intended to help users locate a saved PUMM profile. For example, a user can provide this script with the name of a program that PUMM has already analyzed, and then they can look in PUMM's profile directory for the matching profile file.

ptdump.sh : A helper script to decode a Perf trace into a human-readable log of the Intel PT packets. Users should not need to call this script directly and is intended for debugging.

ptxed.sh : A helper script for decoding a Perf trace into the sequence of executed instructions. Users should not need to call this script directly.

trace.sh : The script for recording execution traces using Perf. See the README in the code repository for usage instructions and examples.

eval : Contains 3 additional scripts to help with evaluation. Specifically, these scripts accept a directory containing 1 or more decoded traces and calculate the number of executed instructions, total size of the traces, and estimate code coverage of the target program.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220912. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenix%20sec2023/.

USENIX'23 Artifact Appendix: Unique Identification of 50,000+ Virtual Reality Users from Head & Hand Motion Data

Vivek Nair
UC Berkeley

Wenbo Guo
UC Berkeley

Justus Mattern
RWTH Aachen

Rui Wang
UC Berkeley

James F. O'Brien
UC Berkeley

Louis Rosenberg
Unanimous AI

Dawn Song
UC Berkeley

A Artifact Appendix

Metaverse Research @ Berkeley RDI

Unique Identification of 50,000+ Virtual Reality Users from Head & Hand Motion Data
rdi.berkeley.edu/metaverse/identification

A.1 Abstract

We present source code and data that can be used to replicate our result of uniquely identifying over 55,000 users based on head and hand motion data in VR. Our source code includes Python scripts for training and testing a LightGBM-based identification model using our novel hierarchical approach. Our dataset includes both 4.7 TB of raw motion data from over 100,000 VR users, as well as about 20 GB of preprocessed features using our described featurization approach. Together, the scripts and data can be used to reproduce the main experiments, results, and figures presented in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact is non-destructive, but requires careful ethical consideration as it involves real user data from over 55,000 users. We have taken steps to anonymize these users before publishing our dataset. Evaluators should be careful to respect the privacy of these users by not attempting to deanonymize or contact these users, to infer sensitive attributes, or to otherwise violate our ethical data use guidelines available at https://rdi.berkeley.edu/metaverse/boxrr-23/dua.pdf.

A.2.2 How to access

The source code and processed features are available on Zenodo at https://zenodo.org/record/7935034.

The raw data from over 100,000 users can be found at https://rdi.berkeley.edu/metaverse/boxrr-23/. It cannot be uploaded to a standard repository due to its size (4.7 TB).

A.2.3 Hardware dependencies

The training and testing code for our machine learning models can be run on any machine or cluster with at least 96 GB of RAM per node. We performed our main evaluation using a distributed machine learning cluster of 10 nodes, each with 16 vCPU cores and 128 GB of RAM. LightGBM also supports CUDA acceleration, and performs slightly better with a GPU. On a single machine with a RTX 3090 GPU, Ryzen 9 5950X CPU, and 128 GB of RAM, the execution time was as follows:

- Training (Layer 1): 1d 7h 38m 43s
- Training (Layer 2): 1d 9h 16m 27s
- Clustering: 4d 16h 42m 49s
- Training (Layer 3): 1h 7m 26s
- Testing: 4d 16h 43m 37s

Finally, we recommend at least 128 GB of free disk space. Alternatively, the subset evaluation (E5) can be run with as little as 8 GB of RAM and 4 GB of disk.

A.2.4 Software dependencies

A wide variety of operating systems are supported; our main cluster used Ubuntu 20.04, while the single-note benchmarking machine used Windows 10. Our scripts were designed for Python 3.10.2 with the following required Python packages:

- PyTorch (torch) v1.13.1
- pandas v1.5.2
- tqdm v4.64.1
- scikit-learn (sklearn) v1.2.0
- NumPy (numpy) v1.24.0
- LightGBM (lightgbm) v3.3.3
- Joblib (joblib) v1.2.0
- NetworkX (networkx) v3.0
- Matplotlib (matplotlib) v3.6.2

A.2.5 Benchmarks

Our evaluation is based on 3.96 TB of raw motion-tracking data from over 2.65 million recordings of over 55,000 virtual reality users. The dataset has since grown in size to 4.71 TB and now includes over 100,000 users, and can be accessed at `https://rdi.berkeley.edu/metaverse/boxrr-23/`.

To help evaluators reproduce our results without downloading the entire 4.71 TB dataset, we have also provided the pre-processed engineered features according to the featurization technique described in our papers. These features are provided in our Zenodo archive as four separate PyTorch (.pt) data files:

- train.pt (14.4 GB): 55540 Users, 150 Events per User
- validate.pt (0.5 GB): 55540 Users, 5 Events per User
- cluster.pt (4.8 GB): 55540 Users, 50 Events per User
- test.pt (4.8 GB): 55540 Users, 50 Events per User

Additionally, we have provided our trained model files and pre-computed identification results at `https://boxrr-23.mfkdf.com/?identification-supplemental=1`. These files cannot be uploaded to Zenodo due to their size (90 GB).

A.3 Set-up

A.3.1 Installation

1. Download and install version 3.10.2 of Python from `https://www.python.org/downloads/`.
2. Use `pip install` to download the suggested version of each of the packages listed in §A.2.4 above.
3. Download the source code and data from Zenodo at `https://zenodo.org/record/7935034`.
4. Optionally, follow the instructions to set up GPU acceleration for LightGBM: `https://lightgbm.readthedocs.io/en/latest/GPU-Tutorial.html`.
   (a) If you prefer to use CPU, change `device_type='gpu'` to `device_type='cpu'` in all three training scripts (scripts 1, 2, and 5).
   (b) Change `n_jobs=16` to `n_jobs=N` where N is the number of physical CPU cores in your machine.
5. Optionally, download the supplemental data (trained models and inference results) at `https://boxrr-23.mfkdf.com/?identification-supplemental=1`.
6. Optionally, download the subset data (scripts and datasets) at `https://zenodo.org/record/8137817`.

A.3.2 Basic Test

After installing the required software, unzip the supplemental data into the same folder as the source code and data from Zenodo, then run this command: `py 7-stats_final.py`

If everything is correctly setup, the script should run without errors, and will calculate the identification accuracy of around 95%. The estimated run time is around 2 minutes.

A.4 Evaluation workflow

Note: A full replication of our main result (C1) requires over 12 days of computing time on a high-end workstation PC. While this is certainly an option (E1), we have also provided an alternative (E2) that allows evaluators to reproduce each step of our training and testing pipeline within a day without completely re-computing every intermediate result for all 55,540 users. Either way, evaluators will be able to efficiently conduct E3 and E4 to validate claims C2 and C3. A third option (E5) allows evaluators with limited computational resources to validate C1, C2, and C3 on a representative subset of the data, which can be extrapolated to the full 55,540 users.

A.4.1 Major Claims

(C1): Our system can uniquely identify over 50,000 VR users with over 90% accuracy from head and hand motion.

(C2): Over 50% of the information used to identify users comes from actual motion rather than static features.

(C3): Our system can correctly classify whether a given user has previously been seen or not with over 90% accuracy.

A.4.2 Experiments

In all experiments, these warnings can be safely ignored:

- `[Warning] min_data_in_leaf is set=20, min_child_samples=20 will be ignored.`
- `LineSearchWarning: The line search algorithm did not converge`
- `UserWarning: Line Search failed`

(E1): [Full Reproduction] [1 human-hour + 320 compute-hour + 128GB disk]: re-run our entire training and testing pipeline to validate our main identification result (C1).
Preparation: Complete steps 1–4 of the installation instructions given in §A.3.1. Do not complete step 5.
Execution: Run the first seven Python scripts in labeled numerical order, starting with `1-train_layer_1.py` and ending with `7-stats_final.py`. See §A.2.3 above for the estimated execution time of each step.
Results: After running `7-stats_final.py`, a results table will be displayed. The final identification accuracy, as shown in the last row, should be 90% or higher.

(E2): [Partial Reproduction] [1 human-hour + 8 compute-hour + 128GB disk]: re-run part of each step of our pipeline on a subset of users to validate our main identification result (C1); a reasonable alternative to (E1).

Preparation: Complete steps 1–5 of the installation instructions given in §A.3.1. Unzip the supplemental data and source code from Zenodo into the same folder.

Execution:

1. Run `py 1-train_layer_1.py` to train the first layer, but only train the first model (you can write `exit()` at the end of the main loop to do so).
   (a) This step should take around 3 hours.
   (b) After running, the final epoch should display a validation error (`valid_0's multi_error`) of 0.40 or less. If so, this step is verified, and it is safe to use our models for the rest of layer 1.
2. Run `py 2-train_layer_2.py` to train the second layer, but only train the first model (you can write `exit()` at the end of the main loop to do so).
   (a) This step should take around 3 hours.
   (b) After running, the final epoch should display a validation error (`valid_0's multi_error`) of 0.40 or less. If so, this step is verified, and it is safe to use our models for the rest of layer 2.
3. Run `py 3-test_and_cluster.py` to test the first two layers. When you see `Testing Accuracy...`, wait a few minutes for the accuracy to stabilize.
   (a) This step should take around 10 minutes.
   (b) The accuracy value should be around 85.0% or higher. If so, this step is verified, and the models in layers 1 and 2 are performing as expected.
4. Run `py 4-generate_groups.py` to cluster users into groups based on the test results.
   (a) This step should take around 5 minutes.
   (b) If you see `Created N groups of size [...]`, this step was successful and you can proceed.
5. Run `py 5-train_layer_3.py` to train all models in the third and final layer.
   (a) This step should take around 1 hour.
   (b) The final epoch of each model should display a validation error (`valid_0's multi_error`) of 0.50 or less on average. If so, this step is verified, and layer 3 is performing as expected.
6. Run `py 6-test_layer_3.py` to test the accuracy of all models in the third layer.
   (a) This step should take around 5 minutes.
   (b) The accuracy of each model should be around 75.0% or higher on average. If so, this step is verified, and layer 3 is performing as expected.
7. Run `py 7-stats_final.py` to test final accuracy.
   (a) This step should take around 5 minutes.

Results: After running `7-stats_final.py`, a results table will be displayed. The final identification accuracy, as shown in the last row, should be 90% or higher.

(E3): [Feature Importance] [5 human-minutes + 5 compute-minutes + 128GB disk]: run our model explainability script to validate our motion feature importance (C2).

Preparation: Complete steps 1–5 of the installation instructions given in §A.3.1. Unzip the supplemental data and source code from Zenodo into the same folder.

Execution: Run `py 8-explain.py`. This script should take at most 3 to 5 minutes to execute.

Results: After running `8-explain.py`, a graph of the results is stored in `stats/features.pdf`. This should be similar to Figure 16 in our paper, and motion features (blue) should constitute most of the *area* of the graph.

(E4): [Open World] [5 human-minutes + 3 compute-minutes + 128GB disk]: run our secondary evaluation to validate our open-world performance claims (C3).

Preparation: Complete steps 1–5 of the installation instructions given in §A.3.1. Unzip the supplemental data and source code from Zenodo into the same folder.

Execution: Run `py 9-open_world.py`. This script should take at most 1 to 3 minutes to execute.

Results: After running `9-open_world.py`, the value of `Overall Accuracy` shown should be at least 0.90.

(E5): [Subset] [5 human-minutes + 1 compute-hour + 4GB disk]: evaluate claims C1–C3 on a subset of users.

Preparation: Complete steps 1, 2, and 6 of the installation instructions given in §A.3.1; unzip subset scripts.

Execution: Run `C1.py`, `C2.py`, and `C3.py` in sequence. These scripts should take about 30 minutes to execute.

Results: For C1, the projected identification accuracy should be at least 90%. For C2, the motion features should account for more than 50% of the entropy gained. For C3, the overall accuracy should be at least 90%.

A.5 Notes on Reusability

We encourage researchers to build upon and improve our work, and to this end have released our entire source code under an MIT license. We further created the "BOXRR-23" dataset, the largest known motion capture dataset, to help researchers try new featurization techniques and model architectures beyond those included in our code. We encourage researchers interested in any topic involving human motion data to consider using the BOXRR-23 dataset, found here:

https://rdi.berkeley.edu/metaverse/boxrr-23/

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Erebus: Access Control for Augmented Reality Systems

Yoonsang Kim* and Sanket Goutam*, Amir Rahmati, Arie Kaufman
Stony Brook University
{yoonsakim, sgoutam, amir, ari}@cs.stonybrook.edu

A Artifact Appendix

A.1 Abstract

The core components of Erebus can be separated into 3 separate modules: (1) Policy Engine that generates Erebus policies from Natural Language inputs. (2) Language Transpiler that converts the intermediate Erebus language into the target platform code (in this implementation C#). (3) Native library implementation (in C#) that is used to build Android APKs for testing.

For reproducibility, we separate our implementation into these separate components and provide instructions on how to reproduce our results. In this Artifact, we provide instructions to reproduce the policy engine and policy generation of Erebus (components 1 and 2). We also open-source our implementation of the Erebus framework itself, developed using Unity and C#. However due to the complexity of setup required, and the number of platform-specific dependencies required to recompile the Android APK files, we skip these components for reproducibility and provide all necessary information to be used as reference.

A.2 Description & Requirements

The two components of the Policy Engine framework, included in this artifact, can be evaluated with the following system requirements.

Run-time requirements Ubuntu 20.04, Python 3.8.10, Dotnet 6.0

Hardware Minimum requirements: 2GB RAM

Expected output Policy code generated by each module into their respective text files. Module-specific instructions are provided in the README under each module.

A.2.1 Security, privacy, and ethical concerns

None

*These authors contributed equally to this work.

A.2.2 How to access

Stable Github Commit: https://github.com/Ethos-lab/erebus-AR_access_control/tree/artifact-final-release-v2

Github Repo: https://github.com/Ethos-lab/erebus-AR_access_control

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

All necessary software dependencies and install directions are provided in the README file.

A.2.5 Benchmarks

Any necessary data sets used for the experiements are included with the Repo. For the natural language policy generator module, we tweaked a publicly available NLP model using a custom training data set. The custom data and the final trained model are included in the repo. For evaluation, there is no need to re-train the model.

A.3 Set-up

Detailed instructions are provided in the README.

A.3.1 Installation

Detailed instructions are provided in the README.

A.3.2 Basic Test

The steps to test each module is provided in the README file under each subsection for the specific module.

A.4 Evaluation workflow

For functional and reproducible evaluation, the README file contains all the steps to evaluate the Policy Engine and Policy Transpiler components of Erebus. In our framework, these two components are sequentially chained together and compiled into an Android APK. In this artifact, we provide the steps to evaluate each of these components individually as recreating the Android APK would require substantial setup effort.

The Evaluation workflow can be summarized as below:

1. Download the Github Repo and make sure all the folders are available as per the documentation provided in README.

2. Verify each component individually based on the instructions provided in README for each component.

3. The folders for *erebus*, *prototype_apps*, and *survey* contain code and survey data used in our paper. But for the purposes of this artifact, they do not need to be evaluated for reproducibility (due to limitations of environment setup).

4. The folders for *policy_gen* and *policy_transpiler* contain the main contribution of our paper, which is the access control framework. The README file contains all the detailed instructions to verify these modules.

A.4.1 Major Claims

The main contribution of our paper is the design of an access control framework for Augmented Reality systems. This framework is designed based on a survey of existing systems, and implemented for an Android system. The main claims of our system (mainly the policy framework) include the following:

(C1): We propose a novel access control framework using a policy language design described in Section 5 and Table 3.

(C2): We also propose a mechanism to derive these policies using natural language input from developer's app descriptions, as shown in Figure 4 and Listing 3.

A.4.2 Experiments

Please refer to the README file for exact steps to test the policy framework of Erebus. Detailed steps are provided for each component, and sample policies that can be tested are also provided in the README file.

(E1): *Setup [30 human-minutes]:* Set up the software packages and install all the dependencies.

(E2): *Policy Engine [15 human-minutes]:* Follow the instructions in README for Reproducing the Policy Engine module. Test with additional sample policies provided, if needed.

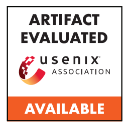
(E3): *Policy Transpiler [15 human-minutes]:* Follow the instructions in README for Reproducing the Policy Transpiler module. Ensure that the generate target code matches the policy statement defined in (E2).

A.5 Notes on Reusability

Apart from the specific modules used for reproducibility, we also release the overall implementation of our framework in C# (refer *erebus* folder) that ties together each of these modules. We advise readers to use this implementation as a reference, along with the *prototype_apps* code released with this artifact.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix
POLIGRAPH: Automated Privacy Policy Analysis using Knowledge Graphs

Hao Cui, Rahmadi Trimananda, Athina Markopoulou, Scott Jordan

University of California, Irvine

A Artifact Appendix

A.1 Abstract

In our main paper, we proposed POLIGRAPH, a type of knowledge graph, for the analysis of data collection statements in a privacy policy. We developed POLIGRAPH-ER, an NLP system to generate POLIGRAPH from the text of a given privacy policy. This appendix provides instructions on how to access our artifacts, how to use POLIGRAPH-ER, and how to reproduce main results in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Web scraping is restricted by certain websites. Inappropriate use of crawlers (*e.g.*, without rate limit) can lead to the banning of your IP address. While building the benchmark dataset, we limited our crawling to publicly accessible privacy policy webpages and enforced an appropriate rate limit.

To our knowledge, no other artifacts associated with the paper pose security, privacy and ethical risks to evaluators.

A.2.2 How to access

Source Code. The Git repository is at https://github.com/UCI-Networking-Group/PoliGraph.git. The version for this artifact evaluation is tagged as `USENIX-AE-v1`.

Dataset. Please see Section A.2.5.

Project Page. For up-to-date information, please check our project page: https://athinagroup.eng.uci.edu/projects/auditing-and-policy-analysis/.

A.2.3 Hardware dependencies

We recommend using an x86-64 Linux machine with a minimum of 32 GiB of memory, 20 GiB free disk space (after setting up conda) and an NVIDIA GPU with 24 GiB of video memory for this artifact evaluation. A GPU is needed to run transformer-based NLP models at a tolerable performance.

For *artifact reviewers*, we will offer SSH access to our GPU server, which serves as the test machine. We will post the login details on HotCRP.

A.2.4 Software dependencies

POLIGRAPH-ER is written in Python and requires several Python libraries. We recommend using conda on Linux to manage dependencies. All the code required for the artifact evaluation has been tested on a Debian 11 GNU/Linux system with Miniconda3 version 23.3.1.

For the full list of software dependencies, please refer to `environment.yml` in our Git repository.

A.2.5 Benchmarks

POLIGRAPH-ER Extra Data. We provide an archive file `poligrapher-extra-data.tar.gz`[1], which contains the NER model, purpose classification model and global entity ontology required by POLIGRAPH-ER. These resources can be generated using the scripts released in our Git repository. We have released them for the ease of reproducibility.

Benchmark dataset. We used privacy policies from the PoliCheck project as the benchmark dataset. The privacy policies, although publicly available, may be copyright protected. Users must sign a consent form before accessing it. Please follow the instructions at our project page to obtain the dataset.

For *artifact reviewers*, we provide the dataset directly on our server (`~/dataset`). Please refer to `README.md` in the directory for details about the content.

A.3 Set-up

We provide step-by-step instructions at `docs/usenix-artifact-evaluation.md` in the Git repository[2].

[1] `poligrapher-extra-data.tar.gz`: https://drive.google.com/file/d/1qHifRx93EfTkg2x1e2W_lgQAgk7HcXhP/view?usp=sharing

[2] Document - Artifact Evaluation: https://github.com/UCI-Networking-Group/PoliGraph/blob/USENIX-AE-v1/docs/usenix-artifact-evaluation.md

A.3.1 Installation

For *artifact reviewers*, we have set up the software environment. Please skip this step and continue to Section A.3.2. The instructions below are provided for completeness.

1. Git clone the POLIGRAPH repository (see Section A.2.2 for the URL) and change to the cloned directory:
```
$ git clone <GITHUB_URL> -b USENIX-AE-v1
$ cd poligraph/
```
2. Download the extra-data tarball (see Section A.2.5) and extract its contents to `poligrapher/extra-data`:
```
$ tar xf /path/to/poligrapher-extra-data.tar.gz \
    -C poligrapher/extra-data
```
3. Create a conda environment named `poligraph` with all the dependencies installed, and activate it:
```
$ conda env create -n poligraph -f environment.yml
$ conda activate poligraph
```
4. Initialize the Playwright library required by the crawler:
```
$ playwright install
```
5. Install POLIGRAPH-ER as a Python module:
```
$ pip install --editable .
```

A.3.2 Basic Test

Here we illustrate how to use POLIGRAPH-ER to generate a POLIGRAPH from the text of a real privacy policy[3].

1. Run the HTML crawler to download the privacy policy webpage (see Footnote 3 for the full URL):
```
$ python -m poligrapher.scripts.html_crawler \
    <PRIVACY_POLICY_URL> example/
```
2. Run the NLP pipeline on the privacy policy:
```
$ python -m poligrapher.scripts.init_document example/
```
3. Run the annotators to discover relations:
```
$ python -m poligrapher.scripts.run_annotators example/
```
4. Build the POLIGRAPH:
```
$ python -m poligrapher.scripts.build_graph \
    --pretty example/
```
   The argument `--pretty` is only needed if you would like to generate the graph in GraphML format.

To view the generated POLIGRAPH, you may use a text editor to open `example/graph-original.yml`. The YAML file describes the graph in a human-readable format. For example, the object below is an edge *we* $\xrightarrow{\text{COLLECT}}$ *statistical user datum* with a purpose "services" as the attribute.

```
- source: we
  target: statistical user datum
  key: COLLECT
  text: ...
  purposes:
    services: ...
```

Please see `docs/view-poligraph.md` in the Git repository for more instructions on how to view a POLIGRAPH.

[3] The privacy policy of "Puzzle 100 Doors": https://web.archive.org/web/20230330161225id_/https://proteygames.github.io/

A.4 Evaluation workflow

A.4.1 Major Claims

(C0): POLIGRAPH *Generation.* We show that POLIGRAPH-ER is used to generate POLIGRAPHs for 6,084 privacy policies in our benchmark dataset.

(C1): *Comparison to Prior Policy Analyzers.* In Section 4.2 of the main paper, we compared the collection relations inferred from POLIGRAPHs to data collection tuples extracted by PolicyLint. We claimed that our approach identifies 40% more collection relations (tuples) than the prior work, with 97% precision.

(C2): *Policies Summarization:* In Section 5.1 of the main paper, we use POLIGRAPH to summarize a large corpus of privacy policies and reveal common patterns among them. We will reproduce main results reported in Figures 8a, 8b and 8c, and Findings 1 and 2.

(C3): *Correct Definitions of Terms:* In Section 5.2 of the main paper, we use POLIGRAPH to access the correctness of definitions of terms in privacy policies. We show examples of different definitions in Table 5, and non-standard terms in Table 6. We will reproduce the results.

A.4.2 Experiments

For easy copy and paste of commands, we recommend following the link provided in Footnote 2. It also provides additional details about the steps and results.

Estimated Time: We expect that the human time required for each experiment is less than 10 minutes. The compute time for E0 is about 4 hours. The compute time for other experiments is negligible (a few minutes).

(E0): *PoliGraph Generation.* This generates POLIGRAPHs for all privacy policies in our benchmark dataset. E1-E3 will be based on these generated POLIGRAPHs.

Preparation: Download and extract the benchmark dataset (see Section A.2.5). For *artifact reviewers*, we have already extracted the dataset to `~/dataset`.

Execution: Do Steps 2-4 in Section A.3.2 on all privacy policies in the benchmark dataset as follows:
```
$ cd ~/dataset
$ python -m poligrapher.scripts.init_document dedup/*
$ python -m poligrapher.scripts.run_annotators dedup/*
$ python -m poligrapher.scripts.build_graph dedup/*
```
On the test machine, `init_document` takes about 2.5 hours, `run_annotators` takes 40 minutes, and `build_graph` takes 15 minutes to complete.

Results: After this experiment, each subdirectory under `~/dataset/dedup`, which corresponds to a privacy policy, should have a generated POLIGRAPH in it:
```
$ ls dedup/*/graph-original.yml | wc -l
6084
```
You may optionally check the generated POLIGRAPHs (`graph-original.yml`) as we explain in Section A.3.2.

(E1): *Comparison to Prior Policy Analyzers.*

Preparation: This experiment requires manually labeled ground truth data and output from PolicyLint. For *artifact reviewers*, we provided:

- Ground truth data at `~/dataset/external/manual-collection-relations.csv`
- PolicyLint's `ext/` directory, which contains all the input and output data of PolicyLint, at `~/dataset/external/policylint-ext`.

Please finish E0 before this experiment.

Execution:

1. The scripts needed for this experiment are in the `evals/tuples/` directory in the Git repository. Copy it to the dataset directory for convenience:

```
$ cd ~/dataset
$ cp -Tr ~/poligraph/evals/tuples ./eval-tuples
```

2. Convert collection relations inferred through POLIGRAPHs into tuples in a CSV file:

```
$ python eval-tuples/export_poligraph_tuples.py \
    -o eval-tuples/result-poligraph.csv s_test/*
```

The contents in `s_test/` link to a subset of 200 privacy policies, which we use as the test set in the main paper.

3. Convert PolicyLint tuples that belong to the same set of privacy policies into a CSV file:

```
$ python eval-tuples/export_policylint_tuples.py \
    -e external/policylint-ext \
    -o eval-tuples/result-policylint.csv s_test/*
```

4. Compare results from both sides to ground truth:

```
$ python eval-tuples/evaluate.py \
    external/manual-collection-relations.csv \
    eval-tuples/result-poligraph.csv
$ python eval-tuples/evaluate.py \
    external/manual-collection-relations.csv \
    eval-tuples/result-policylint.csv
```

Results: The output in Step 4 corresponds to the precisions and recalls reported in Table 4 of our main paper. You may open `eval-tuples/result-poligraph.csv` and `result-policylint.csv` to view tuples inferred through PoliGraph and those found by PolicyLint.

(E2): *Policies Summarization.*

Preparation: Please finish E0 before this experiment.

Execution:

1. The scripts needed for this experiment are in the `analyses/summarization` directory in the Git repository. Copy it to the dataset directory for convenience:

```
$ cd ~/dataset
$ cp -Tr ~/poligraph/analyses/summarization ./summarization
```

2. Generate statistics over the entire dataset:

```
$ python summarization/collect-and-purpose-statistics.py \
    -o summarization/ dedup/*
```

3. Generate figures of policies summarization results:

```
$ python summarization/plot.py \
    summarization/ summarization/figure.pdf
```

Results: In Step 2, the script prints statistics that are reported in Section 5.1 of the main paper. For example:

```
# of policies that disclose the collection of known
  categories: 4093
```

It also produces CSV files in `summarization/` containing statistics of data collection, sharing and usage purposes. In Step 3, the output PDF file `summarization/figure.pdf` reproduces Figure 8 in the main paper.

(E3): *Correct Definitions of Terms.*

Preparation: Please finish E0 before this experiment.

Execution:

1. The scripts needed here are in the `analyses/term-definitions/` directory in the Git repository. Copy it to the dataset directory for convenience:

```
$ cd ~/dataset
$ cp -Tr ~/poligraph/analyses/term-definitions \
    term-definitions
```

2. Run the script to find term definitions that do not align with our global ontologies:

```
$ python term-definitions/check-misleading-definition.py \
    dedup/*
```

3. Run the script to aggregate non-standard terms found in privacy policies into a CSV file:

```
$ python term-definitions/check-self-defined-terms.py \
    -o term-definitions/non-standard-terms.csv dedup/*
```

Results: Step 2 creates a `misleading_definitions.csv` file in each privacy policy's subdirectory that contains all the different definitions. We use `xsv`, a command-line CSV parser, to obtain counts of different definitions in Table 5 in the main paper. For example:

```
$ xsv cat rows dedup/*/misleading_definitions.csv \
  | xsv search -s parent '^non-personal information$' \
  | xsv frequency -s child
```

The output matches results in Table 5 in the main paper:

```
field,value,count
child,ip address,126
child,geolocation,123
......
```

Step 3 generates a CSV file `non-standard-terms.csv`, which contains results in Table 6 in the main paper about non-standard terms. The rows are like:

```
type,term,def_count,use_count,possible_meanings
DATA,technical information,126,311,advertising id|age|...
```

This row indicates that the data type "technical information" is defined in 126 and used in 311 privacy policies, and possible specific data types are listed in the last column.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Calpric: Inclusive and Fine-grained Labeling of Privacy Policies with Crowdsourcing and Active Learning

Wenjun Qiu David Lie Lisa Austin
University of Toronto

A Artifact Appendix

A.1 Abstract

The Calpric project leverages active learning and crowdsourcing techniques to address the challenge of the cost of training accurate deep learning models on privacy policies. In this artifact appendix, we describe the two artifacts available to the public: the Calpric Privacy Policy Corpus (CPPS) and the source codes for Calpric's major components.

We provide the source codes for Calpric as a reference, including the category models and the action models, as well as the privacy policy-based embedding PriBERT. We do not include a full pipeline test for Calpric as the complete system involves additional manual setup and accessing costs, such as the AWS account used to actively crowdsource training labels.

A.2 Description & Requirements

The CPPS data set includes privacy policy segment labels covering 9 data categories (contact, device, location, health, financial, demographic, survey, social media and personally identifiable information) with 3 data actions (collect/use, share, and store). For clarity purposes, duplicated labels have been removed, resulting in a total of 12,585 labels.

A.2.1 Security, privacy, and ethical concerns

The use of human annotators was approved by our institutional review board (IRB). We do not include any personal identifiable information in the publicly accessible dataset.

A.2.2 How to access

The artifacts are accessible via the Calpric GitHub page: https://github.com/dlgroupuoft/Calpric.

A.2.3 Hardware dependencies

The CPPS dataset and PriBERT embedding do not require any specific hardware feature. The source codes support both CPU and GPU processing. Depending on the size of the active querying pool, the required memory size may vary.

A.2.4 Software dependencies

To use the CPPS dataset, no other software dependencies are needed except the Python standard library and the *csv* package.

For the source code example, the following Python packages are required along with the Python standard library: *re, langdetect, numpy, pandas, keras, modAL and tensorflow*.

A.2.5 Benchmarks

Data required by the artifacts: CPPS.

A.3 Set-up

A.3.1 Installation

No other installation is required other than the software dependencies described above.

A.3.2 Basic Test

Run functionality_checks.py. The expected output from the CPPS check should be:
number of health labels: {'health': 1796}

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Measuring and Characterizing Non-Compliance of Apple Privacy Labels

Yue Xiao[1], Zhengyi Li[1], Yue Qin[1], Xiaolong Bai[2], Jiale Guan[1], Xiaojing Liao[1], Luyi Xing[1]

[1]Indiana University Bloomington, [2]Orion Security Lab, Alibaba Group

A Artifact Appendix

A.1 Abstract

[Mandatory] The downloader is utilized to download the app along with its corresponding privacy label. The static analyzer is used to screen apps that make calls to iOS system APIs. The dynamic analysis pipeline is employed to verify whether an app's code behavior complies with its privacy label. To use the tool, you need to provide the binary code of the app in .ipa format, as well as its privacy label from the Apple store (if it is not present in our 366,697 app privacy label dataset). The tool will then identify and output any inconsistencies it detects. The dynamic analysis pipeline is composed of three stages:

- *End-to-end execution, which includes fully automated app UI execution, dynamic instrumentation, and network monitoring.*
- *Inferring data and purpose from the call trace and network traffic information.*
- *Conducting a compliance check to identify any inconsistencies*

A.2 Description & Requirements

[Mandatory] To utilize the tool, the system requirements include Mac OS and a rooted iOS device. We have tested the tool on Mac OS version 12.6.2, which is the minimum version we recommend. Additionally, the iOS device must be running version 12.2 or higher, although lower versions may also be compatible with the tool. We have only listed the minimum versions we have tested, but other versions may still be compatible.

A.2.1 Security, privacy, and ethical concerns

[Mandatory] Rooting an iOS device can create security, privacy, and ethical issues. It grants administrative access to the device's operating system, enabling customization and control over the device's functionality but bypassing built-in security features, which may expose the device to security threats. Rooting can also compromise privacy, granting unauthorized access to personal information and data, particularly when installing third-party software from untrusted sources. Additionally, rooting violates Apple's terms of service, which may lead to legal consequences and could undermine the efforts of developers and manufacturers to create secure devices.

A.2.2 How to access

[Mandatory] You can access the source code in github: `https://github.com/xiaoyue10131748/Lalaine/tree/LalaineStable`

A.2.3 Hardware dependencies

[Mandatory] The tool requires a rooted iOS device to run, which may present a hardware dependency issue.

A.2.4 Software dependencies

[Mandatory] We use Macaca, an open-source automation testing framework that supports different types of applications and provides automation drivers, environment support, peripheral tools, and integration solutions to handle challenges such as test automation and client-side performance. We also set up NoSmoke, a cross-platform UI crawler that scans view trees, performs OCR operations, and creates and runs UI test cases.

- install macaca `https://macacajs.github.io/guide/environment-setup.html#macaca-cli`
- install nosmoke `https://macacajs.github.io/NoSmoke/guide/`

We utilize Frida, a dynamic code instrumentation toolkit. We inject snippets of JavaScript into native apps on iOS. We built our hooking framework on top of the Frida API.

- install Frida's CLI tools on MacOS: `https://frida.re/docs/installation/`

- configure Frida on your rooted iOS device: https://frida.re/docs/ios/

We utilized Fiddler, which is a web debugging proxy tool that monitors, analyzes and modifies the traffic on iOS device.

- install Fiddler in your MacOS: https://docs.telerik.com/fiddler/configure-fiddler/tasks/configureformac
- configure your rooted iOS device: https://docs.telerik.com/fiddler/configure-fiddler/tasks/configureforios

A.2.5 Benchmarks

[Mandatory] The privacy label of apps we crawled from app store are needed to put it under data folder. Please download it from `https://drive.google.com/file/d/1k3FulkLvOhgLV_hU-hkxnuvnP4FF3tXz/view?usp=share_link`. If the app you want to test is not on the list, you can mannully add it to this file to allow further complaince check.

A.3 Set-up

[Mandatory] This section should include all the installation and configuration steps required to prepare the environment to be used for the evaluation of your artifact.

A.3.1 Installation

[Mandatory] Please download the source code `https://github.com/xiaoyue10131748/Lalaine.git` and follow the README to setup environment.

A.3.2 Basic Test

[Mandatory] Check that Macaca, Frida and Fiddler are successfully installed. First, run the command `nosmoke -u <device id>`, to see if nosmoke can launch an app and automatically execute UI events. Second, run the command `frida-ps -U`, to see if the frida can list all the apps installed on the iPhone. Third, launch Fiddler and open any app on the iPhone to see if the traffic generated by the app can be captured.

A.4 Evaluation workflow

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] This section should include all the operational steps and experiments which must be performed to evaluate if your your artifact is functional and to validate your paper's key results and claims. For that purpose, we ask you to use the two following subsections and cross-reference the items therein as explained next.

A.4.1 Major Claims

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Enumerate here the major claims (Cx) made in your paper. Follows an example:

(C1): *The tool is able to download the binary of app and its privacy label.*
(C2): *The tool is able to screen apps that call sensitive iOS system APIs.*
(C3): *The tool is able to gather call trace and network traffic by dynamically executing an app in rooted device.*
(C4): *The tool is able to analyze call trace and network traffic to extract (data, purpose) from code behavior.*
(C5): *The tool is able to perform complaince check.*

A.4.2 Experiments

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available]

(E1): *[download app binary and crawl privacy label] [1 human-minutes + 3-4 compute-minutes + 10GB disk]: In this step, the tool will crawl privacy label from apple store and download its binary ipa file.*
How to: Collect the info of app you want to test. And execute the following down-loader command.
Preparation: *Put the information of the app that you want to download in app_info.json*
Execution: (1) `python privacy_label_crawler.py -input_file ./app_info.json -result_dir ./label/ -driver_path ./chromedriver` (2) `python app_binary_downloader.py -input_file ./app_info.json -result_dir ./ipa/`
Results: *The app binary will be in the folder /ipa/ and the privay label will be in the folder /label/*

(E2): *[screen apps binary ipa file] [0.5 human-minutes + 0.5 compute-minutes + 10GB disk]: In this step, the tool will screen apps that make calls to iOS system APIs.*
How to: Execute the following static analyzer (SAF) command.
Preparation: *Put the binary of app (.ipa) you want to static scan under the folder /app*
Execution: (1) `python find_in_decrypted_ipas.py -f ./API_List.txt -i ./app/`
Results: *The results will be in the file find_in_decrypted_ret.txt*

(E3): *[call trace and traffic gathering] [1 human-minutes + 3-4 compute-minutes + 10GB disk]: In this step, the tool will gather call trace and network traffic by dynamically executing an app in the rooted device.*
How to: *This step will take 3-4 mins. It takes three steps: (1) launch the iPhone, network monitor, and Macaca server for the reviewers. (2) the reviewers run the command `python batch_ui_frida_test.py 0 . -i`*

`<device id>` (3) close the iPhone and macaca server, dump the traffic from the network monitor

Preparation: *Make sure the iPhone, network monitor and Macaca server are launched.*

Execution: `python batch_ui_frida_test.py 0 . -i <device id>`

Results: *The results about are under `result/0/` folder.*

(E4): *[(data, purpose) inference] [0 human-hour + 0.5 compute-minutes]: Analyze call trace and network traffic to extract (data, purpose) from code behavior.*

How to: *Execute the following command.*

Execution: `python analyze_log.py 0 .`

Results: *Analyzing result can be found in `./result/0/prediction_output/`*

(E5): *[Compliance check] [0 human-hour + 0.5 compute-minutes]: Perform compliance check to find any inconsistencies between (data,purpose) extracted from call trace and network traffic and privacy label in its privacy label.*

How to: *Execute the following command.*

Execution: `python compliance_check.py 0 .`

Results: *Analyzing result can be found in `./result/0/inconsistency_output/`*

Evading Provenance-Based ML Detectors with Adversarial System Actions

Kunal Mukherjee, Joshua Wiedemeier, Tianhao Wang, James Wei, Feng Chen, Muhyun Kim,
Murat Kantarcioglu, and Kangkook Jee

Department of Computer Science, The University of Texas at Dallas

A Artifact Appendix

A.1 Abstract

The artifact evaluation process is designed to validate the repeatability and usability of the results presented in the research paper "Evading Provenance-Based ML Detectors with Adversarial System Actions." The paper introduces PROVNINJA, a novel framework designed to discover adversarial samples, also known as gadgets, specifically tailored for path-based Intrusion Detection Systems (IDS) and Graph Neural Network-based IDS. The primary objective of PROVNINJA is to identify actions that can successfully evade state-of-the-art IDSs. The evaluation process comprises two main components: training and testing the IDS and generating adversarial examples to evade the IDSs. As a valuable resource, the authors provide a GitHub link that grants access to the source code, data, and scripts necessary for reproducing the results described in the paper.

By offering these artifacts, the researchers enable fellow researchers and practitioners to replicate and build upon their work in provenance-based ML detectors. The artifacts include comprehensive software, data, and scripts employed to generate the findings presented in the paper. The accessibility of the GitHub repository ensures transparency. It fosters collaboration among researchers, facilitating advancements in the domain of provenance-based ML detectors and contributing to the overall improvement of security systems.

A.2 Description & Requirements

PROVNINJA is a system for generating evasive variants of known attack chains by replacing rare system events with chains of common system events that achieve the same ends. To support the reproduction of our results, we have provided the code, sample data, models, and intermediate files required to produce evasive attacks from the evaluation. Although our artifacts make no particular assumptions about compute power, **25**GB of disk space and **16**GB of memory are required to store and run the models, data samples, and software dependencies.

A.2.1 Security, privacy, and ethical concerns

None. No destructive steps are taken, and no data is collected during the evaluation process. The sample data provided is from our local testbed environments, so real user data is not exposed.

A.2.2 How to access

Our code, sample data, and sample results can be accessed on GitHub at https://github.com/syssec-utd/provninja/releases/tag/USENIX_23. The sample data for the Supply Chain APT is available at https://drive.google.com/file/d/1Jz0ZuiZlUEZdAgqlnfmpN2_X0Cms6S18/view.

A.2.3 Hardware dependencies

Running our experiments requires **25**GB of storage and **16**GB of memory for the data, code, and models.

A.2.4 Software dependencies

Our code is written in Python and uses Miniconda for environment management. Miniconda can be installed from https://conda.io/projects/conda/en/latest/user-guide/install/index.html. Simply follow the installation directions for your machine architecture. Python 3.10 will be installed as part of the environment building process. Our experiments were run on Ubuntu 18. Our ShadeWatcher experiments use a Docker container; to install Docker, please follow the instructions at https://docs.docker.com/engine/install/.

A.2.5 Benchmarks

All the datasets and models required for use with this artifact are provided in the GitHub repository and Google Drive provided in A.2.4.

A.3 Set-up

These instructions assume that you have already installed Miniconda and Docker.

A.3.1 Installation

First, clone our code, data, and configuration files. Next, update, build, and activate the Conda environment.

```
$ git clone \
  https://github.com/syssec-utd/provninja \
  --branch USENIX_23
$ cd provninja
$ conda update conda
$ conda env update -n provng -f provng.yml
$ conda activate provng
```

Next, build the Docker container for the ShadeWatcher experiments.

```
$ cd intrusion-detection-system/shadewatcher
$ docker build . -t shadewatcher
```

A.3.2 Basic Test

Once the environment has been set up, you can verify that all the packages have been installed correctly by running `python test_installation.py`. This script will import all the relevant modules and will print "INSTALLATION VERIFIED" if the Conda environment has been installed correctly. Otherwise, it will print "FAIL", along with a brief exception message. If the Docker build process finishes without issue, the Docker environment is set up.

A.4 Evaluation workflow

We provide detailed instructions on how to reproduce supporting evidence for our major claims. In the README, we also provide the exact commands to run for each major component.

A.4.1 Major Claims

(C1): ProvNinja reduces detection rates of state-of-the-art provenance-based IDS by up to 59%. This is proven by experiments (E1), (E2) and (E3), as described in section 6.4 and presented in tables 2 and 3.

(C2): ProvNinja is able to use event frequency data to construct inconspicuous alternatives to rare events in an attack chain. This is proven by experiment (E4); the gadget chain generation process is discussed in section 4.6 and example gadget chains are presented in tables 1 and 9.

A.4.2 Experiments

(E1): *[Path-based IDS] [10 human-minutes + 1 compute-minute]: Run the trained path-based models on the original attacks and the corresponding Ninja variants.*

How to: In `intrusion-detection-system/path-based/`, run `python sigl.py` and `python provdetector.py`. Record the recall and F1 scores for the original Enterprise APT and the "Gadget" Enterprise APT. The expected results are provided in the README.md file in this directory.

Preparation: To set the working directory, `cd intrusion-detection-system/path-based`. No additional configuration beyond the Conda environment activation from A.3 is required.

Execution: Run `python sigl.py` and `python provdetector.py`.

Results: Record the recall and F1 scores for the original Enterprise APT and the "Gadget" Enterprise APT. The recall and F1 scores for the "Gadget" Enterprise APT should be significantly lower than those of the original Enterprise APT.

(E2): *[Graph-based IDS] [15 human-minutes + 5 compute-minute]: Validate the graph-based IDS on the original Supply Chain APT, then create ninja variants to evade detection.*

How to: In `intrusion-detection-system/graph-based/`, run the S-GAT and Prov-GAT drivers; observe that the weight averge recall and F1 scores are acceptably high (>=0.88). Then, run `python provninjaGraph.py` to generate adversarial examples and observe the decrease in recall and F1 score.

Preparation: To set the working directory, `cd intrusion-detection-system/graph-based`. Download the sample Supply Chain APT data from https://drive.google.com/file/d/1Jz0ZuiZlUEZdAgqlnfmpN2_X0Cms6S18/view and unzip it. To assist in this step, we have provided a shell script `download_sample_supply_chain_data.sh`, which will download and unzip the data (run `./download_sample_supply_chain_data.sh`).

Execution: Run `python gnnDriver.py gat -if 5 -hf 10 -lr 0.001 -e 20 -n 5 -bs 128 -bi -s` and `python gnnDriver.py gat -if 768 -hf 10 -lr 0.001 -e 20 -n 5 -bs 128 -bi` to validate the graph-based IDS. Then, run `python provninjaGraph.py` to create and evaluate the evasive attacks.

Results: From the classification reports, record the weighted average recall and F1 scores for the original Supply Chain APT and the evasive variants. The recall and F1 scores for the evasive variants should be significantly lower than those of the original Supply Chain APT.

(E3): *[ShadeWatcher] [**10** human-minutes + **20** compute-minutes]: Demonstrate that ProvNinja can evade the SOTA provenance-based security detector, ShadeWatcher.*

How to: In the provided Docker container, run `shadewatcher_train.py` and `shadewatcher_eval.py` for the benign and anomalous samples with and without gadgets to observe the gadgets' impact on detection rates.

Preparation: Execute the commands in order:

```
$ docker run -it --mount type=bind,\
source="pwd",target=/data \
-e DATASET_DIR=/data \
--name shadewatcher shadewatcher
$ cd /ShadeWatcher
$ ./prepare_shadewatcher.sh
```

Execution: Run the script to start the evalaution, `run_shadewatcher_experiments.sh`, which will train and evaluate the ShadeWatcher model on the provided benign and anomalous data with and without gadgets. Finally, run `python3.6 stat_eval.py tests` to summarize the results.

Results: Observe the true positive decreases by **35%**, demonstrating PROVNINJA's ability to evade ShadeWatcher.

(E4): *[Gadget Finding] [10 human-minutes + 1 compute-minute]: Using some sample frequency data, create high-regularity gadgets for `winord.exe` executes `notepad.exe`.*

How to: Run `gadget-finder/gadget-finder.py` to generate gadgets and measure their regularity scores. Observe that several usable (regularity > 0.003) gadgets are created for this event.

Preparation: `cd gadget-finder`. Once you are in the right working directory, no additional configuration beyond the Conda environment activation from A.3 is required.

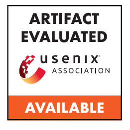
Execution: Run the command:

```
python gadget-finder.py -i input.csv -p
FrequencyDB/SAMPLE_WINDOWS_FREQUENCY_DB.csv
-o output/gadgets.txt
```

Results: Read the `output/gadgets.txt` file and see that five usable (regularity > 0.003) gadgets are provided.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: TreeSync: Authenticated Group Management for Messaging Layer Security

Théophile Wallez
Inria Paris

Jonathan Protzenko
Microsoft Research

Benjamin Beurdouche
Mozilla

Karthikeyan Bhargavan
Inria Paris

A Artifact Appendix

A.1 Abstract

The artifact contains an executable specification of MLS, and proofs for its TreeSync sub-protocol. It also contains code to run the official MLS test vectors.

A.2 Description & Requirements

Running this artifact requires a computer with either Nix or Docker installed. With Nix, the computer architecture must be x86_64.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

https://github.com/Inria-Prosecco/treesync/tree/7ea27ead0abc4e6bf47033f35a7eada233ac244e

A.2.3 Hardware dependencies

A computer with 8GB of RAM is enough to build and run the artifact.

A.2.4 Software dependencies

All the dependencies are managed by Nix or Docker, hence Nix or Docker are the only dependencies required to build the artifact.

A.2.5 Benchmarks

As mentioned in the paper (section 5, table 2), we do run some benchmarks. The benchmarks requires no additional data, and they are executed at the same time as the tests, when running the `make check` command.

A.3 Set-up

A.3.1 Installation

With Nix:

```
# This command will compile Z3, F* and
# other dependencies to the correct
# version, and start a shell with the
# correct environment.
nix develop
```

With Docker:

```
# Build the docker image.
# This will compile Z3 and F* to the
# correct version.
docker build . -t treesync_artifact
# Start the image and start a shell with
# the correct environment.
docker run -it treesync_artifact
```

A.3.2 Basic Test

In a shell with the correct environment:

```
cd mls-star
# This command will verify MLS*
make
# This command will run tests of MLS*
make check
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): MLS* can be used as a reference implementation of MLS. This is proven in the experiment (E1) that runs the official test vectors.

(C2): The security of TreeSync is formally proved. This is proven by experiment (E2) where we link theorem stated in the paper, and formal theorems in F*.

(C3): Doing formal proofs for TreeSync allowed us to propose improvements to the standard. This is proven in the experiment (E3) where we give links to pull-requests.

A.4.2 Experiments

(E1): [15 human-minutes + 1 compute-hour + 4 GB disk]: Run official MLS test vectors.
Preparation: Set up a correct environment, either with Nix or Docker.
Execution: Go in the `mls-star` repository and launch `make check`.
Results: The tests should succeed. For each test vector, it should print something similar to:
Secret Tree: running tests for 2/7 ciphersuites...
Secret Tree: success!

(E2): [30 human-minutes + 1 compute-hour + 4 GB disk]: Check that the formal theorems correspond to the prose.
Preparation: Set up a correct environment, either with Nix or Docker.
Execution: Go in the `mls-star` repository and launch `make`: this will verify with F* all of our theorems. Every theorem mentioned in the paper is listed in the README file, organized by section, with a link to the corresponding source code. Check that they correspond to the prose.
Results: the theorems mentioned in the paper and the formal theorems should correspond.

(E3): [30 human-minutes + 0 compute-minutes]: Look at our MLS pull-requests.
Preparation: Start your favorite web-browser.
Execution: Compare section 6 of our paper, and the following pull-requests:

- (disambiguate signatures) https://github.com/mlswg/mls-protocol/pull/526
- (use tree-hash in parent-hash) https://github.com/mlswg/mls-protocol/pull/527
- (strengthen the parent-hash link) https://github.com/mlswg/mls-protocol/pull/713

Results: the pull-requests and our claims of standard improvement should match.

A.5 Notes on Reusability

This artifact can be used for various purposes.

As a reference implementation, it can serve as a companion to the standard to help understanding it, and understand the security guarantees given by TreeSync.

As a proved specification, it can serve to test changes to the protocol, by modifying the F* specification and update the TreeSync Authentication Theorem (section 4.5).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations

Cas Cremers
CISPA Helmholtz Center for Information Security

Charlie Jacomme
Inria Paris, France

Aurora Naska
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This artifact includes formal models and proofs of the Signal protocol with an abstraction of its session-handling layer Sesame and ratcheting mechanism of the Double Ratchet. We prove that Signal with a session-handling layer does not achieve the Post-Compromise Security (PCS) guarantee, *i.e, the conversation is secure after a healing phase following the compromise*, although it holds in a single-session Signal. Following this, we propose and model a mechanism on how to restore PCS, and in a second step how to detect clone attackers in the conversation.

We provide four models of Signal: a) single-session Signal from the literature, where the PCS guarantee holds, b) Signal with its session-handling layer Sesame, where an attacker breaks PCS, c) Signal with our PCS-fix, with the restored PCS guarantee, and d) Signal with a clone detection mechanism, that soundly detects the clone's activity, *i.e, detection without any false positives*.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our models and proofs are accessible for inspection and reproduction at https://github.com/sesame-symbolic-model/sesame-model. To clone the repository, the user should run:

```
$ git clone --branch "sesame-model-v1"
https://github.com/sesame-symbolic-model/
sesame-model.git
```

A.2.3 Hardware dependencies

We have run our experiments on a Intel(R) Xeon(R) CPU E5-4650L 2.60GHz server with 756GB of RAM, and 4 threads per Tamarin call. The time of the experiments' execution might vary depending on your machine's CPU and RAM.

A.2.4 Software dependencies

To evaluate our models, you need the Tamarin prover[1] tool version v1.7.1. As shown in Appendix A.3.1, either you can download a docker image with the preinstalled Tamarin prover version, or compile it from scratch using the provided source files of the used version.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The following instructions have been tested on a Linux machine, and there may be slight variations on how to install and start the docker client on other systems. There are two ways to install the Tamarin prover and reproduce our results:

- **Using Docker** We provide via dockerhub an anonymous docker with the required preinstalled version of Tamarin.
  1. The user should install the Docker Engine on their machine as instructed in https://docs.docker.com/engine/install.
  2. Fetch the provided Tamarin image via:
     ```
     $ docker pull sesameproof/tamarin
     ```
  3. From the cloned repository, or one that has the Sesame folder inside of it, run:
     ```
     $ docker run -it -v "$PWD:/opt/case-studies"
     sesameproof/tamarin bash
     ```

[1] https://tamarin-prover.github.io

This should give you a shell, where the commands *tamarin-prover* of the experiments can be executed.

- **Compiling Tamarin from source** We provide in the repository the folder `tamarin-prover.zip` containing the source files for the correct version of Tamarin, with installation instructions at https://tamarin-prover.github.io/manual/book/002_installation.html.

A.3.2 Basic Test

To test that the Tamarin Prover is installed correctly, run:

```
$ tamarin-prover test
```

You should see the following message in the terminal:

```
*** TEST SUMMARY ***
All tests successful.
The tamarin-prover should work as  intended.
:-) happy proving (-:
```

A.4 Evaluation workflow

A.4.1 Major Claims

In the following we list the main properties of our formal analysis:

(PCS in single-session): Single-session Signal (*Double Ratchet* model) achieves the PCS guarantee in a conversation between two users, i.e., the conversation is secure after the healing phase and the clone attacker cannot inject or decrypt new messages. In this case, session-based and conversation-based PCS collapse and are both proven by experiment **(E1)** and described in Section 5.3.

(PCS violation in Sesame): Signal with the session-handling layer Sesame (*Sesame* model) does not achieve conversation-based PCS. We show an attack where a clone can impersonate the victim by using the compromised session, even though the honest parties heal after the compromise. The violation is shown in experiment **(E2)** and described in Section 5.4.

(Restored PCS): Signal with session-handling and proposed PCS-fix from Section 4.1 (*Sesame with sequential sessions* model) restores conversation-based PCS and locks the attacker out of the conversation. The property is proven in experiment **(E3)** and described in Section 5.5.

(Sound Clone Detection): In Sesame with sequential sessions and the proposed clone detection mechanism in Section 4.1 (*Sesame with sequential sessions + warning message* model), the parties can soundly detect a clone attacker that impersonates the victim by initiating new sessions with the compromise key. The property is shown in experiment **(E4)** and described in Section 5.5.

A summary of the results is reported in Table 2 of our paper.

A.4.2 Experiments

We now show the experiment steps needed to reproduce our results and prove the main claims we listed in the previous section.

Preparation Before running our experiments, the user should have followed one of the installation steps from Appendix A.3.1. In case of **using Docker**, the user runs the command from the second step to get a shell for the execution of `tamarin-prover` commands. In the second case of **compiling from source**, the user opens a terminal inside the `Sesame` folder in the cloned repository.

(E1): [5 human-minutes + ~1 compute-minute]
We prove the session-based and conversation-based PCS on the Signal model with a single-session restriction.
Execution: In the shell from the preparation step, the user executes:

```
$ tamarin-prover --prove Sesame/
DoubleRatchet.spthy
```

Results: The user should see as a result a list with all the proven properties of the model printed on the terminal. In particular, notice the `PCS` and `PCS_conversation` lemmas, which correspond to session-based and conversation-based PCS. In addition, the results include all other helper lemmas and sanity checks on the model. We present in the following a snippet of the expected output, while we provide the full results at the beginning of the `DoubleRatchet.spthy` file in the repository.

```
==================================================
summary of summaries:
analyzed: DoubleRatchet.spthy

...
PCS (all-traces): verified (9 steps)
StartChainPreceededByAssociate (all-traces):
    verified (119 steps)
SameTidKey (all-traces): verified (597 steps)
PCS_Conversation (all-traces): verified (61 steps)
==================================================
```

(E2): [5 human-minutes + 2.4 compute-seconds]
We show the attack on conversation-based PCS. An attacker can perform a step in the protocol using the compromised session state, even though the parties have healed after the compromise. This is possible since the healing happens in another session of the conversation.
Execution: In the shell, the user executes:

```
$ tamarin-prover Sesame/
Sesame_PCSAttack.spthy
```

Note, that the file `./Sesame_PCSAttack.spthy` contains a stored proof of the attack, therefore we verify

the proof without the `--prove` flag.

[Optional] [5 human-minutes + ∼1 compute-minute]

Using docker: To use Tamarin in the interactive mode, the user needs to run docker with an additional parameter that allows access to the IP address outside of docker:

```
$ docker run -p 3001:3001 -it -v "$PWD:/opt/
case-studies" sesameproof/tamarin bash
```

Then, run Tamarin with an extra argument to make the tool listen on any IP address:

```
$ tamarin-prover interactive
Sesame_PCSAttack.spthy  -i='*4'
```

Compiling Tamarin from source: The user needs to run the tool in interactive mode within the directory with the file as follows:

```
$ tamarin-prover interactive
Sesame_PCSAttack.spthy
```

In the end, in both cases the user opens the interface at `127.0.0.1:3001`, loads the theory `sesame` with origin source `./Sesame_PCSAttack.spthy` and clicks on the **`SOLVED`** green keyword of the attack trace.

Results: The user should see as a result a list with the proven `attack_pcs` lemma printed on the terminal. The expected output is the following:

```
====================================================
summary of summaries:
analyzed: Sesame_PCSAttack.spthy

...
attack_pcs (exists-trace): verified (80 steps)
====================================================
```

[Optional] In Figure 1, we show the attack trace on conversation-based PCS.

(E3): [5 human-minutes + ∼2 compute-minutes]

We prove the session-based and conversation-based PCS on the Signal with sequential sessions model.

Execution: In the shell from the preparation step, the user executes:

```
$ tamarin-prover --prove Sesame/
Sesame_Solution_RestoredPCS.spthy
```

Results: The user should see as a result a list of all the proven properties of the model printed on the terminal. The proven properties include the session-based `PCS` and `PCS_conversation` lemmas, as well as other helper and sanity lemmas. We present in the following a snippet of the expected output, while we provide the full results at the beginning of the `Sesame_Solution_RestoredPCS.spthy` file in the repository.

```
====================================================
summary of summaries:
analyzed: Sesame_Solution_RestoredPCS.spthy

...
PCS (all-traces): verified (9 steps)
SameRootKeyForTid (all-traces): verified (18 steps)
StartPrev (all-traces): verified (78 steps)
distinct_tid (all-traces): verified (53 steps)
SamePartner (all-traces): verified (44 steps)
PCS_Conversation (all-traces): verified (146 steps)
====================================================
```

(E4): [5 human-minutes + ∼1 compute-minute]

We prove the soundness of the clone-detection mechanism on Sesame with sequential sessions and warning message model. Informally, if an honest party detects a clone, there indeed was an attacker that cloned the honest device.

Execution: In the shell from the preparation step, the user executes:

```
$ tamarin-prover --prove Sesame/
Sesame_CloneDetection.spthy
```

Results: The user should see as a result a list of all the proven properties of the model printed on the terminal. In particular, notice the verification of the main guarantee `cd_soundness`. We present in the following a snippet of the expected output, while we provide the full results at the beginning of the `Sesame_CloneDetection.spthy` file in the repository.

```
====================================================
summary of summaries:
analyzed: Sesame_CloneDetection.spthy

...
current_origin (all-traces): verified (429 steps)
CompromiseBeforeStart (all-traces): verified (14
    steps)
cd_soundness (all-traces): verified (115 steps)
====================================================
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

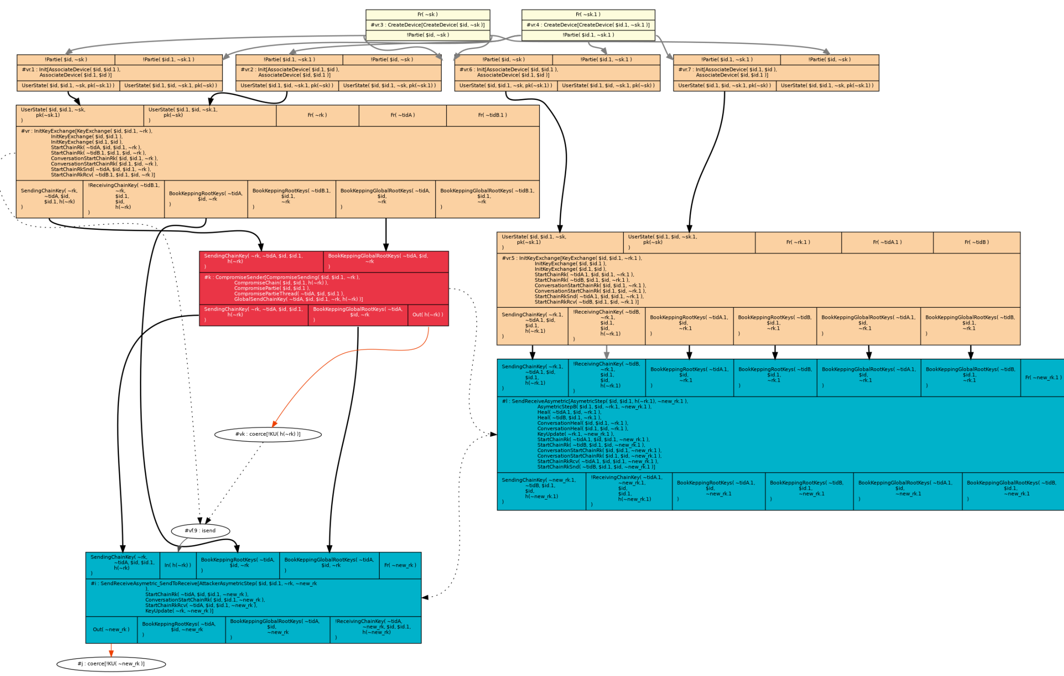


Figure 1: Tamarin output graph of the attack on conversation-based PCS. The attacker can forward the state of a compromised session after the parties have healed in another session.

USENIX'23 Artifact Appendix: MorFuzz: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation

Jinyan Xu
Zhejiang University
phantom@zju.edu.cn

Yiyuan Liu
Zhejiang University
yiyuanliu@zju.edu.cn

Sirui He
City University of Hong Kong
sol.he@my.cityu.edu.hk

Haoran Lin
Zhejiang University
haoran_lin@zju.edu.cn

Yajin Zhou
Zhejiang University
yajin_zhou@zju.edu.cn

Cong Wang
City University of Hong Kong
congwang@cityu.edu.hk

A Artifact Appendix

A.1 Abstract

As introduced in the paper, MorFuzz discovers several new bugs across open-source RISC-V processors with different microarchitectures and significantly improves the efficiency and effectiveness of processor fuzzing. Our artifact provides binaries and scripts to reproduce those results. This appendix describes the steps to set up our prototype and run our evaluation experiments.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is available at `https://github.com/sycuricon/MorFuzz/releases/tag/usenix23`. Both MorFuzz pre-built binaries and the inputs generated by DifuzzRTL that we replayed are available at `https://zenodo.org/record/8055696`.

A.2.3 Hardware dependencies

MorFuzz uses commercial EDA software, so an x86 processor is required. We evaluate MorFuzz on a 48-core dual Intel Xeon Silver 4214 server with 256GB RAM. In addition, at least 280 GB of storage is required. Storing the input generated by DifuzzRTL requires 270 GB, and the evaluation also consumes about 10 GB of storage.

A.2.4 Software dependencies

Our prototype contains three components: an instruction generator, a co-simulation library, and a top-level fuzzing framework. We release the instruction generator and the co-simulation library of MorFuzz as pre-built binaries, they are compiled with GCC 10.2.1 on CentOS 7.9.2009. In order to run MorFuzz the same operating system and compiler are required. MorFuzz uses the Synopsys VCS, a commercial RTL simulator, to simulate processor designs. You need to purchase licenses from Synopsys to use VCS. In addition, in order to cross-compile RISC-V programs, a RISC-V toolchain is also required, which is available at the official riscv-gnu-toolchain repository on the GitHub.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

First, MorFuzz requires the following dependencies, and in order to run the experiment you also need to set up dependencies from riscv-dv and riscv-torture:

```
sudo yum -y groupinstall "Development Tools"
sudo yum -y install redhat-lsb libXScrnSaver
    centos-release-scl dtc
sudo yum -y install devtoolset-10
```

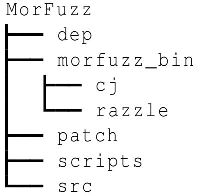
Second, clone the repository and execute the setup script.

```
git clone https://github.com/sycuricon/MorFuzz.git
cd MorFuzz
git checkout usenix23
git submodule update --init --recursive
export ARTIFACT_ROOT=$(pwd)
./scripts/setup.sh
```

Next, download the MorFuzz pre-built binaries `morfuzz_bin.zip` from `https://zenodo.org/record/8055696` and unzip it. You also need to place the decompressed `morfuzz_bin` directory under the root directory of the MorFuzz repository.

Finally, download the input sets `difuzzrtl_[0-4].zip` generated by DifuzzRTL from https://zenodo.org/record/8055696 and unzip them. You do not need to copy them into the repository, making the `DIFUZZRTL_INPUT` environment variable point to one of the input sets is enough.

The final directory structure of the project is as follows:



A.3.2 Basic Test

Before executing each experiment, you need to point the `ARTIFACT_ROOT` environment variable to the directory where the MorFuzz repository was cloned and execute the `env.sh` script to set up the other environment variables.

```
export ARTIFACT_ROOT=#absolute path to MorFuzz#
cd $ARTIFACT_ROOT
source ./scripts/env.sh
```

After executing the script if there are no complaints about missing dependencies, you can execute the basic test script. The script invokes Rocket, BOOM, CVA6, and Spike in turn to execute a normal test case, and if the test passes the "`*** PASSED ***`" message will appear on the terminal.

```
./scripts/basic.sh
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): MorFuzz is compatible with different microarchitectures and identified new bugs. This claim is supported by experiment (E1).

(C2): MorFuzz can efficiently achieve better coverage than DifuzzRTL and other techniques. This claim is supported by experiment (E2).

(C3): MorFuzz is capable of generating more diverse inputs than DifuzzRTL, and is comparable to riscv-dv. This claim is supported by experiment (E3).

(C4): Instruction morphing and state synchronization can help MorFuzz achieve better coverage. This claim is supported by experiment (E4).

A.4.2 Experiments

(E1): Executing test cases that trigger discovered bugs on the corresponding processors to prove that MorFuzz can be used on different microarchitectures, for details see `src/table2/README.md`.

Estimated time: less than 5 human-minute, and less than 5 compute-minute.

Preparation: Go to the MorFuzz repository root directory, set up `ARTIFACT_ROOT` variable and execute `scripts/env.sh`.

Execution: Execute `scripts/tab2.sh`.

Results: Trigger bugs in Table 2. For a detailed analysis of each result, please refer to `src/table2/README.md`.

(E2): Evaluating coverage to prove that MorFuzz achieves better coverage, for details see `src/figure8/README.md`.

Estimated time: less than 5 human-minute, and 24 compute-hour.

Preparation: Go to the MorFuzz repository root directory, set up `ARTIFACT_ROOT` and `DIFUZZRTL_INPUT` variables and execute `scripts/env.sh`.

Execution: Execute `scripts/fig8.sh`.

Results: Reproduce Figure 8, you can find the figure at `scripts/output/fig8.pdf`.

(E3): Evaluating instruction diversity to prove that MorFuzz generates instructions with good diversity, for details see `src/figure9/README.md`.

Estimated time: less than 5 human-minute, and 24 compute-hour.

Preparation: Go to the MorFuzz repository root directory, set up `ARTIFACT_ROOT` and `DIFUZZRTL_INPUT` variables, and execute `scripts/env.sh`.

Execution: Execute `scripts/fig9.sh`.

Results: Reproduce Figure 9, you can find three heatmaps named `heatmap_<name>.pdf` in the `scripts/output` directory.

(E4): Evaluating coverage to prove that MorFuzz's subcomponents can help MorFuzz achieve better coverage, for details see `src/figure10/README.md`.

Estimated time: less than 5 human-minute, and 24 compute-hour.

Preparation: Go to the MorFuzz repository root directory, set up `ARTIFACT_ROOT` variable and execute `scripts/env.sh`.

Execution: Execute `scripts/fig10.sh`.

Results: Reproduce Figure 10, you can find the figure at `scripts/output/fig10.pdf`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets

Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang,
Chunjie Cao, Yuqing Zhang, Flavio Toffalini, Mathias Payer

A Artifact Appendix

A.1 Abstract

The artifact of FISHFUZZ contains the source code, supportive materials along with the documents and scripts to reproduce the results. We release the artifact to help user reproduce the two-stage evaluation results in the paper without too much manual effort.

A.2 Description & Requirements

The FISHFUZZ artifact consists of the following components:

Source Code FISHFUZZ is a novel input prioritization strategy, to demonstrate the generality, we implement FISHFUZZ based on AFL and AFL++ respectively (FF_{AFL} and FF_{AFL++}). In this artifact, we release both of them and provide a wrapper to automate the compilation.

Supportive Materials Due to page limitation, we didn't include all raw data in the paper. In this artifact, we attach the (1) raw data for p-value calculation (2) evaluation results for different hyperparameters (3) full list of new bugs FISHFUZZ found.

Reproduction Scripts In this artifact we provide the scripts and the dockerfile that help users automatically build the benchmark, start evaluations and analysis the results. The benchmark included in the artifact is the two-stage benchmark in the paper.

A.2.1 How to access

- The artifact is available at https://github.com/HexHive/FishFuzz.
- The version we used for artifact evaluation is https://github.com/HexHive/FishFuzz/commit/911637cdf7448b97eccf1c9664ef318aff884b63.

A.2.2 Hardware dependencies

In the evaluation, we use a server equipped with Xeon Gold 5218(22M Cache, 2.30 GHz), 64 GB memory and 1TB disk space. To reproduce this evaluation, more than 50GB disk space is required.

A.2.3 Software dependencies

We plug this evaluation into the docker environment, therefore the user only need to have a Linux server with docker and git installed.

A.2.4 Benchmarks

The benchmark included in the artifact is two-stage benchmark in the paper, which consists of 7 real-world programs that have various types of input format. We provide a dockerfile that automated the build process.

A.3 Set-up

We provide a detailed README in the folder ‘paper/artifact‘. By following the instructions listed, the evaluation results could be easily reproduced.

Tips: we allocate one cpu core for each fuzzer-benchmark pair, and by default the evaluation contains 4 fuzzers * 7 benchmarks, which requires 28 cores to run all campaigns at the same time. Therefore the users could remove some programs or fuzzers according to their hardware bandwidth.

```
1
2  # for two-stage
3  export BENCHMARK_NAME=two-stage
4  export IMAGE_NAME=fishfuzz:ae-twostage
5
6  git clone git@github.com/Hexhive/FishFuzz && \
7  cd FishFuzz/paper/artifact/$BENCHMARK_NAME
8
9  # build base docker images
10 docker build -t $IMAGE_NAME .
11
12 # create scripts for fuzzers to run
13 python3 scripts/generate_script.py \
14        -b "$PWD/runtime/fuzz_script"
```

Listing 1: Build Evaluation Image

To prepare the artifact evaluation, user should first download the repo and build the docker image, as depicted in Listing 1. This step is expected to take about 1.5h. Afterward, run the generate_script.py to generate the scripts for fuzzers to run.

Afterward, we provide a script to generate the docker commands, considering many servers didn't have enough cores (>= 28 cores), we only print the command and the user could copy-paste to run. After 24h, the evaluations are done and we should follow the given instructions for post-processing (Listing 2)

```
1
2  # generate command & copy-paste
3  python3 scripts/generate_runtime.py \
4          -b "$PWD/runtime"
5
6  # wait 24h and stop all
7  docker rm -f $(docker ps -a -q
8      -f "ancestor=$IMAGE_NAME")
9
10 # give w/r permission
11 sudo chown -R $(id -u):$(id -g) runtime/out
12
13 # copy evaluation results to results folder
14 mkdir results/
15 python3 scripts/copy_results.py \
16        -s "$PWD/runtime" \
17        -d "$PWD/results/" -r 0
18
19
20 ...
```

Listing 2: Run the Evaluation

Finally, create a new container, copy the results dir into root dir and run the analysis scripts as follow (Listing 3)

```
1
2  # create container and mount results
3  cp -r scripts/ results/
4  docker run -it -v $PWD/results/:/results \
5  --name validate_twostage $IMAGE_NAME bash
6
7  # run analysis
8  python3 scripts/analysis.py -b /results \
9  -c scripts/asan.queue.json -r 0 -d /results/log/0
10 python3 scripts/analysis.py -b /results \
11 -c scripts/asan.crash.json -r 0 -d /results/log/0
12
13 # plot the results,
14 python3 scripts/print_result.py \
15         -b /results/log/ -r 0 -t all
16
17 ...
```

Listing 3: Evaluation Results Analysis

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): FF_{AFL} and FF_{AFL++} should achieve better coverage than their origin prototype AFL and AFL++. This is proven by experiment and the original results can be found in Table 5 in the paper.

(C2): FF_{AFL} and FF_{AFL++} should find more bugs than the direct competitors (AFL and AFL++). This is proven by the experiment and the original results are available in Table 7 and Figure 5-b.

A.4.2 Experiments

In the analysis step in subsection A.3, a coverage report along with bug reports are generated and can clearly demonstrate our claims C1 and C2.

In the paper we conduct 10 rounds of evaluation to reduce the possible variance, however, due to the time limitation, we suggest reducing the round to run, and 3 rounds might be sufficient. In that case the artifact evaluation can be finished in 3 days given a server equipped with 2 Xeon Gold 5218, 64GB memory and 1TB disk.

Besides, the bug report only consider the callstack and ASan bug type, which might still have lots of duplications, therefore in the paper we manually compare some stack traces to get the actual number of UNIQUE bugs. But the report itself is sufficient enough to support our claim C2.

A.5 Notes on Reusability

To make FISHFUZZ more usable, we develop an all-in-one wrapper that automated all the compilation steps for both FF_{AFL} and FF_{AFL++}. Users can refer to the manual in the README.

Besides, we're also working on the FISHFUZZ fuzzbench integration for better evaluating the capability of FISHFUZZ. Given that lots of fuzzbench targets didn't support LTO mode, we're working on implementing a non-LTO mode FISHFUZZ for the intergration and we're keeping pushing it forward. An experimental fuzzbench configuration file can be found in paper/fuzzbench folder.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems

Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai

A Artifact Appendix

A.1 Abstract

This artifact contains a functional version of PolyFuzz and the necessary dataset for the evaluation. To facilitate the usage of this artifact, we have prepared a Docker image with the necessary components to execute the artifact and visualize the result. Artifact users can compare the results obtained from executing this artifact with those presented in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns with using this artifact.

A.2.2 How to access

We provided three ways to access our artifact package:

1. DOI is provided through FigShare
https://doi.org/10.6084/m9.figshare.20022893.v1

2. An evolving version is maintained on GitHub
The GitHub repository is:
https://github.com/Daybreak2019/PolyFuzz.git
A specific tag is provided:
https://github.com/Daybreak2019/PolyFuzz/releases/tag/v6.0

3. A docker image is provided with PolyFuzz installed
docker pull daybreak2019/polyfuzz:v1.1

A.2.3 Hardware dependencies

The host machine may need at least 16GB memory and 256GB hard disk space.

A.2.4 Software dependencies

PolyFuzz is mainly developed and tested on LLVM 11.0, Soot 4.3.0, Python 3.8/9 (and Python3-dev), and OpenJDK 8/11. For ease of use of PolyFuzz, we have prepared a Docker image with all compilation and run-time dependencies installed.

Moreover, real-world benchmarks have their own particular/additional dependencies. Hence, to fully reproduce the results in the paper, users should install these dependencies successfully, which have been provided via relevant scripts in the PolyFuzz repository on GitHub.

More specifically, we note that the Docker image includes all the libraries/frameworks underlying PolyFuzz; thus it can be used for experimenting with other real-world subjects as well (i.e., saving the time/trouble for installing ubuntu, llvm, etc.) On the other hand, since our real-world subjects are sizable, including the complete compilation and run-time environment (e.g., all the third-party library dependencies) for all of them in the single Docker image would make it clumsy to deploy conveniently. Using a traditional virtual machine would aggregate this concern since they are even heavier. This is why we chose to include in the Docker image only the setup for the subjects in which any vulnerabilities were discovered by PolyFuzz at the paper submission time. Users can still use the scripts in the repository to set up the environments for other benchmarks; we have tested the scripts on our servers.

A.2.5 Benchmarks

For demonstration purposes, we use 5 multi-language benchmarks with vulnerabilities detected as concrete examples and have installed all of their dependencies in the Docker image. Specifically, the installed benchmarks include 4 Python-C benchmarks (i.e., Pillow, Libsmbios, Ultrajson, Bottleneck) and 1 Java-C benchmark (i.e., Jansi).

A.3 Set-up

A.3.1 Installation

- Step 1: Config AFL++

  git clone https://github.com/Daybreak2019/PolyFuzz.git

  sudo PolyFuzz/AFLplusplus/afl-system-config

- Step 2: Download the Docker image and run a Docker container based on the image

  docker pull daybreak2019/polyfuzz:v1.1

  docker run -it daybreak2019/polyfuzz:v1.1

- Step 3: Update PolyFuzz to the latest version and build the project within the container

  cd /root/PolyFuzz
  git pull
  . build.sh

A.3.2 Basic Test

To validate whether the environment is ready, a simple test can be run as follows:
cd /root/PolyFuzz/langspec/python/tests/case4
./build.sh
./sasg_entry.sh

The results should be similar as shown in Figure 1. Then "CTRL + C" can be used to quit the fuzzing.

```
            american fuzzy lop ++3.14a (default) [fast] {0}
+- process timing ------------------------------------+- overall results ----+
|        run time : 0 days, 0 hrs, 0 min, 8 sec       |  cycles done : 0     |
|   last new path : 0 days, 0 hrs, 0 min, 8 sec       |  total paths : 7     |
| last uniq crash : none seen yet                     | uniq crashes : 0     |
|  last uniq hang : none seen yet                     |   uniq hangs : 0     |
+- cycle progress ---------------------+- map coverage+----------------------+
|  now processing : 6*1 (85.7%)        |    map density : 0.00% / 61.90%     |
| paths timed out : 0 (0.00%)          | count coverage : 1.31 bits/tuple    |
+- stage progress ---------------------+- findings in depth -----------------+
|  now trying : splice 4               | favored paths : 1 (14.29%)          |
| stage execs : 3/24 (12.50%)          |  new edges on : 1 (14.29%)          |
| total execs : 1186                   | total crashes : 0 (0 unique)        |
|  exec speed : 153.6/sec              |  total tmouts : 1 (1 unique)        |
+- fuzzing strategy yields ------------+-------------+- path geometry -------+
|   bit flips : disabled (default, enable with -D)  |    levels : 2         |
|  byte flips : disabled (default, enable with -D)  |   pending : 2         |
| arithmetics : disabled (default, enable with -D)  |  pend fav : 1         |
|  known ints : disabled (default, enable with -D)  | own finds : 1         |
|  dictionary : n/a                                 |  imported : 0         |
|havoc/splice : 1/704, 0/424                        | stability : 100.00%   |
|py/custom/rq : unused, unused, unused, unused      +-----------------------+
|    trim/eff : 0.00%/6, disabled                   |          [cpu000:  8%]
+---------------------------------------------------+   [Cross-Language:0]
```

Figure 1: A basic test for PolyFuzz

A.4 Evaluation workflow

We provided scripts for automating all the experiments during the evaluation. Specifically in this artifact, we setup the environment for the experiments on 5 multi-language benchmarks, including 4 Python-C benchmarks (i.e., Pillow, Libsmbios, Ultrajson, Bottleneck) and 1 Java-C benchmark (i.e., Jansi). For each benchmark, the expected fuzzing time is 24 hours.

A.4.1 Major Claims

For the 5 multi-language benchmarks, PolyFuzz should be able to reproduce the vulnerabilities reported in Table 10 of the paper.

A.4.2 Experiments

(E1): [Experiment on Pillow] [30 human-minutes + 24 compute-hour + 128GB disk]: users should rebuild Pillow in the container, then start fuzzing for 24 hours.
How to: Rebuild Pillow and run PolyFuzz on it.
Preparation: None
Execution: Run commands as follows:
1. Build Pillow
cd /root/PolyFuzz/benchmarks/script/multi-benches/Pillow
./build.sh

2. Run fuzzing on Pillow
cd drivers/fig_process
./sasg_entry.sh

Results: PolyFuzz should report crashes/hangs in the end. The corresponding test cases would be generated under the directory fuzz/out/default/crashes and fuzz/out/default/hangs. As an example, Figure 2 shows the fuzzing results of Pillow, which is a snapshot of the fuzzing at 1 hour 41 mins. To validate whether these *hangs* are true positives, we can use the following commands to run the tests one by one:

- 1. entry the *fuzz* directory
  cd /root/PolyFuzz/benchmarks/script/multi-benches/Pillow/drivers/fig_process/fuzz
- 2. list the tests of *hangs* (results as shown in Figure 3.)
  ll out/default/hangs/
- 3. run a single test with the driver (results as shown in Figure 4.)
  python ../fig_process.py out/default/hangs/id:...

```
            american fuzzy lop ++3.14a (default) [fast] {0}ls : 2             |
+- process timing ------------------------------------+- overall results ----+
|        run time : 0 days, 1 hrs, 41 min, 15 sec     |  cycles done : 0     |
|   last new path : 0 days, 0 hrs, 2 min, 7 sec       |  total paths : 228   |
| last uniq crash : none seen yet                     | uniq crashes : 0     |
|  last uniq hang : 0 days, 0 hrs, 5 min, 50 sec      |   uniq hangs : 8     |
+- cycle progress ---------------------+- map coverage+----------------------+
|  now processing : 32.1 (14.0%)       |    map density : 0.80% / 6.09%      |
| paths timed out : 0 (0.00%)          | count coverage : 2.16 bits/tuple    |
+- stage progress ---------------------+- findings in depth -----------------+
|  now trying : splice 14              | favored paths : 21 (9.21%)          |
| total execs : 19.4k                  | total crashes : 0 (0 unique)        |
|  exec speed : 0.00/sec (zzzz...)     |  total tmouts : 11 (8 unique)       |
+- fuzzing strategy yields ------------+-------------+- path geometry -------+
|   bit flips : disabled (default, enable with -D)  |    levels : 2         |
|  byte flips : disabled (default, enable with -D)  |   pending : 220       |
| arithmetics : disabled (default, enable with -D)  |  pend fav : 15        |
|  known ints : disabled (default, enable with -D)  | own finds : 195       |
|  dictionary : n/a                                 |  imported : 0         |
|havoc/splice : 167/8214, 12/1440                   | stability : 100.00%   |
|py/custom/rq : unused, unused, unused, unused      +-----------------------+
|    trim/eff : 0.19%/8689, disabled                |          [cpu000:  7%]
+---------------------------------------------------+   [Cross-Language:0]
```

Figure 2: Example of fuzzing result on Pillow

```
Pillow/drivers/fig_process/fuzz# ll out/default/hangs/
total 680
drwx------ 2 root root  4096 Oct  9 21:08 ./
drwx------ 6 root root  4096 Oct  9 21:14 ../
-rw------- 1 root root 85523 Oct  9 20:58 id:000000,src:000020,time:5092698,op:havoc,rep:4
-rw------- 1 root root 85561 Oct  9 21:00 id:000001,src:000020,time:5229024,op:havoc,rep:16
-rw------- 1 root root 85479 Oct  9 21:05 id:000002,src:000020,time:5522112,op:havoc,rep:4
-rw------- 1 root root 85512 Oct  9 21:05 id:000003,src:000020,time:5548441,op:havoc,rep:4
-rw------- 1 root root 85571 Oct  9 21:06 id:000004,src:000020,time:5581795,op:havoc,rep:4
-rw------- 1 root root 85488 Oct  9 21:07 id:000005,src:000020,time:5629759,op:havoc,rep:8
-rw------- 1 root root 85560 Oct  9 21:07 id:000006,src:000020,time:5634992,op:havoc,rep:2
-rw------- 1 root root 85539 Oct  9 21:08 id:000007,src:000020,time:5725040,op:havoc,rep:2
```

Figure 3: *Hang* tests of fuzzing result on Pillow

(E2): [Experiment on Libsmbios] [30 human-minutes + 24 compute-hour + 128GB disk]: users should rebuild Libsmbios in the container, then start fuzzing for 24 hours.
How to: Rebuild Libsmbios and run PolyFuzz on it.

```
Pillow/drivers/fig_process/fuzz# python ../fig_process.py out/default/hangs/id\:000000\,src\:000020\,time\:5092698\,op\:havoc\,rep\:4
Image size (672976881 pixels) exceeds limit of 178956970 pixels, could be decompression bomb DOS attack.
Pillow/drivers/fig_process/fuzz#
```

Figure 4: A single test result of Pillow with driver fig_process.py

Preparation: None
Execution: Run commands as follows:

- Build Libsmbios
  cd /root/PolyFuzz/benchmarks/script/multi-benches/libsmbios
  ./build.sh
- Run fuzzing on Libsmbios
  cd drivers/op_mem
  ./sasg_entry.sh

Results: PolyFuzz should report crashes/hangs in the end.

(E3): [Experiment on Ultrajson] [30 human-minutes + 24 compute-hour + 128GB disk]: users should rebuild Ultrajson in the container, then start fuzzing for 24 hours.
How to: Rebuild Ultrajson and run PolyFuzz on it.
Preparation: None
Execution: Run commands as follows:

- Build Ultrajson
  cd /root/PolyFuzz/benchmarks/script/multi-benches/ultrajson
  ./build.sh
- Run fuzzing on Ultrajson
  cd drivers/encode
  ./sasg_entry.sh

Results: PolyFuzz should report crashes/hangs in the end.

(E4): [Experiment on Bottleneck] [30 human-minutes + 24 compute-hour + 128GB disk]: users should rebuild Bottleneck in the container, then start fuzzing for 24 hours.
How to: Rebuild Bottleneck and run PolyFuzz on it.
Preparation: None
Execution: Run commands as follows:

- Build Bottleneck
  cd /root/PolyFuzz/benchmarks/script/multi-benches/bottleneck
  ./build.sh
- Run fuzzing on Bottleneck
  cd drivers/random_shape2
  ./sasg_entry.sh

Results: PolyFuzz should report crashes/hangs in the end.

(E5): [Experiment on Jansi] [30 human-minutes + 24 compute-hour + 128GB disk]: users should rebuild Jansi in the container, then start fuzzing for 24 hours.
How to: Rebuild Jansi and run PolyFuzz on it.
Preparation: None
Execution: Run command as follows:

- Build Jansi
  cd /root/PolyFuzz/benchmarks/script/multi-benches/jansi
  ./build.sh
- Build fuzzing drivers of Jansi
  cd drivers
  ./build.sh
- Run fuzzing on Jansi
  cd OutStream
  ./sasg_entry.sh

Results: PolyFuzz should report crashes/hangs in the end.

A.5 Notes on Reusability

Considering the time cost of fuzzing experiments, in the artifact package, we only demonstrated 5 of the multi-language benchmarks used in our paper. However, for all of the benchmarks, we have provided similar scripts (under directory PolyFuzz/benchmarks/script) to the ones demonstrated in the appendix; users can follow the steps above to run PolyFuzz on other benchmarks.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Not All Data are Created Equal: Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks

Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N. Asokan, Danfeng (Daphne) Yao

A Artifact Appendix

A.1 Abstract

This artifact provides a comprehensive guide on installing and utilizing our proposed Data and Pointer Prioritization (DPP) framework. The DPP framework incorporates rule-based heuristics to automatically identify and prioritize/rank sensitive memory objects from an application. Within this artifact, we outline the necessary prerequisites, requirements, and software dependencies for DPP, along with detailed instructions on accessing, setting up, and installing the framework. Additionally, we delve into the usage of the DPP framework, specifically focusing on prioritizing sensitive data objects through a straightforward program.

A.2 Description & Requirements

The source code of DPP consists of a set of LLVM analysis passes and modifications to the AddressSanitizer's (ASan) instrumentation mechanism. To generate the data-flow graph, DPP utilizes the SVF tool (available at `https://github.com/SVF-tools/SVF`). We have made changes to LLVM's build script (CMakeFiles.txt) to include SVF's source as an in-tree build (as a library) during the compilation of the LLVM source code. For ease of use, we provide the build script (build.sh). Additionally, we have also modified SVF's build script and included a customized version of SVF in our repository. It's important to note that compiling and building the LLVM source code requires CMake and the Ninja build systems as prerequisites.

Regarding the datasets, most of them, such as the Juliet Test Suite and the Linux Flaw Project, are publicly available. Additionally, the source codes of other applications used in our evaluation can also be accessed publicly. However, to simplify access and ensure convenience, we have included all of these datasets in our repository.

A.2.1 Security, privacy, and ethical concerns

No destructive steps are taken or no security mechanism are disabled during the build process of DPP. One just needs to install a compatible version (>= 3.13.4) of CMake.

A.2.2 How to access

The source code of DPP is available publicly on GitHub at `https://github.com/salmanyam/DPP` with commit `53cbccb`. The full URL is `https://github.com/salmanyam/DPP/tree/53cbccb6e6eaab6eaabbb06ea21fd31dd83e6eff`.

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

To compile and build the LLVM source code containing DPP's passes, we use Ubuntu 18.04. You will need CMake version 3.13.4 or higher to successfully compile and build the LLVM source code. Additionally, we suggest using the Ninja build system for this process. Ideally, on a system with all the necessary prerequisites, including Ninja and a compatible CMake version, the build process should proceed smoothly without any complications. It's worth noting that while the scripts have been tested on Ubuntu 18.04, they should also be compatible with the latest Ubuntu distributions.

A.2.5 Benchmarks

All the datasets and source codes that have been used in our evaluation are publicly available. However, we have provided the datasets and source code in our repository.

- Juliet Test Suite: `https://github.com/salmanyam/dpp-data/tree/main/juliet-test-suite`
- Linux Flaw Project: `https://github.com/salmanyam/dpp-data/tree/main/linux_flaw_project`
- Other applications' source code: `https://github.com/salmanyam/dpp-data/tree/main/src`
- Besides, we provide the LLVM IR files in `https://github.com/salmanyam/dpp-data/tree/main/IRx86`

A.3 Set-up

To replicate the evaluation environment for DPP, we recommend setting up an Ubuntu 18.04 distribution. To ensure a compatible CMake version is installed, we provide a convenient script called `install_cmake.sh`. After running this script, it is necessary to exit and restart the terminal to apply the installation changes effectively. Additionally, we offer another script, `prerequisites.sh`, which installs all the necessary prerequisites. If you have a fresh installation of the Ubuntu 18.04 distribution, please follow these steps from the root directory of our repository:

```
$ ./prerequisites.sh
$ ./install_cmake.sh
```

By following these steps, you can quickly set up the required environment for DPP and ensure all dependencies are properly installed.

A.3.1 Installation

To compile and build DPP along with the LLVM source, one needs to issue the build script (`build.sh`) provided in our repository. This script will do an in-source compilation of SVF and build LLVM binaries (`clang`, `opt`, `llvm-ar`, `lld`, etc) under `dpp-llvm/build/bin`.

A.3.2 Basic Test

To show simple prioritization results, we provide a simple program (`dpp-data/example/example.c`) and its LLVM IR (`dpp-data/example/example.opt`). The following commands give simple prioritization results.

```
$ LLVM_DIR=${PWD}/dpp-llvm/build/bin
$ ${LLVM_DIR}/opt -S -passes="print-dpp-global"
     --dpp-rule="all" -disable-output < ${PWD}/
    dpp-data/example/example.opt
```

The commands run all rules and prioritize/ranks the data objects in the simple program. The output of the command is the following:

```
######## SUMMARY: 4 data objects #############
AddrVFGNode ID: 17 AddrPE: [34<--35]
   %9 = call noalias i8* @malloc(i64 %8) #6, !
       dbg !25 { ln: 8 cl: 25 fl: example.c } 4
        10
----------------------------------------------
AddrVFGNode ID: 15 AddrPE: [22<--23]
   %4 = alloca i8*, align 8 { ln: 8 fl: example.
       c } 2 10
----------------------------------------------
AddrVFGNode ID: 11 AddrPE: [6<--7]
 @stdin = external dso_local global %struct.
     _IO_FILE*, align 8 { Glob } 1 1
----------------------------------------------
AddrVFGNode ID: 14 AddrPE: [20<--21]
   %3 = alloca [10 x i8], align 1 { ln: 6 fl:
       example.c } 1 0
----------------------------------------------
```

As can be seen from the output, there are four data objects prioritized by DPP for the simple program. Since the simple program is small and most data objects are input-dependent, almost all the data objects have been prioritized.

The following command is used to run the prioritization using a single rule:

```
$ LLVM_DIR=${PWD}/dpp-llvm/build/bin
$ ${LLVM_DIR}/opt -S -passes="print-dpp-global"
     --dpp-rule="rule9" -disable-output < ${PWD
    }/dpp-data/example/example.opt
```

The output of the above command is as follows:

```
AddrVFGNode ID: 17 AddrPE: [34<--35]
   %9 = call noalias i8* @malloc(i64 %8) #6, !
       dbg !25 { ln: 8 cl: 25 fl: example.c }
----------------------------------------------
```

In addition to this simple program, we have provided many LLVM IR files for real-world applications in https://github.com/salmanyam/dpp-data/tree/main/IRx86. We can use the abovementioned command to obtain the prioritized data objects by changing the input to those commands. For example, the following command takes the IR file of wuftpd and runs the prioritization using all rules.

```
$ LLVM_DIR=${PWD}/dpp-llvm/build/bin
$ ${LLVM_DIR}/opt -S -passes="print-dpp-global"
     --dpp-rule="all" -disable-output < ${PWD}/
    dpp-data/IRx86/wuftpd-2.6.0.bc
```

SAFER: Efficient and Error-Tolerant Binary Instrumentation

Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R. Sekar
Stony Brook University, NY, USA.
{spriyadarsha, hnnguyen, rchouhan, sekar}@cs.stonybrook.edu

A Artifact Appendix

A.1 Abstract

The artifact submission is for the paper titled "SAFER: Efficient and Error-Tolerant Binary Instrumentation". Our tool SAFER is an efficient and safe binary-instrumentation suite that is capable of instrumenting complex programs. The tool is compatible with both position independent (PIE) and position dependent (Non-PIE) executables and has a modest overhead of $\approx$2%.

The artifact consists of software and will be submitted in the form of a VM containing a pre-installed version of the software and scripts needed to run the software. It will also contain all the instrumented programs used during evaluation. SAFER was used to instrument 15 real world programs along with their shared libraries, customized coreutils binaries with data embedded in code and SPEC 2006 and 2017 binaries. The total size of all programs was about 1.1GB.

A.2 Description & Requirements

SAFER's current prototype requires Ubuntu 20.04 operating system. It further requires additional packages (*Capstone* and *Ocaml*) for disassembly and static analysis. We are submitting our artifact as an Oracle VirtualBox VM. SAFER is already installed in the VM along with all the prerequisite packages. The VM also contains SAFER's source code.

Requirements to run artifact: A x86-64 system with Oracle VirtualBox is required to use our artifact. Importing the virtual box image through Oracle Virtualbox running on a x86-64 system will recreate the testing environment. The virtual machine image is configured with 8GB of RAM and 100GB of secondary storage. So we recommend to run it on a system with at least 16GB of memory and 256GB of storage space.

Benchmarks like SPEC CPU 2006 and 2017 are already installed and set up with instrumented binaries. Other datasets like instrumented real-world applications and data-in-code coreutils are also present in the virtual machine.

A.2.1 Security, privacy, and ethical concerns

Since we are providing our system in a virtual machine, there is no risk for the host machine of the evaluator.

A.2.2 How to access

SAFER's artifact http://seclab.cs.sunysb.edu/seclab/safer

A.2.3 Hardware dependencies

An x86-64 machine preferably with 16GB of RAM and 256GB of storage space.

A.2.4 Software dependencies

Oracle VirtualBox 6 is required to run the virtual machine. The system setup in the virtual machine is complete and requires no additional software installation.

A.2.5 Benchmarks

For testing the performance overhead we use SPEC CPU 2006 and SPEC CPU 2017 benchmark suites. For functionality evaluation we use 15 pre-built real-world programs and custom-compiled coreutils with data-in-code.

A.3 Set-up

A.3.1 Installation

1. Download the virtual machine image file.
2. Install and open VirtualBox.
3. Select File, Import Appliance.
4. Select the virtual machine image.
5. Click import.

A.3.2 Basic Test

Instrumenting a program **ls** with its shared libraries.

1. Start the virtual machine.
2. Login with the password 'safer'.
3. Copy the original ls binary to the test folder.

```
> cp /usr/bin/ls ~/SBI/test
```

4. Find the dependencies of ls.

```
> cd ~/SBI/testsuite
> ./find_libs.sh /home/safer/SBI/test/ls
> truncate -s 0 randomized.dat
```

5. Run instrimentation on ls.

```
> cd ${HOME}
> ./instrument_prog.sh ls
```

6. Wait for the instrumentation to finish. Printed date means the process is completed.
7. Go to the output directory.

```
> cd ${HOME}/instrumented_libs/
```

8. Run the instrumented ls. Contents of the directory should be printed.

```
> ./ls
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SAFER can handle disassembly errors in the presence of complexities such as data embedded in code.

(C2): SAFER's pointer encoding scheme can detect instrumentation errors at runtime and deterministically abort (FAIL-CRASH).

(C3): SAFERis able to instrument position dependent (Non-PIE) code.

(C4): SAFER's pointer encoding and safe jump table transformation helps in achieving safe instrumentation with a fail-crash while having a modest overhead of $\approx 2\%$.

(C5): SAFERis able to instrument a wide variety of real world programs.

A.4.2 Experiments

(E1): [**Data-in-code test**]: We use coreutils and its built-in tests to ensure that SAFER is successfully able to instrument binaries where data is present in the code section.

Generating the dataset: We make use of a linker-script to compile a version of coreutils where read-only data is embedded in the code. The binaries and pre-built and available in the VM: `/home/safer/coreutils/coreutils-data/bin`.

Test preparation: Pre-instrumented binaries with SAFER's different modes (Table 3 in the paper) are available in `/home/safer/coreutils/coreutils-data` directory as below:

- *bin_FN_PRLG:* Function prologue based pointer classification.
- *bin_FULL_AT:* run time address translation.
- *bin_valid_ins:* valid instruction boundary based pointer classification.
- *bin_ABI:* ABI specification based pointer classification.

To reuse the above pre-instrumented binaries, copy all the binaries from one of the above mention directories to `/home/safer/coreutils/coreutils-8.30/src/`. Alternatively, binaries can be re-instrumented as follows:

```
> cd /home/safer
> ./instrument-coreutils.sh \
    config=<FULL_AT/FN_PRLG/valid_ins/ABI>
```

Testing: Coreutils in-built testsuite is used to test correctness of instrumented binaries:

```
> cd /home/safer/coreutils/coreutils-8.30
> make check
```

Results: FN_PRLG and FULL_AT configurations result in 100% correct instrumented binaries. There is 1 failure due to one of the test making use of LD_PRELOAD to load a dynamically built library. The library is built by the test case at the runtime and used. SAFER requires all program modules to be instrumented for correct execution. Since, this particular library is not available to us during instrumentation time, it is left unchanged and resulted in a crash.

Other approaches (*valid_ins* and *ABI*) specifications result in failures that are detected by SAFER's fail-crash design. The test will not proceed unless the faulty binaries are replaced. The list of failed binaries is in `coreutils-data/ABI_fail.sh` and `coreutils-data/valid_ins_fail.sh`. Running these scripts will replace the faulty binaries with correctly instrumented (with FULL_AT) ones and the test can progress.

(E2): [**Non-PIE binaries**]: SPEC 2006 binaries are compiled as non-PIE and then instrumented. Successful completion of SPEC tests ensure correct transformation. Other non-PIE programs such as gcc and Python have also been tested.

How to: We are providing the instrumented non-PIE SPEC binaries to reduce the testing time. However, if you want to instrument then again follow the steps in the preparation section otherwise skip to the execution section.

Preparation: Clean up and start the instrumentation.

```
> ./instrument-suite.sh \
  /home/safer/spec-06/nopie
```

Execution: • Change the directory to the spec-06 and run the command.

```
> cd /home/safer/spec-06/
> source shrc
> runspec --config=nopie.inst.cfg \
          --noreportable \
          --iterations=1 all
```

Wait for the SPEC run to complete. This may take several hours to complete ($\approx$ 4 hours).

- For real world programs (gcc and python), the testing process is described in the subsequent section.

Results: After the SPEC run is completed check the log file mentioned. There should be results for all the binaries except wrf and gamess whose uninstrumented versions fail on our setup.

(E3): [**Runtime overhead with SPEC 2006**]

How to: SPEC 2006 CPU benchmark suite is used to test the runtime overhead caused by SAFER's instrumentation. Note that the below steps are time consuming. Hence, they should be run in background (e.g., using nohup). Furthermore, we are providing pre-instrumented binaries. Hence, the preparation step can be skipped.

Preparation: Instrument the SPEC binaries.

```
> truncate -s 0
/home/safer/SBI/testsuite/randomized.dat
> cd /home/safer/
> ./instrument-suite.sh \
    /home/safer/spec-06/pie/
```

Wait for the instrumentation to complete.

Execution: Run the testsuite with uninstrumented binaries (base result).

```
> cd /home/safer/spec-06/
> source .shrc
> runspec --config=default.cfg \
    --noreportable all
```

Wait till completion and save the result directory as base. Then, run the instrumented binaries:

```
> cd /home/safer/spec-06
> runspec --config=inst.cfg \
        --noreportable all
```

Save the instrumented version results.

Results: We computed the overhead by importing the corresponding csv files onto a spreadsheet and comparing the timings of instrumented run over base run.

We are also providing the results of our experiments that have been published in the paper (/spec-06/our_results).

(E4): [**Performance vs optimization levels**] We compiled SPEC CPU 2017 benchmarks at 6 optimization levels (O0, O1, O2, O3, Ofast, and Os) and measured the rumtime overhead for each of them.

Preparation: Running the uninstrumented binaries:

```
> cd /home/safer/spec-17/
> source shrc
> runcpu --config=defaultO0.cfg \
        --noreportable intspeed
> runcpu --config=defaultO0.cfg \
        --noreportable fpspeed
```

Execution: Running the instrumented binaries:

```
> cd /home/safer/spec-17
> runcpu --config=default.inst.O0.cfg \
      --noreportable intspeed
> runcpu --config=default.inst.O0.cfg \
      --noreportable fpspeed
```

Results: Other optimizations could be tested by repeating the steps above with O0 replace by either O1, O2, O3, Ofast, or Os. The results can be obtained in the same manner as SPEC 2006.

(E5): [**Safe jump table transformation**]: SAFER's safe jump table analysis (Section 5 in paper), helps in improving performance while maintaining correctness of instrumentation by avoiding instrumentation of indirect jumps related to jump tables.

Preparation: Safe jump table analysis code is present in `/home/safer/SBI/safe_jtable`. Steps to build this code are present in a README file in the same directory.

Execution: Safe jump table results (Figure 5 in paper) were produced on SPEC 2006 binaries. Please follow the steps in README to reproduce the results.

Results: In default mode (enable_ftype=0), no function signature matching is done and it marks 55%jump tables as safe. Enabling the function signature matching (enable_ftype=1) results in 85% safe jump tables.

(E6): [**Real-world programs**]: We used 15 real-world programs (`/home/safer/real-world-instrumented`) to test SAFER's applicability on real-world programs.

Preparation: Programs can be instrumented as mentioned in Section A.3.2. There is no need to generate dependency list again (step 4). Some of the programs are fairly large and cannot be instrumented with the limited amount of RAM that the virtual machine is configured to run with. Hence, we are providing instrumented programs to skip the preparation step.

Execution: Steps to execute test cases are present in `/home/safer/real-world-instrumented/README`.

Results: The instrumented program produces desired output. (e.g., gedit is able to open and edit files).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Reassembly is Hard: A Reflection on Challenges and Strategies

Hyungseok Kim[1,2], Soomin Kim[1], Junoh Lee[1], Kangkook Jee[3], and Sang Kil Cha[1]
[1]*KAIST* [2]*The Affiliated Institute of ETRI* [3]*University of Texas at Dallas*
{witbring,soomink,junoh,sangkilc}@kaist.ac.kr kangkook.jee@utdallas.edu

A Artifact Appendix

A.1 Abstract

REASSESSOR is a tool for finding errors in the implementations of existing reassemblers. This artifact includes the source code of REASSESSOR, the dataset used in our paper, and several scripts for reproducing the results in the paper. As a preprocessing step, one needs to run three existing reassemblers on our dataset, including Ramblr, RetroWrite, and Ddisasm. We provide a dockerized environment to run them. The preprocessing step produces a re-assemblable assembly file for each binary, and REASSESSOR uses those files to find reassembly errors. The next section details each step to reproduce the results in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Not applicable.

A.2.2 How to access

The source code of REASSESSOR is accessible through GitHub at https://github.com/SoftSec-KAIST/Reassessor/tree/v1.0.0. We also provide our dataset at https://doi.org/10.5281/zenodo.7178116.

A.2.3 Hardware dependencies

To reproduce the whole results, it requires at least 2.5TB of disk space. In our experiments, we used a machine equipped with 8 cores of CPUs and 128GB of RAM.

A.2.4 Software dependencies

REASSESSOR is designed to run on a Linux machine, and we tested it on Ubuntu 18.04 and Ubuntu 20.04. Also, REASSESSOR is written in Python 3 (3.6), and it depends on `pyelftools` (>= 0.29) and `capstone` (>= 4.0.2). Besides, Docker needs to be installed on the same machine to run reassemblers within a Docker container. Our scripts assume that you can run Docker commands as a regular (unprivileged) user; thus, no need to run them with `sudo`.

A.2.5 Benchmarks

Our benchmark is accessible through Zenodo: https://doi.org/10.5281/zenodo.7178116. We created our benchmark with various combinations of compilers, linkers, target ISAs, and compiler options.

- ISA: x86 and x86-64 (= 2)
- Compilers: GCC v7.5.0 and Clang v12.0 (= 2)
- PIE/non-PIE: produce a PIE or a non-PIE (= 2)
- Optimization: O0, O1, O2, O3, Os, and Ofast (= 6)
- Linkers: GNU ld v2.30 and GNU gold v1.15 (= 2)

Also, our benchmark was created by compiling two source packages totaling 122 executable programs as follows.[1]

- GNU coreutils (v8.30): 107 executable programs.
- GNU binutils (v2.31.1): 15 executable programs.

A.3 Set-up

A.3.1 Installation

Once you download our source code from the GitHub repository, you can install it using the following command:

```
$ pip3 install -r requirements.txt
$ python3 setup.py install -user
```

A.3.2 Basic Test

You can run REASSESSOR with a sample program as follows:
(Step 1): Build a sample program:

```
$ cd ./example
$ ./make
$ cd ..
```

[1]We exclude 31 programs in SPEC CPU 2006 from the dataset because of a licensing issue. Instead, we provide SSH server to grant access to all datasets we made.

(Step 2): Run `preprocessing.py` to reassemble it:

```
$ mkdir -p output
$ python3 -m reassessor.preprocessing \
./example/bin/hello ./output
$ ls output/reassem/
ddisasm.s retrowrite.s
```

(Step 3): Run REASSESSOR to find reassembly errors

```
$ python3 -m reassessor.reassessor \
example/bin/hello example/asm/ output/ \
--retrowrite ./output/reassem/retrowrite.s
$ ls output/errors/retrowrite/
disasm_diff.txt sym_diff.txt sym_errors.dat
sym_errors.json
```

REASSESSOR produces the following files as output: `ddisasm_diff.txt`, `sym_errors.dat`, `sym_diff.txt`, `sym_errors.json`. Firstly, `disasm_diff.txt` contains a list of disassembly errors (one per line); each line contains the relevant address, reassembler-generated assembly line, and compiler-generated assembly line. `sym_errors.dat` is a raw output file containing a list of symbolization errors. This file is used to generate other two files: `sym_errors.json` and `sym_diff.txt`. `sym_diff.txt` is a human-readable representation of `sym_errors.dat`. Each line of the file contains address, error type, reassembler-generated assembly code, and compiler-generated code, for each error found. Finally, `sym_errors.json` contains detailed information about each symbolization error found, including the relevant assembly file, line number, relocatable expression type, normalized code, repairability, and so on. The file is written in the JSON format.

A.4 Evaluation workflow

A.4.1 Preprocessing step for experiments

[10 human minutes + 5,000 CPU hours + 60 GB disk]

REASSESSOR finds reassembly errors by diffing compiler-generated assembly code and reassembler-generated assembly code. Thus, we need to reassemble benchmark binaries as follows:

(Step 1): Download the dataset:

```
$ cd artifact
$ tar -xzf /path/to/dataset/benchmark.tar.gz .
$ ls dataset/
binutils-2.31.1 coreutils-8.30
```

(Step 2): Run `run_preproc.py` to obtain reassembler-generated assembly code from each reassembler:

```
$ python3 run_preproc.py
```

`run_preproc.py` will then generate assembly files under the `./output` directory:

```
$ ls ./output
binutils-2.31.1 coreutils-8.30
```

```
$ cd \
output/binutils-2.31.1/x64/clang/nopie/o0-bfd/addr2line/
$ ls reassem
ddisasm.s ramblr.s
```

A.4.2 Major Claims

(C1): REASSESSOR *is able to find diverse reassembly errors. This is proven by the experiment (E1) described in Section 5.3 as well as Table 4.*

(C2): *Composite relocation expressions are prevalent in real-world binaries, and precise CFG recovery is a necessary condition for sound reassembly of x86-64 PIEs. This is proven by the experiment (E2) described in Section 5.2.2.*

(C3): *There are previously unknown FN/FP patterns. This is proven by the experiment (E3) described in Section 5.4.1 and 5.4.2.*

(C4): *Preventing data instrumentation can mitigate the symbolization challenge. This is proven by the experiment (E4) described in Section 5.5.2*

A.4.3 Experiments

(E1): [10 human minutes + 140 CPU hours + 330GB disk]
The experiment will search for reassembly errors by running REASSESSOR.
How to: First, run `run_reassessor.py` to find reassembly errors. Second, run `classify_errors.py` to collect the errors. Third, run `get_summary.py` to get the summarized result.
Preparation: The preprocessing step in §A.4.1 is required to have reassemablable assembly files.
Execution: Run `run_reassessor.py`

```
$ python3 run_reassessor.py --core 6
```

Results: First, check the report files described in §A.3.2:

```
$ cd output/binutils-2.31.1/x64/clang/nopie/o0-bfd/
$ cd addr2line/
$ ls errors
ddisasm ramblr
$ ls errors/ddisasm/
disasm_diff.txt sym_diff.txt sym_errors.dat
sym_errors.json
```

Second, run `classify_errors.py` to collect and classify symbolization errors from `sym_diff.txt` files:

```
$ python3 classify_errors.py --core 8
```

Check the results under the `triage` folder:

```
$ ls triage
ddisasm ramblr retrowrite
$ ls triage/ddisasm/x64/nopie/
E1FN.txt E1FP.txt E2FN.txt E2FP.txt E3FN.txt
E3FP.txt ...
```

Each file has a different set of errors, and each line of the files contains a relevant file name, error type, reassembler-generated assembly line, and compiler-generated assembly line.

Third, run `get_summary.py` to get a summarized result presented in Table 4:

```
$ python3 get_summary.py --core 8
```

(E2): [10 human minutes + 2.5 CPU hours + 100MB disk]
This experiment examines all relocatable expressions in our benchmark and reports the distributions of relocatable expressions for a different set of assembly files. Also, the experiment will show that the proportion of label-relative (Type VII) relocatable expressions in x86-64 PIE binaries is not negligible.
How to: First, run `get_asm_statistics.py` to examine compiler-generated assembly files. Second, run `get_e7_errors.sh` to find E7 errors for x86-64 PIE binaries.
Preparation: (E1) experiment needs to be run first since `get_asm_statistics.py` and `get_e7_errors.sh` refer to data files (`gt.dat` and `sym_diff.txt`) that REASSESSOR made.
Execution: Run `get_asm_statistics.py` and `get_e7_errors.sh`:

```
$ python3 get_asm_statistics.py -core 8
$ /bin/bash get_e7_errors.sh
```

Result: First, `get_asm_statistics.py` shows the distribution of relocatable expressions, and the proportion of composite relocatable expressions. Also, it reports how many binaries have abnormal cases including composite relocatable expressions pointing to outside of valid memory ranges and code pointers referring to non-function entries. Second, `get_e7_errors.sh` reports how many x86-64 binaries suffer from E7 errors.

(E3): [10 human minutes + 1 CPU minute + 2.2GB disk]
This experiment will find previously unseen symbolization errors.
How to: Run `dissect_errors.sh` to find previously unseen symbolization errors.
Preparation: (E1) experiment is required since `dissect_errors.sh` examines symbolization errors from `sym_diff.txt` files.
Execution: Run `dissect_errors.sh`:

```
$ /bin/bash dissect_errors.sh
```

Results: `dissect_errors.sh` reports how many reassembler-generated files have previously unseen errors. Also, `dissect_errors.sh` generates the report files: `atomic_fn_cases.txt`, `atomic_fp_cases.txt`, and `label_err_fp_cases.txt`. Each line of the files contains a relevant file name, error type, reassembler-generated assembly line, and compiler-generated assembly line. `atomic_fn_cases.txt` contains false negative cases where reassemblers misidentify atomic relocatable expressions as literals. `atomic_fp_cases.txt` contains false positive cases where reassemblers falsely symbolize atomic relocatable expressions as composite forms. Lastly, `label_err_fp_cases.txt` contains cases where symbolized labels have the same form as in the original one, while only the label values are misidentified.

(E4): [10 human minutes + 1 CPU minute + 6GB disk]
This experiment measures an empirical lower bound of the number of reparable symbolization errors when preventing data instrumentation. Specifically, this experiment will count symbolization errors that satisfy the criteria we suggested in Section 5.5.2.
How to: Run `check_reparable_errors.sh` to find symbolization errors that are reparable.
Preparation: (E1) experiment is required since `check_reparable_errors.sh` examines the error list files that `classify_errors.sh` generates.
Execution: Run `check_reparable_errors.sh`:

```
$ /bin/bash check_reparable_errors.sh
```

Results: `check_reparable_errors.sh` reports how many symbolization errors satisfy the reparable conditions we introduced in Section 5.5.2. Also, `reparable_errors.txt` contains the list of reparable symbolization errors; each line of the file has a relevant file name, error type, reassembler-generated assembly line, and compiler-generated assembly line.

A.5 Notes on Reusability

A.5.1 How to make a new dataset

If you want to use a new dataset, build binaries with '`-g`' option and '`--save-temps=obj`' option. Also, if you want to make non-pie binaries, add '`-Wl,--emit-relocs`' linker option to preserve relocation information.

A.5.2 How to test different versions of reassemblers

If you wish to run REASSESSOR with newer versions of reassemblers, update the execution commands in the `reassemble()` method in `preprocessing.py`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Are Consumers Willing to Pay for Security and Privacy of IoT Devices?

Pardis Emami-Naeini*, Janarth Dheenadhayalan†, Yuvraj Agarwal†, Lorrie Faith Cranor†

*Duke University †Carnegie Mellon University

A Artifact Appendix

A.1 Abstract

By conducting a two-phase online study on Prolific, we quantified the impact of various security and privacy improvements on Internet of Things (IoT) consumers' purchase behavior. Through designing an incentive-compatible experiment using the multiple price list (MPL) methodology, we captured participants' willingness to pay for transparency over security and privacy enhancements of smart devices. We constructed three regression models for each phase of our online study to quantify and explain participants' risk perception, willingness to purchase, and willingness to pay. In this artifact, we provide participants' de-identified survey data that we used to construct these models, the analysis code in R and STATA that we used to build the regression models, and the output files.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Conducting the statistical models for this paper does not introduce any risks. In addition, we de-identified the raw survey data to preserve participants' data privacy.

A.2.2 How to Access

The artifact, including the raw, de-identified survey data, analysis files, model output files, and the README file, is hosted on GitHub and could be accessed via the following stable URL: `https://github.com/pemamina/USENIX23_MonteryValueSP_Artifact/tree/e88e7eb5630996756f14335bf32abc4e9298e97a`.

A.2.3 Hardware Dependencies

None.

A.2.4 Software Dependencies

We used an open source tool, RStudio, to run two of the regression models (risk_clmm and purchase_clmm). Since R currently does not allow constructing mixed effects interval regressions, we used STATA to build the model to explain participants' willingness to pay. We downloaded RStudio via the following link: `https://www.RStudio.com/products/RStudio/download/`. We downloaded R using the following link: `https://cran.RStudio.com/`. We obtained STATA by using the following link: `https://www.STATA.com/`.

A.2.5 Benchmarks

None.

A.3 Set-Up

A.3.1 Installation

After installing RStudio and R, we need to install (`install.packages("ordinal")`) and load (`library(ordinal)`) the ordinal library required to construct CLMM models. This process is shown in lines 2 and 3 of `phase_one_analysis.R`, `phase_two_analysis.R`. No package needs to be installed in STATA to conduct mixed-effects interval regression.

A.3.2 Basic Test

Here we use dataset `ologit.csv` from the OARC website (`https://stats.oarc.ucla.edu/`). This dataset includes four variables: 1) `apply`: nominal categorical variable with three levels (0, 1, 2) showing how likely it is that the student will apply for grad school, 2) `pared`: categorical binary variable, showing whether parents have attended college (1) or not (0), 3) `public`: categorical binary variable, showing whether the school the student has attended is public (1) or not (0), and 4) `gpa`: continuous numeric variable, showing the student's GPA score.

Analysis goal. We would like to understand the impact of parents' college education (`pared`) on students' likelihood of applying to college (`apply`). Similar to our risk perception model and the willingness to purchase model, the dependent variable in this test (`apply`) is ordinal categorical. Therefore, we will construct a cumulative link model (CLM) to explain the impact of `pared` on `apply`.

```
## Loading the required library for ordinal
↪ regression.
library(ordinal)
```

```
## Loading the dataset ``ologit.csv''
dataset <-
↪  read.csv("https://stats.idre.ucla.edu/stat/data/ologit.csv")

## Changing the type of dependant variable
↪  \texttt{apply} from numerical (levels = 0, 1,
↪  2) to ordinal categorical (levels = "unlikely",
↪  "somewhat likely", "very likely")
dataset$apply <- factor(dataset$apply, labels =
↪  c("unlikely", "somewhat likely", "very likely"),
↪  ordered = TRUE)

## Changing the type of independent variable
↪  \texttt{pared} from numerical (levels = 0, 1)
↪  to nominal categorical (levels = "not attend",
↪  "attend")
dataset$pared <- factor(dataset$pared, labels = c("not
↪  attend", "attend"))

## Construction the CLM to explain the impact of
↪  \texttt{pared} on \texttt{apply}.
apply.clm <- clm(apply ~ pared, data = dataset)

# Showing the results
summary(apply.clm)
```

The output should look like:

```
formula: apply ~ pared
data:    dataset

 link  threshold nobs logLik  AIC    niter max.grad cond.H
 logit flexible  400  -361.40 728.79 5(0)  1.25e-10 9.3e+00

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
paredattend   1.1275     0.2634    4.28 1.87e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '
↪  ' 1

Threshold coefficients:
                                Estimate Std. Error z value
unlikely|somewhat likely          0.3768     0.1103   3.415
somewhat likely|very likely       2.4519     0.1826  13.430
```

By exponentiating the estimate in this model, we will calculate the odds ratio of 3.09. This shows that for students whose parents did attend the college, the odds of being more likely to apply to grad school is 3.09 times that of students whose parents did not attend the college.

A.4 Evaluation Workflow

The results of each phase of our study is based on three regression models. The regression results of phase one are included in `phase_one_CLMM_output.txt` and `phase_one_STATA_output.txt` and the regression results of phase two are included in `phase_two_CLMM_output.txt` and `phase_two_STATA_output.txt`. Here we provide the exact R and STATA code that we used to reach these results.

A.4.1 Major Claims

(C1): Our cumulative link mixed models describe participants' risk perception and willingness to purchase behavior in the first phase of our study. This is proven by the experiment (E1) described in Sections 4.2 and 4.3 of the paper, whose results are reported in Table 2 in the paper.

(C2): Our interval regression model describes participants' willingness to pay in the first phase of our study. This is proven by the experiment (E2) described in Sections 4.2 and 4.3 of the paper, whose results are reported in Table 2 in the paper.

(C3): Our cumulative link mixed models describe participants' risk perception and willingness to purchase behavior in the second phase of our study. This is proven by the experiment (E3) described in Sections 5.2 and 5.3 of the paper, whose results are reported in Table 4 in the paper.

(C4): Our interval regression model describes participants' willingness to pay in the second phase of our study. This is proven by the experiment (E4) described in Sections 5.2 and 5.3 of the paper, whose results are reported in Table 4 in the paper.

A.4.2 Experiments

(E1): Cumulative link mixed models in R: Risk perception and willingness to purchase models for phase-one study.

```
## Loading the required library for regression
↪  analysis
library(ordinal)

## Loading the survey data
dataset <- read.csv("phase_one_survey_data.csv")

## Specifying the dependent variables as
↪  ordinal categorical
dataset$risk_perception_coded <-
↪  factor(dataset$risk_perception_coded, order =
↪  TRUE,
                               levels = c("1", "2", "3",
                                  "4", "5"))
dataset$willingness_to_purchase_coded <-
↪  factor(dataset$willingness_to_purchase_coded,
↪  order = TRUE,
                             levels = c("1", "2", "3",
                                "4", "5"))

## Specifying the independent variables as
↪  categorical
dataset$order_scenario <-
↪  factor(dataset$order_scenario)
dataset$correct_definition_number <-
↪  factor(dataset$correct_definition_number)

## Setting the baseline for model independent
↪  variables
dataset$mostProtective_leastProtective_pair <-
↪  as.factor(dataset$mostProtective_leastProtective_pair)
dataset$smart_device <-
↪  as.factor(dataset$smart_device)

dataset$mostProtective_leastProtective_pair <-
↪  relevel(dataset$mostProtective_leastProtective_pair,
↪  "main_personal")
dataset$smart_device <- relevel(dataset$smart_device,
↪  "smoke")
```

```
## Constructing the Risk Perception Model
risk_clmm <- clmm(risk_perception_coded ~
↪ mostProtective_leastProtective_pair + smart_device
↪ +
                    correct_definition_number +
                    ↪ order_scenario +
                       (1|participant), data =
                       ↪ dataset, link = "logit"
)
summary(risk_clmm)

## Constructing the Willingness to Purchase
↪ Model
purchase_clmm <-
↪ clmm(dataset$willingness_to_purchase_coded ~
↪ mostProtective_leastProtective_pair + smart_device
↪ +
                        correct_definition_number +
                        ↪ order_scenario +
                    (1|participant), data = dataset,
                    ↪ link = "logit"
)
summary(purchase_clmm)
```

(E2): Mixed interval regression model in STATA: Willingness to pay model for phase-one study. We first need to import our CSV file. Since this file has long participant quotes, we should ensure the values of the cells do not overflow. We will specify the following parameters when importing the datafile: delimiter(comma), bindquote(strict), and stripquote(yes).

```
* We create a label to show the order of
↪ independent variables.
. label define factor_lab 1 "main_personal"
. label define device_lab 1 "smoke" 2 "speaker"

* We recode the independent variables with the
↪ new baseline
. encode mostprotective_leastprotective_p,
↪ generate(attribute_value) label(factor_lab)
. encode smart_device, generate(device_value)
↪ label(device_lab)

* We construct the mixed interval regression
↪ and set participant as the random effect.
. meintreg minimum_willingness_to_pay
↪ maximum_willingness_to_pay i.order_scenario
↪ i.correct_definition_number i.device_value
↪ i.attribute_value || participant:
```

(E3): Cumulative link mixed models in R: Risk perception and willingness to purchase models for phase-two study.

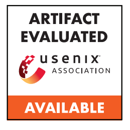
```
## Loading the required library for regression
↪ analysis
library(ordinal)

## Loading the survey data
dataset <- read.csv("phase_two_survey_data.csv")

## Specifying the dependent variables as
↪ ordinal categorical
dataset$risk_perception_coded <-
↪ factor(dataset$risk_perception_coded, order =
↪ TRUE,
                                             levels =
                                             ↪ c("1",
                                             ↪ "2", "3",
                                             ↪ "4", "5"))
dataset$willingness_to_purchase_coded <-
↪ factor(dataset$willingness_to_purchase_coded,
↪ order = TRUE,
                                                      levels
                                                      ↪ =
                                                      ↪ c("1",
                                                      ↪ "2",
                                                      ↪ "3",
                                                      ↪ "4",
                                                      ↪ "5"))

## Setting the baseline for model independent
↪ variables
dataset$label_type_comparison <-
↪ as.factor(dataset$label_type_comparison)
dataset$label_type_comparison <-
↪ relevel(dataset$label_type_comparison, "Z vs Y")

## Constructing the Risk Perception Model
risk_clmm <- clmm(risk_perception_coded ~
↪ label_type_comparison +
                    (1|participant), data = dataset,
                    ↪ link = "logit"
)
summary(risk_clmm)


## Constructing the Willingness to Purchase
↪ Model
purchase_clmm <-
↪ clmm(dataset$willingness_to_purchase_coded ~
↪ label_type_comparison +
                        (1|participant), data =
                        ↪ dataset, link = "logit"
)


* We create a label to show the order of
↪ independent variables.
. label define comparisonOrder 1 "Z vs Y" 2 "X vs Y"
↪ 3 "X vs Z"

* We recode the independent variables with the
↪ new baseline
. encode label_type_comparison,
↪ gen(typeComparisonCat) label(comparisonOrder)

* We construct the mixed interval regression
↪ and set participant as the random effect.
. meintreg min max i.typeComparisonCat || participant:
```

A.5 Notes on Reusability

None.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: PRIVATEFL: Accurate, Differentially Private Federated Learning via Personalized Data Transformation

Yuchen Yang*, Bo Hui*, Haolin Yuan*, Neil Gong†, and Yinzhi Cao
The Johns Hopkins University, †Duke University

A Artifact Appendix

A.1 Abstract

This artifact includes the source code, dataset, setup, and instructions to reproduce the results of PRIVATEFL reported in Section 6, i.e., the evaluation section. This artifact supports our claim that PRIVATEFL can improve the accuracy when applied to FL with DP, and can further improve the accuracy as an add-on to the existing DP-improving method. Our artifacts are available at this open-source repository (https://github.com/BHui97/PrivateFL). Our artifacts require a Linux machine with 64GB of RAM and a GPU with 24 GB of graphics memory.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

We provide the following accesse to our artifacts:

- GitHub repository: https://github.com/BHui97/PrivateFL. You can clone the source code from the main branches and set up the environment following the instruction in README.md file.

A.2.3 Hardware dependencies

We recommend using a 64bit Linux machine with the following requirements:

- GPU: 24G. We test the artifacts on NVIDIA GeForce RTX 3090
- RAM: 64G
- Storage: 16GB

A.2.4 Software dependencies

- Ubuntu 18.04
- Python 3.8+ and Python libraries listed in requirements.txt
- NVIDIA Driver and CUDA for GPU computation. We test the artifacts on NVIDIA Driver Version 510.108.03 and CUDA Version 11.6

A.2.5 Benchmarks

We list the datasets and models the experiments require with this artifact reported in our paper.

- Datasets:
  - CIFAR-10, MNIST, FashionMNIST, CIFAR-100, EMNIST: we use the library `torchvision.datasets`. Details can be found in `dataset.py` of our GitHub repo.
  - CH-MNIST: Dataset can be downloaded from https://github.com/BHui97/PrivateFL/tree/main/data/CHMNIST
  - Purchase: Dataset can be download from https://github.com/BHui97/PrivateFL/tree/main/data/purchase, please unzip the file before using.
- Models:
  - AlexNet, ResNet, 3-layer DNN, 4-layer DNN: details can be found in `modelUtil.py` of our GitHub repo.
  - CLIP: we use pretrained weights from the library `CLIP`, details can be found in `transfer/extract_cifar.py`.
  - SimCLR: model can be downloaded via https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_simclr_imagenet/simclr_imagenet.ckpt

A.3 Set-up

A.3.1 Installation

Source code. Start with the source code and set up the environment with the `README.md`, first install the source and enter it:

- `git clone https://github.com/BHui97/PrivateFL.git`
- `cd PrivateFL/script`

```
Client: 0 ACC: 0.9017160534858704, episilon: 1.9973331713299505
Client: 1 ACC: 0.8433049321174622, episilon: 1.9932507283841159
Round: 1
Acc: 0.5007001400280056
Epsilon: 1.9973331713299505
Use time: 43.09s
==> Test Finished!
```

Figure 1: Expected output of the basic test.

Conda (optional). This step is optional if you don't have an environment/package management tool installed. We use Miniconda to create a virtual environment with Python 3.8. You can install it using the following script for a Ubuntu 18.04 machine or refer to the official document[1]:

- `bash install_conda.sh`
- Note: remember to CLOSE and then RE-OPEN your terminal after running the above script.

Python dependencies. Then run the following script to install the required Python dependencies:

- `bash setup.sh`

Datasets. Datasets except Purchase will be automatically downloaded, we provide the Purchase dataset under `data/purchase/dataset_purchase.zip`. Please remember to unzip it before running the test.

Models. Models except ResNext will be automatically downloaded. Please download the ResNext weight `model_best.pth.tar` manually from this link, and put it under the `transfer/model/` folder.

A.3.2 Basic Test

Run the following script to verify the installation and test the functionality:

- `func_test_1.sh`

The expected successful output follows the structure shown in Figure 1. The specific number may be different for different tests.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PRIVATEFL *can improve the accuracy for FL with DP. This is proven by the experiments (E1), whose results are reported in Section 6.1 (Figure 5) of our paper.*

[1]https://docs.conda.io/projects/conda/en/latest/user-guide/install/linux.html

(C2): PRIVATEFL *can further improve the accuracy for FL with DP as an add-on to the existing DP-improving method. This is proven by the experiments (E2), whose results are reported in Section 6.2 (Table 6) of our paper.*

(C3): PRIVATEFL *consistently improves accuracy when the clients' local training data has different heterogeneity. This is proven by the experiments (E3), whose results are reported in Section 6.4 (Table 9) of our paper.*

(C4): PRIVATEFL *consistently improves the accuracy of FL under DP when the system has a different number of clients. This is proven by the experiments (E4), whose results are reported in Section 6.5 (Table 10) of our paper.*

A.4.2 Experiments

(E1): *[PrivateFL with different epsilon] [Six datasets] [20 human-minutes + 30 compute-hours (from 0.5 to 12 compute-hours for different datasets)]: This experiment tests the accuracy of PrivateFL with different epsilon, i.e., 2 to 8, on six datasets. Each of the datasets can be tested by running* `bash script/E1_[dataset].sh`
How to: The accuracy evaluation on each dataset will be tested via a data-specific script, and the evaluated accuracy for different epsilons will be automatically summarised and shown at the end of the script execution. You can collect the accuracies for each dataset and compare them with Figure 5 in our paper.
Preparation: Follow Section A.3 to finish the setup. Remember to unzip the purchase dataset before running the script for purchase.
Execution: First navigate to the script folder via `cd script`, then run `bash E1_[dataset].sh`. Please replace the `[dataset]` with one of [mnist, fashion-mnist, emnist, purchase, cifar10, chmnist], e.g, `bash E1_mnist.sh`. Note that cifar10 and chmnist need 12+ hours of training due to the large model size, which may vary from different GPUs. The other datasets could be finished within 1 hour. If you encounter CUDA out of memory, please reduce the value `-physical_bs` in `bash E1_[dataset].sh`.
Results: The results are shown as a table with four columns named `[data, mode, epsilon, accuracy]`. The `data` column shows the current evaluated dataset name; the `mode` column shows different DP methods, e.g, LDP; the `epsilon` column shows different epsilon that is being evaluated, e.g., 2; the `accuracy` column shows the testing accuracy for the combination of previous three columns, e.g., `accuracy = 0.946` for `[data = emnist, mode = CDP, epsilon = 2]`.

(E2): *[PrivateFL + DP-improvement with different epsilon] [Two datasets] [10 human-minutes + 2 compute-hours]: This experiment tests the accuracy of Pri-*

vateFL combined with existing DP-improvement methods. We test different epsilon, i.e., 2 to 8, on two datasets, with three different pretrained encoders. Each of the datasets can be tested by running `bash script/E2_[dataset].sh`

How to: The accuracy evaluation on each dataset will be tested via a data-specific script, and the evaluated accuracy for different epsilons and pretrained encoders will be automatically summarised and shown at the end of the script execution. You can collect the accuracies for each dataset and compare them with Table 5 in our paper.

Preparation: Follow Section A.3 to finish the setup. Remember to download the ResNext model following Section A.3.1 before running the script. To save time, we have uploaded the extracted features from different encoders under folder `transfer/feature`. If you want to extract it yourself, delete all folders, e.g., folder named `cifar10_2cpc_100client_clip_64`, under `transfer/feature`.

Execution: First navigate to the script folder via `cd script`, then run `bash E2_[dataset].sh`. Please replace the `[dataset]` with one of [cifar10, cifar100], e.g, `bash E2_cifar10.sh`

Results: The results are shown as a table with four columns named `[data, mode, model, epsilon, accuracy]`. The `model` column shows the pretrained encoder used to extract the feature, e.g., clip; the `data, mode, epsilon` columns are similar to E1; the `accuracy` column shows the testing accuracy for the combination of the previous four columns, e.g., `accuracy = 0.594` for `[data = cifar100, mode = CDP, model = clip, epsilon = 2]`.

(E3): *[PrivateFL with the different data heterogeneity] [Two datasets] [10 human-minutes + 2 compute-hours]: This experiment tests the accuracy of PrivateFL with different data heterogeneity, i.e., 2 to 10 classes per client, on two datasets. Each of the datasets can be tested by running* `bash script/E3_[dataset].sh`

How to: The accuracy evaluation on each dataset will be tested via a data-specific script, and the evaluated accuracy for different data heterogeneity will be automatically summarised and shown at the end of the script execution. You can collect the accuracies for each dataset and compare them with Table 9 in our paper.

Preparation: Follow Section A.3 to finish the setup.

Execution: First navigate to the script folder via `cd script`, then run `bash E3_[dataset].sh`. Please replace the `[dataset]` with one of [mnist, cifar10], e.g, `bash E3_mnist.sh`

Results: The results are shown as a table with four columns named `[data, mode, ncpc, accuracy]`. The `data, mode` columns are similar to E1; the `ncpc` is the number of classes assigned to each client, i.e., 2 (non-iid) to 10 (iid); the `accuracy` column shows the testing accuracy for the combination of the previous three columns, e.g., `accuracy = 0.922` for `[data = mnist, mode = CDP, ncpc = 2]`.

(E4): *[Private FL with the different number of clients] [Two datasets] [10 human-minutes + 5 compute-hours]: This experiment tests the accuracy of PrivateFL with the different number of clients, i.e., 50 to 500, on two datasets. Each of the datasets can be tested by running* `bash script/E4_[dataset].sh`

How to: The accuracy evaluation on each dataset will be tested via a data-specific script, and the evaluated accuracy for different numbers of clients will be automatically summarised and shown at the end of the script execution. You can collect the accuracies for each dataset and compare them with Table 10 in our paper.

Preparation: Follow Section A.3 to finish the setup.

Execution: First navigate to the script folder via `cd script`, then run `bash E4_[dataset].sh`. Please replace the `[dataset]` with one of [mnist, cifar10], e.g, `bash E4_mnist.sh`

Results: The results are shown as a table with four columns named `[data, mode, nc, accuracy]`. The `data, mode` columns are similar to E1; the `nc` is the number of clients, i.e., 50 to 500; the `accuracy` column shows the testing accuracy for the combination of the previous three columns, e.g., `accuracy = 0.892` for `[data = mnist, mode = LDP, nc = 50]`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: "Meta-Sift: How to Sift Out a Clean Subset in the Presence of Data Poisoning?"

Yi Zeng[1,2], Minzhou Pan[1], Himanshu Jahagirdar[1], Ming Jin[1], Lingjuan Lyu[2], and Ruoxi Jia[1]

[1]Virginia Tech, Blacksburg, VA 24061, USA
[2]Sony AI, Tokyo, 108-0075, Japan

A Artifact Appendix

A.1 Abstract

This artifact appendix focuses on road-mapping the **three** main claims we developed in the "Meta-Sift" paper:

- **Defense performance is sensitive to the purity of the base set** (referring to Takeaway #1, **Section 1** and **Section 2.1**): Representative works of defense methods against data poisoning are only effective when they can access a small, clean-held-out dataset (base set). When infiltrated with poisoned samples, the defense effects of these methods are significantly impaired.

- **Both existing automated methods and human inspection fail to identify a clean subset with high enough precision** (referring to Takeaway #2, **Section 1** and **Section 2.3, 2.4**): To evaluate existing methods for identifying a clean base set from a poisoned dataset and conducting a human study, we found that these techniques cannot satisfy the necessary access to the base set required to initiate the defenses mentioned above.

- **Our proposed solution, Meta-Sift** (our main contribution, referring to Takeaway #3, **Section 1**, the implementation of the proposed method in **Section 3**, and results in **Section 4.2**), utilizes a new but intuitive idea that training a model on the clean portion of a (corrupted) dataset will perform poorly on the poisoned portion and vice versa, which is a splitting problem that helps to identify the clean samples and can be described as a bi-level optimization (Eqn. 11, 12). We have introduced a suite of techniques to build Meta-Sift to resolve this splitting problem, resulting in efficient and effective solutions.

A.2 Description & Requirements

The provided artifacts focus on reproducing results with the GTSRB dataset (smaller and easy to download). Our implementation has been tested on our server and can be accessed via SSH, along with the required software environments and dependencies. By running the implementation on our provided backend, you can reproduce examples of the experiments that support our claims. If you choose to re-implement everything on your hardware/software, it might require a machine with the following minimum requirements:
Hardware requirements: CPU: 1×AMD EPYC 7763 64-Core Processor; GPU: 1×NVIDIA RTX A6000 48 GB.
Software requirements: Operating system: Linux (Ubuntu 20.04); Python 3.9; CUDA 11.8; cuDNN 8.7.0; Required Python packages: h5py (version 3.6.0 or later); imageio (version 2.9.0 or later); numpy (version 1.21.5 or later); Pillow (version 9.4.0 or later); torch (version 1.13.0 or later); torchvision (version 0.14.0 or later); tqdm (version 4.64.0 or later); jupyter notebook (version 6.4.8 or later).

A.2.1 Security, privacy, and ethical concerns

We do not collect data or fingerprints while the evaluators use our provided backend for re-implementation. The concern is not applicable if the evaluator uses their own hardware/software platform.

A.2.2 How to access

We provide a stable released version of our implementation via the following stable reference of the GitHub link: `https://github.com/ruoxi-jia-group/Meta-Sift/releases/tag/artifact`. We have created an anonymous SSH account for evaluators to access our hardware platform. Please directly contact the authors for further instructions on using our provided backend.

A.2.3 Hardware dependencies

We highly recommend the evaluators **use our provided hardware backend** for reproducing the results.

A.2.4 Software dependencies

We have tested our artifact in Linux OS (Ubuntu 20.04), along with Python 3.9, CUDA 11.8, and cuDNN 8.7.0. Python packages required, including h5py (version 3.6.0 or later), imageio (version 2.9.0 or later), numpy (version 1.21.5 or later), Pillow (version 9.4.0 or later), torch (version 1.13.0 or later),

torchvision (version 0.14.0 or later), tqdm (version 4.64.0 or later), and jupyter notebook (version 6.4.8 or later). Before evaluating the artifact, please ensure all the necessary software components and packages are installed and configured correctly. For simplicity, we have provided the "**metasift.yml**" file so that one can easily install the required environment with Conda. Detailed instruction is **listed in the GitHub release**.

A.2.5 Benchmarks

Throughout our artifacts, we mainly reproduce the results on the GTSRB dataset [1], which features traffic sign images scaled to 32×32 pixels. This dataset includes 39,209 training samples and 12,630 testing samples, both with class-imbalanced distributions, providing a real-life set of data for our analysis. In the "quick_start.ipynb," we utilized a VGG-16 model that was poisoned with BadNets backdoor attack in "class 38" as the poison model (model structure and poisoned parameters obtained from [2]) that will be using I-BAU [2] for purification. In the subsequent Meta-Sift processes, we used a ResNet-18 [3] model as the feature extractor θ for sifting.

A.3 Set-up

To prepare the environment for evaluating our artifact, one needs to first log in to our server (**contact us!**) or a server with minimum hardware configuration required. After that, simply follow the Conda command provided in the GitHub release will be able to set up the required environment.

A.3.1 Installation

To install the required software environment and the artifacts, first, download all the code from the GitHub release. Once the artifacts are downloaded, one needs to navigate to the directory using the command line and install the required software dependencies. Meanwhile, the GTSRB dataset can be found here. Please **download the required GTSRB dataset and put it under the "./dataset" folder**. By following these steps, one should have a properly configured environment ready for evaluation.

A.3.2 Basic Test

Please **go to the "quick_start.ipynb," and try to run the first block.** If no error pops up, it indicates that the prerequisite environment has been successfully installed.

A.4 Evaluation workflow

This artifact consists of three functional parts that accounts for three experiments:

- "./quick_start.ipynb" contains three example experiments supports each claim (Takeaway #1, #2 and #3);
- "./human_exp" folder provides the tool and a Narcissus [4] poisoned image dataset we built for human study;
- "main.py" implements our proposed method, which accounts for the core contribution, and one can thoroughly evaluate Meta-Sift with different poison settings.

A.4.1 Major Claims

Please refer to Section A.1 for a detailed recap of our claims. Mainly, we have claimed that:

(C1): Defense effects are sensitive to the purity of the base set (Takeaway #1 in **Section 1**, results in **Section 2.1**).

(C2): Both existing automated methods and human inspection fail to identify a clean subset with high enough precision (Takeaway #2 in **Section 1**, results and analysis in **Section 2.3** and **2.4**).

(C3): Our proposed solution, Meta-Sift, can obtain a clean subset with the required budget in many poison situations (Takeaway #3 in **Section 1**, results in **Section 4.2** or **Table 15** for the GTSRB).

A.4.2 Experiments

(E1): *[1 human-minutes + 10 compute-minutes]: **C1**, part of **C2** (Automated methods fail to identify a pure enough clean subset), and one experiment for **C3**.*

How to:

- *First, we reproduce one experiment in Table 1 to support **C1**. The code first loads a poisoned small VGG-16 [2] that has been poisoned with BadNets [5] targeting class 38. The code then executes I-BAU [2] for backdoor removal with a randomly selected 1000-size clean base set. We can observe that the ASR of the model is successfully mitigated. Then, we consider a poisoned base set with **8** poisoned samples mixed in, and we can observe the efficacy of the defense is largely impaired.*
- *Moving forward, we reproduce one experiment in Table 2 to support **C2** on automatic methods. The notebook implements an automated sifting method which we termed Distance to the Class-Means (DCM), to sift out a base set and evaluate the Normalized Corruption Rate (NCR) of the selected base set. The resulting value is much larger than 0, indicating that automated methods fail to identify a clean subset with sufficient precision within the given 1000 selection budget.*
- *Finally, we reproduce Meta-Sift's result on BadNets poisoned GTSRB to confirm **C3**. Upon running the code in the notebook, we can observe that the NCR is 0, indicating that we successfully identified a pure-clean base set within the same 1000 budget.*

Preparation: *Please complete Section A.3 first.*
Execution: *Run "quick_start.ipynb."*
Results: *Expected results are in the current notebook.*

(E2): *[30 human-minutes]: **C2** on humans' inability to identify the poisons with enough precision.*

How to: *We suggest the evaluator should not check the visual examples in our paper before completing this experiment for non-necessary bias. Please first go to the "./exp_human" folder, unzip the "img.zip", and open "huamn_label_interface.html" with your web browser. The interface allows human labelers to assess whether an image is poisoned and output the results with the "yes" or "no" button. Once one successfully goes through all 1000 samples, the browser will automatically download the "result.csv" file. Reviewers can compare these results with the "gound_trouth.xlsx" to calculate the false-negative rate (FNR).*

Preparation: *A web browser is required.*

Execution: *After inspection, the browser will automatically download a "result.csv" file. Please copy the first column from "result.csv" and paste it over column B, "ground_trouth.xlsx," to get the final FNR. One can compare the results with the results in Figure 3.*

Results: *The results should be quite similar to our results in Figure 3, i.e., end up with high FNR.*

(E3): *[1 human-minutes + 20 compute-minutes]: Further evaluation of **C3** (main contribution) with representative poisoning attack under each category.*

How to: *Please refer to the commands in the GitHub release and run them in a terminal with the required Conda environment.*

Preparation: *Please complete Section A.3 first.*

Execution: *Run commands one by one in a terminal.*

Results: *The sifting results over these three poison settings should all end up with an NCR equal to 0 with our proposed method at a selection budget of 1000 (default).*

A.5 Notes on Reusability

In the current release, we provide a plug-in (optional) function that allows for adopting Meta-Sift to identify a clean base set with a specific selection budget from any dataset. When using Meta-Sift on a new dataset, choosing hyperparameters with care is essential. We suggest using a pseudo-poisoned dataset by applying the Narcissus attack [4] with a 10% poison ratio on the top of the provided dataset (i.e., despite what poison the original dataset is poisoned with, we manually introduce the Narcissus attack over the whole dataset). Narcissus is the most stealthy but effective attack we have found in our empirical study. Once you have the Narcissus poisoned dataset on top of the given dataset, input it into Meta-Sift and fine-tune the hyperparameters to reduce the output NCR. When you have a set of hyperparameters that can help you achieve 0 NCR on this pseudo-poisoned dataset, it should perform well in sifting out a 0 NCR base set for your provided dataset.

To adjust the main hyperparameters:

"**-warmup_epochs**" determines the number of rounds the model should be pre-trained on the dataset before starting the sift. For example, in GTSRB, the model's accuracy should be kept around 50% after warmup, while the accuracy of a well-trained model is over 90%.

"**-batch_size**" is an important parameter influencing performance. Keeping the "batch_size" small allows the model to be updated more frequently, which might lead to better NCR over low-resolution datasets.

"**-v_lr**" determines the learning rate of the weight-assigning network, and it should be adjusted for the dataset size. This parameter should be as small as possible for large datasets to prevent overfitting.

"**-top_k**" determines the last few layers of the model that will be selected to compute the gradient for the virtual update. This parameter should be adjusted according to the depth of the model, and the larger it is, the better it is for a model with a deep structure. In RestNet-18, this parameter is set to 15, covering the last residual block.

"**-num_sifter**" controls how many sifters will be trained. Increasing this setting improves the filtering effect but adds to the time/memory overhead. Starting from five and gradually growing, this parameter is recommended.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

References

[1] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The german traffic sign recognition benchmark: a multi-class classification competition," in *The 2011 international joint conference on neural networks*. IEEE, 2011, pp. 1453–1460.

[2] Y. Zeng, S. Chen, W. Park, Z. Mao, M. Jin, and R. Jia, "Adversarial unlearning of backdoors via implicit hypergradient," in *ICLR*, 2022.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[4] Y. Zeng, M. Pan, H. A. Just, L. Lyu, M. Qiu, and R. Jia, "Narcissus: A practical clean-label backdoor attack with limited information," *arXiv:2204.05255*, 2022.

[5] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, "Badnets: Evaluating backdooring attacks on deep neural networks," *IEEE Access*, vol. 7, pp. 47 230–47 244, 2019.

USENIX'23 Artifact Appendix: Towards A Proactive ML Approach for Detecting Backdoor Poison Samples

Xiangyu Qi
Princeton University

Tinghao Xie
Princeton University

Jiachen T. Wang
Princeton University

Tong Wu
Princeton University

Saeed Mahloujifar
Princeton University

Prateek Mittal
Princeton University

A Artifact Appendix

A.1 Abstract

This artifact is mostly based on PyTorch, requiring GPU support. We implemented the proposed defense, Confusion Training (Algorithm 1) of our paper "Towards A Proactive ML Approach for Detecting Backdoor Poison Samples", together with a diverse set of baseline defenses and attacks. The artifact can reproduce our major experimental results (true positive rate, false positive rate, clean accuracy and attack success rate) reported in the main body of the paper. Our source code is available at https://github.com/Unispac/Fight-Poison-With-Poison, with a detailed guide at https://github.com/Unispac/Fight-Poison-With-Poison/blob/master/misc/reproduce.md.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. All backdoor attacks against DNNs in our artifact are conducted on the simulation level, therefore do not lead to any damage in the real world.

A.2.2 How to access

Our artifact source code is hosted at a GitHub repository, available through https://github.com/Unispac/Fight-Poison-With-Poison. For artifact evaluation purposes, this commit is used: https://github.com/Unispac/Fight-Poison-With-Poison/tree/b9ef34d

A.2.3 Hardware dependencies

This artifact minimally requires a Linux server with 300 GB disk storage, 10 GB RAM, 4 CPU cores, and 2 Nvidia GPUs (we use A100 in our experiments).

A.2.4 Software dependencies

This artifact relies on multiple existing Python packages, including Python, PyTorch, scipy and so on (details in requirement.txt). To reproduce only results of our proposed defense (Confusion Training), you may maually install PyTorch following their official guide and all other packages with pip. To produce results of other baseline defenses (specifically, Frequency and SPECTRE), you may also need to manually install Tensorflow (refer to official guide) and Julia (refer to other_cleansers/spectre/README.md). Our guide includes all details to set up the required software dependencies.

A.2.5 Benchmarks

Our experiments with this artifact are reported on 4 benchmark datasets: CIFAR10 (a 10-class common image classification task), GTSRB (a 43-class traffic sign recognition task), ImageNet (a 1000-class standard image classification task), and Ember (a malware classfication task). Among them, CIFAR10 and GTSRB are automatically downloaded and set up in our artifact, and a detailed guide to download and set up ImageNet and Ember datasets is available at misc/reproduce.md#todo-before-you-start.

A.3 Set-up

A.3.1 Installation

Our documentation contains a detailed guide to install our artifact and required environments. Briefly, the installation procedure is as follows:

1. Clone artifact from https://github.com/Unispac/Fight-Poison-With-Poison/tree/f2f02c2.

2. Install PyTorch following the official guide.

3. Install other Python packages via `pip install -r requirement.txt`.

4. (Optional) Install Tensorflow [guide] and Julia [guide].

5. Download ImageNet [link] and Ember [link] datasets. Refer to [link] for more details to set them up properly.

6. Execute command `python create_clean_set.py -dataset=$DATASET -clean_budget=$N` (where `$DATASET = cifar10, gtsrb, imagenet, ember`, $N = 2000 for cifar10 and gtsrb, $N = 5000 for imagenet and ember) to initialize the datasets.

7. Run `data/cifar10/clean_label/setup.sh` to setup data for clean label (CL) attack.

8. Download pretrained models (for Dynamic attack) `all2one_cifar10_ckpt.pth.tar` [link] and `all2one_gtsrb_ckpt.pth.tar` [link] to `models/`.

A.3.2 Basic Test

We provide a simple example (defending against BadNet attack on CIFAR10) involving our whole artifact pipeline (poisoning, training, defense, retraining, etc.) in our detailed guide at misc/reproduce.md. Briefly, one may test our artifact's core functionalities via:

1. Create a BadNet poisoned dataset by running `python create_poisoned_set.py -dataset=cifar10 -poison_type=badnet -poison_rate=0.01`.

2. Train on the poisoned dataset by running `python train_on_poisoned_set.py -dataset=cifar10 -poison_type=badnet -poison_rate=0.01`. The output should include the clean accuracy (ACC) and attack success rate (ASR) of the backdoor model in each training epoch.

3. Launch our Confusion Training defense by running `python ct_cleanser.py -dataset=cifar10 -poison_type=badnet -poison_rate=0.01 -devices=0,1 -debug_info`. The last couple lines of the output should include the defense results (`recall` and `fpr`).

4. Retrain on the cleansed training set by running `python train_on_cleansed_set.py -cleanser=CT -dataset=cifar10 -poison_type=badnet -poison_rate=0.01`. The output should include the clean accuracy (ACC) and attack success rate (ASR) of the defended model in each training epoch.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Confusion Training is an effective approach to identify and remove poisoned samples in the training set. Compared to other baseline defenses, Confusion Training consistently shows better robustness. This is proven by the experiments (E1) in Sec 5.2 whose results are reported in Table 1 and Table 2.

(C2): Confusion Training is generalizable to larger dataset and extends beyond the vision domain. This is proven by the experiments (E2) and (E3) in Sec 5.3 whose results are reported in Table 4 and Table 5.

A.4.2 Experiments

(E1): [Major Experiments on CIFAR10 and GTSRB] [30 human-minutes + 100 compute-hour + 10GB disk]: Experiment (E1) evaluates and compares Confusion Training's effectiveness on CIFAR10 and GTSRB across 11+9 attacks with 11 baseline defenses, therefore proving our first claim (C1). Experiment (E1) corresponds to our reported results in Table 1 and Table 2.
How to: *All the preparation steps have been stated in Artifact Appendix A.3. We provide all necessary commands to reproduce our results of (E1) in misc/reproduce.md#major-results-on-cifar10-and-gtsrb-table-1–table-2.*

(E2): [Experiments on ImageNet] [1 human-hour + 160 compute-hour + 300GB disk]: Experiment (E2) evaluates Confusion Training's effectiveness on ImageNet, a larger vision dataset, and therefore provides support to our second claim (C2). Experiment (E2) corresponds to our reported results in Table 5.
How to: *All the preparation steps have been stated in Artifact Appendix A.3. We provide all necessary commands to reproduce our results of (E2) in misc/reproduce.md#imagenet-table-5.*

(E3): [Experiments on Ember] [1 human-hour + 3 compute-hour + 50 GB disk]: Experiment (E3) evaluates Confusion Training's effectiveness on Ember, a malware classification task, also providing support to our second claim (C2). Experiment (E3) corresponds to our reported results in Table 4.
How to: *All the preparation steps have been stated in Artifact Appendix A.3. We provide all necessary commands to reproduce our results of (E3) in misc/reproduce.md#ember-table-4.*

The expected outcome for the experiments above should be close to our reported results. Our experiments involve randomness in nature. Thus, the outcomes may have some variances. Our table also reports the approximate standard deviation of each result.

A.5 Notes on Reusability

We have already incorporated this artifact into a more comprehensible backdoor research toolbox, available at https://github.com/vtu81/backdoor-toolbox, which will be

constantly maintained in the foreseeable future.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: **"Every Vote Counts: Ranking-Based Training of Federated Learning to Resist Poisoning Attacks"**

Hamid Mozaffari, Virat Shejwalkar & Amir Houmansadr
University of Massachusetts Amherst
{hamid, vshejwalkar, amir}@cs.umass.edu

A Artifact Appendix

A.1 Abstract

This report provides details of our implementation of a federated learning algorithm called Federated Rank Learning (FRL) which is designed to achieve high test accuracy and mitigate the risk of model poisoning attacks in a non-iid client distribution. The report includes the implementation of FRL on CIFAR10 and MNIST datasets with different percentages of malicious clients. The evaluation workflow of the code includes running experiments to validate the major claims of the paper and measuring the test accuracy of the models. The report also includes the installation instructions and requirements for running the code.

A.2 Description & Requirements

A.2.1 How to access

Our artifact, the implementation of our work on Federated Rank Learning (FRL), can be accessed at https://github.com/SPIN-UMass/FRL. The code is written in PyTorch and is publicly available for anyone to use. The repository includes a comprehensive readme file that provides instructions on how to run different experiments, making it easy for others to replicate our results and build upon our work. To ensure that our artifact is easily accessible and can be referenced by others in the future, we have chosen to host it on the popular repository hosting platform GitHub.

A.2.2 Hardware dependencies

To evaluate our artifact, a system with a GPU is required for faster learning. Our experiments were conducted on an NVIDIA GeForce GTX 1080 Ti with 11GB RAM.

A.2.3 Software dependencies

The experiments in this study were conducted using the PyTorch 1.13.1 and Numpy 1.23.5 libraries. PyTorch is a widely-used deep learning framework that provides a seamless integration of computation graphs and tensors, making it an ideal choice for implementing and training neural networks.

Table 1: In our experiments, we use the following, state-of-the-art model architectures.

Architecture	Layer Name	Number of parameters
LeNet (MNIST)	Conv(32)	288
	Conv(64)	18432
	FC(128)	1605632
	FC(10) or FC(62)	1280
Conv8 (CIFAR10)	Conv(64), Conv(64)	38592
	Conv(128), Conv(128)	221184
	Conv(256), Conv(256)	884736
	Conv(512), Conv(512)	3538944
	FC(256), FC(256), FC(10)	592384

Numpy, on the other hand, is a library for the Python programming language that provides support for large, multi-dimensional arrays and matrices, along with a wide range of mathematical functions to operate on these arrays. By using these two libraries in our experiments, we were able to efficiently implement and evaluate the performance of our models.

A.2.4 Benchmarks

We provide two benchmark datasets widely used in prior works on federated learning robustness:
MNIST is a 10-class class-balanced classification task with 70,000 gray-scale images, each of size 28×28. We experiment with LeNet architecture given in Table 1. For local training in each FRL/FL round, each client uses 2 epochs. For training ranks (experiments with FRL), we use SGD with learning rate of 0.4, momentum 0.9, weight decay 1e-4, and batch size 8.
CIFAR10 is a 10-class classification task with 60,000 RGB images (50,000 for training and 10,000 for testing), each of size 32×32. We experiment with a VGG-like architecture given in Table 1. For local training in each FRL/FL round, each client uses 5 epochs. For training ranks (experiments with FRL), we optimize SGD with learning rate of 0.4, momentum of 0.9, weight decay of 1e-4, and batch size of 8.

Table 1 shows the state-of-the-art model architectures and corresponding datasets that we use in our experiments. We also show the number of parameters in each of their layers.

A.3 Set-up

A.3.1 Installation

To install the code for this study, please follow these steps:

1. To download the repository, use the following command: **git clone https://github.com/SPIN-UMass/FRL.git**. The final stable URL is: https://github.com/SPIN-UMass/FRL/tree/4cf2550972e0e6299f61f682579f10b8e32c39d7.

2. Create a new conda environment. you can do so using the following command: **conda create --name FRL_test python=3.10.9**

3. Activate the environment: **conda activate FRL_test**

4. Then, to install the dependencies, run: **pip install -r requirements.txt**

This will download the repository and install all of the necessary dependencies, including PyTorch and Numpy, as specified in the software appendices section. Once the installation is complete, you should be able to run the code and reproduce the results from the study.

A.3.2 Basic Test

To run a simple experiment on the CIFAR10 dataset using the Federated Rank Learning (FRL) algorithm, run the following command: **python main.py --data_loc /CIFAR10/data/ --config experiments/001_config_CIFAR10_Conv8_FRL_1000users_noniid1.0_nomalicious.txt**. This will initiate a federated learning experiment on the CIFAR10 dataset using 1000 clients in a non-iid fashion with a Dirichlet distribution parameter $\beta = 1.0$. The experiment will run for 2000 global FL rounds, with 25 clients selected for local updates in each round. Upon completion, the results of the experiment will be recorded and can be analyzed to evaluate the performance of the FRL algorithm on the CIFAR10 dataset.

A.4 Evaluation workflow

A.4.1 Major Claims

These are the major claims made in our paper:

(C1): FRL can achieve similar performance as FedAvg, Trimmed-Mean and Multi-Krum on CIFAR10 and MNIST distributed over a large number of clients in a non-iid fashion when there is no malicious client, as illustrated in Table 1 in the paper.

(C2): FRL can achieve high test accuracy when 10% of the clients are malicious on CIFAR10 and MNIST distributed over a large number of clients in a non-iid fashion, as illustrated in Table 1 in the paper. FedAvg, Trimmed-Mean and Multi-Krum results in lower test accuracy.

(C3): FRL can achieve high test accuracy when 20% of the clients are malicious on CIFAR10 and MNIST distributed over a large number of clients in a non-iid fashion, as illustrated in Table 1 in the paper. FedAvg, Trimmed-Mean and Multi-Krum results in lower test accuracy.

A.4.2 Experiments

The experiments directory includes the experiments performed in the paper. This section explains the purpose of each experiment and the expected outcome.

(E1): *[FL with 0% malicious client] [30 human-minutes + 16 compute-hour]: For claim C1.*

Execution:

For CIFAR10:

- run FRL: **python main.py --data_loc /CIFAR10/data/ --config experiments/001_config_CIFAR10_Conv8_FRL_1000users_noniid1.0_nomalicious.txt**.
- We also provided more experiments for FedAVG, Trimmed-Mean and Multi-Krum in the experiments directory.

For MNIST:

- run FRL: **python main.py --data_loc /MNIST/data/ --config experiments/004_config_MNIST_LeNet_FRL_1000users_noniid1.0_nomalicious.txt**
- We also provided more experiments for FedAVG, Trimmed-Mean and Multi-Krum in the experiments directory.

Results: The results of the experiment will be stored in the Logs directory. For MNIST, the expected test accuracy should be around 98%, and for CIFAR10, the expected test accuracy should be around 85%.

(E2): *[FRL with 10% malicious client] [30 human-minutes + 16 compute-hour]: For claim C2.*

Execution: For CIFAR10, run python **main.py --data_loc /CIFAR10/data/ --config experiments/002_config_CIFAR10_Conv8_FRL_1000users_noniid1.0_10pmal.txt**. For MNIST, run **python main.py --data_loc /MNIST/data/ --config experiments/005_config_MNIST_LeNet_FRL_1000users_noniid1.0_10pmal.txt**.

Results: The results of the experiment will be stored in the Logs directory. For MNIST, the test accuracy should be around 98%, and for CIFAR10, the test accuracy should be around 79%.

(E1): *[FRL with 20% malicious client] [30 human-minutes + 16 compute-hour]: For claim C3.*

Execution: For CIFAR10, run **python main.py --data_loc /CIFAR10/data/ --config experiments/003_config_CIFAR10_Conv8_FRL_1000 users_noniid1.0_20pmal.txt**. For MNIST, run **python main.py --data_loc /MNIST/data/ --config experiments/006_config_MNIST_LeNet_FRL_1000 users_noniid1.0_20pmal.txt**.

Results: The results of the experiment will be stored in the Logs directory. For MNIST, the test accuracy should be around 98%, and for CIFAR10, the test accuracy should be around 69%.

It is important to note that the results may vary slightly due to the random initialization of the models and the random sampling of clients in each round of federated learning. Therefore, it is recommended to run the experiments multiple times and report the average of the results to obtain a more robust evaluation of the performance of the FRL algorithm.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Smart Learning to Find Dumb Contracts

Tamer Abdelaziz[†]
tamer@comp.nus.edu.sg
(†) National University of Singapore
Singapore

Aquinas Hobor[‡,†]
a.hobor@ucl.ac.uk
(‡) University College London
London, United Kingdom

A Artifact Appendix

A.1 Abstract

The artifact is an implementation of *Deep Learning Vulnerability Analyzer (DLVA)*, a vulnerability detection tool for Ethereum smart contracts based on powerful deep learning techniques for sequential data adapted for bytecode. We benchmark DLVA against nine competitors.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

N/A

A.2.2 How to access

Both DLVA and $DLVA_{large}$ are available as Docker images at https://hub.docker.com/u/dlva.

A.2.3 Hardware dependencies

The test machine is a desktop with a 12-core 3.2 GHz Intel(R) Core(TM) i7-8700 and 16 GB of memory.

A.2.4 Software dependencies

Ubuntu OS and Docker should be installed.

A.2.5 Benchmarks

In this artifact, we use three benchmarks to compare DLVA with the-state-of-the-art tools. $Elysium_{benchmark}$ https://bit.ly/Elysium_benchmark, $Reentrancy_{benchmark}$ https://bit.ly/Reentrancy_benchmark, $SolidiFI_{benchmark}$ https://bit.ly/SolidiFI_benchmark.

A.3 Set-up

A.3.1 Installation

The instructions to install DLVA as follows: Open the terminal and enter the following command to create a "dlva folder" in the home directory: `mkdir ~/dlva`

- To install DLVA (trained on SolidiFI's labels or on Slither's labels of small-length contracts):
  1. Pull the latest version of the DLVA Docker image by running the following command:
     `docker pull dlva/dlva:latest`
  2. Run the DLVA Docker container with the following command:
     `docker run -i -t -v ~/dlva/:/DLVA_Tool/dlva/ dlva/dlva`
  3. After executing the above commands, you will be inside the DLVA Docker container.
- To install $DLVA_{large}$ (trained on Slither's labels of large-length contracts):
  1. Pull the latest version of the $DLVA_{large}$ Docker image by running the following command:
     `docker pull dlva/dlva-large:latest`
  2. Run the $DLVA_{large}$ Docker container with the following command:
     `docker run -i -t -v ~/dlva/:/DLVA_Tool/dlva/ dlva/dlva-large`
  3. After executing the above commands, you will be inside the $DLVA_{large}$ Docker container.

A.3.2 Basic Test

Now, follow the command-line interface instructions to interact with DLVA Docker container.

1. Press 1 to run DLVA trained on SolidiFI labels, or press 2 to run DLVA trained on Slither labels (small contracts).
2. Enter 1 for a single contract mode, or 2 for a batch of contracts mode.
3. For a single contract mode: enter contract address *e.g.* `0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f` or insert the bytecode in "dlva/input.bin" file at "dlva folder" then enter: `b`

4. For batch mode: copy the dataset file (*e.g.*, batch.csv with "address" and "bytecode" columns) to the "dlva folder", then enter `../dlva/batch.csv` Alternatively, the user can use the provided test dataset by entering `Testset10.csv` The analysis results will be written in file named "DLVA_Predictions_batch.csv" at "dlva folder".

To test $\text{DLVA}_{\text{large}}$ Docker container follow the same aforementioned steps without step 1.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): DLVA performs well in practice for smart contract vulnerability detection: Figure 1 illustrates DLVA performance against nine competitors. DLVA is on the far right. We use bar-and-whiskers where star $\star$ represents the mean and plus $+$ represents outliers. Our average Completion Rate (*i.e.*, the percentage of contracts for which a tool produces an answer, the higher the better) is 100.0%. Our average accuracy is 99.7% (the higher the better), with a True Positive Rate (*i.e.*, detection rate; the higher the better) of 98.7% and a False Positive Rate (*i.e.*, false alarm rate; the lower the better) of 0.6%. Our average analysis time per contract (the graph is in log scale, lower better) is 0.2 seconds. Smart learning pays off: DLVA beats Slither on every statistic except for TPR (where it lags by 0.7%). Recall also that Slither requires source whereas DLVA needs only bytecode.
This is proven by the experiment (E1) described in §4.4 whose results are illustrated/reported in Figure 1, the data underlying Figure 1 is in Tables 4, 5, and 6.

A.4.2 Experiments

(E1): *30 human-minutes + machine with 12-core and 16 GB memory + 12 GB disk*
How to: Run the DLVA Docker container with the following command:
`docker run -i -t -v ~/dlva/:/DLVA_Tool/dlva/ dlva/dlva`
Preparation: After executing the previous command, the three benchmarks A.2.5 will be downloaded automatically and saved to the "dlva folder".
Execution: Run DLVA on the three benchmarks:

1. For $\text{Elysium}_{\text{benchmark}}$:
   (a) Press 2 to select DLVA trained on Slither labels (small contracts).
   (b) then, press 2 to select the batch of contracts mode.
   (c) then, enter: `../dlva/Elysium_benchmark.csv`
   (d) then, wait until the analysis is complete, the raw results will be stored at "dlva/DLVA_Predictions_for_Elysium_benchmark.csv"
2. For $\text{Reentrancy}_{\text{benchmark}}$:
   (a) Press 2 to select DLVA trained on Slither labels (small contracts).
   (b) then, press 2 to select the batch of contracts mode.
   (c) then, enter: `../dlva/Reentrancy_benchmark.csv`
   (d) then, wait until the analysis is complete, the raw results will be stored at "dlva/DLVA_Predictions_for_Reentrancy_benchmark.csv"
3. For $\text{SolidiFI}_{\text{benchmark}}$:
   (a) Press 1 to select DLVA trained on SolidiFI labels.
   (b) then, press 2 to select the batch of contracts mode.
   (c) then, enter: `../dlva/SolidiFI_benchmark.csv`
   (d) then, wait until the analysis is complete, the raw results will be stored at "dlva/DLVA_Predictions_for_SolidiFI_benchmark.csv"
4. Press 0 to exit from the DLVA Docker image.

Results: Enter: `cd ~/dlva`
then enter: `pip3 install -r requirements.txt`
then: `python3 print_dlva_results.py` that will show the DLVA results on the three benchmarks.
then: `python3 print_competitors_results.py` that will show the competitors results on the three benchmarks.
Open the "dlva folder", seven files have been added: "tools_results.txt" contains the log performance for all tools based on raw data in "10_tools_files" folder, "tools_results.csv" represents the same results as a spreadsheet, and five files of "tools_predictions_benchmark.csv" contain labels of all tools for each benchmark.
The FN and FP produced numbers in this experiment should match the numbers in Tables 4, 5, and 6.
Any number in Figure 1 is the average for each tool using the three benchmarks.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix
Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge

Nils Bars*, Moritz Schloegel*, Tobias Scharnowski*
Nico Schiller*, Thorsten Holz‡

*Ruhr-Universität Bochum
‡CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

FUZZTRUCTION's artifact contains the source code necessary to run our fuzzer (as well as competing fuzzers). This document describes how to set-up our fuzzer prototype, gives a brief overview of the resource requirements to replicate the experiment (i. e., a coverage comparison with other state-of-the-art fuzzers) conducted in our paper, and contains instructions for reproducing our results.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact is shipped via a Docker image used to spawn a unified containerized environment to ease evaluation and development independent of the underlying system. The container's life cycle is managed by the `env/start.sh` script which by default forwards the `ssh-agent` (if any) and the `.gitconfig` into the container. If this is undesired behavior, the related functionality should be removed from the script before conducting any experiments.

A.2.2 How to access

The artifact's source code is accessible at https://github.com/fuzztruction/fuzztruction/tree/91ba684d2b8fa21ae19e403496b507f3729c4ff5. The repository and sub repositories contain extensive documentation to make the artifact evaluation process as easy as possible.

A.2.3 Hardware dependencies

For evaluation, we used two Intel(R) Xeon(R) Gold 6230R CPUs, totaling 52 cores, 128 GB RAM, and about 1 TB SSD disk space. Since some targets produce many test cases, which are stored in `/tmp`, i. e., in RAM, we advise resizing the `tmp` folder to 600 GiB and backing the amount exceeding the RAM capacity via a swap file. The evaluation script will walk you through these steps.

In our paper, we evaluated 12 targets, which we run five times for 24 hours, and assigned all 52 cores to one experiment. Consequently, a vast amount of computational power is required to replicate the exact experiments conducted in our paper. We believe this computational power is out-of-scope for artifact review, thus we provide instructions on how to approximate our experiments using significantly less resources in Section A.4.2.

A.2.4 Software dependencies

For running the artifact, a working Docker installation is required. All scripts that must be executed on the host system (i. e., outside of the container) have been tested exclusively on Ubuntu 22.04. However, since the scripts are rather simple, they should work on any Linux distribution.

A.2.5 Benchmarks

All data required for the evaluation is part of the linked repository.

A.3 Set-up

The set-up is explained in detail in the main repository's `README.md`. For your convenience, we provide pre-built versions of FUZZTRUCTION. We recommend using these pre-built versions of FUZZTRUCTION since slight changes in, for example, libraries linked into a fuzzing target might cause deviations from the results presented in the paper. Furthermore, compiling all targets takes considerable time, because we use AFL++'s collision free encoding, which causes link time to increase significantly. Overall, the compilation of all targets takes multiple hours.

A.3.1 Basic Test

Testing the set-up is possible using the steps provided in the *Fuzzing a Target using Fuzztruction* section in the `README.md`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We demonstrate the generic capabilities of our approach by fuzzing even complex cryptographic procedures, such as the parsing and validation of encrypted RSA keys, automatically and without custom-crafted seeds.

(C2): We implement and evaluate our prototype, called FUZZTRUCTION, against the state-of-the-art fuzzers AFL++, SYMCC, and WEIZZ. Our results show that our approach achieves significant gains in terms of coverage and number of software faults found.

Please note that claim C1 is a subset of C2, since outperforming all other competitors on a cryptographic target also indirectly shows our approach's applicability to complex cryptographic targets.

A.4.2 Experiments

As described in the requirements section, we conducted an extensive evaluation requiring considerable CPU time for reproduction. Since comparing each individual result from the paper with an experiment using fewer resources is impossible, we suggest concentrating the evaluation efforts on a subset of the fuzzing targets.

According to our statistical analysis presented in Table 2 in our paper, the three targets `objdump`, `readelf` and `unzip` show no statically significant difference between the evaluated fuzzer configurations. Thus, we advise excluding these targets from the reproduction since they are no proxy for our claims. For the remaining targets, we advise only considering the best competitor (cf. Table 2) to further reduce the amount of CPU time required.

Following our recommendations, the artifact evaluation effort is composed of 52 CPUs * 9 targets * 2 fuzzers (FUZZTRUCTION, best competitor) * 24h, which equates to running a single 52 CPU machine for 18 days. If desired, the list of targets might be further reduced by skipping targets not employing cryptographic primitives (i. e.,, targets in Table 2 that are not marked with a lock) since targets using cryptographic primitives are required to support both our claims. For example, if fuzzing only 3 targets (e. g., `rsa`, `vfychain`, and `7zip-enc`) with 2 fuzzers (FUZZTRUCTION, best competitor) for 24 hours, your fuzzing run will take 6 days on a single 52 CPU machine.

Notably, since fuzzing is inherently non-deterministic, a single run per target does not necessarily exactly align with the results presented in the paper. Consequently, reducing the number of tested targets in favor of doing multiple runs for some targets might be desirable. Certainly, this trade-off is primarily driven by the available hardware resources.

As a result of this experiment, we expect a plot similar to Figure 3 in our paper. All steps required to run the experiment, and to plot the data, are explained in the documentation found in the artifact's git repository. Please mind that some of the targets are not supported by SYMCC.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: <FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler>

Junjie Wang[1*], Zhiyi Zhang[2*], Shuang Liu[1+], Xiaoning Du[3], and Junjie Chen[1]

[1]College of Intelligence and Computing, Tianjin University
[2]CodeSafe Team, Qi An Xin Group Corp.
[3]Monash University

A Artifact Appendix

This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of FuzzJIT.

A.1 Abstract

FuzzJIT is a fuzzing tool for JavaScript engines JIT compiler, built on top of Fuzzilli [1]. FuzzJIT maintains a queue that contains all samples that triggered new code coverage in the testing subjects. At the start of each fuzzing round, FuzzJIT selects a test case from the queue and mutates it to generate new test cases. Generated new test cases are executed, and the test cases that triggered new code coverage are then added to the fuzzing queue for further mutation. Our main contributions include a for-loop structure to trigger the JIT compilers, a test function embedding JIT compiler bugs revealing elements, and an enhanced oracle to check if the test function output differently before/after the JIT compilation.

A.2 Description & Requirements

In this section, we list the information necessary to recreate the same experimental setup we have used to run our artifact. We also list the hardware and software requirements to run our artifact. At last, we list benchmarks used to produce results with our artifact.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

FuzzJIT can be obtained from GitHub: https://github.com/SpaceNaN/fuzzjit/commit/a3d3f6da7f7f8577476892d6135eee6c50afc7ad.

A.2.3 Hardware dependencies

In our experiments, we used a workstation with a processor of 12th Gen Intel Core i9-12900K*24 and 32 GB memory. Any similar configuration should work too.

A.2.4 Software dependencies

In our experiment, FuzzJIT runs on a 64-bit Ubuntu 22.04.01 LTS system. Other Linux operating systems should work too. The same as Fuzzilli, FuzzJIT is written in Swift, therefore, the installation of Swift is required. Swift 5.7 and 5.3 are tested working.

A.2.5 Benchmarks

Our testing subjects include four mainstream JavaScript engines, JavaScriptCore (the JavaScript engine of the Safari browser), V8 (the JavaScript engine of the Chrome browser), Spidermonkey (the JavaScript engine of Firefox), and ChakraCore (the JavaScript engine of Edge). In our evaluation, we compared FuzzJIT with four baselines, including Jsfunfuzz [3], Superion [4], DIE [2], and Fuzzilli [3].

A.3 Set-up

In this section, we list the installation and configuration steps required to prepare the environment to be used for the evaluation of our artifact.

A.3.1 Installation

The running procedure of FuzzJIT is the same with Fuzzilli.

1. Download Swift from its download page: https://www.swift.org/download/, for example:

```
wget https://download.swift.org/swift-5.7-release/
    ubuntu2204-aarch64/swift-5.7-RELEASE/swift-5.7-
    RELEASE-ubuntu22.04-aarch64.tar.gz
```

2. Uncompress the downloaded file.

```
tar zxvf ./ swift-5.7-RELEASE-ubuntu22.04-aarch64.tar.gz
```

3. Export the path of Swift to an environment variable.

```
export PATH=~/swift-5.7-RELEASE-ubuntu22.04-aarch64/usr/
    bin:\${PATH}
```

4. Download FuzzJIT from GitHub.

```
git clone https :// github .com/SpaceNaN/fuzzjit
```

5. Compile the FuzzJIT.

```
swift build [-c release ]
```

A.3.2 Basic Test

The user can run the following command to see if FuzzJIT works.

```
swift run FuzzilliCli --help
```

A.4 Evaluation workflow

We list the operational steps and experiments to evaluate FuzzJIT.

A.4.1 Major Claims

Our paper makes four major claims.

C1: FuzzJIT can be used to uncover new bugs in the JavaScriptCore/V8/Spidermonkey/ChakraCore. This is proven by the experiments (E1).

C2: FuzzJIT can achieve better code coverage growth than baselines. This is proven by the experiments (E2).

C3: FuzzJIT can achieve better syntax correctness rate than baselines. This is proven by the experiments (E3).

C4: FuzzJIT can achieve relatively good throughput than baselines. This is proven by the experiments (E4).

A.4.2 Experiments

E1: Finding bugs in testing subjects. One week of fuzzing should work:

How to: Fuzzing given targets for about one week or longer to see any crashes triggered.

Preparation and execution: To fuzz JavaScriptCore with FuzzJIT:

1. Download JavaScriptCore.

```
git clone https :// github .com/WebKit/webkit
```

2. Apply Targets/JavaScriptCore/Patches/*. This step will be a little bit tricky. When the version does not match, the user needs to manually apply the patch.

3. Run the Targets/JavaScriptCore/fuzzbuild.sh script in the WebKit root directory.
4. FuzzBuild/Debug/bin/jsc will be the JavaScript shell for the fuzzer.
5. Fuzz JavaScriptCore.

```
swift run -c release FuzzilliCli --profile=jsc --
    timeout=500 --storagePath=./jsc/ /path/to/webkit/
    FuzzBuild/Debug/bin/jsc
```

To fuzz V8 with FuzzJIT.
1. First download depot_tools.

```
git clone https :// chromium.googlesource.com/chromium/
    tools/depot_tools . git
```

2. Export depot_tools to an environment variable.

```
export PATH=\$PATH:/path/to/depot_tools
```

3. Configure gclient.

```
mkdir v8
cd v8
gclient config https :// chromium.googlesource.com/v8/
    v8
```

4. Synchronize V8's source code.

```
gclient sync
```

5. Run the Targets/V8/fuzzbuild.sh script in the v8 root directory.
6. out/fuzzbuild/d8 will be the JavaScript shell for the fuzzer.
7. Fuzz V8.

```
swift run -c release FuzzilliCli --profile=v8 --
    timeout=500 --storagePath=./v8/ /path/to/v8/out/
    fuzzbuild/d8
```

To fuzz Spidermonkey with FuzzJIT.
1. Download Spidermonkey source code.

```
git clone https :// github .com/mozilla/gecko-dev
```

2. Apply Targets/Spidermonkey/Patches/*. This step will be a little bit tricky. When the version does not match, the user needs to manually apply the patch.
3. Run the Targets/Spidermonkey/fuzzbuild.sh script in the js/src directory of the Firefox checkout.
4. ./fuzzbuild_OPT.OBJ/dist/bin/js will be the JavaScript shell for the fuzzer.
5. Fuzz Spidermonkey.

```
swift run -c release FuzzilliCli --profile=
    spidermonkey --timeout=500 --storagePath=./
    spidermonkey/ /path/to/gecko-dev/js/src/
    fuzzbuild_OPT.OBJ/dist/bin/js
```

To fuzz ChakraCore with FuzzJIT.
1. Download ChakraCore source code.

```
git clone https :// github .com/chakra–core/ChakraCore
```

2. Apply Targets/ChakraCore/Patches/*. This step will be a little bit tricky. When the version does not match, the user needs to manually apply the patch.
3. Run the Targets/ChakraCore/fuzzbuild.sh script in the ChakraCore directory.
4. FuzzBuild/Debug/ch will be the JavaScript shell for the fuzzer.
5. Fuzz ChakraCore.

```
swift run –c release FuzzilliCli ––profile=chakracore
    ––timeout=500 ––storagePath=./chakracore/ /path/
    to/chakracore/FuzzBuild/Debug/ch
```

Results: Fuzzing is a random procedure, but enough time of fuzzing should reproduce the crashes.

E2: Evaluating code coverage. One week of fuzzing should work:
How to: FuzzJIT/Fuzzilli update the fuzzing status per minute, as shown in Figure 1. We can read the code coverage information from the "Coverage:" row of the FuzzJIT/Fuzzilli interface after one week of fuzzing. For Jsfunfuzz, we fail to obtain its coverage information. For Superion/DIE, which are AFL based, we can also read the coverage information from the "map density" row from their interface.

E3: Evaluating syntax correctness rate. One week of fuzzing should work:
How to: FuzzJIT/Fuzzilli update the fuzzing status per minute. We can read the sample syntax correctness rate information from the "Correctness Rate:" row of FuzzJIT interface after one week of fuzzing. For Jsfunfuzz/Superion/DIE, we provide a Python script to calculate the syntax correctness rate, which is at `/path/to/FuzzJIT/script/calculate_syntax_error.py`.

E4: Evaluating throughput. One week of fuzzing should work:
How to: FuzzJIT/Fuzzilli update the fuzzing status per minute. We can read the throughput information from the "Total Execs:" row of FuzzJIT interface after one week of fuzzing. For Jsfunfuzz, its throughput is determined by its timeout threshold since almost all samples can not be finished in given time. For Superion/DIE, we can read its throughput information from its "total execs:" row of their interface.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220912. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenix%20sec2023/.

```
Fuzzer Statistics
-----------------
Total Samples:              45
Interesting Samples Found:  41
Valid Samples Found:        42
Corpus Size:                42
Correctness Rate:           93.33%
Timeout Rate:               0.00%
Crashes Found:              0
Timeouts Hit:               0
Coverage:                   9.54%
Avg. program size:          54.11
Connected workers:          0
Execs / Second:             2.28
Fuzzer Overhead:            0.82%
Total Execs:                938
```

Figure 1: The interface of FuzzJIT.

References

[1] Samuel Groß. Fuzzil: Coverage guided fuzzing for javascript engines. *Department of Informatics, Karlsruhe Institute of Technology*, 2018.

[2] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020.

[3] Jesse Ruderman. Fuzzing tracemonkey. https://www.squarefree.com/2008/12/23/fuzzing-tracemonkey/.

[4] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

USENIX'23 Artifact Appendix: `autofz`: Automated Fuzzer Composition at Runtime

Yu-Fu Fu Jaehyuk Lee Taesoo Kim

Georgia Institute of Technology

A Artifact Appendix

A.1 Abstract

`autofz`'s artifact contains the source code and all the benchmarks used in the evaluation section of the paper. This artifact appendix is used to outline the steps to retrieve the artifact and how to use it to reproduce the experiments. Furthermore, we provide the instructions to extend the framework (i.e. add ad new fuzzer or a new benchmark to `autofz`).

A.2 Description & Requirements

The artifact contains the following components.

1. `autofz` source code
2. A pre-built docker image containing `autofz`, fuzzers, and benchmarks.
3. A VM image which includes all the necessary changes to the host environment and can be used to launch the aforementioned docker image.

A.2.1 Security, privacy, and ethical concerns

During fuzzing, we modify some kernel parameters which docker shares with the host. For example, we enable core dump and disable ASLR for the whole system. Therefore, we recommend that running `autofz` inside a VM.

A.2.2 How to access

1. Source code https://github.com/sslab-gatech/autofz
2. Source code with commit hash
`https://github.com/sslab-gatech/autofz/tree/b9a795dda252aa37406d593434b710b0fbedd177`
3. Docker image: https://hub.docker.com/r/fuyu0425/autofz with SHA256 digest f39fb70af5db and tag v1.0.1.
4. VM image: https://doi.org/10.5281/zenodo.7865366

A.2.3 Hardware dependencies

During the evaluation, we use a cluster of Ubuntu 20.04 machines equipped with AMD Ryzen 9 3900 (12C/24T), 32 GB RAM, and 512 GB SSD disk space. To use the provided docker image or VM image, 30 GB disk space is required.

A.2.4 Software dependencies

To use the docker image, a working Docker/Podman under Linux is required. Alternatively, to use the VM image, VirtualBox/VMware is required.

A.2.5 Benchmarks

All benchmarks required for evaluation are already in the docker image.

A.3 Set-up

We provide a detailed set-up process in `README.md` in the provided GitHub repository. However, building all fuzzers and benchmarks will takes a lot of time and resource. Therefore, we recommend using either the pre-built docker image or the VM image (preferred).

A.3.1 cgroup v2 downgrade

`autofz` uses cgroup v1; therefore, a manual downgrade from v2 to v1 might be required in newer operating systems. This can be done by adding "systemd.unified_cgroup_hierarchy=0" to the kernel command line (e.g. via "/etc/default/grub").

A.3.2 Basic Test

To make sure `autofz` is installed successfully, type the following command:

```
autofz --help
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We demonstrate that `autofz` can achieve better coverage against different target binaries compared with

individual fuzzers (Figure 3 in the paper). The result is supported by E1.

(C2): We demonstrate that `autofz` can achieve better coverage against different target binaries compared with other collaborative fuzzing techniques ENFUZZ and CUPID (Figure 7 in the paper). The result is supported by E2.

A.4.2 Experiments

Setup. To execute the experiments, we need to pull docker images and launch a docker container by the following commands.

```
docker pull fuyu0425/autofz:v1.0.1
docker tag fuyu0425/autofz:v1.0.1 autofz
```

```
docker run --rm --privileged -it autofz
/bin/bash.
```

Note that, the result is not preserved after exiting the container. To preserve the fuzzing output, we need to mount a docker volume.

```
docker run --rm --privileged -v
$PWD:/work/autofz -w /work/autofz -it
autofz /bin/bash.
```

After entering the docker, we need to tune the necessary kernel parameters and create a cgroup for `autofz`; we pack all commands in a script `/init.sh` and can be executed by the following command. Note the security concern mentioned in §A.2.1.

```
sudo /init.sh
```

More detail is in the *running* section of `README.md`.

(E1): [`autofz` v.s. individual fuzzers] [32000 compute-hours + 200 GB disk]: Generate the 24-hour fuzzing output of `autofz` and individual fuzzers on 12 benchmark programs for 10 repetitions for Figure 3 in the paper.
How to: Use `autofz` with different command line arguments to run all the fuzzing. `README.md` in the repository has more information about the arguments.
Execution: To run `autofz` on a target (e.g. `exiv2`), use the following command:

```
autofz -o output-exiv2-autofz -t exiv2
-T 24h -f all
```

To run a individual fuzzer (e.g. AFL) on a target (e.g. `exiv2`), use the following command:

```
autofz -o output-exiv2-afl -t exiv2 -T
24h -f afl --focus-one afl
```

Output directory specified by `-o` needs to be different for each fuzzing repetition.
Results: For each fuzzing run, `autofz` will generate a log file in JSON format, which includes all the coverage and the number unique bugs information.
Additionally, there is a directory called `eval` in the fuzzer output directory. The directory stores the results of crash deduplication and ASAN output of each crash.

(E2): [`autofz` v.s. ENFUZZ/CUPID] [7680 compute-hours + 200 GB disk]: Generate the 24-CPU-hour fuzzing output of `autofz`-10, `autofz`-6, CUPID-4, and ENFUZZ-6 on 8 benchmark programs for 10 repetitions for Figure 7 in the paper.
Execution: To run `autofz`-10 on a target (e.g. `exiv2`), use the following command:

```
autofz -o output-exiv2-autofz10 -t
exiv2 -T 24h -f all -j10 --parallel
```

To run `autofz`-6 on a target (e.g. `exiv2`), use the following command:

```
autofz -o output-exiv2-autofz6
-t exiv2 -T 24h -f afl fairfuzz
qsym aflfast lafintel radamsa -j6
--parallel
```

To run CUPID-4 (ENFUZZ-Q) on a target (e.g. `exiv2`), use the following command:

```
autofz -o output-exiv2-cupid4 -t exiv2
-T 24h -f afl fairfuzz qsym aflfast
--enfuzz 300 -j4 --parallel
```

Results: Note that, in the paper, we draw the graph based on **CPU hours**. Therefore, if we use 10 CPU cores (by specifying `-j 10`), only the first 2.4 hours is draw on the graph.

Because both experiments take enormous resource to replicate, we recommend choosing only a subset of benchmarks. Please note that fuzzing is an inherently a random process; therefore, the reproduced result might not be the same as we have reported in the paper. To alleviate this problem, we recommend increasing the fuzzing repetition (e.g. 10 times as we did) and similar results are expected.

A.4.3 Inspect log files of `autofz`

The log file of `autofz` is in JSON format and can be easily parsed by standard libraries in most programming languages. To inspect the log file (e.g. exiv2.json), we recommend using

a tool called `jq` (https://github.com/stedolan/jq), which can be installed by the package manager in most Linux distributions. We already installed it in both the docker image and the VM image.

There are many fields in the log file. One of them is "log", which can be retrieved by the following command.

```
jq .log exiv2.json
```

The output is an array and each element of the array contains the coverage ("bitmap" field) and unique bugs information and the timestamp for that record. By default, a new log entry is appended for every 60 seconds.

To get the results based on rounds, we can use the following commands.

```
jq .round exiv2.json
```

The output is also an array and each element is the result of one round. Each element records information for different phases in one round (e.g. the coverage before/after preparation/focus phases, resource allocation metadata and current difference threshold θ.)

In the provided VM, we provided one of the fuzzing log with the path
`/home/autofz/output_exiv2/exiv2.json`

A.4.4 Plotting the figures

We also include the scripts to draw the figures used in the paper.

```
autofz-draw -o output-draw -t exiv2 -d
exp -T 24h -pdf
```

Above commands is used to draw figure 3 in the paper but only for `exiv2`.

We have more detailed explanation of each argument in the provided repository.

Note that for timeout parameter `-T`, it specifies **CPU Time**; therefore, for Figure 7, `-T 3h` is enough if you use 10 CPU cores.

A.5 Notes on Reusability

We have included instructions to extend `autofz` (add new fuzzers or new benchmarks) in the provided repository.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing

Dawei Wang[1,2], Ying Li[1,2], Zhiyu Zhang[1,2], and Kai Chen[1,2,3*]

[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Beijing Academy of Artificial Intelligence, China
{wangdawei,liying1998,zhangzhiyu1999,chenkai}@iie.ac.cn

A Artifact Appendix

A.1 Abstract

CarpetFuzz is an NLP-based fuzzing assistance tool specifically designed for extracting constraint relationships between command-line options from documents. Our evaluation of CarpetFuzz involved a comprehensive analysis comprising an end-to-end experiment, a comparative experiment, and four submodule experiments. To facilitate the setup process, we provide a Dockerfile, which helps mitigate potential issues with environment configuration. Additionally, we offer a collection of scripts that automate experiment reproduction and effectively showcase the results obtained.

Given the nature of fuzzing-related work, reproducing the experiments conducted with CarpetFuzz necessitates a substantial amount of computational resources. Replicating all the experiments outlined in the paper requires a total of 33,600 CPU hours (across 5 repetitions). Simplifying the process would still require a minimum of 15,840 CPU hours. Consequently, we recommend employing a server with at least 32 cores to carry out these experiments, which would approximately take around 20.6 days. It's worth noting that having a higher number of cores would further enhance the efficiency of the experiments.

A.2 Description & Requirements

Our paper describes a novel technique for identifying and extracting constraints among program options from the documentation. Our artifact is a prototype of our technique named CarpetFuzz, which contains the models, fuzzers, run scripts, and documentation. We also provide a comprehensive collection of samples, run scripts, and documentation to replicate the experiments outlined in our paper with ease.

*Corresponding author.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is provided as a GitHub repository: `https://github.com/waugustus/CarpetFuzz/commit/50f09eb94d33abbfe3e18184988a0c3a8f0f5612`.

A.2.3 Hardware dependencies

As a fuzzing-related work, reproducing the experiments necessitates a significant allocation of computational resources, ranging from 15,840 to 33,600 CPU hours. To ensure completion within the review process timeframe, we recommend utilizing a server with a minimum of 32 cores, which would require approximately 20.6 days. For enhanced fault tolerance and expediency, we strongly advise opting for a server with a higher core count. In terms of hard disk capacity, our Docker image occupies around 20GB of disk space, so a disk capacity of 50GB is more than sufficient.

For the sole purpose of running CarpetFuzz, we believe that mainstream computers available on the market are sufficient to meet the requirements, such as computers with a 1-core CPU, 8GB RAM, and a 128GB hard drive.

A.2.4 Software dependencies

All software dependencies have been successfully resolved within our provided Dockerfile which is based on Ubuntu 20.04. Therefore, any system capable of running this image is suitable for the task.

A.2.5 Benchmarks

Our benchmark includes a total of 50 executable programs, with 20 sourced from our real-world program dataset and 30 obtained from the POWER dataset. All of these programs

can be readily acquired from the internet. The process of obtaining and compiling each program has been thoroughly documented in our Dockerfile, facilitating automated building using the "docker build" command.

A.3 Set-up

Clone the artifact repository:

```
$ git clone --recursive https://github.com/waugustus/CarpetFuzz; cd CarpetFuzz
```

A.3.1 Installation

For easy installation, we offer a ready-to-use Docker image for download,

```
$ sudo docker pull 4ugustus/carpetfuzz
```

or you can compile the image yourself using the Dockerfile we provide.

```
$ sudo docker build -t 4ugustus/carpetfuzz:latest .
```

Then you can create the container based on the image,

```
$ sudo docker run -it --name "carpetfuzz" 4ugustus/carpetfuzz:latest bash
```

A.3.2 Basic Test

We take the program *"tiffcp"* as an example (in the contatiner),

1. Use CarpetFuzz to analyze the relationships from the manpage file:

   ```
   $ cd /root/programs/libtiff
   ```

   ```
   $ python3 ${CarpetFuzz}/scripts/find_relationship.py --file $PWD/build_carpetfuzz/share/man/man1/tiffcp.1
   ```

2. Use pict to generate 6-wise combinations:

   ```
   $ python3 ${CarpetFuzz}/scripts/generate_combination.py --relation ${CarpetFuzz}/output/relation/relation_tiffcp.json
   ```

3. Rank each combination with its dry-run coverage:

   ```
   $ python3 ${CarpetFuzz}/scripts/rank_combination.py --combination ${CarpetFuzz}/output/combination/combination_tiffcp.txt
   --dict ${CarpetFuzz}/tests/dict/dict.json
   --bindir $PWD/build_carpetfuzz/bin
   --seeddir input
   ```

4. Fuzz with the ranked stubs:

   ```
   $ ${CarpetFuzz}/fuzzer/afl-fuzz -i input/ -o output/ -K ${CarpetFuzz}/output/stubs/ranked_stubs_tiffcp.txt -- $PWD/build_carpetfuzz/bin/tiffcp @@
   ```

A.4 Evaluation workflow

In our paper, we evaluated CarpetFuzz through a total of six experiments, including an end-to-end experiment, a comparative experiment, and four submodule experiments, which collectively required 33,600 CPU hours. Please note that due to the authors of POWER **declining** our request to use their tool during the review process, the comparative experiment was deemed unnecessary (RQ5), resulting in a reduction of 7,200 CPU hours. Furthermore, all experiments in the paper were repeated five times. However, for simplification purposes, we consider three repetitions to be acceptable, resulting in a reduction of 10,560 CPU hours. As a result, the minimum required time for the experiments is 15,840 CPU hours.

If you desire to replicate the experiments in the paper comprehensively (excluding running POWER), you can execute "`$ ./run_fuzzing.sh`" without any options in A.4.1. This will trigger CarpetFuzz to perform fuzzing on the entire benchmark and repeat the process five times, thus amounting to a total runtime of 30,000 CPU hours.

A.4.1 Preprocessing

1. Build docker image for experiments *[1 human-minute + 6 compute-hours + 20GB disk]*:

   ```
   $ git clone https://github.com/waugustus/CarpetFuzz-experiments
   ```

   ```
   $ sudo docker build -t carpetfuzz-experiment:latest .
   ```

   ```
   $ sudo docker run -d --name "carpetfuzz-experiment" carpetfuzz-experiment:latest tail -f /dev/null
   ```

   ```
   $ sudo docker exec -it carpetfuzz-experiment bash
   ```

2. Start all fuzzing instances *[1 human-minute + 15,840 CPU-hours + 10GB disk]*:

   ```
   $ screen -dmS fuzzing bash -c "./run_fuzzing.sh -r 3 2>&1 |tee fuzzing.log"
   ```

3. Analyze the documents from the compiled programs and generate the results of relationship identification and extraction *[1 human-minute + 10 computer-minutes + 1GB disk]*:

   ```
   $ screen -dmS analyze python3 analyze_manpages.py 2>&1 | tee analyze.log
   ```

A.4.2 Major Claims

(C1): Compared to aflfast, mopt, afl++, CarpetFuzz can help afl find more uncovered edges. This is proven by the

experiment (E1) described in Section 5.1 whose results are illustrated in Table 1.

(C2): For explicitly declared relationships, CarpetFuzz achieves an accuracy of 92.90% on the validation set and 98.80% on the documentation of the 20 programs. For implicitly declared relationships, CarpetFuzz achieves an precision of 95.87% and a recall of 90.09%. This is proven by the experiments (E2) described in Section 5.2.

(C3): The precision and recall of conflict were 95.83% and 89.40%, and those of dependency were 100% and 81.82%. The precision and recall of all relationships were 96.10% and 88.85%. This is proven by the experiments (E3) described in Section 5.3.

(C4): With our prioritization technique, CarpetFuzz found more edges on each program that other fuzzers could not discover. This is proven by the experiments (E4) described in Section 5.4 whose results are illustrated in Table 2.

(C5): CarpetFuzz can discover real-world vulnerabilities. This is proven by the experiments (E4) described in Section 5.6 whose results are illustrated in Table 4.

A.4.3 Experiments

(E1): *[5 human-minutes + 1 compute-hour]: This experiment will measure the edge coverage for all Fuzzing instances and present the results in the format shown in Table 1.*

How to: First, run `get_stubs.py` in `experiments/RQ1/scripts` to collect all testcases. Second, run `afl-showmap.py` to obtain the coverage data. Third, run `analyze_results.py` to present the results.

Preparation: The preprocessing step in A.4.1 is required to have fuzzing results.

Execution: Execute the following commands:

```
$ cd experiments/RQ1/scripts
$ python3 get_stubs.py
$ python3 afl-showmap.py
$ python3 analyze_results.py
```

Results: The ouput should match Table 1 of the paper.

(E2): *[5 human-minutes + 5 compute-minutes]: This experiment will measure the relationship identification performance of CarpetFuzz on the validation set and the documentation of the 20 programs.*

How to: Run `analyze_results.py` in `experiments/RQ2/scripts` to obtain the results.

Preparation: The preprocessing step in A.4.1 is required to obtain prediction results.

Execution: Execute the following commands:

```
$ cd experiments/RQ2/scripts
$ python3 analyze_results.py
```

Results: The ouput should match Section 5.2 of the paper.

(E3): *[5 human-minutes + 5 compute-minutes]: This experiment will measure the relationship extraction performance of CarpetFuzz.*

How to: Run `analyze_results.py` in `experiments/RQ3/scripts` to obtain the results.

Preparation: The preprocessing step in A.4.1 is required to obtain extraction results.

Execution: Execute the following commands:

```
$ cd experiments/RQ3/scripts
$ python3 analyze_results.py
```

Results: The ouput should match Section 5.3 of the paper.

(E4): *[5 human-minutes + 1 compute-hour]: This experiment will measure the edge coverage for CarpetFuzz-random instances and present the results in the format shown in Table 2.*

How to: First, run `get_stubs.py` in `experiments/RQ4/scripts` to collect all testcases. Second, run `afl-showmap.py` to obtain the coverage data. Third, run `analyze_results.py` to present the results.

Preparation: The preprocessing step in A.4.1 is required to have fuzzing results.

Execution: Execute the following commands:

```
$ cd experiments/RQ4/scripts
$ python3 get_stubs.py
$ python3 afl-showmap.py
$ python3 analyze_results.py
```

Results: The ouput should match Table 2 of the paper.

(E5): *[5 human-minutes + 1 compute-hour]: This experiment will tally the number of crashes encountered by CarpetFuzz instances.*

How to: Run `analyze_results.py` in `experiments/RQ6/scripts` to tally the number of crashes.

Preparation: The preprocessing step in A.4.1 is required to have fuzzing results.

Execution: Execute the following commands:

```
$ cd experiments/RQ6/scripts
$ python3 analyze_results.py
```

Results: CarpetFuzz should find multiple crashes in the 20 programs.

A.5 Notes on Reusability

A.5.1 How to find the manpage file of a new program?

In our experience, manpage files are typically located in the share directory within the compilation directory, such as `/your_build_dir/share/man/man1`.

A.5.2 How to fuzz the target not recorded in dict.json?

Unfortunately, as mentioned in the paper, CarpetFuzz does not currently support the automatic selection of appropriate option values from the document. To fuzz a new program, you'll need to read the manpage and manually add the synop-

sis and option-value pairs in the JSON, which may not be too time-consuming.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: The Gates of Time: Improving Cache Attacks with Transient Execution

Daniel Katzman ⚘, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen ⚘, Yuval Yarom

⚘ Tel-Aviv University
The University of Adelaide
The University of Melbourne

A Artifact Appendix

A.1 Abstract

We implement a speculative execution attack that improves cache attacks by means of amplifying cache states. In addition to that, the construct that we use to perform this amplification technique is also capable to facilitate robust computation on cache states. This project is available at https://github.com/0xADE1A1DE/GoT. In this Artifact Evaluation, we are applying for:

- "Artifact Available" badge
- "Artifact Functional" badge
- "Results Reproduced" badge

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

A.2.2 How to access

Artifact can be accessed here: https://github.com/0xADE1A1DE/GoT/commit/5bac7ece92f61025b7c1942c59b95e8a999ec231

A.2.3 Hardware dependencies

Amplification, Eviction Set, and Prime + Store Attack
Dynabook TECRA A50- EC, with an Intel(R) Core(TM) i5-8250U CPU
Trace Processing and Stitching
A High Performance Computing (HPC) node with 80 threads and AVX-2 instruction-set support.
Circuit Evaluation
Intel(R) Core(TM) i7-9750H with cores 0,1,6,7 isolated.

A.2.4 Software dependencies

Amplification, Eviction Set, and Prime + Store Attack

Ubuntu 20.04.3 LTS, Chromium commit hash (be87c21d2a7069363dfd66f278739d7e4211145e), emscripten.
Trace Processing and Stitching
A reasonably modern HPC node with Linux.
Circuit Evaluation
Ubuntu 21.04, huge-pages enabled, gnuplot.

A.2.5 Benchmarks

Amplification, Eviction Set, and Prime + Store Attack
None
Trace Processing and Stitching

- **Dataset:** The dataset consists of several components, including the raw traces obtained from the Prime+Store attack on ElGamal, the true key (referred to as the "ground truth") of the ElGamal encryption algorithm, and the frequency analysis of these traces. The latter serves as a guide for selecting the optimal starting trace for key expansion.
- **Location:** The dataset is located in the artifacts repository under the directory: "trace_processing_and_stitching/DATASETS"

Circuit Evaluation

- **Dataset:** The dataset contains raw circuit accuracy data which provides guidance on how to create the figures. Furthermore, scripts that can be used to generate figures that match those in the paper are also included in the dataset.
- **Location:** The dataset can be found in the artifacts repository under the directory: "circuits/FIGURES"

A.3 Set-up

A.3.1 Installation

Amplification, Eviction Set, and Prime + Store Attack

- Install gcc/g++, python3, matplotlib, gnuplot
- Install and activate emscripten

```
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```

- Switch the processor to run in performance mode (meaning the processor is set to its' maximum frequency), by run the following command as superuser ('sudo -s')

```
for i in 0..7; do echo performance > /sys/devices/system/cpu/cpu$i/cpufreq/scaling_governor; done
```

- Clone the GoT repo:

```
git clone https://github.com/0xADE1A1DE/GoT.git
```

- Get Chromium at commit hash `be87c21d2a7069363dfd66f278739d7e4211145e`

  Apply `/GoT/amplification_eviction_set_finding/wasm/v8.diff`, which does the following:

  – Creates a special native function `%CustomFn`
    * Gives access to `clflush`
    * Memory fences `mfence; lfence`
    * Non functioning virtual to physical address resolution (requires running Chromium without sandbox, and some more tedious setup, this part is not used by our experiments)

  – Alters the assembler to emit `rdtsc` when asked to emit a `mov register, imm` with `imm==0xddaa00ccbb00 || imm==0xddaa00ccbb80` This is used to provide our program with the ability to measure with `rdtsc`.

  Note: The experiment requiring this patch is for Figure 9, we only use the `clflush` and `fences` capabilities provided by this patch.

  Note: Due to the size of Chromium, and the limited capacity of the storage in our system, we suggest using our compilation located at `/home/acrypto/Documents/daniel/out/Default/chrome`

Trace Processing and Stitching

Access to an HPC node with 80 threads and AVX-2 instruction set support helps in speeding up evaluation process.

Circuit Evaluation

Ensure that huge pages are enabled and core 0,1,6,7 (for Core i7-9750H with 12 logical cores, these are core 0,1 and their sibling cores) are isolated. On Ubuntu,

This can be achieved by adding the boot parameter:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=0.1,6,7 default_hugepagesz=1G hugepagesz=1G hugepages=3000"
```

to /etc/default/grub.

Then run the command

```
sudo update-grub
```

Then Install gnuplot. In ubuntu, run

```
sudo apt install gnuplot
```

A.3.2 Basic Test

Circuit Evaluation To test that the program compiles successfully and the environment requirements are met. This test require huge pages to be enabled, refer to appendix A.3.1 on how to enable. Run the program with

```
taskset -c 1 ./out gol_glider_demo
```

This will run the game of life glider demo. Expect a glider to hover across your screen. Note that the process is pinned to core 1, as it is isolated from the rest of the system to reduce noise.

A.4 Evaluation workflow

Amplification, Eviction Set, and Prime + Store Attack

Refer to the readme file in `amplification_eviction_set_finding/README.md` and `prime_store_attack_finding/README.md`

Trace Processing and Stitching

Refer to the readme file in `trace_processing_and_stitching/README.md`

Circuit Evaluation

Refer to the readme file in `circuits/README.md`

A.4.1 Major Claims

(C1): Amplification, Eviction Set, and Prime + Store Attack *We can achieve a difference of 100ms between medians in native amplification (graph 8) We can achieve a difference of 1ms between medians in WASM amplification (graph 9) We can find eviction sets in WASM (graph 10). This is proven in experiment (E1)*

(C2): Trace Processing and Stitching *ElGamal Key Recovery using Prime+Store attack. This is proven by the experiment (E2) described in [Section 6.3 - 6.6] whose results are illustrated/reported in [Figure 11-12]*

(C3): Circuit Evaluation *Circuits achieve accuracy as described in the paper. This is proven by the experiment (E3) described in [Section 4] whose results are illustrated/reported in [Figure 3-6, and Table 1].*

A.4.2 Experiments

(E1): Amplification, Eviction Set, and Prime + Store Attack *[3 human-hours + 3 compute-hours + 500MB disk]: In this experiment*

How to: Follow the steps detailed in appendix A.4

Preparation: Install emscripten. As described in appendix A.3.1

Execution: Refer to the steps described in appendix A.4.

Results: Refer to claim (C1).

(E2): Trace Processing and Stitching *[1 human-hour + 1 compute-hour on (80 thread HPC) + 100MB disk]: In this experiment, ElGamal traces obtained from Prime+Store are transformed into square and multiply traces, and then key stitching is performed to combine partial traces into the complete ElGamal key.*

How to: Follow the steps detailed in appendix A.4

Preparation: To compile the trace processing and stitching programs, navigate to the "trace_processing_and_stitching" directory in the repository, and issue the make command.

```
make
```

Execution: Refer to the steps described in appendix A.4.

Results: You can follow the instructions outlined in section appendix A.4. Running the "sigproc" command will produce a set of square and multiply traces. It's important to note that these traces represent only portions of the complete ElGamal key, and must be combined or "stitched" together to retrieve the full key. This process is accomplished using the "stitch_parallel_simd_any_pos" and "stitch_parallel_simd" commands, which will generate the complete ElGamal key as the output in the form of square and multiply representation. Use the "sm_to_exponent" program to convert this into binary format.

(E3): Circuit Evaluation *[30 human-minutes + 3 compute-days + 100MB disk]: This experiment evaluates the accuracy of our circuits. Results are stored in each subdirectory of the experiments under the name "RESULT" which include the raw samples, their mean, and median.*

How to: Follow the steps detailed in appendix A.4

Preparation: 1) Navigate to the "circuits" directory in the cloned repository. 2) Execute `./compile.sh` to compile program.

Execution: Refer to the steps described in appendix A.4.

Results: Refer to the steps described in appendix A.4. Results of each circuit experiment is stored in a directory named "RESULT". This directory contains the raw samples, mean, and median. Use the "final.csv" to recreate the histograms presented in the paper. The "FINAL_MEAN" and "FINAL_MEDIAN" files carry the mean and median values respectively for each circuit, and are the values used in the paper. Follow the steps in appendix A.4 to generate figures identical to the ones found in the paper from collected data.

A.5 Notes on Reusability

All of our experiments are done on specific hardware, and reproducing on others may need additional tuning.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software

Yuanyuan Yuan, Zhibo Liu, Shuai Wang
The Hong Kong University of Science and Technology
{yyuanaq, zliudc, shuaiw}@cse.ust.hk

A Artifact Appendix

A.1 Abstract

We provide code and data of our paper in this artifact. Our artifact is publicly available at https://github.com/Yuanyuan-Yuan/CacheQL with detailed documents. Using our tool, users can quantify the side channel leaks and localize the leakage sites for different software.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact does not violate the security and privacy of evaluators. Nevertheless, since evaluators need to perform real-world side channel attacks (which require the root access) in some evaluations, we suggest evaluators using our artifact on test systems without sensitive data.

More importantly, we clarify that our artifact is provided as-is and is only for research purposes; any users should not use our scripts to attack others.

A.2.2 How to access

An archived copy of the initial version is available at: https://zenodo.org/record/8062035.

Our artifact is actively maintained at: https://github.com/Yuanyuan-Yuan/CacheQL.

A.2.3 Hardware dependencies

We do not have any particular requirements for the hardware. Our artifact may need GPUs to speed up training neural networks; we suggest evaluators having at least one GPU.

A.2.4 Software dependencies

Our tool is built based on Pytorch; evaluators need to first install Pytorch. See detailed instructions in our documents.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Users only need to install Pytorch first. See details in our documents.

A.3.2 Basic Test

Our artifact requires first preparing some data and then analyzing these data. Please see detailed instructions in our documents.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Security and Privacy Failures in Popular 2FA Apps

Conor Gilsenan, UC Berkeley / ICSI
Fuzail Shakir, UC Berkeley
Noura Alomar, UC Berkeley
Serge Egelman, UC Berkeley / ICSI

A Artifact Appendix

A.1 Abstract

The repo on GitHub contains the following artifacts:

Search Terms The list of search terms that we used to identify as many TOTP apps in the Google Play Store as possible (see Section 4.1 - App Selection);

App Checklists For each app, the customized checklist that enumerates exactly which actions to take within the app and which data to enter while recording the network traffic (see Section 4.2.1 - Exploring the App); and

Decryption Scripts For each app that supports encrypted TOTP backups, the golang script that implements the decryption process (see Section 4.2.3 - Performing Cryptanalysis).

Along with the instructions in the README, the app checklists and decryption scripts allow researchers to reproduce the findings we report in Tables 1, 2, and 3 in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The README instructions included in the repo suggest flashing a new Android ROM onto the device in order to use root privileges to capture the plaintext traffic generated by each TOTP app. Rooting a phone can have negative security consequences and flashing a new Android ROM will entirely wipe the phone's data, so we do not recommend using your primary Android device. If available, use an old Android phone that you can flash with a stock Android ROM after evaluation.

The backup mechanisms on several TOTP apps require divulging personal information, such as email address, phone number, name, and date of birth. To protect your privacy, we recommend using fake values where possible. Create a new email address specifically for the purpose of evaluation. Many apps require an active phone number that can receive SMS messages for authentication purposes. During our work, we purchased temporary phone numbers from messagebird.com to protect our privacy. Other telephony APIs available online can achieve the same privacy goals.

A.2.2 How to access

Publicly available at `https://github.com/blues-lab/totp-app-analysis-public`.

The following tag is intended for review by the USENIX 2023 Artifact Evaluation committee: `https://github.com/blues-lab/totp-app-analysis-public/releases/tag/usenix-sec23-ae`.

A.2.3 Hardware dependencies

During our research, we used Pixel 3a Android phones running a custom version of Android 9. However, our results can be replicated using any Android phone running version 9+.

A.2.4 Software dependencies

The README contains detailed instructions on how to install many of the following software dependencies:

- The decryption scripts require golang v1.18 or higher. The golang website provides installation documentation: `https://tip.golang.org/doc/install`
- The Android Debug Bridge (adb) is required to control the Android phone from the researcher's machine: `https://developer.android.com/studio/command-line/adb`
- Magisk is a suite of open source software for customizing Android: `https://github.com/topjohnwu/Magisk/blob/master/docs/install.md`
- mitmproxy is a free and open source interactive HTTPS proxy: `https://mitmproxy.org/`
- (Optional) The scrcpy tool allows the researcher to mirror the Android phone's screen onto their computer and, optionally, record the phone's screen: `https://github.com/Genymobile/scrcpy`

A.2.5 Benchmarks

None

A.3 Set-up

Please see the README in the artifact repo on GitHub.

A.3.1 Installation

Please see the README in the artifact repo on GitHub.

A.3.2 Basic Test

After following the instructions in the repo's README, you should be able to capture plaintext traffic generated by each TOTP app that you are evaluating.

A.4 Evaluation workflow

The README in the linked repo contains detailed steps to reproduce our findings for each TOTP app we analyzed.

Once recorded, the plaintext traffic can be analyzed to confirm whether TOTP fields (secret, issuer, label) are sent in plaintext. Additionally, values from the plaintext can be copy/pasted into the corresponding decryption scripts to verify the cryptographic primitives used in each TOTP backup mechanism.

A.4.1 Major Claims

Our major claims are enumerated in the following tables in the paper:

Table 1: Overview of the backup mechanisms supported in each app.
Table 2: Overview of the backup mechanisms that automatically sync data to the cloud.
Table 3: Cryptographic details of app backup mechanisms.

Evaluators should be able to verify and reproduce the findings reported in each cell of Tables 1, 2, and 3.

A.4.2 Experiments

The linked repo contains a custom checklist for each TOTP app, which enumerates exactly which actions to take within the app and which data to enter while recording the network traffic (see Section 4.2.1 - Exploring the App). It should take about 10-20 minutes to execute the steps defined in the checklist for each app.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: Multi-Factor Key Derivation Function (MFKDF) for Fast, Flexible, Secure, & Practical Key Management

Vivek Nair
UC Berkeley
vcn@berkeley.edu

Dawn Song
UC Berkeley
dawnsong@berkeley.edu

A Artifact Appendix

Multi-Factor Key Derivation Function (MFKDF)
mfkdf.com | pbkdf2.com

A.1 Abstract

We present a JavaScript library implementing all of the factors and methods associated with the Multi-Factor Key Derivation Function (MFKDF) proposal. The library covers all proposed features of MFKDF, including threshold MFKDF, policy-based MFKDF, entropy measurement, authentication, and factor constructions for HOTP, TOTP, HMAC-SHA1, OOBA, and more. In separate repositories, we also include centralized and decentralized proof-of-concept web applications, along with a browser-based benchmarking suite. These repositories together contain about 100,000 lines of JavaScript code.

To aid evaluation, we have detailed documentation that includes usage examples for every supported method, as well as a series of in-depth tutorials. We have also included a unit testing suite with 100.0% code coverage, the results of which can be viewed online. We have compiled and hosted the demo applications and benchmarks for easy online access, and have included video tutorials explaining how to use them.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact is non-destructive, but caution should be taken when using the proof of concept applications, as they are intended for demonstration purposes only and have not been audited for security vulnerabilities. When evaluating the demo applications, evaluators should not use credentials that they have used or intend to use for any other application. Nothing of value should be stored in the ETH demo wallet at this time. We also recommend that any parties wishing to remain anonymous use a fictitious name and disposable email address.

A.2.2 How to access

MFKDF Library
- GitHub Repository (Stable): `https://github.com/multifactor/MFKDF/tree/1427224a709b77312b1b03cfa79ebed7bed316ea`
- Website: `https://mfkdf.com` (or `https://pbkdf2.com`)
- Documentation: `https://mfkdf.com/docs`
- Unit Testing Results: `https://mfkdf.com/tests`
- Code Coverage Report: `https://mfkdf.com/coverage`

Centralized Demo
- GitHub Repository (Stable): `https://github.com/multifactor/mfkdf-application-demo/tree/37ca96c58c050e460e6a3d4d09896eae06ed2720`
- Live Demo: `https://demo.mfkdf.com`
- Video: `https://youtube.com/watch?v=cB44BMGnFIs`

Decentralized Demo
- GitHub Repository (Stable): `https://github.com/multifactor/mfkdf-wallet-demo/tree/1fbc67d2b7505b2185a8ca3ce9ba163e22ae29ab`
- Live Demo: `https://wallet.mfkdf.com`
- Video: `https://youtube.com/watch?v=u3eUsPnv7K8`

Benchmarking
- GitHub Repository (Stable): `https://github.com/multifactor/mfkdf-benchmark/tree/72b81f89818b7b68313e05f17764aadb38fb99e0`
- Live Demo: `https://benchmark.mfkdf.com`

A.2.3 Hardware dependencies

Any system with a stable internet connection and capable of running JavaScript code should be able to run our demo applications and benchmarks. For consistency, the device we used for benchmarking has an AMD Ryzen 9 5950X CPU, NVIDIA GeForce RTX 3090 GPU, and 128GB of DDR4 RAM, although the benchmarking results almost exclusively depend on single-core CPU performance in this case. For the centralized demo, an iOS or Android mobile device capable of installing the Google Authenticator application is required. The decentralized demo application can use (but doesn't require) a YubiKey device supporting HMAC-SHA1.

A.2.4 Software dependencies

We expect that any device with a relatively modern web browser (HTML5/ES6) will be able to run our benchmarks and demo applications. Our evaluations were performed in Chrome Browser v103.0.5060.114 on Windows 10 v21H2.

If manually building any of the packages, each repository contains a `package.json` file with the names and versions of all dependencies. Node.js and NPM are required to build and test each package; we used Node.js v16.15.0 and NPM v8.4.0. Running `npm install` in the root directory of each repository should automatically install all of the dependencies.

A.2.5 Benchmarks

A self-contained benchmark that can run in any modern web browser (usually in less than a minute) is available at `https://benchmark.mfkdf.com`. The relevant source code is visible in index.html. No external data is required.

A.3 Set-up

A.3.1 Installation

1. Download and install the latest version of Google Chrome from `https://www.google.com/chrome`.
2. Download and install the latest LTS version of Node.js (including NPM) from `https://nodejs.org/en`.
3. On a mobile device, download and install the latest version of Google Authenticator for iOS or Android.
4. Clone our main GitHub repository by running `git clone http://github.com/multifactor/MFKDF`
5. Open the root directory of the repository (`cd MFKDF`), and then run `npm install` to download all dependencies.
6. Repeat steps 4 and 5 for each of the demo repositories if you wish to build them instead of using the hosted versions.

A.3.2 Basic Test

If the repository and dependencies have been installed correctly, running `npm run build` should successfully compile the package (you should see a message like `webpack X.X.X compiled with X warnings in X ms`).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The MFKDF algorithms and constructions of §4–§9, as summarized by the pseudocode given in §A, can be used to produce a fully-functional MFKDF implementation satisfying the stated definitions and security goals of §3. This can be verified by experiment (E1), which runs a test suite enforcing the specification of the paper.

(C2): MFKDF has a low computational overhead over PBKDFs in a typical web browser. This can be verified by experiment (E2), which runs browser-based benchmarks for the setup and derive functions of a standard 3-of-3 MFKDF setup, a 2-of-3 threshold MFKDF setup, and all supported authentication factor constructions. The results can be used to verify the performance claims of §11 of the paper, particularly figures 7 and 8.

(C3): MFKDF can be used in place of PBKDFs in common centralized applications like password managers. Such applications can authenticate using derived keys (see §7), enforce arbitrarily specific derivation policies (see §9), and are fully backward-compatible with existing popular authentication factors like TOTP (see §5). This can be verified by experiment (E3), which evaluates a functional MFKDF-based password management application using the standard Google Authenticator mobile app, and can in turn can be used to confirm §10.1 of the paper.

(C4): MFKDF can be used to enable new applications in situations where PBKDFs would not be used, such as in fully-decentralized applications. The public parameters (α) can be stored openly, such as on a public blockchain. This can be verified by experiment (E4), which evaluates a decentralized Ethereum and ERC20 wallet based on MFKDF and stores parameters using IPFS and IPNS, per the description of §10.2 of the paper.

A.4.2 Experiments

(E1): *[Unit Tests] [5 human-minutes + 5 compute-minutes]: Run the included unit testing suite to verify that all tests are passing with 100.0% code coverage.*
Preparation: Clone the main MFKDF repository and install the dependencies as described in §A.3.1.
Execution: Simply run `npm test` to simultaneously generate testing and code coverage results. The tests assert the library's compliance with the specifications of the paper; you can browse the `test` directory to verify.
Results: After about 2 minutes, `337 passing` should be displayed with no tests failing. Below that, the code coverage report should show 100% for all files.

(E2): *[Benchmark] [5 human-minutes + 1 compute-minute]: Run our browser-based benchmark to replicate the main performance evaluation described in the paper.*
Preparation: Visit our hosted benchmarking page at `https://benchmark.mfkdf.com` (recommended), or clone the benchmarking repository and read `README.md`.
Execution: Simply click "run now" to benchmark MFKDF, threshold MFKDF, and all supported factors. You can browse the source code to verify its validity.
Results: After about 1 minute, a results table should be displayed that roughly matches Fig. 7 and Fig. 8 of the paper. The scripts for generating these figures using the benchmarking output are included in the `figs` directory.

(E3): *[Centralized Demo] [30 human-minutes]: Run our browser-based centralized proof-of-concept to verify the compatibility results described in §10.1 of the paper.*
Preparation: Visit our hosted application demo at `https://demo.mfkdf.com` (recommended), or install the repository manually per §A.3.1 and run `npm build`.
Execution: Create an account on the demo application using false information and the Google Authenticator app on your mobile device. Use an anonymous, but valid, email address that can receive mail. Store at least one fictitious password, then log out. To store a password, type a site name (e.g., google.com), then click on the correct option from the dropdown, and type a username and password before clicking save. Log back in using your master password and TOTP, or try the various recovery options, and verify that you can still access the stored password. Our demo video shows the intended usage.
Results: Confirm that the application behaves as described in the paper. Specifically, it should be fully backward-compatible with the Google Authenticator mobile app, and that incorrect login factors can't be used to successfully access encrypted information. You can verify that the source code uses MFKDF as described.

(E4): *[Decentralized Demo] [30 human-minutes]: Run our browser-based decentralized proof-of-concept to verify the functionality described in §10.2 of the paper.*
Preparation: Visit our hosted application demo at `https://wallet.mfkdf.com` (recommended), or install the repository per §A.3.1 and run `npm build`.
Execution: Use false credentials to create a wallet for the Ethereum mainnet or Ropsten testnet. Note the username and recovery code. Optionally, you may transfer nominal funds to the wallet address using a free Ropsten faucet. Sign out, then log back in using your password and recovery code, and verify that you can still access the funds. Our demo video shows the intended usage.
Results: Confirm that the wallet behaves as described in the paper. Verify that incorrect login factors can't be used to successfully access the wallet. You can check that the source code uses MFKDF as described.
***Note:** In the time between submission and publication, the Ethereum merge caused the Ropsten testnet to shut down. Our new and improved demo, using the Sepolia testnet, can be found at `https://ciao.mfkdf.com`.

A.5 Archive of Repositories

The version of each repository evaluated by the USENIX AEC has been tagged with "usenix-ae" on GitHub. A copy of each of these repositories has also been uploaded to Zenodo in their evaluated state for historical preservation:

`https://doi.org/10.5281/zenodo.7859226`

A.6 Notes on Reusability

The MFKDF JavaScript library was built with the express goal of being flexible and easy to deploy in a wide variety of new or existing applications. Its creative commons license puts almost no restrictions on non-commercial use.

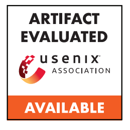
We suggest that interested parties get started by visiting mfkdf.com to learn more, reading our tutorial series, browsing our detailed documentation, and trying our demos.

The MFKDF library is flexible, modular, and easy to extend with new features and factor types. If you are interested in contributing, please read our contributing guide.

The benchmarking code and two demo applications are all offered under the MIT license, and can be modified and redistributed essentially without restriction.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.



USENIX'23 Artifact Appendix: Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants

Gustavo Sandoval*, Hammond Pearce*, Teo Nys, Ramesh Karri, Siddharth Garg, Brendan Dolan-Gavitt
New York University

A Artifact Appendix

A.1 Abstract

This artifact contains raw user data collected during the user study and the scripts used for study evaluation. The user data includes (1) anonymous demographic information, (2) submitted code files, (3) the annotations to those code files performed by the authors' manual security analysis, (4) the complete database record of 'prompts' and 'suggestions' by the utilized language model (`code-cushman-001` by OpenAI). The scripts for study evaluation are written in Python version 3.10.6, and may be executed on any compatible machine.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The code executes locally on resources obtained via the artifact repository. No internet or network access is used beyond this initial download of the artifact repository and the subsequent installation of software dependencies (Python libraries, see Section A.2.4).

The user study data was collected via the involvement of human participants and was approved by New York University's Institution Review Board (IRB) as #IRB-FY2022-6074. There is no known risk for evaluators when executing this artifact as no destructive steps are taken and no evaluator files will be impacted.

A.2.2 How to access

Access via URL: https://zenodo.org/record/7187358

A.2.3 Hardware dependencies

No special hardware dependencies are required, only a machine capable of running Python. The authors used a computer with 16GB of RAM and an Intel i7-10750 with Ubuntu 22.04.

A.2.4 Software dependencies

We assume that evaluation is being undertaken on a Debian-based Linux system. The specific software dependencies are Python 3.10.6, virtualenv 20.13.0, and pip 20.0.2. Installation is described under Section A.3.1.

*Equal Contribution

A.2.5 Benchmarks

The user study data is provided as an input to the artifacts. They are provided in the folder `data` as part of the repository download.

A.3 Set-up

A.3.1 Installation

Ensure build-essential, Python3, pip, virtualenv, and parallel are installed. These should be able to be installed on Debian-based Linux systems with:

```
$ sudo apt-get install build-essential
$ sudo apt-get install python3 python3-pip parallel
$ sudo pip3 install virtualenv
```

Download the repository from the Zenodo URL. Navigate to the root of the downloaded folder, and create and activate a new virtual environment:

```
$ virtualenv venv
$ source venv/bin/activate
```

Now install the necessary Python libraries:

```
$ pip install -r requirements.txt
```

A.3.2 Basic Test

You can test that your system works by running the first (and simplest) generation,

```
$ python plot_fig7.py
```

This should produce:

```
Created FIG7 as figures/functionality.pdf
```

Which you can check by opening that file and seeing that it matches the paper.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1 / RQ1 - Functionality): There are systematic differences between the 'assisted' and the 'control' groups, with the 'assisted' group having a small but consistent advantage over the 'control' group, and the 'autopilot' group outperforming both. The small sample size however means that the comparison does not reach statistical significance. These results are presented in the paper in Section 4.2.Results and Figure 7, and reproduction is described in Experiment E1 of this Appendix.

(C2 / RQ2 - Security 'aggregate'): For all four cases {CWEs/LoC over compiling functions; CWEs/LoC over functions passing unit tests; Severe CWEs/LoC over compiling functions; Severe CWEs/LoC over functions that pass the unit test} the 'assisted' group has fewer bugs compared to the 'control', with up to a 22% lower mean for the 'assisted' group compared to the 'control' for Severe CWEs/LoC over functions that pass unit tests. For severe CWEs, the comparisons are also statistically significant using non-inferiority tests with $\delta = 10\%$. This suggests that in the aggregate case, LLMs may provide a slight benefit to security. These results are presented in the paper in Section 4.3.3 Topline results and Figure 8, and reproduction is described in Experiment E2 of this Appendix.

(C3 / RQ2 - Security 'per-function'): When examining individual functions in the user study, results vary, with different functions being harder or easier for each group and with some differences being statistically significant. As a result of this analysis, it is hard to conclude how LLM suggestions may impact the code security of any arbitrary code-writing task (some functions are made less buggy, some are made more, it likely depends on the complexity of each specific function). These results are presented in the paper in Section 4.3.4 Per-function CWE rates, and Table 3, and reproduction is Experiment E3 of this Appendix.

(C3 / RQ3 - Bug origin): Even though suggestions from the LLM may contain bugs, the human developers introduced a majority of the bugs present in the submitted code from the 'assisted' group. These results are presented in the paper in Section 4.4 RQ3 - On the origin of bugs.

A.4.2 Experiments

The functionality of these artifact scripts are predicated on the completion of a user study as outlined in Preliminary 1 and the formatting of that data in Preliminary 2 as well as bug data encoding in Preliminary 3. Should a fresh user study not be desirable/attempted, evaluators may skip ahead to the data analysis stage starting with Experiment E1.

Preliminaries (User study and first-pass processing):

Preliminary 1 - User study (Manual) [Approximately 1 month]: Each experiment E1-E4 implicitly relies on the data produced by a user study following the setup of the paper (see the paper Section 3). This would use the user study participant files provided in the linked artifacts repository folder `study_participant_instructions`. The data produced by our user study is provided.

Preliminary 2 - First-pass data processing (Automated) [Approximately 15 compute-minutes]: This step converts the given user study data files into ones suitable for the rest of the data processing pipeline. This includes running functional tests against the study-designed test cases and breaking the files into separate functions for split testing (see paper Section 4.2 'Split Testing') and manual bug encoding (see Preliminary 3). Note that executing the linked scripts 'resets' the bug encoding process: as a result, the script will create a new directory rather than overwrite the data that our study collected (should you wish, you may manually overwrite our data by using a copy paste).
Preparation: As per Section A.3.1.
Execution: From the `functionality_tests` directory, execute `./run_all.sh`. This script creates a directory with the necessary testing infrastructure for each user study file in `functionality_tests/repos/{uuid}`, executes the test suites, and saves the results to JSON files in each directory named `api_report.json` (indicating which API functions were implemented, unimplemented, or failed to compile), `orig_testsuite.json` (the results of the basic 11-function test suite), and `ref_testsuite.json` (the results of the expanded 45-function test suite).
This script will produce for each participant a directory containing the results of the split-functional testing and the components of each of their submissions. It also creates extraneous/intermediate files we removed from our own data output folder, which can be found at `data/submitted_assignments/*`.

Preliminary 3 - Bug encoding (Manual) [Approximately 66 person-hours]: To evaluate the security of the code provided by the users, manual analysis was performed. This involved the process described in the paper Section 4.3.1. From a technical point of view, each file in the `data/submitted_assignments/{uuid}/parts/gen_{function}.c` was read by three people who manually looked for security relevant bugs. These annotations are still present and may be modified or evaluated by artifact evaluators. Once this evaluation is done, the bugs must be 'typed up' into a sqlite database with the schema described in `bugs_and_demographics.sqlite3`.

Analysis of user study data:

(E1 / RQ1 - Functionality) [<1 minute total]: This experiment runs several Python scripts.

Preparation: Section A.3.1 and the preliminaries.

Execution: To see the data relevant to this claim run the command: `python plot_fig7.py`. This will produce the file `figures/functionality.pdf` as well as the file `data/derived_data/functionality_stats.txt`.

Results: The file `figures/functionality.pdf` displays the relationship between groups, and should show how the 'autopilot' group appears to function better than the 'assisted' group, which functions better then the 'control' group. Statistically speaking, however, the comparisons between groups for code passing basic and expanded tests does not reach statistical significance as shown in the `functionality_stats.txt` file from line 88 onwards.

(E2 / RQ2 - Security 'aggregate') [<1 minute total]: This experiment runs several Python scripts.

Preparation: Section A.3.1 and the preliminaries.

Execution: To see the data relevant to this claim run the command: `python plot_fig8.py`

Results: The following results will be presented in the terminal and they will create the appropriate images for Figure 8 of the paper:

```
Plotting figures/bugs_per_loc_compiled.pdf
Plotting figures/bugs_per_loc_passing.pdf
Plotting figures/bugs_per_loc_severe_compiled.pdf
Plotting figures/bugs_per_loc_severe_passing.pdf
```

In addition, to run the non-inferiority tests with $\delta = 10\%$, after running the script to run the figure, you may run the script `python inferiority_tests.py`. This script will produce in the command line the results that show the *p-values* for each of the four groups. The two important values for the graph are the values for `Per Loc Severe Compiled` and `Per Loc Severe Passing`, which show that for 'the Severe CWEs per Line of Code Compiled' the non-inferiority test is significant with *p-value* of p=0.04 and the Non-inferiority test for the Severe CWEs per LOC passing which is p=0.06.

(E3 / RQ2 - Security 'per-function') [<1 minute total]: This experiment runs several Python scripts.

Preparation: Section A.3.1 and the preliminaries.

Execution: The results for this claim are made in a two step-process. Firstly, we derive the main Table 3 data by the command `python generate_table3.py` which will produce the file `data/derived_data/table3.tsv`. Then, we perform the non-inferiority tests by running `python inferiority_per_func.py`.

Results: The statistical test results between function group pairings are directly printed to the console. These in conjunction with the derived `table3.tsv` should match Table 3 in the paper.

(E4 / RQ3 - Bug origin) [<1 compute-minutes, up to 6 person-hours]: This experiment runs several Python scripts and may involve a further optional human-annotation step.

Preparation: Section A.3.1 and the preliminaries.

Execution: Demonstrating this is a two-step process. First we run the command `python suggestion_cover.py -o data/derived_data/suggestion_cover.html` to display the origin of code in the final output files.

There is now a human-annotation step, which is required as determining the relationship with the final code and the model suggestions was often beyond simple lexical analysis. For result reproduction, this step may be considered optional as it is laborious. For each bug present in each file in `data/submitted_assignments/recombined_list_files/Active/{uuid}-list.c`, one must scroll to the given bug location in the `suggestion_cover` file. Using a mouse-over, this will display the lexical origin of the suggestion. If the reviewer agrees, then this annotation can be recorded in the original file in the manner described in `bug_origin_all.py`. The authors thus went through each identified bug and visually determined their origin.

Secondly, once bugs were annotated, we can then use the command `python bug_origin_all.py`, which uses `grep` to scan the annotated code files for the human-annotated bugs and their origins and aggregates the statistics. The script prints the results to the terminal, reporting that 63.1 % of bugs come from human developers in the assisted groups.

Results: Results are presented in the terminal and display the count of the origin of each bug.

We can explore the specific reasons for this further by examining a specific bug, CWE-416, with the command `python bug_origin_cwe416.py`. This will, for each user in the assisted group, find each instance of the bug—then reporting if the first time it appeared it was in a suggestion or in the human's own-written code, then count the number of times it was suggested, accepted, and the number of times it appeared in the final document. The results show that for this bug, the LLM typically suggested the bug even if it was not already present in the user's code.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

Aegis: Mitigating Targeted Bit-flip Attacks against Deep Neural Networks

Jialai Wang*, Ziyuan Zhang†, Meiqi Wang*, Han Qiu*§✉, Tianwei Zhang‡,
Qi Li*§, Zongpeng Li*♣✉, Tao Wei♠, and Chao Zhang*§
*Tsinghua University, †Beijing University of Posts and Telecommunications,
‡Nanyang Technological University, ♣Hangzhou Dianzi University, ♠Ant Group,
§Zhongguancun Laboratory

{wang-jl22, wang-mq22}@mails.tsinghua.edu.cn, zhangziy0421@bupt.edu.cn,
{qiuhan, qli01, zongpeng, chaoz}@tsinghua.edu.cn, tianwei.zhang@ntu.edu.sg,
lenx.wei@antgroup.com

A Artifact Appendix

A.1 Abstract

We teach you how to run our experiments in this appendix. If you have any questions, please let us know.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

https://github.com/wjl123wjl/Aegis.git
The GitHub hash is:
95ded45538bf358ce5122a6e3ff920db25525186

A.2.3 Hardware dependencies

You need GPUs to train or run models.

A.2.4 Software dependencies

None.

A.2.5 Benchmarks

Datasets. Our source code could automatically download the datasets, i.e., CIFAR-10, CIFAR-100, STL-10. For Tiny-ImageNet, please download it by yourself.
Models. Our source code could train all models, and you can directly run the scripts to train the models.

✉ Corresponding authors.

A.3 Set-up

A.3.1 Installation

We list the python packages and the corresponding versions to install. Note other versions may work as well, but we haven't tried it.

(1) Please install python 3.6.9.
(2) Please install pytorch 1.7.0.
(3) Please install torchvision 0.8.1.
(4) Please install tensorboardX 2.5.
(5) Please install matplotlib 3.3.4.
(6) Please install tqdm 4.60.0.
(7) Please install pandas 1.1.5.
(8) Please install numpy 1.18.5.

A.3.2 Basic Test

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Aegis could effectively mitigate TBT attacks, and adaptive TBT attacks.
(C2): Aegis could effectively mitigate TA-LBF attacks, and adaptive TA-LBF attacks.
(C3): Aegis could effectively mitigate ProFlip attacks, and adaptive ProFlip attacks.

A.4.2 Experiments

Before conducting experiments, you need to train all models.

- **CIFAR-10: train resnet32.**
  (1) cd cifar10/resnet32
  (2) Train the base model: sh train_CIFAR.sh.
  (3) After finishing training the base model, then train the enhanced model: sh train_finetune_branch.sh

- **CIFAR-10: train vgg16.**
  (1) cd cifar10/vgg16
  (2) Train the base model: sh train_CIFAR.sh.
  (3) After finishing training the base model, then train the enhanced model: sh train_finetune_branch.sh
- **CIFAR-100: train resnet32.**
  (1) cd cifar100/resnet32
  (2) Train the base model: sh train_CIFAR.sh.
  (3) After finishing training the base model, then train the enhanced model: sh train_finetune_branch.sh
- **CIFAR-100: train vgg16.**
  (1) cd cifar100/vgg16
  (2) Train the base model: sh train_CIFAR.sh.
  (3) After finishing training the base model, then train the enhanced model: sh train_finetune_branch.sh
- **STL-10: train resnet32.**
  (1) cd stl10/resnet32
  (2) Train the base model: sh train_STL.sh.
  (3) After finishing training the base model, then train the enhanced model: sh train_finetune_branch.sh
- **STL-10: train vgg16.**
  (1) cd stl10/vgg16
  (2) Train the base model: sh train_STL.sh.
  (3) After finishing training the base model, then train the enhanced model: sh train_finetune_branch.sh
- **Tiny-ImageNet: train resnet32.**
  (1) cd tinyimagenet/resnet32
  (2) Train the base model: sh train_tinyimagenet.sh.
  (3) After finishing training the base model, then train the enhanced model: sh train_finetune_branch.sh
- **Tiny-ImageNet: train vgg16.**
  (1) cd tinyimagenet/vgg16
  (2) Train the base model: sh train_tinyimagenet.sh.
  (3) After finishing training the base model, then train the enhanced model: sh train_finetune_branch.sh

(E1): For TBT attacks.
(1) First enter a folder to attack the target model, e.g., *cd ./Aegis/TBT/resnet32-cifar10/*
(2) If you want to conduct the TBT attack, run the instruction: *python3 TBT_noadaptive.py*. Then, you can observe the ASR.
(3) If you want to conduct the adaptive TBT attack, run the instruction: *python3 TBT_adaptive.py*. Then, you can observe the ASR.

(E2): For non-adaptive TA-LBF attacks. Please enter the folder: cd TA-LBF/non-adaptive. For adaptive TA-LBF attacks. Please enter the folder: cd TA-LBF/adaptive.
(1) on cifar10 and resnet32, run the instruction:
sh ./attack_reproduce_k=50_resnet32_cifar10.sh
(2) on cifar10 and vgg16, run the instruction:
sh ./attack_reproduce_k=50_vgg16_cifar10.sh
(3) on cifar100 and resnet32, run the instruction:
sh ./attack_reproduce_k=50_resnet32_cifar100.sh
(4) on cifar100 and vgg16, run the instruction:
sh ./attack_reproduce_k=50_vgg16_cifar100.sh
(5) on stl10 and resnet32, run the instruction:
sh ./attack_reproduce_k=50_resnet32_stl10.sh
(6) on stl10 and vgg16, run the instruction:
sh ./attack_reproduce_k=50_vgg16_stl10.sh
(7) on tinyimagenet and resnet32, run the instruction:
sh ./attack_reproduce_k=50_resnet32_tinyimagenet.sh
(8) on tinyimagenet and vgg16, run the instruction:
sh ./attack_reproduce_k=50_vgg16_tinyimagenet.sh

(E3): For Proflip attacks.
(1) First enter a folder to attack the target model, e.g., *cd cd ./Aegis/ProFlip/resnet32-cifar10/*
(2) If you want to conduct the ProFlip attack, run the instruction to generate a trigger: *python3 trigger_nonadaptive.py*. Then, run the instruction to attack: *python3 CSB_nonadaptive.py*. Then, you can observe the ASR.
(3) If you want to conduct the adaptive ProFlip attack, run the instruction to generate a trigger: *python3 trigger_adaptive.py*. Then, run the instruction to attack: *python3 CSB_adaptive.py*. Then, you can observe the ASR.

A.5 Notes on Reusability

None.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix

IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks

Neophytos Christou
Brown University

Di Jin
Brown University

Vaggelis Atlidakis
Brown University

Baishakhi Ray
Columbia University

Vasileios P. Kemerlis
Brown University

A Artifact Appendix

A.1 Abstract

This is the artifact appendix for the IvySyn fuzzing framework. It contains instructions about how to setup, run, and reproduce the results of IvySyn, along with information regarding system and resource requirements.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

IvySyn is a fuzzer for discovering security-critical bugs in Deep Learning (DL) frameworks. The list of APIs for which IvySyn has uncovered bugs, during our experiments, is available in the repository of the project. If, during the reproduction of any reported result, IvySyn produces PoVs for APIs not listed in project repository, contact the authors via HotCRP or follow the *Responsible Disclosure* steps in README.md to report the (newly discovered) vulnerabilities to the developers of the corresponding framework(s).

A.2.2 How to access

IvySyn is available at: https://gitlab.com/brown-ssl/ivysyn/-/tree/4b3d26dda0ddea11282c2658e28090a738dfd6c7 (stable ref.)

A.2.3 Hardware dependencies

The provided Docker images are configured to use 4 CPUs and 16GB of RAM, but can also be set to use fewer (or more) resources, as needed.

A.2.4 Software dependencies

We provide a Docker image that builds and runs IvySyn, and hence Docker is required. Some scripts run outside Docker containers and were tested on Debian v11—but they are relatively simple and should work on any Linux distribution.

A.2.5 Benchmarks

All the data required for running our benchmarks are either included in the project repository or can be produced by our scripts during setup. Note that the prototype implementation of IvySyn fuzzes both CPU- and GPU-specific implementations of DL kernels. However, we are not able to provide access to machines with GPUs. Therefore, the benchmarks in the artifact fuzz only kernels with CPU implementations. This does not have any effect on the claims of the paper, other than a smaller number of instrumented and fuzzed kernels.

A.3 Setup

A.3.1 Installation

To setup IvySyn, simply invoke docker/download-prebuilt-image.sh. This script will download a pre-built Docker image of IvySyn. Alternatively, in order to build the IvySyn Docker image from scratch, invoke the script docker/build-docker-image.sh, under the root directory of the project repository. (Note that building the image from scratch requires ≈3.5 hours on a 16-core, 64GB RAM host; and the total size of the fully built image is ≈35GB.)

- To start the container, simply run:
  docker/run-docker-image.sh
  (This script is configured to run the container with access to 4 CPUs and 16GB of RAM.)

- To provide more/less resources to the container, use the optional --memory (e.g., --memory 8g) and --cpus (e.g., --cpus 2) arguments to the run-docker-image.sh script. If you increase the number of CPUs the container can use, you should also increase the amount of RAM, since more parallel jobs may require more memory.

- To get a shell on the container, run:
  docker exec -it ivysyn-instance /bin/bash

If you wish to run the experiments for comparing IvySyn with the two other fuzzers, namely Atheris and DocTer, an additional ≈39GB of storage is required for their corresponding Docker images (i.e., ≈26GB for Atheris and ≈13GB for DocTer). For more details, refer to A.4.1 and A.4.2.

A.3.2 Basic Test

For a quick smoke test, we recommend invoking the do-run.sh script, *inside the running Docker container of IvySyn*, and providing a small number of kernels to be fuzzed with the --nkernels argument. For example:

```
./do-run.sh --seed 1 --pytorch --nkernels 5
```

Once the script done, it should display a summary of the run, containing information about how to inspect the results.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): IvySyn *automatically* fuzzes DL frameworks and produces *Proofs of Vulnerability (PoVs)*—i.e., code snippets that trigger memory errors in low-level (C/C++) code of the respective framework via a high-level (Python) API.

(C2): IvySyn: (i) uncovers *more crashes* in the DL frameworks than the state-of-the-art Python fuzzer Atheris, and (ii) it does so *faster*. This is proven by the experiment described in Section 7.2 of our paper.

(C3): IvySyn produces *more PoVs per unit of time* than DocTer, yet another DL-framework fuzzer. This is proven by the experiment in Section 7.3 of our paper.

A.4.2 Experiments

In what follows, we provide instructions on how to setup, run, and interpret the results of our experiments. All compute-time approximations assume you are using 16GB of RAM and 4 CPUs to run the Docker containers. For additional details, see README.md at the root of the project repository.

(E1): *[Up to 58 compute-hours – Up to 28 compute-hours with suggested configuration + Up to 2GB disk]: Run IvySyn on a selected framework and produce PoVs.*

Preparation: Run and connect to the provided (or custom-built) Docker image.

Execution: To perform a full run of IvySyn, invoke the do-run.sh script, *inside the running Docker container of IvySyn*, by providing an integer as the RNG seed and either --tensorflow or --pytorch to choose the framework to be fuzzed, as follows:

```
./do-run.sh --seed 123 --tensorflow
```

However, note that a full run will require ≈45 hours for PyTorch and ≈12 hours for TensorFlow. We advise restricting the amount of kernels that will be fuzzed, by providing the extra --nkernels argument. Furthermore, we suggest fuzzing 300 kernels for each framework. This will require ≈17 hours for PyTorch and ≈9 hours for TensorFlow, while still producing PoVs. Running on TensorFlow will require an additional 2 hours for the first invocation of the script, in order to compile the C++ developer-provided TensorFlow tests, which are also used by IvySyn. Execute IvySyn as follows:

```
./do-run.sh --seed <rng seed> --tensorflow
--nkernels 300
```

Results: Once fuzzing is done, IvySyn will produce PoVs under results/<framework>/ivysyn_povs (where <framework> is pytorch or tensorflow). To manually run and reproduce the PoVs, do the following (in the IvySyn Docker container):

1. Activate the environment of the pip-installed version of the corresponding framework. In the case of PyTorch, run: source /home/ivyuser/ivysyn/venv/anaconda3/bin/activate; conda activate pytorch-1.11-orig. For TensorFlow, run: source /home/ivyuser/ivysyn/venv/tensorflow-2.6-orig/bin/activate.
2. Run the PoVs produced under /home/ivyuser/ivysyn/results/<framework>/ivysyn_povs using python3 <pov.py> (where <pov.py> is the filename of the selected PoV).

(E2): *[Atheris-comparison] [Up to 400 compute-hours – Up to 26 compute-hours with suggested configuration + Up to 40GB disk space]: Run IvySyn and a selected variant of the Atheris fuzzer, and compare their efficiency at uncovering crashing inputs.*

Preparation: We provide a separate Docker image that sets-up the two variants of Atheris, namely Atheris+ and Atheris++, which are described in Section 7.1 of our paper. Similarly to the IvySyn image, you can build this image from scratch, by running:

```
comparisons/atheris_comp/docker_env/docker
/build-docker-image.sh
```

or download a pre-built version of the image by running:

```
comparisons/atheris_comp/docker_env/docker
/download-prebuilt-image.sh
```

Execution: To perform the experiment that compares IvySyn to Atheris, invoke the compare-fuzzers.sh script *outside the IvySyn container*.

You need to specify an RNG seed, the framework to fuzz (either `--tensorflow` or `--pytorch`), the `Atheris` variant (`--atheris1`, which corresponds to `Atheris+`; or `--atheris2`, which corresponds to `Atheris++`), and whether you want to limit the number of kernels to be fuzzed using the `--nkernels` argument. For example:

```
./compare-fuzzers.sh --tensorflow
--atheris2 --seed 123 --nkernels 50
```

We suggest running at least 50 kernels to get a decent approximation of the overall results.

The script above will:

1. Instrument the subset of 308 TensorFlow and 283 PyTorch kernels we performed our experiment on (depending on the chosen framework) and re-compile the target framework.
2. Execute `IvySyn` on the target kernels, limiting the fuzzed kernels if `--nkernels` was specified.
3. Run the selected `Atheris` variant on the same subset of fuzzed kernels.
4. Output a summary of the results.

Results: The `compare-fuzzers.sh` script will display a summary of the results. Specifically, a new directory will be created at `results/<framework>/atheris_comp/` (where `<framework>` is `pytorch` or `tensorflow`), containing the following:

- `results.csv`: start/end timestamp, as well as whether a crash was found, for each API (CSV file).
- `total_time.txt`: total time of the run (text file).
- `fuzzer_logs_dir.txt`: name of the directory in the Docker container with the raw logs produced by the `IvySyn` fuzzer (text file).

The corresponding CSV that contains similar entries for the `Atheris` run can be found at `comparisons/atheris_comp_fuzzed_<framework>.csv` (where `<framework>` is either `pytorch` or `tensorflow`). The raw `Atheris` logs can be found in the `Atheris` Docker container at `/home/ivyuser/ivysyn-atheris/fuzzer_output`. To connect to the `Atheris` Docker container, in order to manually inspect the logs, run `docker exec -it atheris-instance /bin/bash`.

(E3): *[DocTer-comparison] [Up to 34 compute-hours – Up to 26 compute-hours with suggested configuration + Up to 15GB disk space]: Run `IvySyn` and `DocTer` and compare their effectiveness at producing PoVs.*

Preparation: We provide a separate Docker image that sets-up `DocTer`. Similarly to the `IvySyn` image, you can either build it from scratch, by running:

```
comparisons/docter_comp/docker_env/docker
/build-docker-image.sh
```

or download a pre-built version of the image, by running:

```
comparisons/docter_comp/docker_env/docker
/download-prebuilt-image.sh
```

Execution: To perform the experiment that compares `IvySyn` to `DocTer`, invoke the `compare-fuzzers.sh` script *outside the IvySyn container*. You need to specify an RNG seed, the framework to fuzz (either `--tensorflow` or `--pytorch`), the `--docter` flag, and whether you want to limit the number of kernels to be fuzzed using the `--nkernels` argument. For example:

```
./compare-fuzzers.sh --tensorflow --docter
--seed 123 --nkernels 50
```

We suggest running at least 50 kernels to get a decent approximation of the overall results.

The script above will:

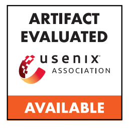
1. Instrument the subset of 125 TensorFlow and 105 PyTorch kernels we performed our experiment on (depending on the chosen framework) and re-compile the target framework.
2. Execute `IvySyn` on the target kernels, limiting the fuzzed kernels if `--nkernels` was specified.
3. Run `DocTer` on the same subset of fuzzed kernels.
4. Output a summary of the results.

Results: The `compare-fuzzers.sh` script will display a summary of the results. A new directory with the `IvySyn` results will be created at `results/<framework>/docter_comp/` (where `<framework>` is `pytorch` or `tensorflow`), and will contain files similar to the ones mentioned in the Atheris experiment (i.e., `results.csv`, `total_time.txt`, and `fuzzer_logs_dir.txt`).

The corresponding CSV that contains similar entries for the `DocTer` run can be found at `comparisons/docter_comp_fuzzed_<framework>.csv` (where `<framework>` is `pytorch` or `tensorflow`). The raw `DocTer` logs can be found in the `DocTer` Docker container at `/home/workdir/<framework>`. To connect to the `DocTer` Docker container, in order to manually inspect the logs, run `docker exec -it docter-instance /bin/bash`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: Spying through your voice assistants: Realistic voice command fingerprinting (#462)

Dilawer Ahmed
North Carolina State University

Aafaq Sabir
North Carolina State University

Anupam Das
North Carolina State University

A Artifact Appendix

A.1 Abstract

The artifact, written in Python, contains the code and dataset for the major part of our work on fingerprinting voice commands across the three most popular voice assistants. The artifact contains 7 datasets that we used along with code for our data collection process which we used to collect the datasets and the code we used to process the data and get the results that we presented in our paper.

A.2 Description & Requirements

A.2.1 Environment setup

A.2.2 Security, privacy, and ethical concerns

There are no major security or privacy considerations except that you will be collecting network traffic from the routers which can include traffic from other devices connected to this network. You would need to be careful about who has access to this dataset

A.2.3 How to access

The code can be accessed from `https://github.com/dilawer11/va-fingerprinting/tree/0dd1ec3a65e843e366e81ffd29721593bc8043b1`.
The datasets can be downloaded from `https://privacy-datahub.csc.ncsu.edu/publication/ahmed-usenix-2023/`. An archived version through Zenodo is available at `https://doi.org/10.5281/zenodo.8037394`

A.2.4 Hardware dependencies

For analysis following specifications are minimum:

- CPU: (x86-64) 8 cores
- RAM: 16 GB
- OS: Debian Linux (Ubuntu recommended)

For data collection following specifications are recommended:

- CPU: (x86-64) 4 cores
- RAM: 8 GB
- OS: Debian Linux (Ubuntu recommended)
- Speaker: Any Stereo PC Speakers
- Routers: 2x OpenWRT (ssh-capable) Linux routers
- Voice Assistant: Alexa, Google Assistant or Siri supported smart speaker

A.2.5 Software dependencies

You can either use the Docker image (provided at `https://privacy-datahub.csc.ncsu.edu/publication/ahmed-usenix-2023/` or setup your own environment (using either the Dockerfile or manually installing all dependencies). The github repository contains the more detailed instructions on how to setup the environment manually and on Docker. The software dependencies which can be installed using `apt-get` are:

- python3.9 (*analysis*, *data collection*)
- python3-pip (*analysis*, *data collection*)
- tshark (*analysis*)
- tcpdump (*router*, *data collection*)
- dumpcap (*data collection*)

The following python packages also need to be installed for analysis: `autogluon.tabular[fastai, xgboost, ray, lightgbm]`, `pandas`, `scikit-learn`, `matplotlib`, `tqdm`, `seaborn`, `datetime`, `plotly`, `jupyterlab`, `python-dotenv`. For data collection the following packages need to be installed: `python-dotenv`, `google-cloud-texttospeech`, `gtts`, `tqdm`, `selenium`.

A.2.6 Benchmarks

The following datasets are available at the URLs provided above:

- simple_50_alexa
- simple_50_siri
- simple_50_google
- simple_100_alexa
- skills_100_alexa
- stream_15_alexa
- mix_100_alexa

The results presented in Tables 2 and 3 of our paper are mostly computed on these datasets and can be used as benchmarks for the setup.

A.3 Set-up

The following subsections describe how to complete the setup

A.3.1 Docker Setup

To set up via the provided Docker container you need to download it from the given link. After you have downloaded it use the following command to load the docker container image into your Docker system. Replace the `path/to/dockerimage` with the absolute path of the downloaded docker image

```
$ docker load < path/to/dockerimage
```

After you have loaded the Docker image you can then create and start a container by using the following command from inside the root repository directory

```
$ docker run -it
  -v $('pwd'):/va-fingerprinting
  vafingerprint
```

The command above starts the container and mounts the repository directory (including source and data) to the container to any changes are reflected and persistent in local storage.

A.3.2 Manual Installation

To manually install the software dependencies use the following commands:

```
$ apt-get update && apt-get install
  packagenames
```

Replace the *packagenames* with the packages that need to be installed based on the type of setup i.e. *analysis*, *data collection*. The *analysis* packages need to be installed on the machine where the raw data is input (e.g our datasets) and results need to be computed. The *data collection* packages need to be installed on the machine where you want to dump the data and want to run the speaker control script for data collection. The *router* packages need to be installed on the router which you want to capture traffic from. In this appendix, we are primarily focused on the *analysis* part.

To install the Python packages use the following commands:

```
$ pip3 install -r path/to/requirements.txt
```

Use the requirements file from the *setup* directory in the GitHub repo. The `requirements_analysis.txt` file contains the pip dependencies for the analysis and the `requirements_collection.txt` contains the pip dependencies for data collection.

To download and setup the datasets you can download them from the URL mentioned above. Then unzip them in the data directory of the cloned GitHub repostory. The detailed instructions can be found in the `data/README.md` file.

A.3.3 Basic Test

After the setup is complete and you have set up the dependencies and the *test* dataset. Use the following command from the root of the cloned GitHub repository to run a basic test that dependencies are properly installed (for the analysis).

```
$ sh src/scripts/test.sh
```

If the script runs without any errors and outputs the results then everything should work fine.

A.4 Evaluation workflow

We provide the details and steps necessary to recompute the results of activity detection and invocation detection which are a major portion of our work in realistic voice activity fingerprinting.

A.4.1 Major Claims

(C1): We achieved more than 98% accuracy while fingerprinting invocations on voice assistants. This is described in Section 4.2 of our paper and in Table 2.

(C2): We improved the state-of-the-art accuracy in voice command fingerprinting on a variety of datasets we collected. The results are presented in Section 5.5-5.7 and in Table 3 of our paper

A.4.2 Experiments

To re-compute the results for the datasets make sure you have already set up the datasets as described previously and conducted a simple test.

A common step to both experiments is to process the PCAPs to remove the unnecessary information and convert to CSV format for easier processing with Python and Pandas. To achieve this you can use the script provided as such

```
$ python3 src/scripts/PCAP2CSV.py -i
  path/to/data/dataset
```

(E1): Invocation Detection [10 human-minutes + 3 compute-hour]: This experiment seeks to evaluate the invocation detection performance on the datasets. The results, once computed, should look similar to Table 2 of our paper

How to: Running this experiment is fairly simple, however, it might take long to actually compute the results. We will initially need to convert the PCAP files to CSV and then create sliding windows based on *invoke records*. Then we will extract features from the windows and finally train the model. As a final step, we can aggregate results across datasets and create a table

Preparation: After you have completed the setup mentioned above. You can then use the `src/scripts/PCAP2CSV.py` script to convert PCAP files to CSV. If you have already converted (for another experiment) you should skip this step.

Execution: After the CSV files are created you would then, for each dataset, need to create windows, extract features and train models. We have provided a default short hand argument to the scripts which can automatically do this. For each dataset you can compute the results using the following command

```
$ python3 src/scripts/InvocationDetection.py
auto-train -i path/to/data/dataset
```

For further options and information you can either use the `-h` option or see the GitHub README file.

Results: To create the table with all datasets you can run the following script with the options as follows

```
$ python3 src/scripts/PostProcessing.py
id-table -d path/to/data
```

This will display a table with result metrics across different models and datasets similar to Table 2 of the paper

(E2): Activity Detection [10 human-minutes + 3 compute-hour]: This experiment is to evaluate the performance of our voice activity fingerprinting method on the datasets we collected. This experiment will reproduce the results in Table 3 of our paper

How to: Similar to last experiment we will need the CSV files (converted from PCAPs) from which windows are created based on *invoke records*. We then extract features and train the AutoGluon model.

Preparation: You can skip this step if the PCAPs for the dataset are already converted to CSV. Otherwise, run the following script to convert PCAP to CSVs.

```
$ python3 src/scripts/PCAP2CSV.py -i
path/to/data/dataset
```

Execution: To create windows, extract features and train the AutoGluon model you can use the following command.

```
$ python3 src/scripts/ActivityDetection.py
auto-train -i path/to/data/dataset
```

Alternatively, instead of the `auto-train` option you can pass the `windows`, `features`, `train` arguments to the script (in this order) to complete this experient.

More information on this script is provided in the GitHub README file

Results: To create the table with all datasets you can run the following script with the options as follows

```
$ python3 src/scripts/PostProcessing.py
ad-table -d path/to/data
```

This will display a table with result metrics across different datasets similar to Table 3 of the paper

A.5 Notes on Reusability

Our artifact was designed to be a prototype and hence is not nearly optimized enough for production. To help with reusing and adding additional functionality we have tried to make the code easier to read and functionality separated into modules. For the extension of a particular module, only that module can be extended while keeping the rest of the code base largely untouched.

The `src/iotpackage/Utils.py` file contains some utility functions that can help with extending the functionality and also help future researchers by optimizing some workflows. For example, the `DataFrame2LatexTable` can convert a Pandas dataframe into a latex table format to save time.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Pushed by Accident: A Mixed-Methods Study on Strategies of Handling Secret Information in Source Code Repositories

Alexander Krause, CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

Our paper contains a mixed-methods study, a survey with developers on their experiences handling secrets in source code, and an interview study with developers that experienced code secret leakage in the past. In order to support our paper and make it more useful for the readers, we provide all the necessary artifacts available in a replication package. The replication package includes: 1. The full survey and interview recruitment materials (including Upwork post and invitation, as well as GitHub invite messages). 2. The survey screening questions and interview pre-survey questionnaire. 3. The survey and interview consent form. 4. The survey questionnaire and interview guide. 5. The survey and interview codebook. 6. The background section on version control.

A.2 Description & Requirements

In this section, we provide the descriptions of all the applicable subsections for our use case (i.e., "artifacts available" badge).

A.2.1 Security, privacy, and ethical concerns

The data we provide is not harmful to viewers. All data we provide has been anonymized to protect our participants' privacy.

A.2.2 How to access

Our artifact can be accessed using the following URL: https://doi.org/10.25835/xfc2h3pg.

The complete replication package can be downloaded as a .zip file through the provided link. This replication package is hosted on the Research Data Repository of our university (data.uni-hannover.de).

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

None

A.2.5 Benchmarks

None

A.3 Set-up

Our artifacts can be downloaded as a .zip file from the URL we provided in the section A.2.2. The file contains the following seven .pdf files:

1. **README.md** This .md file contains a list of all provided resources.
2. **index.html** This .html file is used to render the replication package as a website.
3. **background.md** This .md file contains an additional background section on version control, source code platforms, and secret information.
4. **interviews/codebook.txt:** This .txt file contains the high level codebook, including counts of the codes that we assigned to participants' answers.
5. **interviews/consentform.html:** This .pdf file contains the consent form we used in the pre-survey.
6. **interviews/Interview_Guide.pdf** This .pdf file contains the semi-structured interview guide we used in our interview study.
7. **interviews/invite_mail.md** This .md file contains our recruitment material. We sent this invite mail text to developers from GitHub.
8. **interview/pre-survey.md** This .md file contains the pre-survey we used to collect demographics and screen participants.
9. **survey/codebook.md** This .md file contains the high level codes that emerged from the survey to identify code secret leakage prevention and remediation approaches.

10. **survey/consentform.html** This .html file contains the consent forms used for participants recruited from Upwork and GitHub.

11. **survey/github_invite_mail.md** This .md file contains our recruitment material. We sent this invite mail text to developers from GitHub.

12. **survey/survey.md** This .md file contains the survey questionnaire.

13. **survey/survey-matrix.png** This .png file contains the survey question on participants' threat models because it could not be displayed in a .md file.

14. **survey/upwork_recruitment_material.md** This .md file contains all recruitment materials used to recruit developers from Upwork.

A.3.1 Installation

N/A

A.3.2 Basic Test

N/A

A.4 Notes on Reusability

To replicate the study, we suggest using the survey questionnaire excluding the open-ended questions that did not work well; we detailed on that in the paper). When replicating the interview study, we suggest using our template of the interview guide that we provide within this artifact.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: DeResistor: Toward Detection-Resistant Probing for Evasion of Internet Censorship

Abderrahmen Amich, Birhanu Eshete, Vinod Yegneswaran and Nguyen Phong Hoang

.1 Abstract

DeResistor is a research project that provides a system extension to protect Probing for Evasion of Internet Censorship from detection. Specifically, it offers IP address protection for internet users that are running automated tools for censorship measurments and evasion (e.g. Geneva).

In this artifact, we provide an instance of DeResistor implemented on top of Geneva code: https://github.com/Kkevsterrr/geneva. DeResistor leverages Machine Learning techniques to model a censor-side flow-level detector and use it to guide Geneva genetic evolution towards more detection-resilient evasion strategies. Additionally, DeResistor introduces guided-pauses of censorship evasion attempts and interleaving them with normal user-driven network activity to confuse IP-level detection.

.2 Description & Requirements

.2.1 Security, privacy, and ethical concerns

For Docker experiments, evaluators have no risk to execute this artifact. However, real-world experiments are intended to test DeResistor in censored regimes (e.g., China, India, etc). Evaluators should not try to reproduce these experiments in one of these countries using there personal machines. Instead, they need to get access to vantage points that are remotely controllable and do not require the involvement of real user credentials that may identify individuals.

.2.2 How to access

Our artifact can be accessed via https://github.com/um-dsp/DeResistor.

.2.3 Hardware dependencies

No specific Hardware dependencies are needed for docker experiments. However, real-world experiments have to be performed in censored regimes. If needed, we can provide you with `ssh` access to one of our vantage points in China.

.2.4 Software dependencies

DeResistor has been developed and tested on Ubuntu. However, it should support Centos or Debian-based systems. Similar to Geneva, due to limitations of netfilter and raw sockets, this code does not work on OS X or Windows at this time and requires python3.6. To reproduce In-situ experiments, docker has to be properly installed.

.2.5 Benchmarks

None

.3 Set-up

.3.1 Installation

- Install netfilterqueue dependencies:

```
$ sudo apt-get install build-essential
python-dev libnetfilter-queue-dev
libffi-dev libssl-dev iptables python3-pip
```

- Create a new python3.6 environment and install Python dependencies:

```
$ python3 -m pip install -r requirements.txt
```

- **If needed**, for Debian 10 systems, you can install netfilterqueue directly from Github:

```
$ sudo python3 -m pip install --upgrade -U
git+https://github.com/kti/python-
netfilterqueue
```

- **If needed**, on Arch systems, you can make liblibc.a available for netfilterqueue:

```
$ sudo ln -s -f /usr/lib64/libc.a
/usr/lib64/liblibc.a
```

- After you make sure you install and run docker on your system use the dockerfile provided in */docker* folder to build the base image:

```
$ sudo docker build -t
base:latest -f docker/Dockerfile .
```

.3.2 Basic Test

- to manually run/inspect the docker image to explore the image, run:

```
$ sudo docker run -it base
```

- To check that all docker containers and harpoon are running correctly, We can run the genetic algorithm with small number of *Individuals* (`--population`) and *Generations* (`--generation`):

```
$ sudo /path/to/python_environment/bin/python
evolve.py --censor censor3 --server
forbidden.org -- log debug --workers 1
--runs 1 --population 5 --
generation 1 --jump 1
```

[To add: Output description]

.4 Evaluation workflow

.4.1 Major Claims

(C1): DeResistor offers detection-resilience to Geneva's probing traffic. This is discussed in §6.2. We record the flow-level detection rate and IP-level detection result of DeResistor and Geneva in Table 1.

(C2): Effectiveness of DeResistor to produce working strategies that can evade the censor.

.4.2 Experiments

Experiments (E1) and (E2) reproduce results related respectively to major claims (C1) and (C2). If the evaluators do not have access to controlled vantage points in China, India or Kazakhstan, in order to reproduce results in the first 3 rows of Table 1 and working strategies in Table2, they can rely on the docker experiments to reproduce results in rows 4-8 in Table 1 (addresses (C1)) and monitor the traffic logs of newly generated strategies that they have found against the mock censors to address (C2). If evaluators are able to generate strategies against real-world censors (e.g., startegies in Table2 in the paper), they can test those strategies directly against the censor using Geneva engine.

(E1): [Testing Detection-Resilience] [30 human-minutes + 1 to hours compute-hour according to the considered number of individuals and generations. 2GB disk should be sufficient to store all the results]:

We run DeResistor vs. one of the 11 mock censors, while enabling real-time detection using `--real-time-detection`. During execution time, we keep track of the *Detection Rate* after every strategy evaluation, displayed to the screen. Additionally, we check whether the real-time detector blocks the IP address and stops Geneva/DeResistor training.

How to: We start by running Geneva without DeResistor protection, using `--Geneva` to test its flow-level and IP-level detection. Then, we run DeResistor and compare the results between both runs. We provide more detailed description later in the *Execution* paragraph.

Preparation: To prepare this experiment, we need to make sure all Setup points in §3 are taken care of.
You can ignore the following if you are running only docker experiment: – For real-world experiments, You need to have internet access. For experiments in china, we need to first bypass DNS poisoning as exaplained in §6.1 in the paper, paragraph 3. For Linux systems we can point the URL `hrw.org` to its correct IP `23.185.0.2`, by adding the line `23.185.0.2 www.hrw.org` to `/etc/hosts`. Similarly, to run this experiment in Kazakhstan we need to add `93.184.216.34 www.youporn.com` to `/etc/hosts` file.

Execution:
- Running Geneva without DeResistor protection.

```
$ sudo /path/to/python_environment
/bin/python evolve.py --censor censor1
--server forbidden.org -- log debug
--workers 1 --runs 1 --population 200
--generation 10 --Geneva
--real-time-detection
```

Before performing a second run make sure all docker containers related to Geneva are killed using: `"sudo docker kill $(sudo docker ps -q)"`. The execution automatically stops after testing only two flows which is a evidence of IP-level detection (reported in Table1 in the paper). To reproduce Geneva's flow-level detection value, we need to complete all Geneva training by disabling `--real-time-detection` (remove it from the command) to avoid early blocking of Geneva. After every iteration, we observe how *The detection Rate* value changes to reach $\approx 99\%$ as reported in Table1 in the paper rows 4-8.

For real-world experiments (e.g., China):

```
$ sudo /path/to/python_environment/bin
/python evolve.py --external-server
--server www.hrw.org --test-type http
--log debug --workers 1 --runs 1
--population 500 --generation 20
--real-time-detection
--local-model rfc_gfw.joblib
--censor-model rfc_gfw2.joblib
--Geneva
```

`www.hrw.org` is not necessarily censored outside of China. For India, we can use `bannedthought.net, xnxx.com, vidwatch.me` and for Kazakhstan, we

can use `youporn.com`. For india and Kazakhstan experiments we can use more appropriate models provided in `/ML detectors` folder, respectively called `rfc_india.joblib` and `rfc_kz.joblib` to update `-local-model` and `-censor-model`. Before performing a second run make sure you reset the iptables: "`sudo iptables -F`".

- Running DeResistor:

```
$ sudo /path/to/python_environment
/bin/python evolve.py --censor censor1
--server forbidden.org -- log debug
--workers 1 --runs 1 --population 200
--generation 10 --jump 1
--real-time-detection
```

Before performing a second run make sure all docker containers related to Geneva are killed using: "`sudo docker kill $(sudo docker ps -q)`". According to Table 1 in the paper, DeResistor should be able to complete its training without IP-level detection. Similar to the previous run, we also track the changes occurred on the *Detection Rate* as it regularly displays in the console.

For real-world experiments, we can use the same command as before without `--Geneva` and adding *--jump 1*. Similarly, we need to flush the iptables after each run with: "`sudo iptables -F`".

To reproduce results against all 11 mock censors, we can run the same commands using a different censor (e.g., `--censor censor2`, etc).

Results: Results of every run are stored in the folder `/trials/[date-and-time-of-execution]`. It contains , network traces in `/packets` and their csv counterparts after features extraction in `/csv` (using only the client-side packets). All generated strategies are located in the final `hall.txt` file (e.g., `/generations/hall9.txt`) with their fitness values. Strategies with the highest fitness values are most likely to evade the censor.

As illustrated before, detection resilience results should be observable during run-time. Particularly, if the program raises a `detection` exception, then an IP detection is observed. Additionally, the flow-level detection rate is regularly displayed during run-time as `Detection Rate`. We also store the flow-level detection results of all flows in `preds.csv`. We can re-compute the final value of the flow-level detection rate by counting the percentage of zeroes (0: *detected as Geneva flow*) compared to all flow-level predictions.

(E2): [Evasion Effectiveness:] [1 to 2 human-hours + 0 compute-hour. 2GB disk should be sufficient to store all the results]: In this experiment, we leverage results stored in previous executions of DeResistor and Geneva to select strategies that are effective for censorship evasion.

How to: We inspect collected strategies of DeResistor in `/generations/hall[final].txt` located in the folder corresponding to the DeResistor run. For real censors (e.g., China), we can evaluate the effectiveness of each strategy against the censor using Geneva startegy-testing Engine. More details about testing a strategy with Geneva engine is provided in *Execution*. You can also refer to Geneva documentation related to how to run a strategy in `https://geneva.readthedocs.io/en/latest/intro/gettingstarted.html#running-a-strategy`. For startegy generated against mock censors, We cannot evaluate them using Geneva engine. Instead we can manually inspects the logs of the most fit strategies and check whether the client finally had access to the forbidden server (evaded censorship). We note that, the most fit startegies are the ones that have the highest fitness values which can be negative in case we run DeResistor.

Preparation: To perform this experiment we need to generate appropriate results files using commands described in (E1). Similar to (E1), to test strategies against real censors (e.g. GFW), you need access to controlled vantage points in the desired country.

Execution:

- **Evaluate a strategy against real censors:** To evaluate a strategy that you selected, you first need to run Geneva engine to apply the strategy later on using:

```
$ sudo /path/to/python_environment
/bin/python engine.py --server-port 80
--strategy "[your-strategy]" --log debug
```

In a separate console (e.g. terminal), you can perform `curl` commands to attempt connections to a censored website. For instance, in china you can try:

```
$ curl -L --no-keepalive --local-port
[random-port-number] --connect-to
::23.185.0.2: 'http://hrw.org' -D -
```

The port number has to be changed across runs to avoid Resisdual censorship performed by GFW (ths is discussed in the paper in §6.1 paragraph 3.

To automate the strategy evaluation process, we provide a script in `/test.py` that evaluates a list of strategies 30 times and stores their success rate in `success_rate.txt`. Using this script, we can reproduce results in Table2.

Results:

.5 Notes on Reusability

[Optional] This section is meant to optionally share additional information on how to use your artifact beyond the research presented in your paper. In fact, a broader objective of an artifact evaluation is to help you make your research reusable by others.

You can include in this section any sort of instruction that you believe would help others re-use your artifact, like, for example, scaling down/up certain components of your artifact, working on different kinds of input or data-set, customizing the behavior replacing a specific module/algorithm, etc.

USENIX'23 Artifact Appendix: How the Great Firewall of China Detects and Blocks Fully Encrypted Traffic

Mingshi Wu
GFW Report

Jackson Sippe
University of Colorado Boulder

Danesh Sivakumar
University of Maryland

Jack Burg
University of Maryland

Peter Anderson
Independent researcher

Xiaokang Wang
V2Ray Project

Kevin Bock
University of Maryland

Amir Houmansadr
University of Massachusetts Amherst

Dave Levin
University of Maryland

Eric Wustrow
University of Colorado Boulder

April 20, 2023, v1.0

A Artifact Appendix

A.1 Abstract

As introduced in the paper, we measure and characterize the GFW's new system for censoring fully encrypted traffic. We find that, instead of directly defining what fully encrypted traffic is, the censor applies crude but efficient heuristics to exempt traffic that is unlikely to be fully encrypted traffic; it then blocks the remaining non-exempted traffic. These heuristics are based on the fingerprints of common protocols, the fraction of set bits, and the number, fraction, and position of printable ASCII characters. In this artifact, we provide the data and code to support our major claims. Additionally, we conducted a follow-up experiment to confirm that the GFW had stopped blocking fully encrypted traffic dynamically as of Wednesday, March 15, 2023.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

As detailed in the ethics section of our paper, our measurement tools have already employed the best practices by default.

A.2.2 How to access

The artifact is available on GitHub: `https://github.com/gfw-report/usenixsecurity23-artifact/commit/ad45e63b4a708bda5ce39f48fc25ebbae013ee51`.

A.2.3 Hardware dependencies

We have prepared a VPS in China and a VPS in the US, on which the AE reviewers can perform remote experiments. The VPS in China is located in AlibabaCloud Beijing Datacenter (AS37963), which uses one core of Intel Xeon Platinum 8163 and 1GB RAM. The VPS in the US is located in DigitalOcean San Francisco Datacenter (AS14061), which uses one core of Intel DO-Regular and 1GB RAM. To SSH into the VPSes, reviewers need to install the provided credentials, as detailed in `artifacts/setup-vps/README.md`.

For people other than the AE reviewers who want to perform experiments, they need to prepare a VPS in China and a VPS outside of China themselves.

A.2.4 Software dependencies

The VPS in China runs Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-56-generic x86_64). The VPS in the US runs Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-88-generic x86_64). The following tools and environment are required:

- GNU make utility
- Go 1.17+
- Python 3.8+

In particular, the two VPSes do not require Go environment. As detailed in `README`, reviewers may compile the Go code on their local machines and copy the binaries to the VPSes.

A.2.5 Benchmarks

None.

A.3 Set-up

As detailed in `artifacts/setup-vps/README.md`, we have prepared a VPS in China and a VPS in the US, on which the AE reviewers can perform remote experiments. Since we have installed the dependencies and required tools on the VPSes, all reviewers need to do is to use the provided credentials to SSH into the VPSes to run experiments. We also provide one-click scripts, allowing reviewers to initialize test-ready VPSes themselves. Be very cautious that running setup scripts will disrupt other reviewers' ongoing experiments.

A.3.1 Installation

- Set up the Go environment: https://go.dev/dl/.
- Retrieve the artifacts: `git clone https://github.com/gfw-report/usenixsecurity23-artifact`.
- Compile the client-side experiment tools and install them to the CN VPS: `cd artifacts/setup-vps && ./setup-client/to_alibaba_server.sh`.
- Compile the sink server and install it to the US VPS: `cd artifacts/setup-vps && ./setup-server/to_digitalocean_server.sh`.

A.3.2 Basic Test

First login to the VPS in China using the provided credentials: `ssh usenix-ae-client-china`.

Then send random probes to the port 80 of the $*serverIP* with the following command: `echo $serverIP | ./utils/affected-norand -p 80 -log /dev/null`.

The program outputs in CSV format. If the `affected` field is `True`, the blocking is successfully triggered. If the `affected` field is `False`, the blocking is not triggered.

A.4 Evaluation workflow

A.4.1 Major Claims

(C0): As of Tuesday, March 7, 2023, the GFW was still blocking random traffic. This is supported by the experiment (E0).

(C1): As of Wednesday, March 15, 2023, the GFW had stopped blocking random traffic dynamically. This is supported by the experiment (E1).

(C2): The GFW exempts a connection if the first TCP packet `pkt` satisfies: $\frac{popcount(\texttt{pkt})}{len(\texttt{pkt})} \leq 3.4$ or $\frac{popcount(\texttt{pkt})}{len(\texttt{pkt})} \geq 4.6$. This is supported by the experiment (E2) described in Section 4.1 of the paper. This detection rule is introduced in Algorithm 1 (Ex1) and illustrated in Figure 1.d.

(C3): The GFW exempts a connection if the first six (or more) bytes of the first TCP data packet `pkt` are [`0x20`, `0x7e`]. This is supported by the experiment (E3) described in Section 4.2 of the paper. This detection rule is introduced in Algorithm 1 (Ex2) and illustrated in Figure 1.a.

(C4): The GFW exempts a connection if the first TCP data packet `pkt` has more than 50% of `pkt`'s bytes in [`0x20`, `0x7e`]. This is supported by the experiment (E4) described in Section 4.2 of the paper. This detection rule is introduced in Algorithm 1 (Ex3) and illustrated in Figure 1.b.

(C5): The GFW exempts a connection if the first TCP data packet `pkt` has more than 20 contiguous bytes in [`0x20`, `0x7e`]. This is supported by the experiment (E5) described in Section 4.2 of the paper. This detection rule is introduced in Algorithm 1 (Ex4) and illustrated in Figure 1.c.

(C6): The GFW exempts a connection if the first few bytes of the first TCP data packet `pkt` match the protocol fingerprint for TLS or HTTP. This is supported by the experiment (E6) described in Section 4.3 of the paper. This detection rule is introduced in Algorithm 1 (Ex5) and illustrated in Figure 1.e.

A.4.2 Experiments

Experiment E0 tests if the GFW still blocks random traffic dynamically by sending random probes from China to a single server in US. If the reviewers can trigger the blocking in experiment E0, they can proceed to test experiments (E1-E6); otherwise, they only need to run experiment E1 to further confirm the GFW has stopped blocking random traffic.

Experiments E0 and E2-E6 follow the same testing logic: we craft different payloads that will be either exempted or blocked by the GFW. We send them, from VPS in China to the VPS in US, through the GFW, to observe whether each payload can trigger the blocking or not. If the blocking or exemption results match with what Algorithm 1 predicts, it shows that the GFW is indeed using the detection rule described in Algorithm 1. For reviewers' convenience, we implement Algorithm 1 as `utils/detect.py`, which reads a list of given payloads in hex format and writes if each payload will be exempted by any of the detection rules.

Unless explicitly specified, all operations described below are performed on the VPS in China. And $*serverIP* corresponds to the IP address of the VPS in US.

(E0): *[test-random] [5 human-minutes + 5 compute-minutes]:* This experiment tests if the GFW blocks random traffic. It also familiarizes the reviewers with the testing tools and logic.

Preparation: `cd artifacts`

Execution: Execute this command to generate a random probe of 200 bytes and check if Algorithm 1 thinks it will be blocked by the GFW or not: `head -c200 /dev/urandom | xxd -p -c256 | tee random.txt | ./utils/detect.py`.

Execute this command to repeatedly send the probe to the same port of the US server: `cat random.txt | ./utils/affected-payload -host $serverIP -p $serverPort`.

Results: If the `affected` field in the program output is `True`, it means that your generated probe has triggered the blocking by the GFW. This result should be consistent with what `detect.py` predicts.

(E1): *[confirm-ceased-blocking] [15 human-minutes + 2 compute-days]:* This experiment tests if the GFW has stopped blocking random traffic dynamically. Specifically, it performs an Internet scan from a VPS machine

in China to all 142,827 IP addresses that were previously marked affected as of August 22, 2022. For each IP address, The test uses two types of probes: 50-bytes of random data and 50-bytes of zero (as the control group). For each type of probe, the program makes up to 25 connections, and when five consecutive connections to an IP address fail, the program mark it as possibly affected. We then remove any probes that were also marked as affected in the control group to rule out most of the false positives due to network failure rather than censorship.
Preparation: `cd ceased-dynamic-blocking`
Execution: Execute this command to perform the 2-day test: `make`.
One then compares the results between the two tests using two different types of probes, to find the IP addresses that are marked as blocked (`true`) in the random probe test but marked as not blocked (`false`) in the zero probe test: `make compare`.
Results: The number of affected IP addresses should be as low as around six thousand out of the 142,827 IP addresses tested. One can further test these IP addresses recursively to make sure they are all false positives.

(E2): *[test-entropy] [30 human-minutes + 30 compute-minutes]:* This experiment test if the GFW exempts a connection whose first TCP packet `pkt` satisfies: $\frac{popcount(\texttt{pkt})}{len(\texttt{pkt})} \leq 3.4$ or $\frac{popcount(\texttt{pkt})}{len(\texttt{pkt})} \geq 4.6$.
Preparation: `cd test-entropy`
Execution: Execute this command to generate a list of payloads: `make`. As shown in the output of `detect.py`, some of the probes will be exempted by the GFW; while other probes will not.
Use this command to test if each probe is exempted by the GFW: `make test`.
Use this to compare the blocking results against the results predicted by the `detect.py`: `make compare`.
Results: The testing results should match with what `detect.py` predicts.

(E3): *[test-printable-prefixes] [15 human-minutes + 30 compute-minutes]:* This experiment tests if the GFW exempts a connection whose first six bytes are printable characters.
Preparation: `cd test-printable-prefixes`
Execution: Execute this command to generate a list of payloads: `make`. As shown in the output of `detect.py`, some of the probes will be exempted by the GFW; while other probes will not.
Use this command to test if each probe is exempted by the GFW: `make test`.
Use this to compare the blocking results against the results predicted by the `detect.py`: `make compare`.
Results: The testing results should match with what `detect.py` predicts.

(E4): *[test-printable-fraction] [15 human-minutes + 30 compute-minutes]:* This experiment tests if the GFW exempts a connection whose first TCP data packet has more than 50% of printable characters.
Preparation: `cd test-printable-fraction`
Execution: Execute this command to generate a list of payloads: `make`. As shown in the output of `detect.py`, some of the probes will be exempted by the GFW; while other probes will not.
Use this command to test if each probe is exempted by the GFW: `make test`.
Use this to compare the blocking results against the results predicted by the `detect.py`: `make compare`.
Results: The testing results should match with what `detect.py` predicts.

(E5): *[test-printable-longest-run] [15 human-minutes + 15 compute-minutes]:* This experiment tests if the GFW exempts a connection whose first TCP data packet has more than 20 bytes of contiguous printable characters.
Preparation: `cd test-printable-longest-run`
Execution: Execute this command to generate a list of payloads: `make`. As shown in the output of `detect.py`, some of the probes will be exempted by the GFW; while other probes will not.
Use this command to test if each probe is exempted by the GFW: `make test`.
Use this to compare the blocking results against the results predicted by the `detect.py`: `make compare`.
Results: The testing results should match with what `detect.py` predicts.

(E6): *[test-protocol-fingerprints] [15 human-minutes + 2 compute-hours]:* This experiment tests if the GFW exempts traffic that matches the protocol fingerprints.
Preparation: `cd test-protocol-fingerprints`
Execution: Execute this command to generate a list of payloads: `make`. As shown in the output of `detect.py`, some of the probes start with a fingerprint that will be exempted by the GFW; while other probes do not.
Use this command to test if each probe is exempted by the GFW: `make test`.
Use this to compare the blocking results against the results predicted by the `detect.py`: `make compare`.
Results: The testing results should match with what `detect.py` predicts.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: ARI: Attestation of Real-time Mission Execution Integrity

Jinwen Wang*, Yujie Wang*, Ao Li*, Yang Xiao†,
Ruide Zhang‡, Wenjing Lou‡, Y. Thomas Hou ‡, Ning Zhang*

* Washington University in St. Louis
† University of Kentucky
‡ Virginia Polytechnic Institute and State University

A Artifact Appendix

A.1 Abstract

This artifact includes all the source code of Attestation of Real-time Mission Execution Integrity (ARI), a policy-guided real-time mission execution integrity attestation framework. It mainly contains three key component. A compartmentalization mechanism, runtime mission information measurement mechanism, and a mission integrity verification engine. For this artifact evaluation, we will illustrate ARI 's functionality by employing an example policy to verify the execution integrity of a copter flight mission. Specifically, the policy will utilize controller-based compartmentalization, with the attitude controller as the critical compartment. We aim to streamline the Artifact Evaluation (AE) process by providing a pre-configured virtual machine (VM) with all necessary dependencies. Additionally, we provide a hardware flight controller accessible via SSH. For further convenience, remote VM access is made available through Teamviewer.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Carrying out the Artifact Evaluation (AE) for ARI does not pose any security, privacy, or ethical concerns. All AE tasks are carried out within the Virtual Machine (VM) and the remote hardware flight controller, which are host on remote machine. This ensures that the AE process remains isolated and does not interact with the reviewer's personal or sensitive code/data.

A.2.2 How to access

Source Code: https://github.com/WUSTL-CSPL/ARI

A.2.3 Hardware dependencies

To properly evaluate our artifact, please ensure that your host system has stable network to access the remote VM through Teamviewer. Furthermore, the quadcopter mission runs in a simulation environment on a Raspberry Pi 3, equipped with a Navio2 daughter board. To access the remote Raspberry Pi 3 with the Navio2 daughter board, you can utilize the following command and corresponding password in remote VM:

A.2.4 Software dependencies

To undertake an effective evaluation of the ARI artifact, it is recommended to use the Ubuntu 16.04 LTS operating system. Our customized compiler is built upon LLVM 3.9.0, while the software quadcopter flight controller utilizes ArduPilot 3.9.0. The underlying verification engine has been developed based on Capstone 4.0.2, and employs the Black2s hash algorithm. The cross compiler is gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf. The OS on Raspberry Pi3 is Linux navio 4.14.95-emlid-v7+.

A.2.5 Benchmarks

None

A.3 Set-up

The setup section is intended for reviewers who wish to construct the system from scratch. We have made available a well-configured remote VM accessible via Teamviewer. Thus, reviewer can skip section 3.

A.3.1 Installation

Dependencies Installation: Install the dependencies for the three key components of ARI by using the following commands.
$ sudo apt-get install python-pip
$ python -m pip install –upgrade "pip < 19.2"

$ sudo python -m pip install –upgrade "pip < 21.0"
$ sudo apt-get install clang-3.9 && cd /usr/bin && sudo ln ./clang-3.9 ./clang && sudo ln ./clang++-3.9 ./clang++
$ sudo apt install git cmake build-essential make texinfo bison flex ninja-build ncurses-dev texlive-full binutils-dev python-networkx python-matplotlib python-pygraphviz python-serial
$ sudo pip2 -q install -U future lxml pymavlink MAVProxy
$ pip install pydotplus python-louvain bitarray capstone enum34 pyelftools pyblake2
$ wget http://launchpadlibrarian.net/356067403/gcc-5-aarch64-linux-gnu_5.4.0-6ubuntu1~16.04.9cross1_amd64.deb
$ sudo dpkg -i ./gcc-5-aarch64-linux-gnu_5.4.0-6ubuntu1 ~16.04.9cross1_amd64.deb

Customized LLVM Installation: *ari_dir* is the root directory of ARI project. In VM *ari_dir* is */home/ari-new-ae/conattest*
$ git clone https://github.com/WUSTL-CSPL/ARI.git
$ cd ./conattestllvm
$ chmod +x ./compiler_for_1st_part.sh
$ ./compiler_for_1st_part.sh
$ mkdir build && cd ./build
$ cmake -DLLVM_ENABLE_ASSERTIONS=OFF ..
$ make
$ echo 'export PATH=$PATH:ari_dir/ conattestllvm/build/bin' ≫ ~/.bashrc
$ source ~/.bashrc

ArduPilot Installation:
$ echo 'export PATH=$PATH:ari_dir/ gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf/bin' » ~/.bashrc
$ source ~/.bashrc
Download the Pi3 image in following link. Decompress it into *pi3_img_dir*.https://drive.google.com/drive/folders/1WOiFES-zJf6JkdWjziMnFrqsJJlmlBwy?usp=sharing.
$ cd pi3_img_dir/my-working-image && ./load_image.sh

A.3.2 Basic Test

You can verify the success of the LLVM installation by using the command *llvm-config –version*. A successful installation will return *3.9.0svn* as the result

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): ARI is capable of compartmentalizing and instrumenting the system in accordance with the policy.
(C2): ARI can automatically instrument the CPS, recording control flow events during runtime.
(C3): ARI has the ability to verify mission integrity based on runtime measurements.

A.4.2 Experiments

(E1): [Automatic Compartmentalization and Runtime Measurement Instrumentation] [5 human-minutes + 30 compute-minutes]
How to: Given CPS software such as ArduPilot, ARI uses command line instructions to automatically compartmentalize and instrument the software.
Preparation: Implement the compartmentalization policy in *ari_dir/graph_analysis/analyze.py* (*ari_dir* refers to the root directory of ARI , i.e., */home/ari-new-ae/conattest in VM*). We have provided the default one, *partition_by_controller*. Identify the critical compartments in *ari_dir/ardupilot/crit_cpt*; we have designated the attitude controller as the critical compartment. **Please note that the terminal might display a 'Build Failed' message due to linking issues during the ArduPilot build, but this is normal; we will link it later.**
Execution: *$ cd ari_dir*
$ cd ./conattestllvm && ./compiler_for_1st_part.sh
$ cd ./build && make -j2
$ cd ../../ardupilot
$ source ./compile_1st_part.txt
$ cd ../conattestllvm && ./compiler_for_2nd_part.sh
$ cd ./build && make -j2
$ cd ../../ardupilot
$ source ./compile_2nd_part.txt
Results: The resulting binary is stored in *ardu_dir/build/sitl/bin/arducopter* and is also transferred to the Pi3. You can check the compartmentalization of the application into 8 regions with the following command:
$ readelf -S ardu_dir/ardupilot/build/sitl/bin/arducopter

(E2): [Mission Execution and Measurement Collecting] [5 human-minutes + 10 compute-minutes]:
How to: Execute a takeoff mission with Ardupilot on Pi3. The instrumented code will automatically record the runtime measurements.
Preparation: (1) Log into the remote Pi3 via SSH from VM. (2) Open a terminal in the VM.
Execution: (1) On the remote Pi3, execute the following commands:
$ ssh pi@10.228.106.170
$ cd ~&& sync
$ sudo ./arducopter -S -I0 –model + –speedup 1 –defaults ./copter.parm
(2) In the VM, run the following single line command:
$ "mavproxy.py" "–master" "tcp:10.228.106.170:5760" "–sitl" "10.228.106.170:5501" "–out" "10.228.106.170:14550" "–out" "10.228.106.170:14551" "–map" "–console"

You will then see a *Console* and a *Map* in the VM. Wait for approximately 1 minute until the *Console* displays *AP: EKF2 IMU0 is using GPS* and *AP: EKF2 IMU1 is using GPS*.

Next, type the following commands in the terminal (only the command after > in the following):
STABILIZE> mode guided
GUIDED> arm throttle
GUIDED> takeoff 20

After the flight takes off (wait for about 10 seconds), stop the processes in the VM terminal and remote Pi3 using *Ctrl + C*. Then, transfer the measurement from the remote Pi3 to the VM terminal using these commands:
$ cd ardu_path/ardupilot/
$ source ./tsf_measurement.txt

Results: The mission measurements are stored in *ardu_path.txt*, including *ARI_branch.txt*, *ARI_ind_jmp.txt*, *ARI_ret_hash.txt*, and *ARI_tsf.txt*.

(E3): [Measurement Verification] [1 human-minutes + 3 compute-minutes]
How to: Perform the verification using runtime measurements as input. The verification engine will issue an alert if the verification fails.
Preparation: Open a terminal in the VM.
Execution: *$ cd ../oat-verify-engine*
$ source ../ardupilot/mission_verify.txt

Results: The verification engine will validate the mission integrity. If the verification passes, it will display the hash of all return addresses obtained from both runtime and replay. A successful verification will also show *Return Integrity Verification Pass!*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: XCheck: Verifying Integrity of 3D Printed Patient-Specific Devices via Computing Tomography

Zhiyuan Yu[†], Yuanhaur Chang[†], Shixuan Zhai[†], Nicholas Deily[†],
Tao Ju[†], XiaoFeng Wang[§], Uday Jammalamadaka[‡], Ning Zhang[†]

[†] Washington University in St. Louis
[§] Indiana University Bloomington
[‡] Rice University

A Artifact Appendix

A.1 Abstract

XCheck is a defense system designed to verify the integrity of 3D-printed medical devices, by crosschecking their CT scans against the original designs with shape comparison techniques. Our artifact comprises the source code, CT scans (in the DICOM format), and designed model files (in the STL format). To operate the system, the user needs to run the program in the command line, providing input file paths and specifying acceptable thresholds adapted to different types of medical devices. The expected output includes interactive visualization for identifying malicious areas, and quantitative scores indicating whether the print is benign or malicious.

Given the complex computation and geometry rendering involved in XCheck, a machine with a moderate CPU and GPU, as well as memory of at least 16 GB is recommended. Please note that run-time may vary depending on the user's hardware. We have compiled a list of required dependencies into a YML configuration file.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact provided does not contain any harmful materials that may compromise machine security or pose threats to human health. The models and CT scans used by the system were not derived from real human subjects, and there is no privacy concern associated with the use of artifacts. We have taken the utmost care to ensure that the artifact meets all necessary safety standards and guidelines.

A.2.2 How to access

We have made the code, models, and CT scans publicly available on GitHub. The stable URL link pointing to the commit is `https://github.com/WUSTL-CSPL/XCheck/commits/5ee4b4820671fc215795ccb09daa70670a29e4f3`.

A.2.3 Hardware dependencies

The system can run on a machine with a moderate CPU and at least 16GB of available RAM. The system was tested stable on a desktop computer with AMD Ryzen 9 3900X 12-Core Processor, NVIDIA GeForce RTX 3070 Ti, and 32GB memory; and a laptop with Intel i9-9880H Processor, AMD Radeon Pro 5500M, and 16GB memory. Both setups are equipped with OpenGL of version 4.1. No other specific hardware is required, but the variance in hardware can lead to differences in run-time.

A.2.4 Software dependencies

XCheck was implemented in Python, and the system's environment was set up using Miniconda 4.12.0 on Ubuntu 22.0.4. The required packages include vedo, vtkplotter, open3d, point-cloud-utils, numpy, matplotlib, pydicom, seaborn, and scipy. For the detailed installation process please see Section A.3. The models and CT scans for testing are included in the artifact and do not need to be downloaded from external sources.

A.2.5 Benchmarks

The data required by the experiments are the design models (in the STL format) and CT scans (in the DICOM format) of the printed medical devices. Nine malicious models underwent geometry or material modifications provided; three malicious bone scaffolds where certain internal regions were filled in solid, a lung-on-chip with solid internal bulges, a dental guide with added sphere volume, two bone screws with enlarged thread distance and shortened non-threaded shank, a bone screw and a dental guide printed with a different material. The CT scans and design models can be found in the directory *./Geometry*.

A.3 Set-up

A.3.1 Installation

Conda or Miniconda is recommended for setting up the environment. It can be installed via the official link https://docs.conda.io/en/latest/miniconda.html and the process can differ based on the user's OS. The commands for setting up the environment with the *xcheck.yml* file are:

```
$ cd <the_path_to_the_folder>
$ conda env create -f xcheck.yml
$ conda activate xcheck
```

A.3.2 Basic Test

The basic functionality can be tested with an additional flag *-basic* in the command line. For testing, please run the following command:

```
$ python3 run.py -basic Bone_12
```

The expected output is an interactive visualization window, with an example shown in Figure 6 in the manuscript. The boxes in the top right corner are clickable, each corresponding to an analysis method. When clicking on *Registration*, the CT-reconstructed model should overlap with the design model, indicating that they are properly aligned after registration. When clicking on *Added Voxel* or *Missing Voxel*, an additional box *Colormap* becomes clickable. The colormap enables filtering out areas with lower distances, therefore highlighting the areas where geometric discrepancies are high. When clicking on *Ray-based*, it shows results where malicious regions are highlighted in red in the 3D space.

A.4 Evaluation workflow

A.4.1 Major Claims

In summary, our major claim is that XCheck can verify the integrity of printed devices in terms of geometry and material, by providing both interactive visualization applications and quantitative risk scores.

(C1): XCheck can validate the geometry integrity by measuring and visualizing discrepancies in geometric features. This is proven by the experiment (E1) described in Section 6 whose results are illustrated/reported in Figure 6 and Figure 11.

(C2): XCheck can validate the material of printed objects by comparing them to prints with similar geometry but different materials. The material verification result is reflected in the corresponding *Gamma_m* value. This is proven by the experiments (E2) whose results are illustrated in Table 3 in the appendix.

(C3): XCheck provides a quantitative risk score with gamma analysis, which aggregates geometry and material deviations. This is proven by the experiments (E3) whose results are illustrated in Table 3 in the appendix.

A.4.2 Experiments

(E1): *[Geometry Verification] [16 compute minutes + 1GB disk]:*

Preparation: Detailed instructions regarding environment setup and activation are included in Section A.3.1.

Execution: The experiments are conducted using command lines. For each of the included CT scans and models under *./Geometry*, run the command by replacing the "-f1" and "-f2" arguments for the files to be compared, and the "-etdist", "-ets", and "-etg" command to set the appropriate thresholds. For instance, to compare the CT scans of a 3D printed bone scaffold (*./Geometry/Bone_12*), that is manipulated by adding various internal solid regions, to its original model (*./Geometry/Bone.stl*), use the following command:

```
$ python3 run.py \
     -f1 "Geometry/Bone_12" \
     -f2 "Geometry/Bone.stl"
     -o Bone_12 -etdist 1.7 \
     -ets 0.05 -etg 0.005 -etm 1
```

Results: XCheck begins to first reconstruct the CT scanned DICOM images, then aligns them to the original model. Due to the complexity of models, the registration process of XCheck adopts a non-deterministic design to significantly reduce the runtime. Therefore, the iterative registration will prompt the user to visualize the aligned shapes to easily verify the registration. After the user confirms and closes the visualization window, the user can either press "enter" to proceed, or press "r" to restart the registration.

When the program run to its termination, an interactive visualization window appears, through which the user can navigate to visualize the difference between the original model and the CT-scanned reconstruction. The user can click on *Added Voxel* or *Missing Voxel* coupled with selecting *Colormap* to visualize the added/missing voxel of the printed model. During this, the user can also control different aspects of the visualization using the following sliders. (1) *Opacity-Divergence Isolation*: filters out voxels below a certain distance threshold; (2) *Colormap-Upper Bound*: paints all voxels with distance above the upper bound red, then normalizes and assigns colors with the new bounds; (3) *Colormap-Lower Bound*: paints all voxels with distance below the lower bound blue, then normalize and assign colors with the new bounds; and (4) *Point Size*: adjust voxel size ranging from 1 to 10. By customizing these parameters, users can identify whether a region of interest has been manipulated. For instance, in the verification of the model *./Geometry/Screw_5*, adjusting *Opacity* can effectively isolate part of voxels on threads and render in red, because the enlarged pitch size results in threads shifting to

shank regions. Such visualization enables a more flexible and intuitive presentation of malicious regions.

Ray-based analysis leverages the principle that all geometry attacks have to alter the volume of devices. It will identify and visualize such volume in the interactive window. An example is given in *./Geometry/Bone_14* model, where the internal solid region is too small to be captured by voxel analysis but can be visualized by ray-based analysis when clicking the *Ray-based* box.

(E2): *[Material Validation] [1 compute minute]:*

Preparation: Material validation follows the geometry verification automatically.

Execution: Material validation should run automatically following the geometry verification.

Results: Material validation extracts the HU values from the CT scans, using kernel density estimation (KDE) to find the features describing its shape. This information is compared with known benign values of the same type of 3D-printed devices, to decide whether the material is tampered with. The expected result is reflected in the *Gamma_m* value, which is produced by gamma analysis and will be printed out in the terminal. The material will be considered malicious if the value is higher than 1, otherwise it is considered benign.

(E3): *[Gamma Analysis] [Less than 1 compute minute]:*

Preparation: Gamma analysis follows the geometry verification and material validation automatically.

Execution: Gamma analysis should run automatically following the previous analysis.

Results: The gamma analysis is automatically calculated at each run. If any of the four Gamma terms is less than 1, it is set to 1 to avoid value compensation when aggregating different terms (Section 5.6 in the manuscript). Each individual term of Gamma is squared and aggregated to output a total Gamma value. If the total Gamma value is no greater than 2, it is deemed benign, otherwise, it is deemed malicious. Besides, each term can be analyzed independently to infer the exact attack type; for instance, a material term *Gamma_m* larger than 1 indicates the existence of material attacks in the examined device. XCheck then proceeds to run to its termination, with the expected output of an interactive visualization window described in Section A.3.2.

```
[Computed Gamma_s: 5.130472175211942]
[Computed Gamma_d: 6.949988175539468]
[Computed Gamma_v: 1.8414619560913288]
[Computed Gamma_m: 0.43136242250997886]
[Total Computed Gamma: 8.889041709683562]
[Final Decision: Malicious]
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs

Tobias Scharnowski[1], Simon Wörner[1], Felix Buchmann[2], Nils Bars[1],Moritz Schloegel[1], and Thorsten Holz[1]
[1]CISPA Helmholtz Center for Information Security
[2]Ruhr-Universität Bochum

A Artifact Appendix

A.1 Abstract

Hoedur is a rehosting-based firmware fuzzer that introduces a multi-stream input format that extends the concepts contained in previous rehosting-based fuzzing works such as P2IM, uEmu, and Fuzzware. This format helps the fuzzer to mutate its input effectively.

We provide GitHub repositories containing data and scripts to install our prototype, to automatically reproduce our new firmware targets/configurations, and to auto-generate configurations to rerun our experiments based on the available computation resources. Our experiments run in Docker containers. These can be rebuilt via the provided scripts, or prebuilt Docker containers can be used to rerun our experiments. To facilitate the reproduction, we provide an alternative experiment profile that reduces the CPU requirements while still, in our view, supporting our major claims.

In our experiments, we compare our fuzzer against itself in different configurations (with and without multi-stream inputs) and against the related work Fuzzware given that this tool outperformed P2IM and uEmu in an evaluation. We test Hoedur's speed and reliability in finding known bugs, producing code coverage, and finding previously unknown vulnerabilities.

A.2 Description & Requirements

Access Privileges. The user under which the experiments are supposed to run requires access to Docker. `root` privileges are also required to configure the environment for running the related work `Fuzzware` (see scripts/fuzzware/set_limits_and_prepare_afl.sh).

OS and software. Our fuzzing experiments require Linux hosts. We recommend a recent Ubuntu Server installation. On these hosts, Python 3, rsync, and Docker must be installed and the user must be part of the `docker` group.

Computation Resources. Our original experiments took around 30 CPU years to run. To provide a less resource intensive option, we created an alternative experiment profile for the artifact evaluation that should reproduce our major findings in around 3 CPU years worth of computation time. If changes to these configurations are desired, we provide a configuration format that allows customizing the experiment duration and the repetition counts.

To rerun the experiments within 15 days, the equivalent of the following computation resources will be required:

1. Reduced experiment: 2 servers, 50 physical cores each.
2. Full experiment: 20 servers, 50 physical cores each.

Storage. In total, the experiments will produce a maximum of 300 GB of data which needs to be pulled onto one server to compute and aggregate all metrics.

A.2.1 Security, privacy, and ethical concerns

Running the experiments will require root access and access to Docker on the reproduction machines. No security mechanisms are disabled, and the machines are not exposed to additional security risks.

Hoedur has found previously-unknown vulnerabilities which have been responsibly disclosed to the respective vendors. Our prototype may find additional vulnerabilities, possibly even in the provided targets. Please handle these findings responsibly as well.

A.2.2 How to access

The code and experiment data are available on GitHub under github.com/fuzzware-fuzzer/hoedur-experiments. We created the tag `sec23-ae-accepted` as a stable reference.

From there, the complete reproduction is done in Docker containers. These can be built from Dockerfiles, or our prebuilt containers can be used. The repository README contains all information required to build and run the experiments.

A.2.3 Hardware dependencies

None (no special hardware, for general compute requirements, see Descriptions & Requirements above).

A.2.4 Software dependencies

To run the experiments, a Linux distribution is required. We recommend a recent version of Ubuntu Server. On these hosts, Python 3, rsync, sudo, and Docker need to be installed and set up for the user. For the optional step of rebuilding our target binaries, a small number of Python packets need to be installed via the provided requirements.txt file.

A.2.5 Benchmarks

As a part of our firmware fuzzing targets, we use the firmware binaries previously published in the P2IM, uEmu, and Fuzzware experiments. We include copies of these binaries in the hoedur-experiments GitHub repository. Section 6.1 of the paper explains that we adapt targets from the coverage measurement data set by applying binary patches. You can find these patches along with comments and reproduction scripts at 02-coverage-est-data-set/binary-patching.

A.3 Set-up

All of our experiments use Docker as an execution environment. Docker needs to be installed on the system, and the user under which the experiments will be run needs to be added to the `Docker` group. Also, ensure that Python 3 is installed on the system. In case you would like to rebuild our target binaries using the reproduce_targets_and_configs.py script, `pip` is required.

For each host on which the related work FUZZWARE might be run, prepare the system to run FUZZWARE by running scripts/fuzzware/set_limits_and_prepare_afl.sh as `root`.

The hosts to be used for the experiment are configured in experiment-config/available_hosts.txt (see also experiment-config/README.md) and looks like the following:

```
my-host-1 <number_of_physical_cores_1>
my-host-2 <number_of_physical_cores_2>
```

For example, if everything should be run and metrics be generated on the same, single host with 70 cores, the configuration is as simple as:

```
localhost 70
```

If two servers (`my-host-1` and `my-host-2`) with 50 cores each are to be used, and the results should be aggregated on `host-1`, then the configuration would look like the following:

```
localhost 50
my-host-2 50
```

To allow some of the experiment scripts to access each host via `SSH`/`rsync`, create an SSH config entry in `~/.ssh/config`:

```
Host my-host-1
    Hostname 12.34.56.78
    User user
    IdentityFile ~/.ssh/my_privkey.key
    AddKeysToAgent yes
```

A.3.1 Installation

To install Hoedur, first, make the Docker containers available to your system. To re-build the Docker containers, run install.py. To obtain the pre-built Docker containers, run `./install.py --prebuilt`.

As an optional step, you may wish to reproduce our new target firmware binaries and auto-converted configurations and bug detection hooks. To reproduce these, run reproduce_targets_and_configs.py.

A.3.2 Basic Test

To make sure that the basic installation has been successful, first run:

```
./scripts/check_install.py
```

You may also like to run one of the bug reproducing inputs provided in the experiments repository. To run the reproducer for CVE-2022-41873, run:

```
cd ./04-prev-unknown-vulns/repro-run-scripts
./run_reproducer_CVE-2022-41873.sh
./run_reproducer_CVE-2022-41873.sh --trace
```

This should indicate that the bug `new-Bug-CVE-2022-41873` has been triggered by the reproducing input.

A.4 Evaluation workflow

Our experiments test four different aspects of Hoedur: The ability to trigger bugs, to produce code coverage, to make previously unavailable mutation types (by the example of dictionaries) effective, and to find previously unknown bugs.

We organized the hoedur-experiments repository in such a way that one subdirectory corresponds to one section in the evaluation of the paper. We provide scripts to run these experiments, and try to facilitate this process by providing the additional utility generate_host_run_config.py which accepts a description of computation resources (SSH-available hosts as well as the number of cores to use on each host) and generates run scripts for each host that together allow reproducing our results. We also created experiment profiles that reproduce the major insights of our experiments while reducing the CPU requirements.

A.4.1 Major Claims

(C1): HOEDUR outperforms the state of the art reference FUZZWARE in terms of its ability to find bugs quickly. This is proven by the experiment *Bug Finding Ability* described in Section 6.2 in the paper and the results of which are reported in Table 1 and Table 2. This corresponds to the directory 01-bug-finding-ability in the artifact.

(C2): HOEDUR is either on par with or outperforms the state of the art reference FUZZWARE and its version SINGLE-STREAM-HOEDUR (which does not use our multi-stream input representation) in terms of code coverage. This is proven by the experiment *Code Coverage: Established Data Set* described in Section 6.3 in the paper, whose results are reported in Figure 6 and Figure 8. This corresponds to the directory 02-coverage-est-data-set in the artifact.

(C3): Our multi-stream input representation allows using dictionary mutations effectively. These dictionaries do not provide a significant benefit when using a flat binary input format. This is proven by the experiment *Advanced Mutations via Dictionaries* described in Section 6.4 in the paper, whose results are reported in Figure 7. This corresponds to the directory 03-advanced-mutations in the artifact.

(C4): HOEDUR is able to find previously unknown vulnerabilities. This is proven by the experiments *Bug Finding Ability* and *Finding Unknown Vulnerabilities* described in Section 6.2 and Section 6.5 in the paper. The results are reported in Table 2 and Table 3. This corresponds to the directories 01-bug-finding-ability and 04-prev-unknown-vulns in the artifact.

A.4.2 Experiments

In summary, the experiment workflow is the following:

1. Configure the experiment hosts via available_hosts.txt.
2. Install and setup: install.py and set_limits_and_prepare_afl.sh on each experiment host.
3. Generate the experiment run configurations: generate_host_run_config.py. Upload via `--upload`.
4. Run experiment configs on the respective experiments hosts (make sure to use `tmux` or similar).
5. Pool data together: sync_experiment_data.py
6. Generate metrics: compute_metrics.py.
7. Inspect the results (see per-experiment description).

Now we include more details on each step. Steps 1 to 6 need to be taken once to generate the data corresponding to the results reported in our paper for the different experiments. From there, the final experiment-specific step is to check the relevant output for each experiment manually.

We provide the utilities available_hosts.txt, generate_host_run_config.py and run_experiment.py to help with scheduling and running the fuzzing experiments for **E1**, **E2**, **E3** and **E4**. Before running these experiments, we assume that the Docker containers are already installed on each host on which any parts of the experiments are supposed to be run (install.py), and that each host is available via an SSH configuration (see Section A.3.1) and set up to run the reference fuzzer FUZZWARE (scripts/fuzzware/set_limits_and_prepare_afl.sh).

The first manual step after this installation is to determine the hosts on which to run the experiments and to configure them in available_hosts.txt. The fuzzing run experiment configurations can then be generated using generate_host_run_config.py. The experiment configuration `YAML` files will be located in experiment-config/host-run-configs as `<hostname>.yml`. These configurations need to be copied to the corresponding hosts. Uploading the configurations can be done either manually or by running:

```
./generate_host_run_config.py --upload
```

Given the experiment configuration file on a host, the experiment can be started via:

```
./run_experiment.py <HOST_CONFIG>.yml
```

Please note that these are long-running experiments, such that `tmux` or similar tools should be used to run them. After running the fuzzers to completion, the workflow will be to synchronize all results into the `hoedur-experiments` directory of the main host via sync_experiment_data.py and running the post processing script compute_metrics.py to generate all metrics.

From here, the generated results need to be confirmed by inspecting the `results` directory of each experiment. We document which files within the `results` directory contain the data from our paper for each experiment in the descriptions below. You may also refer to the README of each experiment directory of the published hoedur-experiments repository for more info on the available data.

(E1): *[01-bug-finding-ability] [60 human-minutes + 750 compute-days + max 100GB disk]:*
How to: Reproduce claim **C1** / Section 6.2 in the paper.
Preparation: Generate run configurations using generate_host_run_config.py (see above) and copy them to the respective hosts. Make sure the required Docker containers are installed and the hosts are set up on all experiment hosts via install.py and set_limits_and_prepare_afl.sh (see above).
Execution: Run the fuzzers using the run scripts. Ensure a stable execution environment when running via SSH, such as `tmux`. After the fuzzers have finished executing, sync the results via sync_experiment_data.py and then compute all remaining metadata via compute_metrics.py.
Results: After generating all metadata for the completed experiments, the results can be found in 01-bug-finding-ability/results/bug-discovery-timings. In

this directory, the bug raw discovery timing data can be found in `timings.txt` and `timings.json` and represent the data also contained in Table 1 in the paper. The table representations as present in the paper can be found in the `results` directory under table_1_cve_discovery_timings.tex and table_2_add_bugs_discovery_timings.tex.

The overall results should resemble those in the paper: It is expected that HOEDUR finds bugs more quickly than FUZZWARE overall. Keep in mind that due to the non-deterministic nature of fuzzing, these numbers may vary. Also keep in mind that based on the experiment profile used, the number of iterations and fuzzing duration may be altered from the original experiments. For example, the default experiment configuration shortens the fuzzing runs from the original 15-day duration. For the exact numbers of each experiment configuration/profile, please refer to the `experiment-config` README. To fully rerun the original configuration, please use the corresponding, pre-supplied experiment configuration/profile name `full-eval`.

(E2): *[02-coverage-est-data-set] [60 human-minutes + 180 compute-days + max 100GB disk]:*
How to: Reproduce claim **C2** / Section 6.3 in the paper.
Preparation: Already done in **E1**.
Execution: Already done in **E1**.
Results: After generating all metadata for the completed experiments, the results can be found in 02-coverage-est-data-set/results/coverage. The raw plot data can be found in the directory tree in compressed format under `charts/<fuzzer_name>`. It contains the raw data also shown in Figure 6 in the paper. The graphical plot representations as present in Figure 6 and Figure 8 in the paper can be found in the parent directory under figure_6_baseline_coverage_plot.pdf and figure_8_appendix_baseline_coverage_plot.pdf, respectively. It is expected that HOEDUR is on par with or generates more coverage than FUZZWARE and SINGLE-STREAM-HOEDUR. See also the general disclaimer about fuzzing experiment variance and experiment configurations/profiles under item **Results** of **E1**.

(E3): *[03-advanced-mutations] [60 human-minutes + 40 compute-days + max 50GB disk]:*
How to: Reproduce claim **C3** / Section 6.4 in the paper.
Preparation: Already done in **E1**.
Execution: Already done in **E1**.
Results: After generating all metadata for the completed experiments, the results can be found in 03-advanced-mutations/results/coverage. The raw plot data can be found in the directory tree in compressed format under `charts/<fuzzer_name>`. It contains the raw data also shown in Figure 7 in the paper. The graphical plot representations, as present in Figure 6 and Figure 8 in the paper, can be found in the parent directory under figure_7_baseline_dict_coverage_plot.pdf. It is expected that HOEDUR+DICT performs best overall, while HOEDUR+DICT does not provide the same level of improvement over its single-stream version.

(E4): *[04-prev-unknown-vulns] [60 human-minutes + 50 compute-days + max 10GB disk]:*
How to: Reproduce claim **C4** / Section 6.1 and Section 6.5 in the paper.
Preparation: Already done in **E1**.
Execution: Already done in **E1**.
Results: For this experiment we provide pre-extracted samples of crashing inputs that trigger each reported bug in the repository. For a full overview of the reported bugs, see the tables in 04-prev-unknown-vulns/README.md. Some of the new bugs have been found in the CVE target set of FUZZWARE. As such, the corresponding bug reproducing inputs can be found in 01-bug-finding-ability/results/bug-reproducers. The other bug reproducing inputs can be found in 04-prev-unknown-vulns/results/bug-reproducers.

As part of the reproduction, we also run HOEDUR on the remaining targets for a limited amount of time. As some new bugs require more computation power for HOEDUR to find and are found with some variance, it is expected that this will find some of the newly reported bugs, but may not reproduce all of them. The results can be found in 04-prev-unknown-vulns/results/bug-reproducers. The reproducers can be found in the directory tree of `bug-reproducers`. They represent triggers for the bugs listed in Table 3 in the paper. It is expected that HOEDUR triggers at least some of the bugs within the fuzzer reruns and as a result, a set of reproducers can be found and run via the scripts located in 04-prev-unknown-vulns/repro-run-scripts.

A.5 Notes on Reusability

We designed our experiments to be extendable and configurable in the computation resources required. Our configurations allow adding more experiments, such as `05-my-other-experiment`. Our scripts are built to be able to compute metadata for new fuzzing runs, such that one should only need to add another experiment and metrics as configurations. Coverage metadata, alongside their plots, can be configured to be computed using the provided scripts and setup. Some additional metrics are already computed, even though they are not referenced in this document. After running the experiments, one can find these metrics under the `results` experiment sub-directories.

Regarding the fuzzer itself, our prototype is published open source under https://github.com/fuzzware-fuzzer/hoedur. The source code separates the emulator from the fuzzer rather strictly, such that the community may make use of both components.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Forming Faster Firmware Fuzzers

Lukas Seidel[1], Dominik Maier[2], and Marius Muench[3]

[1]*Qwiet AI, jlseidel@qwiet.ai*
[2]*TU Berlin, dmaier@sect.tu-berlin.de*
[3]*VU Amsterdam and University of Birmingham, m.muench@bham.ac.uk*

A Artifact Appendix

A.1 Abstract

This artifact allows the replication of the experiments and results described in Section 6. We provide the following: (i) A stand-alone repository containing the full source code for our rehosting and fuzzing engine, ready to be compiled and used (https://github.com/pr0me/SAFIREFUZZ), (ii) a repository containing documentation, build- and setup scripts for replicating our experiments and a copy of the data we gathered during our evaluation (https://github.com/pr0me/safirefuzz-experiments).

The artifact has been validated on a HoneyComb LX2 ARM workstation containing 16 ARM Cortex-A72 cores with a clock rate of up to 2 GHz, 32 GB DDR4 memory with a frequency of 3200 MT/s and a 128 GB m.2 SSD running Ubuntu 18.04.05.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

While running and evaluating SAFIREFUZZ does not require destructive steps, small changes to the host system weakening its security guarantees are needed to run our system.

In particular, we require ASLR to be disabled, to increase determinism and avoid mapping of, e.g., linked libraries in segments we need otherwise and expect to be empty. Additionally, we enable allocating virtual memory down to address 0 by adjusting `mmap_min_addr`, as we need to place parts of the firmware image in low memory regions. Both can be configured by running the `SAFIREFUZZ/prepare_sys.sh` script. Those changes should be reverted after usage of our system, either by manually reverting the changes or rebooting.

A.2.2 How to Access

We provide public access to our code and experiment setups and data through the following GitHub repositories at specific tags for artifact evaluation:

1. SAFIREFUZZ main repository: https://github.com/pr0me/SAFIREFUZZ/tree/post_ae
DOI: https://zenodo.org/record/8223057

2. Artifact Evaluation data: https://github.com/pr0me/safirefuzz-experiments/tree/post_ae
DOI: https://zenodo.org/record/8223055

The repositories contain detailed information on building, running, and reproducing our experiments.

A.2.3 Hardware Dependencies

SAFIREFUZZ rehosts low-level Cortex-M firmware onto more powerful Cortex-A cores. As such, a system containing a Cortex-A core with 32-bit support is required, which can for instance be found on a Raspberry Pi 4b featuring four Cortex-A72 cores. Installation instructions for Raspberry Pis can be found in our main repository. Additionally, due to interoperability issues, our Fuzzware-specific experiments were run on an x86-64 VM.

During artifact evaluation, we provided the reviewers with access to the same hardware we used during our evaluation: (**M1**) a *HoneyComb* ARM workstation, and **(M2)** an Ubuntu 18.04 x86-64 VM hosted on an AMD EPYC 7662 server.

A.2.4 Software Dependencies

1. **Rust:** Our artifact is implemented in the Rust programming language. Per the `rust-toolchain` file provided in the main repository [1], we pin the installation environment to compiler version `rustc 1.62.0-nightly`.

2. **Cross-Compilation:** A cross compilation toolchain is required. On Ubuntu, the corresponding packets are `gcc-arm-linux-gnueabihf` and `g++-arm-linux-gnueabihf`.

   Install the `armv7-unknown-linux-gnueabihf` rust target for the above-mentioned compiler version. Note that these steps are even required when directly building in an ARM environment such as the HoneyComb. While the processor can execute programs targeted for both

ARMv7 and ARMv8 versions, if the OS is built for aarch64, cross-compilation is required as the artifact binary will execute in ARMv7's 32-bit mode.

3. **External Dependencies:** The main artifact requires the *LibAFL* and *Keystone* external dependencies that cannot be automatically fetched by Rust's package manager. We include the dependencies as git submodules, pinned to specific versions.
   The evaluated third-party frameworks introduce their own dependencies and can be set up as documented in the HALucinator[1] and Fuzzware[2] repositories.

4. **Python:** For multiple build and automation scripts provided with the AE experiment repository, we require a Python version > 3.9. Additionally, we require the following Python libraries for analyzing and plotting the results of our experiments: `jupyter`, `numpy`, `matplotlib`, `seaborn`, `scipy`, `pandas`.

A.2.5 Benchmarks

To evaluate SAFIREFUZZ, we use a collection of 12+2 firmware samples: 12 samples from the original HALucinator evaluation, and 2 previously untested samples (*JPEG Decoder* and *STM32 Sine*). We include all samples in the experiment repository under `00_firmware`.

Using these samples, we evaluate our approach against the following fuzzing setups:

1. **HALucinator.** State-of-the-art high-level-emulation-based rehosting and fuzzing framework. We include the fuzzing-ready *hal-fuzz* version as a submodule in `safirefuzz-experiments/01_fuzzing/hal-fuzz`.

2. **HALucinator - LibAFL.** We replace HALucinator's legacy AFL forkserver with a LibAFL-based forkserver. This new version is identical in configuration to the forkserver backend we use in SAFIREFUZZ. We conduct this comparison to eliminate variables such as differences in mutation strategies. Details can be found in the *safirefuzz-experiments* repository under `01_fuzzing/forkserver_LibAFL`.

3. **Fuzzware.** A recent peripheral-modeling-based rehosting approach. This is the only experiment we conducted in an x86-64 environment, as, even after consulting the authors, Fuzzware could not be brought to run in our default ARM environment. We provide usage information and link the necessary submodule under `01_fuzzing/fuzzware`.

We include setup guides and detailed usage instructions for all evaluated frameworks under `01_fuzzing/README.md`.

[1] https://github.com/ucsb-seclab/hal-fuzz
[2] https://github.com/fuzzware-fuzzer/fuzzware

A.3 Set-up

A.3.1 Installation

If you are using the provided access to the experiment machines, all systems are already set-up and below instructions can be skipped. To manually install SAFIREFUZZ, the main artifact, please follow these steps:

1. Checkout our experiments repository 2 and initialize the submodules recursively.
2. Inside the experiments repository:
   `$> cd 01_fuzzing/SAFIREFUZZ`
3. Install the Rust programming language.[3]
4. Install the cross-compilation toolchain with `'rustup target add armv7-unknown-linux-gnueabihf'` and cross-arch linkers, e.g., on Ubuntu by running `'sudo apt install gcc-arm-linux-gnueabihf g++-arm-linux-gnueabihf'`.
5. Specify the correct linker by adding the following lines to your `~/.cargo/config`:
   `[target.armv7-unknown-linux-gnueabihf]`
   `linker = "arm-linux-gnueabihf-gcc"`
6. Specify the target harness you want to execute / fuzz in `src/engine.rs`:
   `use crate::harness::`**`wycinwyc`**` as harness;`
7. Build with
   `$> cargo build -release -target armv7-unknown-linux-gnueabihf`
8. Run the `prepare_sys.sh` script as root.

A.3.2 Basic Test

After installing and compiling the main artifact, you will find the `safirefuzz` binary under `./target/armv7-unknown-linux-gnueabihf/release/`. Compilation is always specific to a single target or harness, so make sure to change the target (cf. Section A.3.1, step 6.) and re-compile before trying to execute a new firmware image.

Start fuzzing a specific firmware image with a directory of seeds by running:

```
./safirefuzz -b 00_firmware/wycinwyc.bin -i
01_fuzzing/seeds/wycinwyc/ -c 1.
```

When starting a fuzzing campaign, you should see LibAFL's status reports scrolling by. For running a test on the WYCINWYC target, you should be able to see rapidly increasing numbers for *corpus*, around 400-600 after approx. 30 seconds, which are interesting inputs leading to unique new coverage, at roughly 7000 executions per second. You can find these inputs in the *queue* directory while crashing inputs are stored in *crashes*.

To then execute a single input, execute:

[3] https://www.rust-lang.org/tools/install

```
./safirefuzz -b 00_firmware/wycinwyc.bin -i
01_fuzzing/crashes/SOMECRASHID
```

We automated most of these steps with the `safirefuzz_target.py` script included in the experiments repository under `01_fuzzing`. For instance, running `$> ./safirefuzz_target.py nxp_http` will automatically build SafireFuzz for the correct target and start fuzzing. This script defaults to running on the third core (`-c 2`), change this if you are running multiple tests in parallel.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): SafireFuzz achieves statistically significant more exec/s (ca. 690x on avg.) and coverage than HALucinator, except for coverage on the P2IM PLC and P2IM Drone targets. This is proven by experiment E1.

(C2): SafireFuzz achieves more exec/s (ca. 1100x on avg.) and coverage than HALucinator-LibAFL, except for coverage on the UDP Echo Server, STM PLC, P2IM PLC and P2IM Drone targets. These results are statistically significant, except for coverage on the P2IM PLC, STM PLC, WYCINWYC, and UDP Echo Client targets. This is proven by experiment E2.

(C3): SafireFuzz achieves more exec/s (ca. 145x on avg.) and coverage than Fuzzware, except for coverage on P2IM PLC[4] and P2IM Drone. The results are statistically significant except for coverage on the 6LoWPAN RX/TX, STM PLC, and WYCINWYC targets and for execution speed on the SAMR21 target. This is proven by experiment E3.

(C4): SafireFuzz reliably re-discovers previously found bugs during fuzzing (E0). This includes vulnerabilities in the WYCINWYC and 6LoWPAN RX/TX targets as discussed in Section 6.4.

(C5): SafireFuzz finds crashes in the previously untested firmware images *JPEG Decoder* and *STM32 Sine*. This can be replicated with experiment E4. We discuss the findings in Section 6.4 of our paper.

For C1, C2 and C3, we discuss our results in Section 6.2 in the main paper. Table 3 reports numbers gathered during our experiments and Figure 3 illustrates achieved coverage over the course of a 24-hour fuzzing campaign for all targets and frameworks.

[4]The paper as published as part of the proceedings contains an error in which the list of valid basic blocks for the P2IM PLC target was calculated incorrectly. This led to underreporting achieved coverage, impacting Fuzzware the most. We would like to thank Chris Boyce for pointing this out, based on the experiment data we published. A version with updated numbers and graphs can be found under https://download.vusec.net/papers/safirefuzz_sec23.pdf.

A.4.2 Experiments

As a working SafireFuzz installation is required for the subsequent steps, refer to Section A.3.1 of this Appendix and the README of our main repository [1] for instructions. The following steps assume you work in the pre-configured environments.

(E0): SafireFuzz Baseline [20 human-minutes fuzzing set-up time + up to 5x12x24 compute-hours + 10 human-minutes coverage collection set-up time + 2-6 compute-hours queue replay time]: Use the `./safirefuzz_target.py` script in `01_fuzzing` of the experiments repository [2] to start a 24-hour fuzzing campaign for the specified target with SafireFuzz.

(E1): HALucinator Comparison [20 human-minutes fuzzing set-up time + up to 5x12x24 compute-hours + 10 human-minutes coverage collection set-up time + 2-6 compute-hours queue replay time]: HALucinator is readily set-up, you can start fuzzing with this framework by executing the corresponding script in the *hal-fuzz* submodule. For further details, refer to the HALucinator section in `01_fuzzing/README.md`.

(E2): HALucinator-LibAFL Comparison [20 human-minutes fuzzing set-up time + up to 5x12x24 compute-hours + 10 human-minutes coverage collection set-up time + 2-6 compute-hours queue replay time]: For details how to start a HALucinator-LibAFL fuzzing campaign, refer to the corresponding section in `01_fuzzing/README.md`.

(E3): Fuzzware Comparison [15 human-minutes fuzzing set-up time + up to 5x12x24 compute-hours + 15 human-minutes coverage collection set-up time + 2-6 compute-hours queue replay time]: In order to set up and start fuzzing with Fuzzware, please refer to the detailed instructions provided `01_fuzzing/fuzzware` as part of our experiments repository.

(E4): Vulnerability discovery [15 human-minutes fuzzing set-up time + up to 24 compute-hours + 15 human-minutes replay & verification]: To compile and fuzz the previously untested targets, please refer to the `README` included in `03_case_studies` inside the experiment repository.

Collecting Coverage. For all experiments except E3, coverage can be collected using the `eval_bbs_halucinator.py`

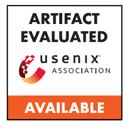
script in `02_coverage_collection`. For detailed instructions on this, refer to the `README` provided within the directory. For E3, use the scripts `fuzzware_genstats_with_hal.sh` and `fuzzware_genstats_without_hal.sh` provided in `02_coverage_collection` and refer to the `README` for details.

Analyzing Results. We provide scripts to test whether achieved coverage and execution speeds are statistical significant under `04_eval_data` inside the experiment repository. Please use the `bb_mann_whitney.ipynb` and `execs_mann_whitney.ipynb` jupyter notebooks inside the `coverage` and `executions` directories. We further provide a `gen_fig3.ipynb` notebook to plot coverage data over time. To use these notebooks with data from your experiments, you will need to exchange the `.data` and `.csv` in the according according subdirectories. Please refer to the `README` for more details.

Time & Resource Considerations. Due the extent of the experiments carried out during evaluation, it may not be possible to run all experiments for all reviewers in the time frame allocated for artifact evaluation. Hence, we provide the raw data collected from our runs under `04_eval_data` in our experiment repository. The raw data allows to reproduce our claims without, or only partially, running the experiments.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: Understand Users' Privacy Perception and Decision of V2X Communication in Connected Autonomous Vehicles

Zekun Cai
The Pennsylvania State University
zuc204@psu.edu

Aiping Xiong
The Pennsylvania State University
axx29@psu.edu

A Artifact Appendix

A.1 Abstract

Our understanding of human drivers' privacy perception and decision of V2X communication in connected autonomous vehicles (CAVs) is based on descriptive statistics and inference statistical tests of quantitative data, as well as thematic analysis of qualitative data. We provide the data and explain the analysis methods we used to replicate all results reported in the paper. The artifact includes our collected data (for review only), our quantitative analysis via R code, and the qualitative analysis via excel spreadsheets and Python, such that one can recreate all results in tables, figures, statistical tests, and reported themes throughout the paper.

A.2 Description & Requirements

A.2.1 Artifact check-list (Meta-information)

Data set: Survey responses from the participants (quantitative and qualitative data sets) in our study; non-public.
Run-time environment: We did our analysis on Windows 11 system.
Security, privacy, and ethical concerns: Maintaining the confidentiality of participant data; the dataset will not be publicly available based on the approved IRB protocol.
Metrics: Perceived benefits, perceived risks, willingness to share data, and confidence of sharing decision.
Output: The artifact produces all results, containing tables, figures, and the code counts in users' answers to the open-ended questions.
Experiments: Descriptive statistics, inference statistical tests, and qualitative analysis of responses to the open-ended questions.
How much disk space required (approximately)?: Negligible, less than 1 GB.
How much time is needed to prepare workflow (approximately)?: This depends on whether the required environment (RStudio and PyCharm) and the required packages are already installed. If none of the aforementioned are present, the set-up should take less than 20 min on a modern computer.
How much time is needed to complete experiments (approximately)?: This depends on hardware, but should take less than 30 min on any recent PC or laptop.
Publicly available (explicitly provide evolving version reference)?: All scripts and code are made publicly available[1].
Code licenses (if publicly available)?: The R code and the Python script are licensed under Creative Commons Attribution 4.0 International.
Archived (explicitly provide DOI or stable reference)?: The DOI provided by Zenodo is 10.5281/zenodo.7707330.

A.2.2 Security, privacy, and ethical concerns

All personal identifiable information has been removed from both data sets. There is no risks in executing the analysis. However, we cannot exclude the possibility of those data being used to deanonymize participants. Based on the approved IRB protocol, the data will not be publicly available.

A.2.3 How to access

Along with the supplementary materials, we make all the scripts and code used to analyze data publicly available (see Footnote 1). The artifact includes four main parts: (1) the "CLMM Tests" folder that contains anonymized *Quantitative_Data.csv*, the R script "CLMM_Analysis.Rmd" for data analysis and the expected output "CLMM_Analysis.pdf"; (2) the "Thematic Analysis" folder that contains spreadsheets *Coder1_Coding.xlsx*, *Coder2_Coding.xlsx*, *Final_coding.xlsx*, and the script "Thematic_analysis.py" calculating the inter-rater agreement of the coders and counting the agreed codes. *Code_book.txt* describes the meanings of the codes; and (3) "Supplementary Materials.pdf." Additionally, the "README.md" provides a detailed overview of all files.

[1] https://zenodo.org/record/7707330#.ZAh0q3bMIQ8

A.2.4 Hardware dependencies

No specific hardware is needed. Our analysis requires less than 1 GB of disk space.

A.2.5 Software dependencies

The quantitative analysis requires R to run. We use RStudio (2022.12.0 Build 353) and R (version 4.2.2). RStudio can be obtained online for free[2]. The following R packages are needed to run the script: `ordinal` and `emmeans`. For the qualitative analysis, the coding results are listed in Microsoft Excel (Version 2301 Build 16.0.16026.20196). We use Python (3.10) and PyCharm (2021.2.3), both of which are publicly available[3]. To calculate inter-rater reliability (i.e., Cohen's Kappa) and count the frequencies of the themes, the following Python packages are needed: `pandas` and `numpy`. Our analysis is done on Windows 11 system.

A.2.6 Benchmarks

Datasets. We use the survey responses collected in our user study. The quantitative analysis (i.e., Cumulative Link Mixed-effects Model (CLMM) analysis) is conducted using the dataset *Quantitative_Data.csv*. The qualitative analysis (i.e., thematic analysis and Cohen's Kappa) is conducted based on *Coder1_Coding.csv*, *Coder2_Coding.csv*, and *Final_Coding.csv*. The description of the codes can be found at *Code_book.txt*.

The quantitative and qualitative data are provided for artifact evaluation only. To maintain participants' privacy, we do not release the data publicly.

Models. We run CLMMs on the *Quantitative_Data.csv* via the "CLMM_Analysis.Rmd."

A.3 Set-up

A.3.1 Installation

Quantitative analysis. Installation time: about 15 min. The quantitative evaluation is performed by using R. The set-up consists of two steps.

Software. RStudio 2022.12.0 Build 353 with R 4.2.2 is recommended because the authors used these versions. R and RStudio are all publicly available and their instructions for version-specific installation can be found at their respective websites.

R packages. When RStudio is installed, it must be started and the analysis script can be opened using the "File" menu. Then the following R packages need to be installed: `ordinal` and `emmeans`. To install these packages using RStudio, open the "Tools" menu and then select "Install packages". In the search box enter the first package. Then click "Install". Repeat these two steps for the second package. Installation of the packages might take some time if they need to be compiled. Once the two packages are installed, the analysis script can be run. In our experiment, we generate a PDF to view the results, which requires pdflatex. If there is no pdflatex in your computer, you can install the R package `tinytex` in RStudio to meet this requirement using the same method described above.

Qualitative analysis. Installation time: about 5 min. Same as the quantitative analysis, the set-up consists of two steps.

Software. Python (3.10) and PyCharm (2021.2.3) are both publicly available and their instructions for version-specific installation can be found at their respective websites.

Python packages. To run the `.py` script for thematic analysis, the following dependencies also need to be installed: `pandas` and `numpy`. You can install them one by one from the terminal using pip (which is automatically installed with Python). In PyCharm, the packages can also be installed directly through "Python Packages" tool bar at the bottom-left corner.

A.3.2 Basic test

After installing the dependencies (the R packages and the Python packages), you can run "Basic_Test.R" and "Basic_Test.py" to see whether the required dependencies can be loaded, respectively. There should not be any error messages if the packages are successfully installed.

A.4 Evaluation Workflow

A.4.1 Major claims

The major claims made in the paper are as follows:

(C1): While participants perceived more benefits but fewer risks in the three driving-related scenarios, they perceived more risks but fewer benefits in the infotainment scenarios (**RQ1**). This is proven by **E1**. Statistical inference test results are described in **Section 4.1** and **Table 3**. The descriptive statistics are illustrated in **Figure 2 (a)** and **(b)**.

(C2): Only the privacy priming was effective in reducing participants' perceived benefits than those in the control. Instead of augmenting their privacy concerns, the privacy&security priming condition showed similar results as those in the control (**RQ2**). This is proven by **E1**. Statistical inference test results are described in **Section 4.1** and **Table 3**. The descriptive statistics are illustrated in **Figure 2 (a)** and **(b)**.

(C3): Participants made more liberal privacy decisions in the driving-related scenarios, which could have been caused by perceiving both more benefits and fewer risks (**RQ1**). Moreover, they made more conservative privacy decisions as long as they were primed (**RQ2**). This is proven

[2] https://posit.co/download/rstudio-desktop/

[3] https://www.jetbrains.com/pycharm/download/#section=windows

by **E1**. Statistical inference test results are described in **Section 4.2** and **Table 3**. The descriptive statistics are illustrated in **Figure 2 (c)**.

(C4): We observed a non-significant trend that participants with much experience in driving assistance and connectivity functions perceived more benefits and more risks of data sharing. Moreover, there was a non-significant trend that they showed higher willingness in sharing the data (**RQ3**). This is proven by **E1**. Statistical inference test results are described in **Section 4.3** and **Table 3**. The descriptive statistics are illustrated in **Figure 3**.

(C5): Our thematic analysis verified the privacy-safety trade-off. The analysis revealed not only common factors similar to other settings, but also some unique factors for the CAV context (**RQ1**). This is proven by **E2**. Inter-rater agreement via Cohen's Kappa and thematic analysis results are described in **Section 4.5**.

A.4.2 Experiments

(E1): Quantitative analysis
Execution Time: about 15 min. The results from **Sections 4.1** to **4.3**, **Table 3**, **Figures 2** and **3** are produced in the R code via the following steps:
1. Open the RStudio.
2. In RStudio, open the document "CLMM_Analysis.Rmd" by clicking "File", then "Open File...", and selecting "CLMM_Analysis.Rmd".
3. Then click the drop-down arrow of "Knit" and select "Knit to PDF".
4. Once completed, you may view the produced PDF: "CLMM_Analysis.pdf".

The generated PDF will include results verifying claims 1-4 (**C1**-**C4**) (i.e., **Findings 1-4** in the paper).

(E2): Qualitative analysis
Execution Time: about 1 min. Cohen's Kappa (inter-rater reliability) with the initial codes of two coders and the frequencies of each theme with the final codes from **Section 4.5** are produced in the Python code via the following steps:
1. Open the PyCharm.
2. In PyCharm, click "File", then "Open...", and select the folder where "Thematic_Analysis.py" is located.
3. Then click "Run" and select "Run...". In the pop-up window, select "Thematic_Analysis" to run it.

Once completed, you will see the output results (should be the same with "Thematic_Analysis.pdf"). The generated results will verify claim 5 (**C5**) (i.e., **Finding 5** in the paper).

A.5 Notes on Reusability

Our artifact (R and Python scripts) can be reused to analyze other human-subject studies' results using CLMM, Cohen's Kappa, and thematic analysis.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926.

USENIX'23 Artifact Appendix: NRDelegationAttack: Complexity DDoS attack on DNS Recursive Resolvers

Yehuda Afek *
Tel-Aviv University
afek@tauex.tau.ac.il

Anat Bremler-Barr †
Tel-Aviv University
anatbr@tauex.tau.ac.il

Shani Stajnrod
Reichman University
shaaniba93@gmail.com

A Abstract

To fully understand the root cause of the NRDelegationAttack and to analyze its amplification factor, we developed mini-lab setup, disconnected from the Internet, that contains all the components of the DNS system, a client, a resolver, and authoritative name servers. This setup is built to analyze and examine the behavior of a resolver (or any other component) under the microscope. On the other hand it is not useful for performance analysis (stress analysis).

Here we provide the code and details of this setup enabling to reproduce our analysis. Moreover, researchers may find it useful for farther behavioral analysis and examination of different components in the DNS system.

A.1 Description & Requirements

DNS-FullProtocolSimulator is an Inner-Emulator environment for DNS protocol which was built as part of NRDelegationAttack research. DNS-FullProtocolSimulator includes a client, resolver and three authoritative name servers. The resolver is a BIND9 recursive resolver with both the NXNSAttack patched version (BIND9 version 9.16.6) and a pre-NXNS version (BIND9 version 9.16.2). The three authoritative name servers are: a 'root', an attacker, and a malicious delegation authoritative name servers.

Most of the NRDelegationAttack measurements were carried out on a BIND9 version 9.16.6 resolver compiled to work with the local 'root' authoritative name server. The authoritative name servers are implemented with Name Server Daemon (NSD) version 4.3.3. The clients are deployed on the same machine, which was configured to send DNS queries directly to the local recursive resolver. The setup configuration and environment are provided in GitHub (*https://github.com/ShaniBenAtya/dnssim*).

In order to use DNS-FullProtocolSimulator, a docker docker is required.

*Member of the Checkpoint Institute of Information Security.

†The majority of this research was carried out while the author was at Reichman University. Member of the Checkpoint Institute of Information Security.

A.1.1 Security, privacy, and ethical concerns

To ensure that no harm may be done outside of the setup, the environment runs locally in closed Docker container environment. It is thus important to use "–dns 127.0.0.1" flag to configure this. Changing the "resolv.conf" configuration inside the docker container is not enough (see Appendix A.2).

A.1.2 How to access

DNS-FullProtocolSimulator source code can be found at DNS-FullProtocolSimulator GitHub (1.4 Tag). The environment docker image can be accessed through DockerHub (1.7 Tag).

A.1.3 Hardware dependencies

There is no hardware dependencies required for using DNS-FullProtocolSimulator. During our research, we used an Ubuntu computer or Virtual Machine (we recommend using Ubuntu 20.04 or above) which is capable of running Docker images according to "Install Docker Engine on Ubuntu" specification.

A.1.4 Software dependencies

1. Docker
2. WireShark (To install WireShark on Ubuntu use: `apt install wireshark`).
3. Kcachegrind (To install Kcachegrind on Ubuntu use: `apt install kcachegrind`).

A.1.5 Benchmarks

In order to conduct the experiments described in NRDelegationAttack paper (Section 5), the setup should contain a resolver with a non vulnerable to NXNSAttack version (e.g. bind-9.16.6) and at least two authoritative servers (local root authoritative and at least one more authoritative to simulate the "referral.com" authoritative in Figure 1). In addition, a malicious zone file is required for the attacker authoritative (i.e., "home.lan" server). The malicious zone file may contain

as many malicious name servers as necessary for the specific test, we used malicious referral list with 1500 NRDelegation name servers which all delegate the resolver to a server IP address that is non-responsive to DNS queries through another authoritative server (the local root server can also be used to delegate the resolver to this IP address).

For instance, if the malicious request from the client is "attacker.home.lan", the malicious referral response should include a long list of name servers. In order to create such referral list the "/env/nsd_attack/home.lan.forward" zone file needs to have 1500 records per one malicious request. That is, the malicious zone file includes 1500 records for the malicious request which leads to a none existent domain name (e.g., attacker IN NS ns0...1500.fake0...1500.fake) and the root authoritative server, which does not have any record to the non-responsive domain name, includes a wildcard record delegating the resolver to a non responsive IP address (e.g., * IN A 127.0.0.89).

A.2 Set-up

The following tree structure represent relevant folders and file in the environment with description for each one of them.

```
env
├── client ... (127.0.0.1).
├── resolver ... (127.0.0.1).
├── nsd_root ... (127.0.0.2) - Root
│   │             authoritative server
│   │             configuration folder.
│   ├── lan.forward ...
│   │   Zone file for SLD server
│   │   ".lan".
│   ├── lan.reverse
│   ├── net.forward ...
│   │   Zone file for root server
│   │   ".net".
│   ├── net.reverse
│   ├── nsd.conf ... Configuration file for NSD,
│   │                contains the IP address of
│   │                the root server.
│   └── nsd.db ... NSD DB, for internal NSD
│                  usage.
├── nsd_attack ... (127.0.0.200) "home.lan"
│   │               malicious authoritative
│   │               server configuration
│   │               folder, (which simulates
│   │               'referral.com" server in
│   │               Figure 1).
│   ├── home.lan.forward ...
│   │   Zone file for sld
│   │   ".home.lan", this sld
│   │   represents the malicious
│   │   authoritative.
│   ├── home.lan.reverse
│   ├── nsd.conf ... Configuration file for NSD,
│   │                contains the IP address of
│   │                the malicious authoritative
│   │                server.
│   └── nsd.db ... NSD DB, for internal NSD
│                  usage.
│      .3 named.conf ...
│      Bind9 configuration,
│      contains the IP address
│      of the local environment.
├── bind9_16_6, bind9_16_2 OR bind9_16_33 ...
│   Bind source code with
│   modification to use local
│   root server.
└── nsd ... NSD source code from
            https://github.com/NLnetLabs/nsd,
            this folder relevant in
            case of changes to the
            original NSD code (In our
            experiment we didn't change
            this code).
```

A.2.1 Installation

1. Pull the docker image from Docker Hub (`docker pull shanist/dnssim:1.7`).
2. Run the docker image as a container interactively so you can control the environment (`docker container run --dns 127.0.0.1 --mount type=bind,source =<local_folder_path>,target=/app -it shanist/ dnssim:1.7 /bin/bash`).
   It is important to use the `dns 127.0.0.1` flag so the environment DNS will be local, changing the `resolv.conf` file inside a Docker container does not work. Note that we are mounting `<local_folder_path>` to the folder `/app` inside the docker container so it will be easier to copy files to and from the docker container.
3. Now you have a terminal inside the environment.
4. In order to open another terminal for the environment first run `sudo docker container ls`, look for `dnssim` docker image name and copy its `<CONTAINER ID>`.
   Then, run `sudo docker exec -it <CONTAINER ID> bash`.
   To open more terminals into the environment, repeat this process.

To conduct the experiments described in NRDelegationAttack paper, the setup needs to include a resolver and at least two authoritative servers.

Environment IP address:

1. 127.0.0.1 – Client
2. 127.0.0.1 – Our own Resolver (The client and resolver have the same IP address)

3. 127.0.0.2 – Root authoritative
4. 127.0.0.200 – "home.lan" TLD authoritative (which simulates "referral.com" server in Figure 1).
5. 127.0.0.53 – The default resolver – DO NOT USE IT WHILE TESTING!

Authoritative Servers: Our authoritative servers are located at "/env/nsd_root" and "/env/nsd_attack". To use them, first configure their zone files which are located inside their folder and called "ZONE_NAME.forward". After changing the zone file a restart to the authoritative is required in order to apply the changes.

Resolver: First, go to the resolver implementation folder (We have bind-9.16.2 (Which is vulnerable to NXNSAttack), Bind-9.16.6 (Which is non-vulnerable to NXNSAttack and vulnerable to NRDelegationAttack) and bind-9.16.33 (Which is non-vulnerable to both attacks)). You can easily replace the Bind9 version by going to the correct Bind9 version folder (e.g.,"/env/bind9_16_6", "/env/bind9_16_2" or "/env/bind9_16_33") and run: `make install`. NOTE: The environment is pre-installed with Bind 9.16.6 which was the main Bind9 resolver version tested in NRDelegationAttack paper.

Starting the environment: Open three terminals in the Docker container: First, turn on the Resolver using the following commands:

```
cd /etc
named -g -c /etc/named.conf
```

If there is a key-error run `rndc-confgen -a` and try to start it again. If you are getting the error: "loading configuration: Permission denied", use the following commands to correct the error:

```
chmod 777 /usr/local/etc/rndc.key
chmod 777 /usr/local/etc/bind.keys
```

Now, turn on the Authoritative servers in a different environment terminal: Navigate to the Authoritative server folder (/env/nsd_attack and /env/nsd_root), then run in each authoritative server folder: `nsd -c nsd.conf -d -f nsd.db`

If there is an error stating that the port is already in use, run `service nsd stop` and start it again.

A.2.2 Basic Test

To make sure that the setup is ready and well configured, the following steps are required:

1. Run another shell inside the docker container using `docker exec -ti <container id> bash` and run `tcpdump -i lo -s 65535 -w /app/dump`
2. Query the resolver from within the docker container `dig firewall.home.lan` and make sure that the correct IP address is received, you should see `Address: 127.0.0.207`
3. Stop `tcpdump` (you can use `^C`), Open WireShark, load the file `<local_folder_path>/dump` and filter DNS requests. You should observe the whole DNS resolution route for the domain name requested (`firewall.home.lan`).
   (a) `firewall.home.lan` query from client to resolver (ip `127.0.0.1` to ip `127.0.0.1`)
   (b) Resolver query to the root server (from `127.0.0.1` to `127.0.0.2`)
   (c) Root server return the SLD address (from `127.0.0.2` to `127.0.0.1`)
   (d) Resolver query the SLD (from `127.0.0.1` to `127.0.0.200`)
   (e) SLD return the address for the domain name (`127.0.0.207`)
   (f) Resolver return the address to the client (`127.0.0.207`)

NOTE: The address `firewall.home.lan` is configured in `/env/nsd_attack/home.lan.forward` and by performing the above test ensures that the resolver accesses the authoritative through the root server.

A.3 Evaluation workflow

As explained in Appendix A.1.5, in order to test NRDelegationAttack using DNS-FullProtocolSimulator a client, a resolver (The environment is pre-installed with Bind 9.16.6 which was the main Bind9 resolver version tested in NRDelegationAttack paper) and at least two authoritative servers with pre-configured zone files are required. See Appendix A.1.5 for detailed example of such zone file configuration.

A.3.1 Major Claims

(C1): If the number of names in the referral list is large, e.g., 1,500, then each NRDelegationAttack malicious packet costs at least 5,600 times more CPU instructions relative to a benign query. This is proven by experiment (E1) below, as described in Section 5.2 and whose results are reported in Figure 3. The reproduction (proof) of this claim does not require significant resources of any sort, neither compute nor memory (10 human minutes + 2 compute minutes).

(C2): The resolver exhibited a significant performance degradation in its throughput measurements during the NRDelegationAttack. This is may be proven by experiment (E2) below, as described in Section 6.2 and whose results are reported in Figure 5 in the paper. While in the paper this test and measurements were done on a cloud setup with each DNS component implemented on a separate server, here we show that the same test can be carried out on the closed virtual setup. The results on this isolated virtual setup provide an indication of the phenomena and are

not reliable as those presented in Section 6 in the paper. The reproduction (proof) of this claim does not require significant resources of any sort, neither compute nor memory (20 human minutes + 4 compute minutes).

(C3): The attack is empowered mainly due to the NXNSAttack mitigations: NRDelegationAttack is much worse for NXNS-patched servers than unpatched servers (See Section 4). This is proven by experiment (E3) below. The reproduction (proof) of this claim does not require significant resources of any sort, neither compute nor memory (10 human minutes + 2 compute minutes).

(C4): The proposed NRDelegationAttack mitigation greatly reduces the attacks effectiveness (see Section 8). This is proven by experiment (E4) below. The reproduction (proof) of this claim does not require significant resources of any sort, neither compute nor memory (10 human minutes + 2 compute minutes).

(C5): NRDelegationAttack affects open resolvers as well as the vendors. This claim is problematic to reproduce due to ethical considerations, in addition, most of the open resolvers patched their implementations to NRDelegationAttack as part of the responsible disclosure procedure (see Section 7 in the paper).

A.3.2 Experiments

For the following experiments, Resperf, Valgrind and Kcachegrind tools are needed.

(E1): Instructions measurement experiment

Preparation: For this experiment Valgrind and Kcachegrind are required:
First, make sure that your resolver is configured to use Bind9.16.6 resolver which is patched to NXNSAttack (first run: `cd /env/bind9_16_6` and then run: `make install`). Turn on the resolver with the Valgrind tool with the following command (make sure to run the resolver from "/etc") `valgrind --tool=callgrind named -g -c /etc/named.conf`. In addition, the malicious referral response should include a long list of name servers, in order to create such referral list the "/env/nsd_attack/home.lan.forward" zone file needs to have 1500 records per one malicious request. For example, you can create one malicious request using a short script we provided (`python /env/reproduction/genAttackers.py`) that generates the malicious request configuration and copy its output from the "attackerNameServers.txt" output file into the zone file For your convenience we uploaded our "/env/nsd_attack/home.lan.forward" zone file which includes our attackers to "/env/reproduction" folder.

Execution: Query the resolver with a malicious query (e.g., "dig attack0.home.lan"). Stop the resolver and restart it using with Valgrind as explained before. Query the resolver with a legitimate query (e.g., "dig test.home.lan").

Results: Copy the results file from the docker container `/etc/callgrind.VALGRIND_TEST_NUMBER` (which is the folder from which the resolver is executed) to the host, `cp /path/in/docker/callgrind.VALGRIND_TEST_NUMBER /app/` so you could access the file in `<local_folder_path>` alternatively you can use (`docker cp <CONTAINER_ID>:/path/in/docker/callgrind.VALGRIND_TEST_NUMBER /path/in/host`.
Note that the `VALGRIND_TEST_NUMBER` is a number given by Valgrind. Open the results files in Kcachgrind: first add permissions to open the file (`sudo chown USERNAME:USERNAME OUTFILE_NAME`) and then open the file: (`kcachgrind ./OUTFILE_NAME`).
In the tool, choose `Instructions Fetch` tab and record the `Incl.` value of `fctx_getaddresses` function.
Please make sure that the "relative" button is unchecked.
Repeat this step with each file and compare the results.
Benign query results should be around 200,000 instructions, while the malicious query should have more than 2,000,000,000.

(E2): Throughput measurement experiment

Preparation: For this experiment Resperf is required:
To configure the malicious authoritative zone file ("/env/nsd_attack/home.lan.forward" file) with multiple malicious domain names (multiple attackers, each of them configured as explained in E1 and multiple benign domain names, you can use the script we provided (`python /env/reproduction/genAttackers.py`) and change the number of attackers generated by changing the ATTACKERS_NUM variable (e.g. ATTACKERS_NUM = 50). Note that the zone file length is bounded by the file size, therefore we used only 50 different attacker malicious requests in our measurements. In addition, we uploaded a script that generates a list of names for the Resperf tool (`python /env/reproduction/genNamesToCheck.py`), the script creates two output files: "benignNamesE2.txt" for benign user domain names and "attackerNamesE2.txt" for that attacker.

Execution: First make sure that the resolver is using Bind 9.16.6 version (checking Bind9 version can be done by running `named -v` on the resolver terminal and changing its version can be done by running: `cd /env/bind9_16_2` and then: `make install`) and that both the resolver and authoritative servers are running.
Benign and malicious users commands are being executed from two terminals inside the docker container, so two instances of Resperf tool are required:
The first simulates the attacker and issues queries each time at a fixed rate, and the second tool ramps up the benign user requests until things start to fail.
The malicious user command should be run first but ultimately in parallel to the benign user command.
In your benign user run: `resperf -d INPUT_FILE -s`

`127.0.0.1 -v -R -P OUTPUT_NAME.`
And from the malicious user run: `resperf -d INPUT_FILE -s 127.0.0.1 -v -m 15000 -c 60 -r 0 -R -P OUTPUT_NAME`
Where: -m is the number of QPS that are sent, -c is the duration of time in which Resperf tries to send the queries, -r is the duration of time in which Resperf ramps-up before sending the packets in a constant time, we want the ramp-up to be zero and OUTPUT_NAME is the output file name of your choice (make sure to use different file names for each test).
The benign and malicious input files should include only benign or malicious domain names respectively.
You should run two "sub"-experiments: First, you need to measure the effect of the attack on benign users throughput, in this experiment "INPUT_FILE" = "benignNamesE2.txt" for benign user, and "INPUT_FILE" = "attackerNamesE2.txt" for attacker user. Then, to measure the resolver throughput without any attack restart the resolver (in order to test with clean cache), and run the experiment using the "benignNamesE2.txt" file created using the "genNamesToCheck.py" script as the "INPUT_FILE" to both Resperf commands.

Results: Open only the benign output files from both "sub"-experiments using a text editor (from both the benign and attacker terminal) and compare the benign user throughput presented in the "responses_per_sec" column. A major obstacle in reproducing the experiment was the use of docker instead of multiple clients. The experiment required different computers to send attack and benign queries simultaneously and was originally done using our cloud setup environment as described in Section 6.2 of, but using the docker forced us to use only one machine for all the tests. Nevertheless when performing the test within the docker, we are still able to observe that the resolver throughput for benign users while the attack takes place is degraded. The difference was much smaller than in the original experiment (Sec 6.2), but it was statistically significant and can be mainly seen at the first 10 lines of the output file (in which the attack QPS is high and the resolver throughput is degraded by more than 90% (actual_qps are around 5000 but the responses_per_sec are around 100) or even complete denial of service).
Nevertheless, we are still able to observe that the resolver throughput for benign users while NRDelegationAttack takes place is degraded. The difference was much smaller than in the original experiment, but it was statistically significant and can be mainly seen at the first 10 lines of the file (in which the attack QPS is high and the resolver throughput is degraded by more than 90% ("actual_qps" are around 5000 but the "responses_per_sec" are around 100) or even complete denial of service).

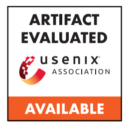
(E3): Instructions measurement experiment - NXNSAttack unpatched server
Preparation: For this experiment follow E1 instructions using Bind9.16.2 resolver (which is not patched to NXNSAttack) instead of Bind9.16.6 resolver (which is patched to NXNSAttack). Checking Bind9 version can be done by running `named -v` on the resolver terminal and changing its version can be done by running: `cd /env/bind9_16_2` and then: `make install`).
Execution: Follow E1 instructions.
Results: Follow E1 instructions. Benign query results should be around 200,000 instructions, while the malicious query should be around 200,000,000.

(E4): Instructions measurement experiment - NRDelegationAttack mitigation
Preparation: For this experiment follow E1 instructions using Bind9.16.33 resolver (which is not patched to NRDelegationAttack). Checking Bind9 version can be done by running `named -v` on the resolver terminal and changing its version can be done by running: `cd /env/bind9_16_2` and then: `make install`).
Execution: Follow E1 instructions.
Results: Follow E1 instructions. Benign query results should be around 200,000 instructions, while the malicious query should less than 10,000,000.

Acknowledgements: The authors are grateful to the USENIX Security artifact referees for their dedicated careful review and discussions which have significantly improved the artifact.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix
ACon^2: Adaptive Conformal Consensus for Provable Blockchain Oracles

Sangdon Park† *Osbert Bastani** *Taesoo Kim*†

†Georgia Institute of Technology *University of Pennsylvania

A Artifact Appendix

A.1 Abstract

Our paper proposes an online learning algorithm, called Adaptive Conformal Consensus. Our artifact consists of source code, datasets, docker files, and scripts to generate paper results. We aim for *Artifacts Available*, *Artifacts Functional*, and *Results Reproduced* badges.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Code of our artifact will run a proposed machine learning algorithm over Python without external communication and a local blockchain with a forked Ethereum mainnet, so we do not expect to see any security, privacy, or ethical concerns. Note that in forking Ethereum mainnet, a script will use an author's API key for Alchemy, so we would not expect related security, privacy, and ethical issues.

A.2.2 How to access

Our artifacts are accessible via Github `https://github.com/sslab-gatech/ACon2/tree/AEStableVersion`[1].

A.2.3 Hardware dependencies

We expect a standard computing environment, i.e., a computing machine with CPU, HDD, and Internet access. In particular, a 4 or 5 core CPU machine would be preferred for multi-processing. The results and docker require about 4 GB HDD. Internet access is required to fork the Ethereum mainnet during experinents.

A.2.4 Software dependencies

Docker is required, as we provide docker images for reproducing our results.

[1] git clone –depth 1 –branch AEStableVersion git@github.com:sslab-gatech/ACon2.git

A.2.5 Benchmarks

We include required datasets (i.e., USD/ETH data and INV/ETH data) into docker images; thus, additional actions to get datasets are not required.

A.3 Set-up

A.3.1 Installation

Our code repository is cloned via `git clone -depth 1 -branch AEStableVersion git@github.com:sslab-gatech/ACon2.git`. We provide docker files, so Docker needs to be installed. Other than these, all executions are done over docker images.

A.3.2 Basic Test

Once two docker images are installed and the code repository is cloned, (1) change the working directory to `python` and execute `./docker_scripts/docker_plot_INV_ETH_precomp.sh`; and (2) change the working directory to `solidity` and execute `./docker_scripts/plot_sim_precomp.sh`. These two scripts sould not introduce errors if set-up is right.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *ACon^2 generates consensus sets that follows well USD/ETH price data change when $K = 1$. This is proven by the experiment (E1) whose results are illustrated in Figure 4(a).*

(C2): *ACon^2 generates consensus sets that follows well USD/ETH price data change when $K = 2$. This is proven by the experiment (E2) whose results are illustrated in Figure 4(b).*

(C3): *ACon^2 generates consensus sets that follows well USD/ETH price data change when $K = 3$. This is proven by the experiment (E3) whose results are illustrated in Figure 4(c).*

(C4): *ACon^2 generates consensus sets that satisfy a desired pseudo-miscoverage rate over USD/ETH price data when $K = 1$. This is proven by the experiment (E4) whose results are illustrated in Figure 5(a).*

(C5): *ACon^2 generates consensus sets that satisfy a desired pseudo-miscoverage rate over USD/ETH price data when $K = 2$. This is proven by the experiment (E5) whose results are illustrated in Figure 5(b).*

(C6): *ACon^2 generates consensus sets that satisfy a desired pseudo-miscoverage rate over USD/ETH price data when $K = 3$. This is proven by the experiment (E6) whose results are illustrated in Figure 5(c).*

(C7): *ACon^2 generates reasonable small consensus sets over USD/ETH price data when $K = 3$. This is proven by the experiment (E7) whose results are illustrated in Figure 6(a).*

(C8): *a baseline algorithm σ-ACon^2 generates large consensus sets and conservative pseudo-miscoverage rates over USD/ETH price data when $K = 3$. This is proven by the experiment (E8) whose results are illustrated in Figure 9(a) and 9(b).*

(C9): *ACon^2 generates meaningful consensus sets under price manipulation, while trigger alarms for downstream applications over INV/ETH price data. This is proven by the experiment (E9) whose results are illustrated in Table 1 and Figure 1.*

(C10): *ACon^2 generates consensus sets that follows well INV/ETH price data change when $K = 1$. This is proven by the experiment (E10) whose results are illustrated in Figure 7(a).*

(C11): *ACon^2 generates consensus sets that follows well INV/ETH price data change when $K = 2$. This is proven by the experiment (E11) whose results are illustrated in Figure 7(b).*

(C12): *ACon^2 generates consensus sets that follows well INV/ETH price data change when $K = 3$. This is proven by the experiment (E12) whose results are illustrated in Figure 7(c).*

(C13): *ACon^2 generates consensus sets that satisfy a desired pseudo-miscoverage rate over INV/ETH price data when $K = 1$. This is proven by the experiment (E13) whose results are illustrated in Figure 8(a).*

(C14): *ACon^2 generates consensus sets that satisfy a desired pseudo-miscoverage rate over INV/ETH price data when $K = 2$. This is proven by the experiment (E14) whose results are illustrated in Figure 8(b).*

(C15): *ACon^2 generates consensus sets that satisfy a desired pseudo-miscoverage rate over INV/ETH price data when $K = 3$. This is proven by the experiment (E15) whose results are illustrated in Figure 8(c).*

(C16): *ACon^2 generates reasonable small consensus sets over INV/ETH price data when $K = 3$. This is proven by the experiment (E16) whose results are illustrated in Figure 6(b).*

(C17): *ACon^2 generates reasonable small consensus sets and achieves a desired pseud-miscoverage rate over local Ethereum network data when $K = 3$. This is proven by the experiment (E17) whose results are illustrated in Figure 10(a) and 10(b).*

(C18): *ACon^2 achieves a desired pseudo-miscoverage rate over local Ethereum network data with different K and α. This is proven by the experiment (E18) whose results are illustrated in Figure 11(a), 11(b), and 11(c).*

(C19): *ACon^2 uses a reasonable gas amount for computation. This is proven by the experiment (E19) whose results are illustrated in Table 2.*

A.4.2 Experiments

This section includes detailed instructions to reproduce results. Also, see https://github.com/sslab-gatech/ACon2/tree/AEStableVersion, which contains instructions with pre-computed data, which do not require heavy computation. Note that the measured compute-hours are estimated based on a server-level environment (i.e., 128 2GHz-CPUs with 500G memory); we expect one CPU with at least 500MB memory as minimal requirements, but the actual computation time could vary, depending on a HW setup.

Common preparation step.

1. Install Docker
2. Pull docker images via `dockerpullghcr.io/sslab-gatech/acon2:latest` and `dockerpullghcr.io/sslab-gatech/acon2-sol:latest`
3. Clone our code repository

(E1-8): *[0 human-minutes + 30 compute-hour + 5GB disk]: This experiment generates results for Figure 4, Figure 5, Figure 6(a), and Figure 9.*
How to: *First collect required data by executing a script.*
Preparation: *change the working directory to* `python`
Execution: *Run* `./docker_scripts/docker_run_USD_ETH.sh` *and Run* `./docker_scripts/docker_plot_USD_ETH.sh`
Results: *Ways to interpret results are described in* ***(E1-8)***

(E1): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 4(a).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 4(a), see* `output_docker/one_source_USD_ETH_UniswapV2_K_1_beta_0/figs/plot_ps.pdf`

(E2): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 4(b).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`

Results: *For Figure 4(b), see* `output_docker/two_sources_USD_ETH_UniswapV2_coinbase_K_2_beta_1/figs/plot_ps.pdf`

(E3): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 4(c).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 4(c), see* `output_docker/three_sources_USD_ETH_UniswapV2_coinbase_binance_K_3_beta_1/figs/plot_ps.pdf`

(E4): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 5(a).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 5(a), see* `output_docker/one_source_USD_ETH_UniswapV2_K_1_beta_0/figs/plot_miscoverage.pdf`

(E5): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 5(b).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 5(b), see* `output_docker/two_sources_USD_ETH_UniswapV2_coinbase_K_2_beta_1/figs/plot_miscoverage.pdf`

(E6): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 5(c).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 5(c), see* `output_docker/three_sources_USD_ETH_UniswapV2_coinbase_binance_K_3_beta_1/figs/plot_miscoverage.pdf`

(E7): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 6(a).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 6(a), see* `output_docker/one_source_USD_ETH_UniswapV2_K_1_beta_0_two_sources_USD_ETH_UniswapV2_coinbase_K_2_beta_1_three_sources_USD_ETH_UniswapV2_coinbase_binance_K_3_beta_1/figs/plot_size.pdf`

(E8): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 9(a,b).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 9(a), see* `output_docker/three_sources_OneSigma_USD_ETH_UniswapV2_coinbase_binance_K_3_beta_1/figs/plot_ps.pdf` *and for Figure 9(b), see* `output_docker/three_sources_OneSigma_USD_ETH_UniswapV2_coinbase_binance_K_3_beta_1/figs/plot_miscoverage.pdf`

(E9-16): *[0 human-minutes + 2 compute-hour + 5GB disk]: This experiment generates results for Table1, Figure 1, Figure 7, Figure 8, and Figure 6(a).*
How to: *First collect required data by executing a script.*
Preparation: *change the working directory to* `python`
Execution: *Run* `./docker_scripts/docker_run_INV_ETH.sh` *and Run* `./docker_scripts/docker_plot_INV_ETH.sh`
Results: *Ways to interpret results are described in* ***(E9-16)***

(E9): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Table 1 and Figure 1.*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Table 1, see* `stdout` *of* `./docker_scripts/docker_plot_INV_ETH.sh` *and for Figure 1, see* `output_docker/highlight/figs/plot_ps.pdf`

(E10): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 7(a).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 7(a), see* `output_docker/one_source_INV_ETH_SushiSwap_K_1_beta_0/figs/plot_ps.pdf`

(E11): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 7(b).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 7(b), see* `output_docker/two_sources_INV_ETH_SushiSwap_UniswapV2_K_2_beta_1/figs/plot_ps.pdf`

(E12): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 7(c).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 7(c), see* `output_docker/three_sources_INV_ETH_SushiSwap_UniswapV2_coinbase_K_3_beta_1/figs/plot_ps.pdf`

(E13): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 8(a).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 8(a), see* `output_docker/one_source_INV_ETH_SushiSwap_K_1_beta_0/figs/plot_miscoverage.pdf`

(E14): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 8(b).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 8(b), see* `output_docker/two_sources_INV_ETH_SushiSwap_UniswapV2_K_2_beta_1/figs/plot_miscoverage.pdf`

(E15): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 8(c).*
How to: *Check a generated figure.*

Preparation: *change the working directory to* `python`
Results: *For Figure 8(c), see* `output_docker/three_sources_INV_ETH_SushiSwap_UniswapV2_coinbase_K_3_beta_1/figs/plot_miscoverage.pdf`

(E16): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 6(b).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `python`
Results: *For Figure 6(b), see* `output_docker/one_source_INV_ETH_SushiSwap_K_1_beta_0_two_sources_INV_ETH_SushiSwap_UniswapV2_K_2_beta_1_three_sources_INV_ETH_SushiSwap_UniswapV2_coinbase_K_3_beta_1/figs/plot_size.pdf`

(E17-19): *[0 human-minutes + 30 compute-hour + 5GB disk]: This experiment generates results for Table 2, Figure 10, and Figure 11.*
How to: *First collect required data by executing a script.*
Preparation: *change the working directory to* `solidity`
Execution: *Enter into the docker image via* `./docker_scripts/enter.sh`*, execute* `./scripts/run.sh`*, execute* `./scripts/run_baseline.sh`*, exit from the docker image, and generate plots via* `./docker_scripts/plot_sim.sh`*.*
Results: *Ways to interpret results are described in **(E17-19)***

(E17): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 10(a,b).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `solidity`
Results: *For Figure 10(a), see* `output_docker/figs/acon2/plot-ps-K-3-alpha-0d01-iter-1.pdf` *and for Figure 10(b), see* `output_docker/figs/acon2/plot-error-var-K-3-alpha-0d01.pdf`

(E18): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Figure 11(a-c).*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `solidity`
Results: *For Figure 11(a), see* `output_docker/figs/acon2/plot-error-var-K-3-alphas.pdf`*, for Figure 11(b), see* `output_docker/figs/acon2/plot-error-var-K-4-alphas.pdf`*, and for Figure 11(c), see* `output_docker/figs/acon2/plot-error-var-K-5-alphas.pdf`*,*

(E19): *[1 human-minutes + 1 compute-minutes + 5GB disk]: This experiment generates results for Table 2.*
How to: *Check a generated figure.*
Preparation: *change the working directory to* `solidity`
Results: *For Table 2, see* `stdout` *of* `./docker_scripts/plot_sim.sh`*.*

In all of the above blocks, please provide indications about the expected outcome for each of the steps (given the suggested hardware/software configuration above).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: SANDDRILLER: A Fully-Automated Approach for Testing Language-Based JavaScript Sandboxes

Abdullah AlHamdan, Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security
{abdullah.alhamdan, staicu}@cispa.de

A Artifact Appendix

A.1 Abstract

In this artifact, we showcase SANDDRILLER, the first tool for testing language-based sandboxes JavaScript, supporting both client-side and server-side sandboxes. SANDDRILLER takes as input a set of self-contained JavaScript files (corpus), interposes oracles using instrumentation, and executes each instrumented test case inside a target sandbox. Additionally, SANDDRILLER also recombines ingredients from known exploits to generate variants of the files in the corpus that are more likely to trigger a bug in the target sandboxes. We present experiments that demonstrate how SANDDRILLER works overall (single input file, variant generator) and partially replicate important results from the paper (V8 tests corpus with the `vm2` sandbox). SANDDRILLER is publicly available at https://github.com/vdata1/SandDriller. For this artifact evaluation, we seek the following badges: available and functional.

A.2 Description & Requirements

In this section, we describe the software and hardware requirements for running SANDDRILLER and explain how to obtain the corpora used in the evaluation of the paper.

A.2.1 Security, privacy, and ethical concerns

There are no risks for the reviewers of this artifact with respect to security and privacy of their machines. SANDDRILLER identified 12 zero-day vulnerabilities in widely-used, open-source sandboxes. A security advisory was published for each of these findings.

A.2.2 How to access

The source code is available on our GitHub repository https://github.com/vdata1/SandDriller/releases/tag/1.0, including an important README file that describe in detail the usage of the tool and the different configuration options available.

A.2.3 Hardware dependencies

For the paper's evaluation, we ran SANDDRILLER on a server with 64 Intel Xeon E5-4650L@2.60GHz CPU cores and 768GB of memory. However, SANDDRILLER does not require any specific hardware feature, so it can run on any other machine running a Linux distribution. Nonetheless, since SANDDRILLER makes extensive use of multi-threading during testing, we recommend that the number of parallel workers be set to maximum the number of physical threads available on the machine. In Section A.4.2, we explain how to set this important configuration option.

A.2.4 Software dependencies

While SANDDRILLER might run on other operating systems, we require a Linux distribution for the evaluation. For obtaining the results described in this artifact, we require running the tool using Node.js version 14.15. We recommend using the Node Version Manager (nvm) [1] to install this exact version of Node.js. We also expect the machine to have `git` and `npm` installed and correctly configured. To showcase the tool's capability we run tests using two sandboxes: `vm2` and `safe-eval`. Both these tools are declared as third-party dependency in the `package.json` file, so they do not need to be installed separately. However, since we do not ship them with our tool, the input corpora need to be downloaded and configured separately, as described below.

A.2.5 Benchmarks

To test the sandboxes, we used ECMAScript Conformance Test Suite and V8 engine's test suites as benchmarks. SANDDRILLER takes each of these tests, instrument them, and run them inside the sandbox under test. In our experiments, we used ECMAScript test cases available at https://github.com/tc39/test262/tree/99b2a70789b27d433f9036b98572a4443d91e01f/test, and V8 test cases available at https://github.com/nodejs/node/tree/e46c680bf2b211bbd52cf959ca17ee98c7f657f5/deps/

[1] https://github.com/nvm-sh/nvm/

v8/test/mjsunit. As described below, during installation, these repositories need to be cloned into a specific subfolder.

A.3 Set-up

A.3.1 Installation

First, clone the SANDDRILLER repository locally in a directory we will call $PATH_TO_SANDDRILLER. To install the required third-party packages, run in the project's main directory `npm -prefix . install ..` This will install all the required dependencies and all the npm-available (vulnerable) sandboxes used in our experiments.

Next, clone the V8 corpus inside the $PATH_TO_SANDDRILLER/Dataset/ folder:

```
cd Dataset
git clone https://github.com/nodejs/node/
git reset --hard e46c680
cd ..
```

A.3.2 Basic Test

To run a simple test with SANDDRILLER, go to test directory by typing on the command line `cd $PATH_TO_SANDDRILLER/test` and run `node run-multi-proc.js`.
As a result, RESULTS.csv will be written on $PATH_TO_SANDDRILLER/Results/ showing the test results of SANDDRILLER. Results will also be shown on the terminal, i.e., a JSON result object for each test, followed by a summary of the results.

When executing SANDDRILLER on a fresh installation, it uses a toy corpus containing three JavaScript files. SANDDRILLER should report two security violations and a crash for this corpus.

A.4 Evaluation workflow

A.4.1 Major Claims

SANDDRILLER is a testing approach for automatically detecting sandbox escape vulnerabilities in real-world JavaScript sandboxes. To successfully isolate untrusted code, JavaScript sandboxes must block access to foreign references and prevent the side effects of hosting third-party code during runtime, such as getting stuck in an endless loop. SANDDRILLER aims to automatically synthesize exploits that violate this objective by escaping the sandbox. It first interposes checks that at execution time detect foreign references pointing outside of the sandbox, and subsequently exploits such references, creating an end-to-end exploit.

We make the following claims about our prototype:

(C1): *Starting with (a set of) benign JavaScript file(s) as input,* SANDDRILLER *can detect problematic references at runtime, which can be used to escape the sandbox. Concretely,* SANDDRILLER *synthesizes exploits that attempt to access privileged operations outside the sandbox, and/or test write values into the global scope, outside the sandbox. We have conducted experiment (E1) and (E3) as described in Section A.4.2 to demonstrate this capability.*

(C2): SANDDRILLER *can construct exploits by synthesizing variants of the input files. By using a variant generator it provides more comprehensive tests for verifying the security of a sandbox. We have demonstrated this through experiment (E2) as described in Section A.4.2.*

These claims collectively support our paper's main assertion that SANDDRILLER is an effective testing approach for automatically detecting zero-day sandbox escape vulnerabilities.

A.4.2 Experiments

Since replicating our entire testing campaign would incur a significant effort on the reviewers' side, we show how SANDDRILLER works with a single file as input and with a fairly large corpus (5,000+ JavaScript files from the V8 tests), for both using a single sandbox and a single Node.js version.

(E1): *[First exploit] [For one valid test case: 15 human-minutes + 5 compute-minutes]:* In this experiment, we show step-by-step how to come up with the first working exploit using vm2 sandbox and a single test case.

How to: To successfully accomplish this experiment, we start with configuring and running our tool, then run the instrumented code on the sandbox, and finally, apply delta debugging to reach the minimal working code for the exploit.

Preparation: First, we start editing the source code of the tool for the corpus for this experiment by uncommenting line number **138** in test/run-multi-proc.js (set regress-746909.js as the single file in the corpus). Moreover, go to test/process-runner.js and make vm2 as the sandbox to test by editing line number **10** to `const sandbox = "vm2";`.

Execution: Run SANDDRILLER as mentioned in A.3.2. As a result, SANDDRILLER will write the successful instrumented test case on path /tmp/res/. First, let us inspect the content of the generated file regress-746909.js, which is an instrumented version of the file in our corpus https://github.com/nodejs/node/blob/main/deps/v8/test/mjsunit/regress/regress-746909.js. It contains code for verifying potential foreign references and for attempting to escape the sandbox using such references. For example, for each function invocation, SANDDRILLER obtains the root prototype of its result `let grtA = getRootPrototype(r);` and attempts to modify the corresponding root prototype `grtA.FIA = 'FI: Got it?';`. Copy the exploit (instrumented

test case) and host it on the target sandbox, vm2. To this end, we provided template for each sandbox in `templates_sandboxes` for time-saving. Paste the generated exploit in `exploit.js` and run the `vm2.js` file using Node.js. Observe how the global root prototype changes as a result of running the code inside the sandbox, i.e., a new property is added on the root prototype. Optionally, apply manual delta debugging by deleting the unnecessary lines of code in the exploit to get the minimal working exploit. To that end, after any removing spurious lines in the code, rerun the exploit to verify that the desired side-effect in the global scope is still present. To get the minimal working exploit, a sequence of lines of code deletion and reruning of the code might be required. For this specific test case, we recommend deleting the first **122, 139-148, 150-180, 180-192, 204-end of the code** lines to get a working exploit. The code at this stage requires more editing to produce the final working exploit showed in Figure 1 of the paper. However, we provided a working exploit for this experiment, showing the possible steps performed during delta debugging in `demo_results/E1.js`.
Results: After completing the experiment, the detailed result can be found in `Results/RESULTS.csv`

(E2): *[Use the Variant Generator] [For one valid test case 20 human-minutes + 5 computing-minutes]:*
How to: To successfully accomplish this experiment, we start with configuring our tool to **enable** the generator, then run the instrumented code on the sandbox, and finally, apply delta debugging to reach the minimal working code for the exploit.
Preparation: First we enable the generator in `test/process-runner.js` by assigning `const useGenerator = true` at line **11** and using `safe-eval` as the sandbox to test. Then, uncomment line number **`141`** in `test/run-multi-proc.js`, and comment line 138.
Execution: Run SANDDRILLER as mentioned in A.3.2. As a result, SANDDRILLER will write variants of successful instrumented test cases with on path `/tmp/res/`. Copy the instrumented test case, named "`array-push2_v1.js`" and host it on the target sandbox, `safe-eval`. As before, we provided a template for each sandbox in `templates_sandboxes` for time-saving: paste the explit in the `exploit.js` file and run `safe-eval.js` with Node.js to observe its side effect in the global scope (a new property on the root prototype). We Optionally, apply delta debugging by deleting the unnecessary lines of code in the exploit to get the minimal working exploit. For this specific test case, we recommend deleting **3-123, 127-157, 129-162, 165-196, 199-261, 266-284, 289-422, 426-514** lines to get working exploit. The code at this stage requires more editing to come up with the final working exploit. However, we provided a working exploit for this experiment, showing a possible result of delta debugging in `demo_results/E2.js`. We draw the reader's attention to the code `throw function thrower() {...}`, which is a code fragment that was not part of the original file in the corpus, but was injected by the variant generator.
Results: After completing the experiment, the statistical result can be found in `Results/RESULTS.csv`

(E3): *[Running* SANDDRILLER *on a benchmark] [5 human-minutes + 1 computing-hour]:*
How to: To successfully accomplish this experiment we run SANDDRILLER on a fairly large corpus consisting of 5,000+ V8 tests. For simplicity, the variant generator is disabled.
Preparation: First make sure to delete all generated test cases on `/tmp/res` and delete the results file to work in a clean environment. Then, uncomment line number **`144`** in `test/run-multi-proc.js` and change the number of threads to 16 at line 12 by `const POOL_SIZE = 16;`. Make sure the generator is disabled by setting `const useGenerator = false;`. The sandbox from the previous experiment should remain the same for this experiment.
Execution: Run SANDDRILLER as mentioned in A.3.2.
Results: After completing the experiment, the detailed results can be found in `Results/RESULTS.csv`, and a summary of the the experiment will be printed in the terminal. The results list all the tests in which SANDDRILLER succeeded to break the sandbox, test cases that cause a hard crash of the sandbox, and more details such as execution time or number of oracle checks performed at runtime. Feel free to attempt hosting any of the produced exploits inside the vulnerable sandbox (`safe-eval`) and observe its side effect in the global scope, as before.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Instructions Unclear: Undefined Behaviour in Cellular Network Specifications

Daniel Klischies*, Moritz Schloegel†, Tobias Scharnowski†
Mikhail Bogodukhov‡, David Rupprecht§, Veelasha Moonsamy*

*Ruhr University Bochum, †CISPA Helmholtz Center for Information Security
‡Independent, §Radix Security

A Artifact Appendix

A.1 Abstract

The artifacts for *Instructions Unclear: Undefined Behaviour in Cellular Network Specifications* consist of two main parts: The **TLA+ models** used to discover undefined behaviours and the **modified `srsRAN` implementations** used to test smartphone implementations of undefined behaviour. For each of the three LTE features, PWS, SMS, and RRC that we evaluate against, there is a separate TLA+ model and `srsRAN` version.

This document describes how to use the models to derive concrete examples of undefined behaviour in LTE specifications, and then employ our `srsRAN` forks to replay these concrete examples against a commercial UE. This way, one can verify the presence of undefined behaviour in multiple LTE feature specifications, as well as determine their real world impact.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Evaluating the TLA+ models for undefined behaviour does not entail any direct risks.

Replaying the counter examples using `srsRAN` and an SDR against real UEs, however, imposes multiple risks:

1. You will create a (small) LTE cell; doing so without an RF shielding box might violate local laws.

2. Since you will replay network packets (PDUs) that you cannot target to a specific phone, any phone in the vicinity of your SDR will receive these PDUs and might be temporarily or permanently affected by that PDU. As our experiments have shown, this may cause a Denial of Service attack, information leakage, or potentially even more severe problems. This is why it is absolutely essential to use an RF shielding box, even if your local laws would permit running the experiment without it.

3. You might brick any phones that you test. We had at least one non-reproducible case where a phone temporarily refused any connection until we reset its NVRAM.

4. If misconfigured, you might brick the SDR that you are using. To mitigate this, we recommend that you read and obey its instruction manual carefully.

A.2.2 How to access

The artifact is available for download at https://zenodo.org/record/8013704.

A.2.3 Hardware dependencies

TLA+ models. The TLA+ models are CPU intensive. To reproduce the CPU time in Table 1 of our paper, a dual Intel Xeon Gold 6230R system, totaling 52 cores with 128 GB RAM is required. Using a different setup is possible, but might result in a different core-hour measurement. All other results will remain the same, no matter which CPU is used.

Replaying counter examples via `srsRAN`. To be able to replay concrete examples of undefined behaviour, the following components are required:

- Computer with Gigabit LAN
- Ettus Research USRP X300 with SFP+ to Gigabit RJ45 module
- 2x Ettus Research VERT900 antennas
- GPSDO for USRP X Series (PCB-Mounted GPS-Disciplined OCXO)
- Programmable SIM-card (e.g. Sysmocom sysmoISIM-SJA2)
- RF shielding box (faraday cage)
- Smartphone(s): We evaluate against a Samsung A41, Samsung S20 5G (European edition), Oppo A73 5G, Huawei P40 Lite 5G and OnePlus 8. To reproduce our exact results, all of these phones must be updated to firmware patch level *April 2022*

for the SMS and PWS tests and *January 2023* to reproduce the RRC results.

A.2.4 Software dependencies

TLA+ models. To run the TLA+ models, a `tlc` is required. We recommend to use https://github.com/pmer/tla-bin. This requires Linux or a BSD derivative with at least `Java 11` and `curl` installed.

Replaying counter examples via `srsRAN`. Installing srsRAN 4G requires multiple external libraries, drivers, and firmware for the USRP. As the requirements of our forks mostly match the requirements of the upstream version, we recommend following the official setup guide at https://docs.srsran.com/projects/4g/en/latest/general/source/1_installation.html#installation-from-source.

SMS: For the SMS fork, you will also have to install `libosmocore` (we tested with version `1.6.0`), `Python 3.8` and the `pyzmq` package (version `22.3.0`).

Virtualisation: We recommend *against* using virtual machines to run srsRAN, as the overhead induced by the virtualisation interferes with the delicate timing requirements of the connection between computer and USRP.

A.2.5 Benchmarks

The data set (concrete counter examples that represent undefined behaviour) used for the UE evaluation is derived via the TLA+ models. If you do not want to rerun the TLA+ model checking procedure, you can use the counter examples that we prepopulated our srsRAN forks with.

A.3 Set-up

A.3.1 Installation

TLA+ models. Assuming that `tlc` has bin installed via `tla-bin` as described in A.2.4, no more setup steps are required.

Replaying counter examples via `srsRAN`. Assuming that all dependencies of `srsRAN` have been installed as described in A.2.4, the next step is compiling one of our `srsRAN` forks. To do so, `cd` into the directory of the fork, and then run the following commands:

```
mkdir build
cd build
cmake ../
make
sudo make install
srsran_install_configs.sh service
```

The next step is to configure srsRAN correctly. To do so, please follow the instructions at https://docs.srsran.com/projects/4g/en/latest/app_notes/source/cots_ue/source/index.html. Since srsRAN has undergone multiple name changes, files might be named slightly differently in our forks (e. g., `srsran_install_configs.sh` instead of `srsran_4g_install_configs.sh`).

A.3.2 Basic Test

TLA+ models. Open a shell in the `models/rrc` directory of the artifact and run `tlc -deadlock rlc`. After 5-20 minutes (depending on your system), this results in a message saying `Model checking completed. No error has been found.`. The number of threads can be controlled using the `-workers n` parameter.

You can perform the same test for the SMS and PWS models. They require that you also supply `-maxSetSize 10000000` and take significantly longer (e. g., the SMS model will arrive at the same output after 75 days on 100 threads).

Replaying counter examples via `srsRAN`. Change into the directory of one of the three `srsRAN` clones. Then compile and install according to Section . You can now run the clone as follows:

1. (Only applies to SMS testing: `cd zmq_server && python3 zmq_server.py`).
2. Launch srsepc: `sudo srsepc \ --config /usr/local/share/srsran/epc.conf`
3. Launch srsenb: `sudo \ UHD_IMAGES_DIR=/usr/share/uhd/images/ \ srsenb /usr/local/share/srsran/enb.conf`
4. On the phone under test: Ensure that your APN is set to `srsapn`
5. Open an adb connection to the phone and run `ping 8.8.8.8`. You should see that 8.8.8.8 is reachable.

If you receive any error regarding missing UHD firmware, double check with the srsRAN and Ettus documentation that the supplied path matches the location of the firmware binaries. This is not specific to our modifications.

Similarly, if srsRAN does not work or the internet connection of you phone does not work, we recommend following the srsRAN documentation – unless you are running a test case (more on that later), our srsRAN forks operate exactly the same as the upstream versions, such that all their troubleshooting guides apply.

A.4 Evaluation workflow

A.4.1 Major Claims

For your convenience, we summarize major claims our paper makes:

(C1): Modelling via TLA$^+$ of LTE specification parts is computationally feasible. This is demonstrated by experiment E1, described in Sections 4.1 to 4.3 in the paper, with the results shown in Table 1 in the paper.

(C2): Using our approach, one can find more undefined behaviours than using the state-of-the-art approach "DoLTEst". This is discussed in Table 2 and Section 5.1 of our paper and demonstrated using Experiment E2.

(C3): Our approach can find undefined behaviours that lead to real world vulnerabilities. We list these in Table 2 of our paper and describe the vulnerabilities in sections 5.3.1-5.3.3. We reproduce these results in E3.

A.4.2 Experiments

(E1): [< 1 human-hour + up to 180,000 compute-hours]:
Execution: To measure the number of *States* and *CPU hrs.* according to Table 1 of our paper, run `tlc` with the same parameters described previously in A.3.2.
To measure the number of *Undefined Behaviours* and number of *PDUs* shown in Table 1, you have to read the `.cfg` file of the model (e. g., `models/rrc/rrc.cfg`) and modify the corresponding `.tla` file. The `.cfg` file contains a `CONSTANTS` section listing each test case. These test cases also correspond to the test cases listed in Tables 4-6 in the Appendix of our paper. To verify that each of these test cases is an undefined behaviour, you change the `ConstTestCase` variable in the `.tla` file to the test case you want to run. The line in the file is marked by a comment saying `Modify this value to choose the behaviour that you want to generate a counter example for`. An example of a correct assignment in the RRC model is `ConstTestCase == RRCConnectionReject_AFTER_SECURITY`.
Results: for the first part, the model checking will eventually terminate. Note that this takes 180,000 core hours for the SMS model, so we recommend parallelising using the `-workers` flag. If you do not supply a number of workers, TLC might choose a (suboptimal) number below the number of available CPU threads. The command line output should look like the following (for the RRC model).

```
3019102 states generated,
955 distinct states found,
0 states left on queue.
[...]
Finished in 07min 00s at ([...])
```

The number of distinct states (here, 955) and the "Finished in" time multiplied by the number of threads should match the corresponding columns in Table 1. Note that the timing will vary a bit for the shorter test cases (RRC and PWS), as it is dominated by startup time. The measurement for the SMS test case is much more stable as it is dominated by the actual model checking time.
When evaluating a test case (i. e., after modifying `ConstTestCase`), the model checking will terminate early. You should see the following command line output (this example corresponds to `ConstTestCase == RRCConnectionReject_AFTER_SECURITY`):

```
Error: Invariant Invariant is violated.
Error: The behavior up to this point is:
[...]
State 4: <Next line 603,
col 9 to line 713,
col 43 of module rrc>
[...]
/\ currentSequence =
  << << "RRCConnectionSetupMessage",
        [ rrcTransactionIdentifier |-> 0,
          [...]
        ] >>,
     << "SecurityModeCommandMessage",
        [ rrcTransactionIdentifier |-> 0,
          [...]
        ] >>,
     << "RRCConnectionReject",
        [ criticalExtensions |->
              [...]
     >> >>
[...]
```

This illustrates that to trigger this undefined behaviour, a counter example has been generated that contains 3 separate PDUs and their value assignments: RRCConnectionSetupMessage and SecurityModeCommandMessage to setup the state, followed by a RRCConnectionReject that ultimately triggers the transition into an undefined state. By counting the number of different undefined behaviours (assignments of `ConstTestCase`) and the number of PDUs (entries in `currentSequence` of the final state before model checking procedure terminates), you can reproduce the ***#UBs*** and calculate *Avg. PDUs* columns of Table 1 in our paper.

(E2): [10 human-hours + 5 compute-hours]:
Execution: Repeat the procedure of setting `ConstTestCase` for each undefined behaviour that we found, to generate a counter example for each of these. Then, compare these to the RRC section of Table 5 in the DoLTEst paper[1]. To match our and their test cases, you must compare their message setup[2] to the sequences generated by our approach.

[1]https://www.usenix.org/system/files/sec22-park-cheoljun.pdf

[2]https://github.com/SysSec-KAIST/DoLTEst/blob/e2251bfa8cd74f49b23369619722255ed895ef5e/srsepc/src/mme/fzmanager_epc.cc#L560

Results: For each test case, you should get a different `currentSequence`. In the cases where the *Guideline* column in Table 2 of our paper contains a number, you will find a test case in the DoLTEst code that corresponds to the final message of the generated sequence. For cases where our table does not show a number, the DoLTEst authors do not provide a corresponding test case.

(E3): [3 human-hours + 200 compute-hours]:

Execution: To reproduce our CVEs, you will need to use the PWS and SMS srsRAN forks. To verify that our test cases match what was generated by our TLA$^+$ models, open the following files: `srsran/pws/srsenb/src/stack/rrc/rrc.cc` (PWS) and `srsran/sms/srsepc/src/mme/sms.cc` (SMS) and compare the values assigned to the structs to the TLA$^+$-generated counter examples. As the counter example generation is not deterministic, you might end up with different assignments (counter examples), but the same undefined behaviours.

To run the tests, turn on the phone and the SDR, and start `srsenb` and `srsepc` as described previously in Section A.3.2. For the PWS samples, you have to schedule SIB12 in the `sib.conf`, by including `12` in `si_mapping_info` and adding a `sib12` entry to the same file. For the SMS test case, `zmq_server.py` must be started. To choose a test case, you enter the test cases index into the enb window (PWS) or enter it in the `zmq_server.py` shell (SMS) and press enter. For sequences longer than one PDU, you have to repeat this as many times as there are PDUs in the sequence. The indexes can be found in the code, the ZMQ command line output, or by using the row indices of our result tables 4 and 5. Remember to restart enb, epc and the phone between each test case. Also remember to increase the system time of the phone by a week for each PWS test, as described in the paper (Section 3.5).

Results: You should see the behaviours described in Sections 5.3.1, 5.3.2 (PWS), and 5.3.3 (SMS) of our paper. To trigger a modem crash you might have to make the adjustments described in the paper. To determine the modem indeed crashed we recommend setting the phone into debug mode by dialling `*#9900#` and setting "Debug Level" to "HIGH" in the resulting hidden menu. Note that this setting only exists on Samsung phones (A41 and S20 in our case).

For the OOB read on Samsung S20 phones, please note that you might have to perform multiple attempts, as it depends on the current heap memory contents. We recommend indicating 5-7 pages (first two digits of the first `warning_msg_segment_r9`), as this demonstrates the attack while working relatively reliably.

We have reported the issues to Mediatek and Samsung, and the vulnerabilities might have been patched in firmware versions released after April 2022.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: MOBILEATLAS: Geographically Decoupled Measurements in Cellular Networks for Security and Privacy Research

Gabriel K. Gegenhuber
University of Vienna*

Wilfried Mayer
SBA Research

Edgar Weippl
University of Vienna

Adrian Dabrowski
CISPA Helmholtz Center for Information Security†

A Artifact Appendix

MOBILEATLAS is a scalable, cost-efficient test framework for cellular networks that takes international roaming measurements to the next level. It implements the promising approach to geographically decouple SIM card and modem, which boosts the scalability and flexibility of the measurement platform. It offers versatile capabilities and a controlled environment that makes a good foundation for qualitative and fine-grained cellular measurements.

A.1 Abstract

Physically moving devices and SIM cards between countries to enable measurements in a roaming environment is costly and does not scale well. Therefore, we introduce an approach to geographically detach the SIM card from the modem by tunneling the SIM card's protocol over the Internet and emulating its signal on the cellular modem. This allows us to test roaming effects on a large number of operators without physically moving any hardware between different countries.

To make it accessible to other researchers, we fully release the hard- and software documentation of our measurement framework.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

MOBILEATLAS is meant to be used in live mobile networks. Our measurement experiments usually mimic normal user behavior and just transmit minimal data traffic (i.e., several MB of data) to get deeper insights into the operator's core network mechanisms and interplay during roaming scenarios. However, due to SIM tunneling, SIM cards could change the "country" in an irregular and fast fashion, which might spark confusion among the operator's systems or trigger fraud control alerts. In order to exercise caution, we manually imposed a waiting time of 2 hours between country switches in our experiments. When testing for potential free-riding possibilities, we always made sure not to enrich ourselves by not oversizing the generated test traffic and by letting an equal or greater amount of our monthly traffic allowance expire at the end of the month (as if we were billed for the traffic).

*Supported by the UniVie Doctoral School Computer Science DoCS.
†Partly as postdoc at University of California, Irvine.

A.2.2 How to access

The hard- and software documentation of the MOBILEATLAS measurement platform that is presented in our paper is hosted on GitHub.

A.2.3 Hardware dependencies

SIM Providers require a host system (e.g., a linux laptop), a SIM reader device (e.g., PC/SC reader) and a SIM card that will be made accessible (i.e., to an external *Measurement Probe* via a SIM tunnel).

Measurement Probes require a dedicated hardware setup. To support future upgrades, most of the used hardware components are easily interchangeable. We based our current probe version on a Raspberry Pi 4 and a Quectel EG25G modem. Furthermore, we use a HAT adapter to connect the modem to the Pi. To tunnel and emulate the SIM card protocol we leverage the Pi's GPIO ports and connect them to the modems SIM socket via a self-made SIM PCB.

More details and some pictures of the used hardware can be found on GitHub in a dedicated README file.

A.2.4 Software dependencies

Our python-based source code relies on external dependencies that need to be installed via the package manager or via pip (cf. Section A.3). Besides common and officially available python

packages, we also require a customized version of pySIM.

Furthermore, we use ModemManager [1] to interact with the modem during measurement experiments. Therefore, good ModemManager support is essential when using different hardware (i.e., a different modem) for the *Measurement Probe*.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

We use Ansible to patch and setup the system of our *Measurement Probes*. The Ansible playbooks are meant to be executed on a fresh installation of the Raspberry Pi OS, as described in a dedicated README file.

The software dependencies that are needed for the *SIM Provider* are referenced in the next section.

A.3.2 Basic Test

To run the *SIM Provider* and *Measurement Probe* software components it is required to setup a Python virtual-environment as described in the responsible README file.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

[1] https://modemmanager.org/

USENIX'23 Artifact Appendix: BASECOMP: A Comparative Analysis for Integrity Protection in Cellular Baseband Software

Eunsoo Kim*[1], Min Woo Baek*[1], CheolJun Park[1], Dongkwan Kim[2], Yongdae Kim[1], Insu Yun[1]

[1] *KAIST*,
[2] *Samsung SDS*

A Artifact Appendix

A.1 Abstract

This artifact implements comparative analysis in a static approach for detecting discrepancies with the specification in the integrity protection of cellular baseband software. The system comprises mainly two components; probabilistic inference and symbolic execution. Probabilistic inference locates the integrity protection function and is implemented with Python APIs provided by IDA Pro. Symbolic execution reports the mismatches and is implemented above *angr*. We evaluate the system's probabilistic inference by the effectiveness of finding the genuine integrity protection function within baseband firmware. We then evaluate the system by the number of bugs found. Further, we evaluate the capability of finding different types of bugs compared to dynamic testing methods. All of the artifact evaluation results refer to Section 7 and the Appendix of the paper. The artifact evaluation aims for the three badges: available, functional, and reproducible.

A.2 Description & Requirements

Here we describe the hardware and software requirements to run the artifact, as well as the tested targets of our evaluation.

A.2.1 Security, privacy, and ethical concerns

There are no risks for the evaluators while executing the artifact to their machine's security, data privacy, or other ethical concerns. This artifact has been used to detect 29 bugs in 16 images of baseband software and all have been responsibly disclosed to the vendors.

A.2.2 How to access

The artifact is available on GitHub at the address `https://github.com/kaist-hacking/BaseComp`.

*These two authors equally contributed.

A.2.3 Hardware dependencies

We perform the experiments on AMD Ryzen 9 5900X 12-Core Processor CPU, 3.70GHz, 64GB DDR4 RAM. No specific hardware feature is required for the artifact evaluation.

A.2.4 Software dependencies

We perform the experiments on Windows 11 Pro. For analyzing baseband firmware, we require IDA Pro v7.6. Codes are written for Python 3.10.5 and require the packages pgmpy, NetworKit, and angr. To build libraries for supporting MIPS16e2, Visual Studio Build Tools are also required.

A.2.5 Benchmarks

We provide the 16 images from Table A1 of the paper. The root directory of the artifact repository contains a folder named `artifact`. The corresponding folder contains folders for each image which is named after the "Nick" column of Table A1. In each folder, the image is provided.

A.3 Set-up

To prepare the environment to be used for the evaluation of our artifact, clone the BaseComp repository `https://github.com/kaist-hacking/BaseComp` and checkout commit `cd6d118`.

To load the provided images to IDA Pro, follow the steps below. We provided the .idb files for images from MediaTek separately through a link to external storage.

(S1): Run `python parse_modem.py` in the `idb-creation` folder and provide the target image's path.

(S2): Load the binary created with `MAIN` in its name to IDA Pro. Set ARM Little Endian as the architecture and select `Manual Load`.

(S3): Set the `ROM start address` and `Loading address` as the starting address written in the binary's name. This should look something like 0x40010000.

(S4): Load the script file `analyze.py` in the `idb-creation` folder to IDA Pro.

(S5): Repeat the steps above for images to be newly loaded.

To support the MIPS16e2 architecture later on in the symbolic execution phase, follow the steps below after installing all the software dependencies described in Section A.3.1.

(S1): Run `python build_pyvexlib.py` with `mips16e2` as the working directory in x64 Native Tools Command Prompt for VS.

(S2): Copy the `pyvex.dll` and `pyvex.lib` file created under the `pyvex_c` folder to `your_path_to_python/Lib/site-packages/pyvex/lib`. This should replace the library files originally located there. Back up the original files if needed.

All the instructions are also described in the `README` files of each directory of the artifact.

A.3.1 Installation

The experimental evaluation requires the following software.

(I1): pgmpy: https://pgmpy.org

(I2): NetworKit: https://networkit.github.io

(I3): angr: https://docs.angr.io

(I4): Visual Studio Build Tools: https://visualstudio.microsoft.com/downloads/?q=build+tools

A.3.2 Basic Test

We prepared a simple functionality test inside the `function-identification` folder of the artifact. The execution of command `python -m run_tests` from the directory `function-identification/tests` performs construction of the call graph on a test code and checks its values. The test should end within seconds and no assertions should be raised.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The utilization of probabilistic inference in our system significantly reduces the number of functions to be analyzed manually. The average rank of the genuine integrity protection function we seek is 1.56 as illustrated in Table 4 of the paper. This is evaluated in **(E1)**.

(C2): Our system detects a total of 34 mismatches from the 3 vendors we test. 29 of them are actual bugs including those that can lead to NAS AKA bypass. This is evaluated in **(E2)** and the results are summarized in Table 5 of the paper.

(C3): Our system complements dynamic testing in terms of completeness for analyzing integrity protection. Compared to DoLTEst and DIKEUE, which are previous works that use dynamic testing for analyzing integrity protection, our system covers more types of integrity protection bugs. This is evaluated with the same results of **(C2)** and the comparison results are illustrated in Table 6 of the paper.

A.4.2 Experiments

(E1): Identifying Integrity Protection [5 human-minute + 30 compute-minute] for each image: Ranks possible integrity protection functions in the target image.

Preparation: Follow the steps in Section A.3 to prepare the .idb files for analysis. To test with a different probability parameter value, change the `PROBABILITY_PARAMETER` value in the `utils.py` file under `function-identification/scripts/analyses`.

Execution: Load the `identify_integrity_function.py` file in `function-identification/scripts` to the .idb file of the target image.

Results: The results of the analysis should be written in the `results.txt` file of the target image's folder. A list of functions should be under the line written `Integrity Function Probability`. Starting with the function with the highest rank, the address of the function and probability is listed. The address of the genuine integrity protection function (reference result) is in the `symbolic-execution/config_firmware.yaml` file written as `integrity_func`. The rank should be the same as in Table 4 in the paper. Probability values may slightly differ by the number of functions identified. The time consumed for each step is also at the end of the file. We repeated the experiment several times while removing the cache every iteration and recorded the average in Table 7 of the paper.

(E2): Symbolic Execution [5 human-minute + 5 compute-minute] for each image: Finds mismatches with the specification in the integrity protection functions.

Preparation: Additional information about the image earned by manual analysis is required to be written in `symbolic-execution/config_firmware.py`. However, those for the provided images are already written down. Therefore, there is no preparation required for the evaluators to process.

Execution: Run the command `python analyze_base.py -fn {name_of_target}` in the `symbolic-execution` directory for each image.

Results: The results will be created under the `symbolic-execution/results/{name_of_target}` folder with the current time as the file name. The `Errored Results` and `Errored States` indicate the mismatches found by the system. The list under `Errored Results` consists of `[security state, security header, protocol discriminator, message type, reason_why_it_is_errored]` and the list under `Errored States` just indicates the reason

it is errored. We gathered the mismatches by the vendor and Table 5 in the paper indicates the results.

A.5 Notes on Reusability

To use our system on baseband firmware other than those provided in the artifact, mainly 2 steps would be required.

(S1): Based on whether our current implementation supports the vendor, a vendor-specific module might be needed to be written. The instructions for writing the module are well specified in the `README` files in the artifact.

(S2): Collect firmware-specific information such as the addresses of the security state, functions to skip, and so on.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs

Jianhao Xu[1] Kangjie Lu[2] Zhengjie Du[1] Zhu Ding[1] Linke Li[1] Qiushi Wu[2] Mathias Payer[3] Bing Mao[1]

[1]State Key Laboratory for Novel Software Technology, Nanjing University
[2]University of Minnesota [3]EPFL

A Artifact Appendix

A.1 Abstract

This artifact provides a dataset of Compiler-Introduced Security Bugs (CISBs). Our dataset comprises various types of CISBs, which we manually identified from GCC and Clang Bugzilla reports and Linux git history. We organize these CISBs in a taxonomy based on their root causes, formation, and security impacts. Additionally, we include test cases and their triggering oracles for all the reproduced CISBs. Please note that the user study data cannot be shared due to ethical considerations. We have promised our participants that we will only share statistics of their data.

To validate the results of our paper, we also prepare scripts to obtain statistics on the bugs in our dataset, reproduce the evaluation of compiler mitigations based on our dataset, and obtain statistics on the targeted bugs in our dataset for automatic prevention works in a console.

The minimum required disk space for the program is approximately 20 GB. We have tested it on Ubuntu 20.04. The software prerequisite for the program is an operating system that can run Docker and supports Ubuntu 20.04 as a container image. The whole experiment takes about 3 human-hours and 60-70 compute-hours.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The included scripts should not pose a greater risk to evaluators than regular benign Python scripts when running them.

A.2.2 How to access

The aritifact and dataset can be got from Github `https://github.com/H0w1/CISB-dataset/tree/aac22565c96744a13f0786854b3257d64421acef`.

A.2.3 Hardware dependencies

To run our evaluation, a x64 machine with a network connection is required.

A.2.4 Software dependencies

To evaluate the artifact, you need an operating system that can run Docker and supports Ubuntu 20.04 as a container image.

A.2.5 Benchmarks

Please note that running one experiment requires SPEC CPU 2006, which is not provided as it is not a free software. Reviewers will need to obtain their own copy of SPEC CPU 2006 to run this experiment.

A.3 Set-up

We provide a Dockerfile that automatically downloads the dataset and evaluation materials, as well as installs all the necessary software requirements.

Instructions for downloading and using the Dockerfile can be found on the `README` page `https://github.com/H0w1/CISB-dataset#aritifact-setup`.

A.3.1 Installation

After installing the Docker container from the Dockerfile, the dependencies and main artifact will be automatically prepared.

A.3.2 Basic Test

A basic functionality test can be easily executed by running the Python script 'check-compiler.py' using the command '`python3 check-compiler.py`'. Upon successful execution, the script will output a list of notes indicating that each compiler used has been installed correctly. You can find this script in the main directory of the Git repository `https://raw.githubusercontent.com/H0w1/CISB-dataset/`

b122ac42ff52ecb59b94a319c4558b1275cc9166/check-compiler.py.

A.4 Evaluation workflow

The overall workflow comprises the following steps:

1. Obtain the Dockerfile;
2. Obtain the dataset and test scripts and install the necessary dependencies automatically using the Dockerfile;
3. Execute a Python script to check the statistics of CISBs in the dataset;
4. Execute scripts and review some CISB details in the dataset to evaluate the effectiveness of current mitigations and the performance overhead of current compiler mitigations;
5. Execute a Python script to obtain statistics on the percentage of CISBs in our dataset that can be targeted by automatic prevention works. Check the statistics by reviewing related CISBs.

A.4.1 Major Claims

(C1): *We identify a large set of different kinds of CISB in the real world.* This is proven by the CISBs in the dataset. The statistic of these bugs can be viewed by running the script 'statistic.py'.

(C2): *We investigate and show the risks of existing mitigations.* Specifically, (i) we show CISB prevention performed by programmers is risky, derived from real cases; (ii) we perform a comprehensive evaluation of existing mitigations provided by compilers, with our dataset. We can prove (i) by pointing to the existence of a few bugs in our dataset. As for (ii), we can provide the script (('statistic.py')) we used to re-run the analysis and generate Table 6 (an evaluation of the mitigations provided by the compiler.)

(C3): *The CISBs we studied have not been extensively studied before.* This is proven by the script ('statistic.py') to get the results shown in Table 7, which shows the statistics of CISBs that can theoretically be prevented by automatic prevention works.

A.4.2 Experiments

We provide a guide of the experiments in the Github repository. https://github.com/H0w1/CISB-dataset#aritifact-experiments

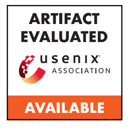
(E1): *[CISB statistics] [30 human-minutes + 1 compute-second]:* Execute the Python script to obtain the statistics of CISBs in our dataset. Check the dataset and script for mistakes. The result should be in line with the data in Figure 2 and Figure 3 of the paper.

(E2): *[Evaulation of mitigations] [30 human-minutes + 60-70 compute-hours]:* (i) Review a list of bugs where the prevention performed by programmers failed. This list can be obtained by executing a script. It takes one compute-second. (ii) Run a script to obtain statistics on the effectiveness of compiler mitigations. It takes about two compute-minutes. (iii) Run scripts to measure the overhead of different compiler prevention strategies using the SPEC CPU 2006 benchmark. It takes about 60 compute-hours. For (i), the expected result is those CISBs exist. For (ii) and (iii), the output results should be in line with the data shown in Table 6 of the paper.

(E3): *[Target bugs of automatic prevention works] [2 human-hours + 2 compute-minutes]:* (i) Execute the script to obtain the statistics of CISBs that can theoretically be prevented by automatic prevention works. (ii) Check the lists of CISBs we summarized and shown in the script. For (i), the result should be in line with the data in Figure 7 of the paper. For (ii), these bugs should be within the scope of the corresponding prevention work.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs

Gertjan Franken
imec-DistriNet, KU Leuven

Tom Van Goethem
imec-DistriNet, KU Leuven

Lieven Desmet
imec-DistriNet, KU Leuven

Wouter Joosen
imec-DistriNet, KU Leuven

A Artifact Appendix

A.1 Abstract

`BugHog` is a comprehensive framework designed to identify the complete lifecycles of bugs, from their introduction to mitigation, and potential regression. For each bug's proof of concept (PoC) integrated in the `BugHog` experiment web server, the framework can perform automated and dynamic experiments using Chromium and Firefox revision binaries.

Each experiment is performed within a dedicated Docker container, ensuring the installation of all necessary dependencies, in which `BugHog` downloads the appropriate browser revision binary, and instructs the browser binary to navigate to the locally hosted PoC web page. Through observation of HTTP traffic, the framework determines whether the bug is successfully reproduced. Based on experiment results, `BugHog` can automatically bisect the browser's revision history to identify the exact revision or narrowed revision range in which the bug was introduced or fixed.

The framework offers a graphical user interface, accessible through a locally hosted web page. The experiment results are visualized using a Gantt chart, facilitating easy interpretation and analysis.

In our study, BUGHOG was employed to identify the lifecycles of 75 bugs related to the Content Security Policy, across its complete development history in Chromium and Firefox.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Executing our artifact on evaluators' machines poses no risk.

A.2.2 How to access

The stable version of `BugHog` as part of the USENIX artifact evaluation process is available at https://github.com/DistriNet/BugHog/tree/usenix23-artifact-stable. The most recent version, which will also be maintained and updated in the future, can be found at https://github.com/DistriNet/BugHog. To clone the repository, use the following command:

```
git clone https://github.com/DistriNet/BugHog.git
```

Or use the GitHub web interface.

A.2.3 Hardware dependencies

To run the framework, the following minimum hardware specifications are required:

- 2 CPU cores (more cores may be necessary for concurrent experiments).
- 8 GB of RAM.
- 5 GB of disk space.
- Internet connection.

Ideally, the number of CPU cores should match or exceed the number of concurrent experiments the user allows `BugHog` to perform. Insufficient disk space may cause the framework to crash since this could prevent it from temporarily storing downloaded binaries.

A.2.4 Software dependencies

The BUGHOG framework only relies on Docker, making it compatible with any operating system that supports Docker (e.g. Windows, macOS, Linux). All necessary dependencies are included in the Docker images, eliminating the need for additional software installations.

A.2.5 Benchmarks

No specific benchmarks are associated with this artifact.

A.3 Set-up

The installation instructions that follow can be found in our GitHub repository's `README.md` as well.

A.3.1 Installation

Follow these steps to install BugHog:

1. Clone the repository, and navigate to the root directory:

   ```
   git clone https://github.com/DistriNet/BugHog.git
   cd BugHog
   ```

   The project can also be downloaded through GitHub's web interface at https://github.com/DistriNet/BugHog instead.

2. Obtain the required BugHog Docker images:

   - Option A: Pulling (fastest)

     Use the following command to pull the necessary pre-built Docker images:

     ```
     docker compose pull core worker web
     ```

   - Option B: Building

     If you intend to modify the source code, use the following commands to build the required Docker images. Rerun this script if you make any changes to the source code:

     ```
     docker compose up node_install_deps
     docker compose up node_build
     docker compose build core worker web
     ```

     For reference, building the images takes approximately 4 minutes on a machine with 8 CPU cores and 8 GB of RAM.

3. (**Optional**) Use your own MongoDB instance.

   If you prefer using your own MongoDB instance, provide the connection parameters in a `.env` file at the project's root:

   ```
   bci_mongo_host=[ip_address_of_host]
   bci_mongo_database=[database_name]
   bci_mongo_username=[database_user]
   bci_mongo_password=[database_password]
   ```

   If not provided, BugHog will spin up a MongoDB instance in a Docker container. The data is persisted between runs within the `database` folder, allowing you to safely stop and start BugHog without losing any data.

A.3.2 Basic Test

To start the framework, execute the following command:

```
docker compose up core web
```

Depending on the installation option chosen earlier, a pulled or locally built image will be used. You can switch between the two options by executing the appropriate installation step before starting the framework.

To access the web interface, open your web browser and navigate to http://localhost:5000. BugHog is ready for use when the following message is logged in either the terminal window or the web interface:

```
[INFO] bci.master: BugHog is ready!
```

Perform a simple test by following these steps:

1. Select the `CSP` project and `Chromium` browser from the dropdown menus in the upper left corner of the web interface.

2. Choose the `c1064676` experiment from the `Experiments` pane. [1]

3. Set the `Evaluation range` with a lower version of 20 and an upper version of 110.

4. Input 1 for the `Number of parallel containers`. [2]

5. Click the green `Start evaluation` button.

As the evaluation progresses, the framework will provide updates in the terminal window or `Log` pane at the bottom of the web interface. The information and Gantt chart in the `Results` pane will be updated automatically as well, if `c1064676` is selected in the `Select an experiment` dropdown menu. Please note that the Gantt chart requires a minimum of two completed experiments before it can be generated. Each dot in the Gantt chart represents whether the bug can be reproduced in the corresponding revision binary. By hovering over a dot, the associated revision number and browser version can be observed. To prevent the Gantt chart from refreshing automatically, the `Auto-refresh Gantt chart` checkbox can be unchecked. This might be necessary when zooming in on a specific part of the chart, since the zoom level will be reset when the chart is refreshed. A refresh can be triggered manually by clicking the `Refresh` button.

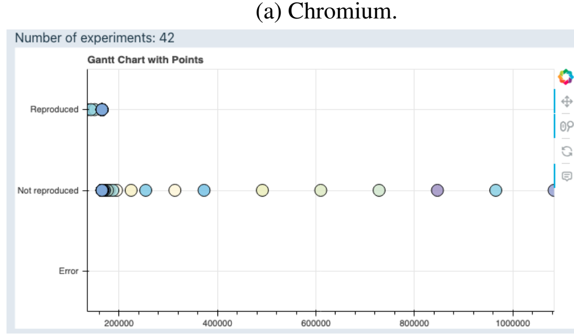
You have the option to stop the evaluation at any time by clicking either the yellow `Stop gracefully` button, which allows the ongoing experiments to finish before stopping, or the red `Stop forcefully` button, which immediately attempts to halt all experiments. When all experiments have ended, either by user intervention or because the last available revision has been evaluated, the following line will be logged:

```
[INFO] bci.master: BugHog has finished the evaluation!
```

[1] Experiments are named after the bug report ID. Experiments with a `c` as prefix are Chromium reported bugs, while experiments with an `f` as prefix are Firefox reported bugs.

[2] Feel free to increase this number if you have more available CPU cores.

(a) Chromium.



(b) Firefox.

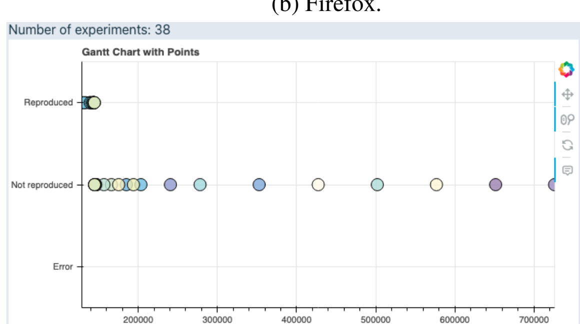


Figure 1: Resulting Gantt chart of experiment (E1).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BugHog is capable of running all browser revision binaries that support CSP enforcement[3] required for our study, without the need for manual dependency management. This claim is supported by experiment (E1) conducted on both browsers.

(C2): We successfully identified the complete lifecycles of 75 bugs in Chromium and Firefox using BugHog. Considering the impracticality of evaluators individually pinpointing the lifecycles of all bugs, we will provide our comprehensive MongoDB lifecycle dataset.[4] Evaluators can utilize BugHog to pinpoint the lifecycle of any bug in our dataset and compare them with our findings. Furthermore, as an example, we describe experiment (E2) where we trace the lifecycle of one of the CSP bugs.

A.4.2 Experiments

The experiments were conducted on a machine equipped with 8 CPU cores and 8 GB of RAM, where BugHog was configured to use 8 parallel containers.

(E1): *[Support for the complete CSP browser development history] [20 human-minutes + 15 compute-minutes]*:

Description: BugHog is capable of running all necessary browser revision binaries that support CSP without the need for manual dependency management. To demonstrate this capability, we conduct an experiment to identify the introduction of CSP in Chromium and Firefox. For this experiment, we utilized a PoC that employs a CSP policy blocking all resource loading through the `default-src` directive, which is supported since CSP's introduction. In revisions without CSP support, requests are allowed to reach our server, indicating successful reproduction. The PoC can be found in the `experiments/pages/Support/CSP` directory.

Execution: Select the following evaluation parameters:

- `Project`: Support
- `Browser`: Chromium for one evaluation, Firefox for the other
- `Experiment`: CSP
- `Evaluation range`: 20 to 110
- `Search strategy`: Binary search
- `Reproduction id`: csp
- `Number of parallel containers`: any number from 2 to 8 (higher numbers will result in faster evaluation)

Click the green `Start evaluation` button to begin the evaluation.

Results: By refreshing the `Results` pane in the web interface, you will eventually observe that `BugHog` successfully identifies a specific revision range in which the introduction of CSP occurred. The Gantt chart for both browsers will also exhibit a distinct pattern indicating the utilization of binary search. Figures 1a and 1b show the resulting Gantt charts for Chromium and Firefox, respectively.

By solely using downloaded publicly available revision binaries, we can infer that the introduction of CSP in Chromium took place at revision 165317.[5] This revision corresponds to a WebKit roll, where WebKit's revision range [133029 - 133116] was integrated into Chromium. Through manual analysis of revision metadata, we determined that revision 133131[6] is the exact revision responsible for introducing CSP.

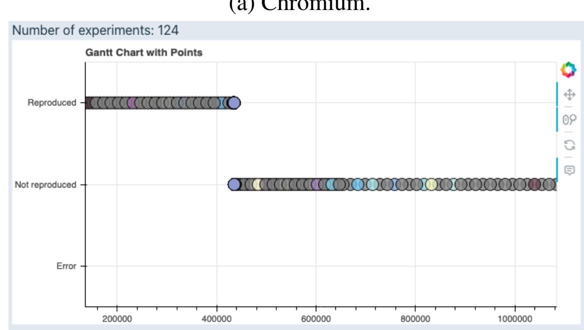
[3] CSP support starts from revision 165317 and 144546 for Chromium and Firefox, respectively.

[4] A dump of this dataset is available at: https://github.com/DistriNet/lifecycle-data

[5] https://crrev.com/165317

[6] https://trac.webkit.org/changeset/133131/webkit

(a) Chromium.



(b) Firefox.

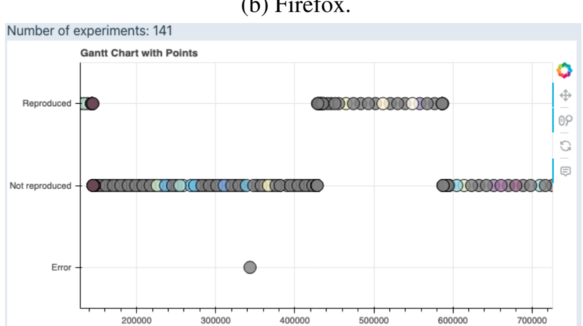


Figure 2: Resulting Gantt chart of experiment (E2).

For Firefox, the framework narrows down the revision range to [144529 - 144643]. Here, through a combination of self-built binaries and manual analysis, we inferred that revision 144546[7] indeed introduced CSP.

(E2): *[Full lifecycle analysis] [20 human-minutes + 40 compute-minutes]*:

BUGHOG provides a comprehensive lifecycle analysis of bugs through the `Composite search` method. This approach involves two stages:

- In the first stage, N evenly spread out revisions are evaluated over the whole indicated range, with N determined by the value of `Sequence limit`.
- In the second stage, the analysis focuses on identifying smaller revision ranges where reproducibility shifts are observed.

In this experiment, we will conduct a lifecycle analysis of bug `f1441468`, which was reported for Firefox.[8] Although we did not find this bug reported for Chromium, `BugHog` has revealed that the bug is reproducible in Chromium as well in our cross-browser analysis. Therefore, we will perform this evaluation on both browsers again in this experiment.
Unlike the previous experiment, our goal here is not just to determine when the bug was introduced. We aim to obtain a complete view of the bug's lifecycle. To achieve this, we will employ the `Composite search` strategy. To ensure that we evaluate approximately one binary per release version, we will set the `Sequence limit` to 100.

Execution: Select the following evaluation parameters:

- `Project`: CSP
- `Browser`: Chromium for one evaluation, Firefox for the other
- `Experiment`: f1441468
- `Evaluation range`: 20 to 110
- `Search strategy`: Composite search
- `Sequence limit`: 100
- `Reproduction id`: f1441468
- `Number of parallel containers`: any number from 2 to 8 (higher numbers will result in faster evaluation)

Click the green `Start evaluation` button to begin the evaluation.

Results: Figures 2a and 2b show the resulting Gantt charts for Chromium and Firefox, respectively.
For Chromium, BUGHOG shows that this bug is foundational, as the bug is reproducible since the introduction of CSP. BUGHOG also identifies a narrow revision range, specifically [435165 - 435177], where an effective fix was applied. By manually analyzing revision metadata, we can determine that the bug was fixed in revision 435165.[9]
In the case of Firefox, the results indicate that the bug is non-foundational since it could not be reproduced at the time of CSP introduction. Instead, the bug was introduced within revision range [428395 - 428677], and subsequently fixed within revision range [587121 - 587215]. Through a combination of self-built binaries and manual analysis of revision metadata, we can identify the introducing revision as 428568[10] and the fixing revision as 587202.[11]
Both results can be cross-referenced against the lifecycle data dump by opening the `json` file in any text editor with string search functionality, and searching for the bug ID without the single-letter prefix (i.e. 1441468). The object associated with this ID contains the bug's life-

[7] https://hg.mozilla.org/releases/mozilla-release/rev/6b181afc9fadbd4bb9d04648aa24a34bd9731e82
[8] https://bugzilla.mozilla.org/show_bug.cgi?id=1441468
[9] https://crrev.com/435165
[10] https://hg.mozilla.org/releases/mozilla-release/rev/428568
[11] https://hg.mozilla.org/releases/mozilla-release/rev/587202

cycles for both browsers, in which the aforementioned revisions are listed.

A.5 Notes on Reusability

As mentioned in our paper, BugHog is not limited to evaluating CSP bugs but can also be used to analyze other types of bugs, including those impacting other (security) policies and functionalities. By integrating the bug's PoC into the framework, BugHog can uncover its complete lifecycle. This integration is performed by adding the necessary web page files to the `experiments/pages` folder and by including the URL queue of pages to be visited during the experiment in the `experiments/url_queues` folder. Since the integration format may evolve in the future, we provide more detailed instructions on how to integrate new bug PoCs in the `README.md` file of our GitHub repository.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: <Remote Code Execution from SSTI in the Sandbox: Automatically Detecting and Exploiting Template Escape Bugs>

Yudi Zhao[1, ¶], Yuan Zhang[1, ¶], and Min Yang[1]

[1] *School of Computer Science, Fudan University, China*
[¶] *co-first authors*

A Artifact Appendix

A.1 Abstract

This artifact is a tool named TEFuzz for paper <Remote Code Execution from SSTI in the Sandbox: Automatically Detecting and Exploiting Template Escape Bugs>. TEfuzz can detect the template escape bugs in the template engine and generate exploit synthesis.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact don't have any risk for evaluators while executing your artifact to their machines security, data privacy or others ethical concerns. Evaluators can run this artifact with confidence.

A.2.2 How to access

Evaluators can access the artifact and source code by github repository (https://github.com/seclab-fudan/TEFuzz).

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

The TEFuzz part of the artifact run under ubuntu 18.04 and require support for Python3.8 and related libraries, a list of which is included in the *requirements.txt* file in the github repository. The TE driver part of the artifact needs to run in Apache2 and PHP7.2.34 environment. We provide a complete docker image on github repository, and evaluators can directly pull the docker image without manual configuration.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

First clone the source code from the Github repository, set up Python3.8 environment.

```
1 git clone https://github.com/seclab-fudan/TEFuzz
2 sudo apt-get update
3 sudo apt-get install python3.8 python3-pip
4 python3.8 -m pip install -r requirements.txt
5 cd $YOUR_TEFUZZ_PATH/CodeWrapper
6 composer install
7 sed -i 's/protected $attributes;/public
     $attributes;/g' vendor/nikic/php-parser/lib/
     PhpParser/NodeAbstract.php
```

Listing 1: Bash command

Then pull the docker image , evaluators need to mount docker's '/var/www/html/tefuzz' directory to the host so that tools can read the information.

```
1 docker pull altm4nz/tefuzz:1.0
2 docker run -itd -p 80:80 -v /var/www/html/tefuzz:/
     var/www/html/tefuzz --name tefuzz altm4nz/
     tefuzz:1.0
3 docker cp tefuzz:/tmp/tefuzz/ /var/www/html/
4 docker cp tefuzz:/tmp/seed/ $YOUR_TEFUZZ_PATH/
     result/
5 docker exec -it tefuzz /bin/bash -c 'service
     apache2 start'
```

Listing 2: Docker command

A.3.2 Basic Test

After installing the artifact, evaluators can perform a basic test using the *'python3 check.py '* command to determine whether the artifact was successfully installed. If *success* is displayed, the evaluators can continue the follow test evaluation.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): TEFuzz can detect template escape bugs in the template engine of the data set and generate exploit synthesis.

A.4.2 Experiments

(E1): [Vulnerability detection] [30 human-minutes + 50 compute-hour + 5GB disk]
Preparation: After the artifact have been successfully installed, set *TARGET_IP* as your docker ip, set *TE_NAME* in *'config.py'* to the target template engine name, such as 'smarty'.
Execution: Run the artifact using *'python3 main.py'*. Detecting a different template engine requires re-running the artifact after replacing the *TE_NAME* parameter.
Results: After completion of the run, the TEFuzz will output the number of generated exploit synthesises, this result corresponding to the third column of Table 2 in the paper. For example, "generate 3 EXP" means 3 exploit synthesises were generated. Meanwhile, TEFuzz will output intermediate data in the process of vulnerability detection, including the number of seeds, the number of Testcases generated, the number of interesting Testcases , etc., which correspond to Table 6 in the paper.

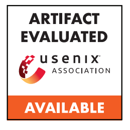
A.5 Notes on Reusability

If you want to evaluate more template engines by using my artifact, first you need to collect the seed data for that template engine, store it in the *result/seed* directory, and configure the appropriate adaption rules (optional). The most important thing is that you need to set up a TE driver environment for the new template engine, The TE driver environment can refer to the environment in the docker image.

If you want to test new seeds in an existing template engine, it's easier to just update your seeds into the *result/seed* directory.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: SMACK: Semantically Meaningful Adversarial Audio Attack

Zhiyuan Yu, Yuanhaur Chang, Ning Zhang
Washington University in St. Louis

Chaowei Xiao
Arizona State University

A Artifact Appendix

A.1 Abstract

SMACK is an adversarial audio attack that leverages manipulation of the prosody attributes to craft adversarial speech examples. Our artifact comprises the source code, the generative model for controlling speech prosody, along with the automatic speech recognition (ASR) and speaker recognition (SR) models for attack testing. To operate the attack framework, the user needs to run the program in the command line, providing attack types (i.e., against ASR or SR system) and specifying attack targets (i.e, targeted transcription or speaker label). The expected results are the adversarial audio samples.

Given the complexity of the speech generative model involved in SMACK, a machine with a moderate CPU and a GPU of at least 8GB VRAM is recommended. Please note that run-time may vary depending on the user's hardware. We have compiled a list of required dependencies into a YML configuration file.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact provided does not contain any harmful materials that may compromise machine security or pose threats to human health. The generative model was trained on public speech corpus, and there is no privacy concern associated with the use of artifacts. We have taken the utmost care to ensure that the artifact meets all necessary safety standards and guidelines.

A.2.2 How to access

We have made the code and models available on Zenodo. The stable URL link pointing to the repository is `https://github.com/WUSTL-CSPL/SMACK/commits/895f19b35350c5aded3362508c4a770f5e36342f`.

A.2.3 Hardware dependencies

The attack framework can run on a machine with a moderate CPU, at least 16GB of available RAM, and a GPU with at least 8GB VRAM. The system was tested stable with AMD Ryzen 9 3900X 12-Core Processor accompanying RTX 3070Ti and 32GB memory. No other specific hardware is required, but the variance in hardware can lead to differences in run-time.

A.2.4 Software dependencies

SMACK was implemented in Python, and the environment was set up using Miniconda 4.12.0 on Ubuntu 22.0.4. The used machine learning framework is Pytorch, and other associated dependencies are encapsulated in a YAML file. For the installation process please see Section A.3. Due to the file size limit of GitHub, some of the files used for testing need to be downloaded from the Google Drive link, `https://drive.google.com/file/d/12vUxRaIRDaD_prg8F-vpb5oUvWMOPqsl/view?usp=sharing`, containing custom scripts and pre-trained models. Please refer to README.md for detailed guidance.

A.2.5 Benchmarks

The data required by the experiments are the generative model and speech audio used in adversarial optimization. For speech recognition, we provide two ASR cloud services, iFlytek and Google Speech-to-Text. For speaker recognition, we provide two state-of-the-art models (GMM-UBM and ivector-PLDA) as targets. The SR models are provided in the artifact, which can be found in the *./FAKEBOB* directory.

A.3 Set-up

A.3.1 Installation

Conda or Miniconda is recommended for setting up the environment. It can be installed via the official link `https://docs.conda.io/en/latest/miniconda.html` and the process can differ based on the user's OS. The commands for setting up the environment with the *smack.yml* file are:

```
$ cd <the_path_to_the_folder>
$ conda env create -f smack.yml
$ conda activate smack
```

For the setup of speaker recognition systems, we follow the existing work *FAKEBOB* and use the Kaldi toolkit. Notably, this process can be time-consuming and requires modification

of the shell configuration file. Therefore, we wrote a dedicated tutorial detailing all steps in the *setup_SR.md* file.

A.3.2 Basic Test

The SMACK attack framework consists of two main components, the adapted genetic algorithm (AGA) and gradient estimation scheme (ES), each can be individually tested. Due to limited space, some printed outputs are omitted in this appendix and some of the commands are broken into multiple lines to fit the template. The full basic test can be found in the *README.md* document in the *Basic Tests* section, please copy the commands from there.

The command for basic tests of the AGA is:

```
$ python3 genetic.py
```

You are expected to see printed outputs comprising the unique ID of individuals in the population and their fitness value breakdown, including fitness value, confidence score, adversarial term value, and audio quality term value.

Similarly, the basic functionality of the gradient estimation part can be tested with:

```
$ python3 gradient.py
```

Note that the above two tests are based on speaker recognition systems, so they examine the setup and functionality of both attack algorithms and SR models. If the SR models are not properly setup prior to tests, the basic tests would fail automatically.

Besides, the normal functionality of the target models can be tested by running corresponding scripts. For ASR, we can use an adversarial example to test *iflytekASR* model:

```
$ python3 iflytek_ASR.py \
"SMACK_Examples/iflytekASR_THEY DID NOT HAVE \
A LIGHT.wav"
```

And you are expected to see the output as follows:

```
iflytek ASR Result after 1 connection retries:
THEY DID NOT HAVE A LIGHT
```

Similarly, the command and expected output for Google speech-to-text model is:

```
$ python3 google_ASR.py \
"SMACK_Examples/success_gmmSV_librispeech_p1089.wav"

Google ASR Result after 0 retries:
MY VOICE IS THE PASSWORD
```

For SR models, the testing command with the provided adversarial example is:

```
$ python3 speaker_sv.py \
"SMACK_Examples/success_gmmSV_librispeech_p1089.wav" \
gmmSV librispeech_p1089
```

And you are expected to see the output saying the tests on GMM-UBM models passed, with additional information on the decision, acceptance threshold, and score. The complete commands and associated outputs can be found in the "*README.md*" in the artifact.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SMACK can be used to generate natural speech that misleads the state-of-the-art automatic speech recognition models, including commercial products such as iFlyTek and Google speech-to-text. This is proven by the experiment (E1) and (E2) described in Section 8.1 whose results are reported in Table 1 and Table 2.

(C2): SMACK can be used to generate natural speech that misleads the state-of-the-art speaker recognition models. The attack can also be achieved for the challenging inter-gender attack scenario, where the speaker of the adversarial example and targets are of different genders. This is proven by the experiment (E3) and (E4) described in Section 8.2 whose results are reported in Table 3.

A.4.2 Experiments

(E1): *[iFlyTek ASR Attack] [10 human-minutes + 4 compute-hour + 15GB disk]:*

Preparation: The environment installation is detailed in the previous section. Since the target models are commercial products, the speech recognition query is limited and comes with a cost. Please don't hesitate to contact us if you find the service no longer available. We will replace a valid token for your use.

Execution: In the attack against ASR systems, we provide two real-world speech recognition services as the target models, iFlytek and Google. In this experiment against iFlyTek, please use the following command:

```
$ python3 attack.py \
--audio "./Original_TheyDenyTheyLied.wav" \
--model iflytekASR \
--content "They deny they lied" \
--target "They did not have a light"
```

Results: You are expected the see printed outputs similar to the basic tests. All the intermediate audio files generated throughout the attack process are stored in the "*./SampleDir*" directory. An example terminal output produced by this attack is recorded in the "*SMACK_Examples/iflytekASR_THEY DID NOT HAVE A LIGHT.txt*" file, and the resulted adversarial example is provided as "*SMACK_Examples/iflytekASR_THEY DID NOT HAVE A LIGHT.wav*". The example is named as its transcription by the target model. To validate the adversarial example, please use the same command in the basic test:

```
$ python3 iflytek_ASR.py \
"SMACK_Examples/iflytekASR_THEY DID NOT \
HAVE A LIGHT.wav"
```

(E2): *[Google ASR Attack] [10 human-minutes + 1 compute-hour + 15GB disk]:*

Preparation: The environment setup is detailed in the previous section. The Google speech-to-text is a commercial cloud service that requires user token. It is already provided in the "*google_token.json*" file in the artifact and no other setup is needed as long as the token is still valid. Please contact us for a renewed token if it expires.

Execution: Similar to the attack against iFlyTek, please use the following command to launch the attack against Google ASR:

```
$ python3 attack.py \
--audio "./Original_SamiGotAngry.wav" \
--model googleASR \
--content "Sami got angry" \
--target "Send me that"
```

Results: The expected results are similar to the experiment (E1). We also provide a sample adversarial example "*SMACK_Examples/googleASR_SEND ME THAT.wav*", along with its terminal prints recorded in the "*SMACK_Examples/googleASR_SEND ME THAT.txt*" file. To validate the adversarial example, please use the command:

```
$ python3 google_ASR.py \
"SMACK_Examples/googleASR_SEND ME THAT.wav"
```

(E3): *[GMM-based SR Attack] [10 human-minutes + 8 compute-hour + 15GB disk]:*

Preparation: It requires the setup of both attack framework and speaker verification encapsulated in the Kaldi toolkit. The detailed guidance for installing Kaldi and associated dependencies are included in the "*setup_speakerRecog.md*" file in the artifact.

Execution: In this attack, we target a well-established model GMM-UBM deployed with the Kaldi toolkit. The command for running the attack is as follows:

```
$ python3 attack.py \
--audio "./Original_MyVoiceIsThePassword.wav" \
--model gmmSV
--content "My voice is the password" \
--target librispeech_p1089
```

By using this command, we conduct an inter-gender attack, that is, the reference audio "*./Original_MyVoiceIsThePassword.wav*" is uttered by a woman while the target speaker *librispeech_p1089* is a man.

Results: All the intermediate audio files generated throughout the attack process are stored in the "*./SampleDir*" directory. The adversarial examples that successfully achieve the adversarial goal will be stored in the "*./SuccessDir*" directory. The terminal prints also reveal the process of that attack and intermediate results (such as loss values). An adversarial example generated by the attack is provided in "*SMACK_Examples/success_gmmSV_librispeech_p1089.wav*, and its success can be validated in the basic test 4. The associated printed output throughout the optimization process is also provided in "*SMACK_Examples/success_gmmSV_librispeech_p1089.txt*.

(E4): *[ivector-based SR Attack] [10 human-minutes + 8 compute-hour + 15GB disk]:*

Preparation: The setup required by this experiment is the same with experiment (E3).

Execution: In this attack, we target another well-established model ivector-PLDA deployed with the Kaldi toolkit. The command for running the attack is as follows:

```
$ python3 attack.py \
--audio "./Original_MyVoiceIsThePassword.wav" \
--model ivectorCSI
--content "My voice is the password" \
--target librispeech_p1089
```

The launched attack specified by this command is also an inter-gender attack.

Results: The expected results are similar to that of the experiment (E3). We also provide two adversarial examples generated by the attack in "*SMACK_Examples/success_ivectorCSI_librispeech_p1089.wav*", "*SMACK_Examples/success_ivectorCSI_librispeech_p1089_1.wav*", and their success can be validated in the basic test. The associated printed output throughout the optimization process is also provided in "*SMACK_Examples/success_ivectorCSI_librispeech_p1089.txt*.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: URET: Universal Robustness Evaluation Toolkit (for Evasion)

Kevin Eykholt*
IBM Research

Taesung Lee*
IBM Research

Douglas Schales
IBM Research

Jiyong Jang
IBM Research

Ian Molloy
IBM Research

Masha Zorin
University of Cambridge

A Artifact Appendix

A.1 Abstract

The provided artifact contains URET as described in the paper. The tools provided are sufficient to allow users to perform custom adversarial evaluations on machine learning classifiers regardless of input domain. Specifically, this version includes input transformer definitions for the common input types (e.g., numerical, text, and categorical) as well as the binary file input type described in the paper. With respect to results reproduction, it contains the model checkpoints, evaluation data, and notebooks used to generate most of the results in Table 6.

Some components described in the paper have been purposely left out of the provided artifact for proprietary reasons:

- No implementation of the "Model Guided" algorithm. This implementation was deemed proprietary.
- No data/notebooks for the Malware experiments. The malware samples used for evaluation are proprietary and may pose a risk if improperly handled.
- No data/notebooks for the DGA experiments. The training and evaluation data are proprietary. The model code is also proprietary.

Despite these limitations, the provided artifact can be used to perform the custom evaluations described in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There should be no security, privacy, and ethical concerns.

A.2.2 How to access

URET is an evaluation toolkit for adversarial evasion attacks. The public URET repository is accessible at https://github.com/IBM/URET. It should contain the code necessary to perform an evaluation as well as notebook examples of how to use URET. The stable URL used for Artifact Evaluation is https://github.com/IBM/URET/tree/8bd1b4f4d78ac19f026e862b31ae933983c99551.

*These authors contributed equally.

A.2.3 Hardware dependencies

Our artifact does not have any hardware requirement. Of note, a GPU is not required to run the evaluation notebooks and pre-trained model checkpoints have been provided. We tested our artifact on an Ubuntu 18.04 system with 8 CPU cores.

A.2.4 Software dependencies

The artifact repository contains a setup script for installing the required python libraries to use URET, independent of any machine learning libraries (e.g., Tensorflow, PyTorch, etc.). However, the example notebooks require a different setup script, which is included in the artifact, as the model checkpoints were trained using older libraries. The example notebooks were tested using Python 3.8. In Section A.3., we detail the necessary steps to install the required python libraries. We recommend evaluators create a virtual environment prior to running the setup script.

A.2.5 Benchmarks

The artifact requires the 2018 Home Mortgage Disclosure Act (HMDA) dataset to run the evaluation notebooks. We have already included a copy of the dataset in the artifact.

A.3 Set-up

A.3.1 Installation

Here, we provide instructions to deploy URET and run the evaluation notebooks. This was tested using Python 3.8.

1. Clone the artifact from the stable URL: https://github.com/IBM/URET/tree/8bd1b4f4d78ac19f026e862b31ae933983c99551

2. In the artifact directory, replace the existing setup.py file with the version from from notebooks/setup.py.
3. Run the setup script in the top level directory (i.e. pip install -e .) to install the evaluation libraries. It is suggested you do this in a virtual environment.

After step 3, URET should be ready for use. To reproduce most of the results in Table 6, move in to the notebooks/ directory and run HMDA_results.ipynb. We have included pre-computed adversarial examples generated from running each of the exploration algorithms described in the paper. This samples are stored in notebooks/data/HMDA_adv_samples. Note that running a generation notebook (e.g., notebooks/HMDA_random.ipynb) will overwrite the saved samples by default.

A.3.2 Basic Test

After setting up URET, the easiest method to test functionality is to run one of the adversarial generation notebooks. We recommend running notebooks/HMDA_random.ipynb as it is the fastest algorithm to run. The notebook should run without errors, though you may get some warning messages. Cell 4 should display several progress bars and text related to the model being evaluated and the adversarial success rate of the generated samples.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *URET can be used to perform generating adversarial evasion examples for a variety of input domains and formats. This claim proven by experiment E1 and Sections 6.2 and 6.3 in the paper. Experiment E1 is described in Section 6.1 of the main paper and its results are reported in Table 6.*

(C2): *URET can generate adversarial examples for inputs containing a multiple features in the input. This claim proven by experiment E1.*

(C3): *URET can generate adversarial examples for inputs containing a single feature. This claim proven by Sections 6.2 and 6.3 in the paper.*

(C4): *URET presents several different exploration confirmations that can be selected based on user needs. Experiment E1 shows results for several exploration configurations as well as a baseline (Random) to highlight the success rate/speed tradeoff.*

A.4.2 Experiments

(E1): *[HMDA Adversarial Examples][6 Human-hours]: Generate adversarial examples for five HMDA classification models using a baseline four exploration configurations.*

Preparation: Follow the installation instructions in Section A.3.1. Once URET has been installed along with its required libraries, change to the notebooks/ directory before beginning the experiment.
Execution: We have pre-computed adversarial examples for each of the generation notebooks. If the evaluator wants to re-generate the adversarial examples, then run the following notebooks:

1. notebooks/HMDA_random.ipynb - [25 Human-minutes] This generates adversarial examples where every transformation edge is selected randomly.
2. notebooks/HMDA_brute.ipynb - [1.5-2 Human-hours] This generates adversarial examples by exploring every edge.
3. notebooks/HMDA_lookup.ipynb - [1 Human-hour] This generates adversarial examples using the lookup table algorithm. Each generation process will first compute a transformation weight lookup table followed by generation.
4. notebooks/HMDA_simanneal.ipynb - [2.5-3 Human-hours] This generates adversarial examples using the simulated annealing algorithm. The default configuration file assigns 1 sec of attack time per sample.

Of the models, we found that the random forest and multi-layer perception models require the most amount of time to generate.
Results: To generate attack success rate and transformation count results on generated adversarial examples, run notebooks/HMDA_results.ipynb. It expects that there are adversarial examples for each exploration configuration and model.
To get the per sample generation times, we divide the generation time shown in the generation notebooks by the number of samples (2000). We have noticed that the simulated annealing generation time can be sometimes longer than the specified amount.

We have provide configuration files for each of the experiments shown in Table 6 of the paper. Due to randomness, there may be some slight variation in the success rate, transformation count, and per sample generation time between adversarial example generations. The configuration files can be found in notebooks/configs/HMDA. If interested, the evaluator can alter these configuration files to try different exploration settings. For the non-simulated annealing configuration files, consider modifying the beam width (i.e., how many potential transformation candidates are considered) and beam depth (i.e., how many transformations can be applied). For simulated annealing, consider modifying the transformation parameters:

- max_transform_i_sampled - Upper limit on feature transformation applied in a single transformation step

- global_max_transforms - how many transformations can be applied

or the attack time. We note that for simulated annealing, modifying the number of transformations without increasing attack time may result in decreasing the success rate given its random exploration process.

Note that we do not include the code, models, or data for the experiments show in sections 6.2 and 6.3 in main paper. We are unable to share the relevant material due to the proprietary nature of data. Section 6.1 is the only experiment that uses entirely non-proprietary data.

A.5 Notes on Reusability

URET is intended to be an evolving set of tools that can be used to evaluate adversarial robustness of classifiers with respect to evasion. If users find the current set of modules insufficient for their needs, they are encouraged to implement their own custom modules using the common interfaces exposed by URET. Specifically, we expect users may need to customize some or all of the following component:

- Input Transformers and Subtransformers (found in uret/transformers) - For data types beyond the basic and binary types we include in URET, users will need to provide new implementations, which the exploration algorithms can use.
- Custom Loss Functions (found in uret/transformers) - URET uses two common loss types: 1) classification loss based on ground truth labels or model predictions and 2) Distance based loss function. Users that require alerted or unique loss functions (e.g., a loss based on time series input data) can define their own function to provide to the explorer during initialization.
- Dependencies - Some feature relationships need to be handled outside of the transformation interface, such as normalization of a multi-feature vector input. These dependencies can be functionally defined and specified in the URET configuration file for the explorer to enforce during example generation.

The goal of URET was to provide a basic, but easily expandable set of tools to be used for adversarial evaluations. We hope that as users customize URET for their own needs, their implementations can be integrated into the public repository to expand URET's capabilities and help other users.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Authenticated private information retrieval

Simone Colombo
EPFL

Kirill Nikitin
Cornell Tech

Henry Corrigan-Gibbs
MIT

David J. Wu
UT Austin

Bryan Ford
EPFL

1 Artifact Appendix

1.1 Abstract

The source code for our single- and multi-server authenticated-PIR schemes and the Keyd public-key server is available at https://github.com/dedis/apir-code under open-source license. The same repository contains unauthenticated-PIR schemes that we implemented as baselines for comparison; as single-server PIR baseline we use the original implementation of SimplePIR [HHCMV22]. Our implementation and the implementation of SimplePIR use C for the performance-critical functions. We perform all the experiments on machines equipped with two Intel Xeon E5-2680 v3 (Haswell) CPUs, each with 12 cores, 24 threads, and operating at 2.5 GHz, and 256 GB of RAM.

1.2 Description & Requirements

1.2.1 Security, privacy, and ethical concerns

None.

1.2.2 How to access

The source code for all the authenticated-PIR schemes, the classic-PIR schemes and Keyd under which this artifact evaluation was tested is available at https://github.com/dedis/apir-code/tree/af3202e3776d4cb880256372dd51613ee34532ba.

1.2.3 Hardware dependencies

We perform all the experiments on machines equipped with two Intel Xeon E5-2680 v3 (Haswell) CPUs, each with 12 cores, 24 threads, and operating at 2.5 GHz. Each machine has 256 GB of RAM, and runs Ubuntu 20.04 and Go 1.17.5. Machines are connected with 10 Gigabit Ethernet. In the experiments for the multi-server schemes and Keyd (Sections 7.1, 7.2 and 7.4), the client and the servers run on separate machines—the experiments use at most six machines. For single-server schemes we use a single machine that runs both client and server. However, it is possible to run the code, together with the accompanying tests, benchmarks and experiments, on any machine equipped the software dependencies listed in the next section.

1.2.4 Software dependencies

Running run the code requires Go (tested with Go 1.17.5 and 1.19.5) and a C compiler (tested with GCC 9.4.0).

Reproducing the evaluation results requires GNU Make, Screen, Python 3[1], Fabric, Tomli, Numpy and Matplotlib.

1.2.5 Benchmarks

None.

1.3 Set-up

1.3.1 Installation

Installation instructions are given in the `Setup` sections of https://github.com/si-co/vpir-code/blob/main/README.md and we report them here. To run the code in the repository install Go (tested with Go 1.17.5) and a C compiler (tested with GCC 9.4.0). To reproduce the evaluation results, install GNU Make, Screen, Python 3, Fabric, Numpy and Matplotlib.

1.3.2 Basic Test

To run all basic tests, users should clone the repository, and download the dump of the SKS PGP key directory using the command

```
bash scripts/download_sks_parsed.sh
```

in the repository's root directory.

To run the basic test, use the following command:

```
go test
```

in the repository's root directory. This command takes about six minutes to run and outputs the time taken by each test. If all the tests pass, the output ends as follows:

```
PASS
ok    github.com/si-co/apir-code
```

[1]The package `python-is-python3` might be needed.

1.4 Evaluation workflow

1.4.1 Major Claims

Our paper claims what follows.

Multi-server point queries (Section 7.1).

(C1): The maximum overhead for our multi-server authenticated-PIR scheme for point queries, in comparison with classic unathenticated PIR is 2.9× for user time and 1.8× for bandwidth. This is the outcome of experiment (E1), whose results are presented in Fig. 3.

(C2): The impact of the number of servers on our multi-server authenticated-PIR scheme for point queries is almost negligible for user time and imposes a linear increase for bandwidth. This is the result of experiment (E2), whose results are reported in Fig. 4 in the body of the paper.

(C3): The preprocessing cost for our multi-server authenticated PIR scheme for point queries is linear in the database size. This is the result of experiment (E3), whose results are reported in Fig. 8 in Appendix C.

Multi-server complex queries (Section 7.2).

(C4): The user time and bandwidth overheads of the authenticated-PIR schemes for complex queries against classic unauthenticated-PIR schemes are less than 1.1×. This is the outcome of experiment (E4), whose results are presented in Fig. 5.

Single-server point queries (Section 7.3).

(C5): The authenticated-PIR schemes from the decisional Diffie-Hellman assumption (DDH) and from the learning-with-errors assumption (LWE) have integrity error 2^{-128}. The DDH construction has a smaller digest, i.e., lower offline bandwidth, but has twice the online bandwidth of the LWE construction. The LWE construction is also faster (3-79×). The scheme with integrity amplification (LWE^+) has integrity error 2^{-64} but the same classic-PIR privacy as SimplePIR [HHCMV22]. LWE^+ is faster than LWE for the 1 KiB and 1 MiB databases, but slower (1.4×) for the 1 GiB database. SimplePIR is 30-100× faster than LWE^+. These results are the outcome of experiment (E5), whose results are presented in Fig. 6 in the body of the paper.

Application evaluation (Section 7.4).

(C6): For classic key look-ups we measure the wall-clock time needed to retrieve a PGP public-key with authenticated PIR, classic PIR without authentication, and by direct download. We measure 1.11 seconds for authenticated PIR, 1.10 seconds for unauthenticated PIR and 0.22 seconds for non-private direct look-up. This is the result of experiment (E6), whose results are discussed in Section 7.4 in the body of the paper.

(C7): To analyze the performance of Keyd in computing private statistics over keys, we measure user-perceived time and bandwidth of different predicate queries. For all the predicates, the user-perceived time and bandiwdth overheads of authenticated PIR are upper bounded by a factor of 1.05×. This is the outcome of experiment (E7), whose results are presented in Table 7 in the body of the paper.

1.4.2 Experiments

The experiments use at most six server machines (to run the client and servers) and an additional machine (that we call *local*) to manage the experiments. The local machine can be a commodity computer, since it is used only to run light scripts and gather results. Clone the repository on all the server machines and on the local machine.

(E1): [*15 human-minutes + 2 compute-hour*]: This experiment measures the user-time and bandwidth overheads of *two*-server authenticated-PIR schemes for point queries in comparison with unauthenticated PIR. This experiments uses three server machines: one client and two servers.

Preparation: Edit `simulations/multi/config.toml` on the *local* machine to indicate the IP address of the client machine and the IP addresses and ports of the two server machines. The default port numbers are safe to use.

Execution: Run the following commands from the repository's root on the *local* machine:

```
cd simulations/multi
APIR_USER=<username>
APIR_PASSWORD=<password>
APIR_PATH=<path>
python simul.py -e point
```

where `<username>` and `<password>` are the username and password for the servers, respectively, and `<path>` is the path of the repository's root on the servers.

Results: Run the following commands from the repository's root:

```
cd simulations/multi
python plot.py -e point
```

The command stores the figure in `simulations/multi/figures/point.eps`.

(E2): *[15 human-minutes + 18 compute-minutes]*: This experiment measures the impact of the number of servers on our multi-server authenticated-PIR schemes for point queries. This experiments uses six server machines: one client and five servers.

Preparation: Edit `simulations/multi/config.toml` on the *local* machine to indicate the IP address of the

client machine and the IP addresses and ports of the five server machines. The default port numbers are safe.
Execution: Run the following commands from the repository's root on the *local* machine:

```
cd simulations/multi
APIR_USER=<username>
APIR_PASSWORD=<password>
APIR_PATH=<path>
python simul.py -e point_multi
```

where `<username>`, `<password>` and `<path>` are as in experiment E1.
Results: Run the following commands from the repository's root:

```
cd simulations/multi
python plot.py -e point_multi
```

The command stores the figure in `simulations/multi/figures/multi.eps`.

(E3): *[5 human-minutes + 9 compute-minutes]*: This experiment measures the cost of preprocessing for our multi-server authenticated-PIR scheme for point queries. This experiment uses one server machine.
Preparation: Nothing.
Execution: Run the following commands from the repository's root on the *server* machine:

```
cd simulations
make preprocessing
```

Results: Run the following commands from the repository's root:

```
cd simulations
python plot.py -e preprocessing
```

The command stores the figure in `simulations/figures/preprocessing.eps`.

(E4): *[15 human-minutes + 36 compute-minutes]*: This experiment measures the user-time and bandwidth overheads of two-server authenticated-PIR schemes for complex queries in comparison with unauthenticated PIR. This experiments uses three server machines: one client and two servers.
Preparation: As in experiment E1. The file `simulations/multi/config.toml` on the *local* machine must list *only two* servers.
Execution: Run the following commands from the repository's root on the *local* machine:

```
cd simulations/multi
APIR_USER=<username>
APIR_PASSWORD=<password>
APIR_PATH=<path>
python simul.py -e predicate
```

where `<username>`, `<password>` and `<path>` are as in experiment E1.

Results: Run the following commands from the repository's root:

```
cd simulations/multi
python plot.py -e predicate
```

The command stores the figure in `simulations/multi/figures/complex_lines.eps`.

(E5): *[15 human-minutes + 21 compute-hour]*: This experiment measures the user-time and bandwidth overheads of single-server authenticated-PIR schemes for point queries in comparison with SimplePIR [HHCMV22]. This experiment uses one server machine.
Preparation: Nothing.
Execution: Run the following commands from the repository's root on the *server* machine:

```
cd simulations
make single
```

To evaluate SimplePIR, clone the following repository: `https://github.com/si-co/simplepir`. The code is the same as the original repository, but it runs the evaluation on the same database sizes as authenticated PIR and produces a compatible JSON file for the results. Run the following command (45 compute-minutes) from the repository's root on the *server* machine:

```
cd pir
go test -timeout 0 -run=PirSingle
```

Copy the file `simplePIR.json` (in the `pir` directory) in `simulation/results`.
Results: Run the following commands from the repository's root:

```
cd simulations
python plot.py -e single
```

The command stores the figure in `simulations/figures/single_bar.eps`.

(E6): *[20 human-minutes + 10 compute-minutes]*: This experiment measures the user-time needed download a PGP public-key with authenticated PIR for point queries, classic unauthenticated PIR for point queries and by direct download. This experiment uses three machines: one client and two servers.
Preparation: Download the dump of the SKS PGP key directory using the command

```
bash scripts/download_sks_parsed.sh
```

in the repository's root directory on the *two* servers. Set the IP addresses of the two servers in `simulations/real/real_client_pir.sh` and in `config.toml` (in the repository's root) on the client machine.
Execution: Run the following commands from the repository's root on the *first server*:

```
cd simulations/real
bash real_server_pir.sh 0
```

Similarly, on the *second server* run:

```
cd simulations/real
bash real_server_pir.sh 1
```

A server is running properly when it logs:

```
gRPC server started at <ip>:<port>
```

Once both servers started, run the following command on the *client machine*:

```
cd simulations/real
bash real_client_pir.sh
```

This command executes 30 look-ups with unauthenticated PIR and 30 with authenticated PIR. At the end, the client automatically shuts both servers down.

Results: Copy `simulations/results/stats_*` from the three machines (two servers and the client) on the *local* machine in the folder `/simulations/results`. Run the following commands:

```
cd simulations
python plot.py -e real
```

The command prints the results directly on the terminal.

(E7): *[20 human-minutes + 5 compute-hours]*: This experiment measures the user-time needed to compute statistics on the PGP public-keys with authenticated PIR for predicate queries and unauthenticated PIR. This experiment uses three machines: one client and two servers.

Preparation: As in Experiment E6, but set the IP addresses of the two servers in `simulations/real/real_client_fss.sh`.

Execution: Run the following commands from the repository's root on the *first server*:

```
cd simulations/real
bash real_server_fss.sh 0
```

Similarly, on the *second server* run:

```
cd simulations/real
bash real_server_fss.sh 1
```

For this experiment, it is not needed to wait for the servers to properly start. Run the following command on the *client machine*:

```
cd simulations/real
bash real_client_fss.sh
```

Results: Copy `simulations/results/stats_*` from the three machines (two servers and the client) on the *local* machine in the folder `/simulations/results`. Run the following commands:

```
cd simulations
python plot.py -e realcomplex
```

The command prints the results directly on the terminal. Table 7 has a different formatting, but values are the same as the one that the command prints on screen.

1.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: GigaDORAM: Breaking the Billion Address Barrier

Brett Falk*
University of Pennsylvania

Rafail Ostrovsky*
UCLA

Matan Shtepel*
UCLA

Jacob Zhang*
UCLA

A Artifact Appendix

A.1 Abstract

The artifact for the USENIX'23 paper "GigaDORAM: Breaking the Billion Address Barrier" is this GitHub repository. The repository contains a C++ implementation of the GigaDORAM protocol, benchmarking scripts, and explanations helpful to reproducing the experiments in the paper. The explanations are composed of a standard README file and a **detailed video walkthrough** which we believe to be the most convenient way to install GigaDORAM and reproduce our results.

A.1.1 What is GigaDORAM?

GigaDORAM is a 3-party state-of-the-art Distributed ORAM protocol (DORAM) protocol. DORAM is a stateful multiparty cryptographic protocol. We envision the protocol holding a secret shared state, `Memory`, with `N` 0-initialized cells, `Memory[0],...,Memory[N-1]`. An execution of the protocol takes secret shared variables X_{query}, Y_{new}, `IsWrite` as input. The output of the protocol is secret shared `Memory[`X_{query}`]`. If `IsWrite=1`, a stateful update `Memory[`X_{query}`] :=` Y_{new} is preformed. Under certain non-collusion assumptions, an execution of the protocol does not reveal *any* information to the participating parties.

GigaDORAM is a 3-party DORAM protocol specialized for the low-latency, large `N` setting. In these settings, GigaDORAM significantly outperforms previous protocols. In other settings, GigaDORAM preforms comparably to previous protocols. See the paper, and in particular Section 9, for more details.

A.2 Description & Requirements

Roughly speaking, we benchmarked GigaDORAM in 2 different settings

- *Single machine tests:* we execute GigaDORAM through 3 processes on the same machine. This enables us to artifically restrict the network between the machines processes via the `tc` command and benchmark the performance of GigaDORAM in a variety network settings.
- *Multi machine tests:* we execute GigaDORAM on 3 different AWS EC2 instances in the same AWS region. These tests are meant to demonstrate the "real world" potential of GigaDORAM.

In Section A.3 and Section A.4 we show how to set-up and execute both kinds of tests, respectively. Again, we suggest that the best way to follow along with setup, installation, and experiment-replication is our **detailed video walkthrough**

A.2.1 Security, privacy, and ethical concerns

`single_server_experiments.py` runs `sudo tc qdisc replace dev lo root` to simulate network latency and bandwidth limits on the loopback interface, which can slow down other programs running on the machine. We recommend running on a dedicated VM. If anything strange happens, the network changes can be undone with `sudo tc qdisc del dev lo root`

A.2.2 How to access

Our artifact can be accessed via this GitHub repository. It is not our development repository and will not be evolving.

A.2.3 Hardware dependencies

For evaluation, a processor supporting the Intel SSE2 instruction set is and we recommend at least 8 CPU cores and 8GB of RAM. While local evaluation of our artifact is possible, benchmarking on AWS is necessary for replicating our results. If AWS credits are not available to reviewers, please reach out to us via the anonymous submission portal.

A.2.4 Software dependencies

- A Linux machine with processor supporting the Intel SSE2 instruction set.
- `sudo` access is unfortunately required; this isn't an issue on AWS.

*Authors are in alphabetical order.

- For compilation, the EMP-toolkit library. Follow the instructions at EMP-toolkit repository to install EMP and its dependencies.
  - EMP's dependencies are extremely basic (`python3 cmake git build-essential libssl-dev`) and are all needed to build GigaDORAM. Those can be installed using the `apt` package manager.
- On a typical Linux system there are no dependencies needed to run the compiled binary.

A.2.5 Benchmarks

No data is needed to run our tests.

A.3 Set-up

In this section we describe set-up for single-server GigaDORAM tests, multi-server GigaDORAM tests, and tips for optionally testing other DORAM constructions.

A.3.1 Installation

Single server tests.

1. Clone our repository
2. Run `compile.sh`. If this fails, make sure you've installed EMP-toolkit.

Multi server tests. The multi-machine tests are designed to be run on AWS EC2. *Warning*: Follow the directions below carefully; it is easy to miss something which will cause the benchmarks to not be able to run.

- You will need choose an AWS region and create and start AWS EC2 instances named `DORAM_benchmark_1`, `DORAM_benchmark_2`, `DORAM_benchmark_3` in that region.
  - Use the same SSH key for access to all 3 instances, as it will be an argument to the script later.
  - Set security group settings to allow TCP traffic between the 3 instances.
  - We used `c5n.metal` instances, which guaranteed that our parties were not running on the same physical host, and also provided high multi-threaded performance.
  - We used Ubuntu 22.04 but any modern enough Linux distribution should work. Recall that Intel processors are required.
- For most tests, the instances should be created in a cluster placement group. A quick how to is covered in our video tutorial. For more information about cluster placement groups, see here.
- In addition to the requirements for single server tests, the AWS CLI (package `awscli` in apt) needs to be installed on the machine used for builds.
- After installing, configure your region and access keys with `aws configure`.
- Additionally, you will need to add these lines to `~/.ssh/config/` on the build machine: `Host * StrictHostKeyChecking no`

  Without disabling StrictHostKeyChecking, the experiment script will be unable to ssh to new hosts in the background, since user input would be required to continue connecting to a new host.
- Nothing needs to be installed on the benchmark instances!

Optional: other DORAM constructions. In this section, we briefly comment on the procedure we took to install and set-up other DORAM constructions.

- *DuORAM*: We benchmark DuORAM via their well documented dockerization.Detailed information can be found in their README.
- *3PC-ORAM*: We benchmarked 3PC-ORAM via the convenient dockerization graciously provided by the DuORAM team. Again, details can be found in the README.
- *Sqrt ORAM, Circuit ORAM, fss-FLORAM, cprg-FLORAM*: We benchmark Sqrt ORAM, Circuit ORAM, fss-FLORAM, and cprg-FLORAM via Doerner and shelats' original code. Due to a reliance on the somewhat aged obliv-c framework, we ran into some difficulties running their code. We try to give some helpful tips here (note: some of this steps may be redundant or incomplete – we simply note here what worked for us):
  - We started two Ubuntu 18.04.6 EC2 instances in a cluster placement group, one to be the "server" and the other to be the "client"
  - To install the needed old version of `ocaml`, run `sudo apt install opam`, `opam switch create 4.06.0`, and `eval $(opam env --switch=4.06.0)`
  - To install the needed old version `gcc`, `sudo apt install -y gcc-9 g++-9 cpp-9`
  - Then follow the obliv-c README to install.
  - Then follow the FLORAM README to install
  - *PFEDORAM*: PFEDORAM is proprietary, and we obtained benchmarks directly from Bingsheng Zhang, one of the authors of the paper.

A.3.2 Basic Test

Single server. `./benchmark_doram_locally.sh 100us 10Gbit -prf-circuit-filename LowMC_reuse_wires.txt -build-bottom-level-at-startup false -num-query-tests 10 -log-address-space 20 -num-levels 3 -log-amp-factor 4 -num-threads 4`

Multi server. `./run_3_server_experiment.sh ../my_key.pem -prf-circuit-filename LowMC_reuse_wires.txt -build-bottom-level-at-startup false -num-query-tests 10 -log-address-space 20 -num-levels 3 -log-amp-factor 4 -num-threads 20`

Optional: other DORAM constructions. Other DORAMs contain simple tests in their respective READMEs

A.4 Evaluation workflow

A.4.1 Major Claims

The main claim of our paper is the performance of GigaDORAM that can be seen in various network settings / configurations. These can be found in Figure 5, Figure 6, Table 1, and Table 2.

A.4.2 Experiments

Below we provide descriptions of how to execute our main results. In the repositories README we also provide descriptions on how to run any set of parameters in both the single server and multiserver setting

(E1): *Reproducing Figures 6a and 6b, 10 human-minutes + 1.5 compute hour + 20 compute threads and 20 GB RAM. These tests reproduce the performance of GigaDORAM in simulated varying latency and varying bandwidth settings.* :

Preparation: noted in Section A.3.

Execution: The single server experiments are run by `./single_server_experiments.py` which prints usage information with the `-h` flag.

To reproduce the tests used to generate the GigaDORAM data in Figures 6a and 6b in the paper, run

```
./single_server_experiments.py Figure6a
./single_server_experiments.py Figure6b
```

We ran these tests on a machine with 96 CPUs and saw best results with 20 threads per party (which is the default). If you are running on a smaller machine, you should use a num_threads which is less than 1/3 the number of CPUs available, for example:

```
$ ./single_server_experiments.py Figure6a
    --threads 2
$ ./single_server_experiments.py Figure6b
    --threads 2
```

Results: The concatenated output from all experiments will be written to `single_server_results/doram_timing_report${i}.txt` for party i in 1, 2, 3.

These files are are human readable and formatted in blocks as, for example:

```
DORAM Parameters
Number of queries: 1000
Build bottom level at startup: 0
Log address space size: 16
Data block size (bits): 64
Log linear level size: 8
Log amp factor: 4
Num levels: 3
PRF circuit file: LowMC_reuse_wires.txt
Num threads: 1

Timing Breakdown
Total time including builds: 2.89467e+06 us
Time spent in queries: 2.83478e+06 us
Time spent in query PRF eval: 1.21668e+06 us
Time spent querying linear level: 751191 us
Time spent in build PRF eval: 8775 us
Time spent in batcher sorting: 0 us
Time spent building bottom level: 0 us
Time spent building other levels: 20811 us

SUMMARY
Total time including builds: 2.89467e+06 us
Total number of bytes sent: 2.7828e+07
Queries/sec: 345.462
```

The `SUMMARY` section is the most important to look at, as it gives the total time and communication to run the test.

WARNING: single_server_experiments.py runs `sudo tc qdisc replace dev lo root` to simulate network latency and bandwidth limits on the loopback interface, which can slow down other programs running on the machine. We recommend running on a dedicated VM. If anything strange happens, the network changes can be undone with `sudo tc qdisc del dev lo root`

(E2): *Multi server tests, reproducing Figure 5, Table 1, and Table 2, 10 human-minutes + 1 compute-hour + 3 strong, running AWS EC2 machines*

Preparation: Noted in Section A.3.

Execution: `./multi_server_experiments.py -h` for syntax help. Run the following experiments

one by one, checking the results output after each

```
$ ./multi_server_experiments.py Figure5
    ../my_pem_file.pem
$ ./multi_server_experiments.py Table1
    ../my_pem_file.pem
$ ./multi_server_experiments.py Table2
    ../my_pem_file.pem
$ ./multi_server_experiments.py Figure8
    ../my_pem_file.pem
```

Results: The concatenated output from all experiments will be written to `multi_server_results/doram_timing_report${i}.txt` for party i in 1, 2, 3, following the same format as the single server results.

(*Optional* E3): *reproducing the results of other DORAMs [1.5 human-hours (including setup) + 4 compute-hour + several AWS EC2 instances (number pending on how many tests are ran in parallel)]: benchmark the performance (in queries/sec) of other DORAM constructions.*

Preparation: Noted in Section A.3.

Execution: In this section, we briefly comment on the procedure we took to benchmark other DORAM constructions. We describe below which command For the settings we tested each construction in and additional discussions, please see Section 9, Figures 5 and 6, and Appendix E of the paper. As each figure describes the benchmark setting, here we only describe the code commands we used.

- *DuORAM*: For varying `numops` and `size`, we summed `./run_experiment read size numops preproc 3P` and `./run_experiment readwrite size numops online 3P` to account for both the online and offline costs of protocol.
- *3PC-ORAM*:Using `./run-experiment size numops` we benchmarked 3PC-ORAM's *reads*, which are no more expensive than writes.
- *Sqrt ORAM, Circuit ORAM, fss-FLORAM, cprg-FLORAM*: On the "server" EC2 machine call `./bench_oram_write -e ADDRESS_SPACE_SIZE -o TYPE -i 1024` and then on the "client" EC2 machine call `./bench_oram_write -e ADDRESS_SPACE_SIZE -o TYPE -i 1024 -c ADDRESS_OF_SERVER` where `ADDRESS_SPACE_SIZE` is `N`, not `log_N`, `TYPE` can be either `{sqrt, circuit, fssl, fssl_cprg}`, `-i` marks the number of writes to be done, and `ADDRESS_OF_SERVER` is the IP address of server (make sure AWS security group is set to allow for TCP traffic).
- PFEDORAM: PFEDORAM is proprietary, and we obtained benchmarks directly from Bingsheng Zhang, one of the authors of the paper.

Results: Each of these constructions respective READMEs explains the output format.

A.5 Notes on Reusability

We believe the GigaDORAM implementation and benchmarks we provide reflect the performance of GigaDORAM accurately in various network settings. However, our implementation is *not* production ready. For instance, it may contain timing attacks and lacks several important optimizations.

It is possible to execute GigaDORAM on data other than the (dummy) benchmark data, but at this stage that might require some slight understanding of our codebase.

To start, we suggest `https://github.com/jacob14916/GigaDORAM-USENIX23-Artifact/blob/main/doram/doram_array.h` in our repository which, via several subclasses, implements the GigaDORAM protocol. For black-box usage, other than the constructor, the only function from this class that that should be called publicly is `read_and_write`. Unfortunately, because GigaDORAM is a multi-machine, multi-threaded program, running GigaDORAM is not as simple as calling the constructor and then the method. For an example we suggest `https://github.com/jacob14916/GigaDORAM-USENIX23-Artifact/blob/main/test/doram/doram.cpp` which shows initialization the resources (e.g. Psuedorandom function seeds) necessary for executing GigaDORAM, proceeds to construct a GigaDORAM object, and then calls `read_and_write` on it repeatedly. To see how `doram.cpp` gets called, we suggest `https://github.com/jacob14916/GigaDORAM-USENIX23-Artifact/blob/main/run_3_server_experiment.sh` which, for example, is called by `https://github.com/jacob14916/GigaDORAM-USENIX23-Artifact/blob/main/multi_server_experiments.py`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval

Alexandra Henzinger
MIT

Matthew M. Hong
MIT

Henry Corrigan-Gibbs
MIT

Sarah Meiklejohn
Google

Vinod Vaikuntanathan
MIT

A Artifact Appendix

A.1 Abstract

Our source code for our two new, high-throughput PIR schemes, SimplePIR and DoublePIR, is available under the MIT open-source license at https://github.com/ahenzinger/simplepir. SimplePIR and DoublePIR, including their extensions to support databases with long records and batch queries (cf. Sections 4.3 and 5.2), are implemented in roughly 1,400 lines of Go code, along with 200 lines of C (for the performance-critical matrix-multiplication routines). For each PIR scheme, the code implements the Setup, Query, Answer, and Recover routines. Our repository additionally contains a suite of correctness tests and performance benchmarks. To obtain our performance numbers, we run our benchmarks on an AWS EC2 `c5n.metal` instance.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The source code for SimplePIR and DoublePIR is available at https://github.com/ahenzinger/simplepir/tree/438b4590aceedf76c7588b03125dfc0db39e361f.

A.2.3 Hardware dependencies

We run our evaluation on an AWS EC2 `c5n.metal` instance running Ubuntu 22.04. However, it is possible to run the SimplePIR and DoublePIR code on any machine with an installation of Go and of a C compiler (see Section A.2.4), though this might require amending the command-line flags passed to the C compiler.

A.2.4 Software dependencies

Running SimplePIR and DoublePIR requires installations of Go and GCC. We additionally require Python, NumPy, and Matplotlib to generate our evaluation plots.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Instructions for installing the required dependencies are given in the `Setup` section of https://github.com/ahenzinger/simplepir/blob/main/README.md. Users should install Go (tested with version 1.19.1) and GCC (tested with version 11.2.0) to run SimplePIR and DoublePIR. Users should additionally install Python (tested with Python3), NumPy, and Matplotlib to generate our evaluation plots.

A.3.2 Basic Test

To run all SimplePIR and DoublePIR correctness tests, users should run the command

```
go test
```

in the `simplepir/pir` directory. The suite of correctness tests runs SimplePIR and DoublePIR on random databases of fixed dimensions and checks that the PIR outputs are correct. The test suite should take roughly 2.5 minutes to run.

This command should produce logging output, followed by this message to indicate that all tests have passed:

```
PASS
ok        github.com/ahenzinger/simplepir/pir
```

A.4 Evaluation workflow

A.4.1 Major Claims

Our paper makes the following claims:

(C1): SimplePIR performance. On a database 1 GB in size containing 2^{33} 1-bit entries, SimplePIR has:

1. a throughput of roughly 10 GB/s/core when running on an AWS EC2 `c5n.metal` instance,
2. 121 MB of offline download, and
3. 242 KB of online communication.

This is shown by experiment (E1), whose results are displayed in Table 8 in the body of our paper.

(C2): DoublePIR performance. On a database 1 GB in size containing 2^{33} 1-bit entries, DoublePIR has:

1. a throughput of roughly 7.4 GB/s/core when running on an AWS EC2 `c5n.metal` instance,
2. 16 MB of offline download, and
3. 345 KB of online communication.

This is shown by experiment (E2), whose results are displayed in Table 8 in the body of our paper.

(C3): Throughput with batching. SimplePIR and DoublePIR's throughput increases when the client makes batches of many queries at once. This is shown by experiment (E3), whose results are displayed in Figure 9 in the body of our paper.

(C4): Application evaluation. On a database 8 GB in size containing 2^{36} 1-bit entries, DoublePIR has:

1. a throughput of roughly 7 GB/s/core when running on an AWS EC2 `c5n.metal` instance,
2. 16 MB of offline download, and
3. 756 KB of online communication.

This is shown by experiment (E4), whose results are reported in Section 8.2 in the body of our paper.

A.4.2 Experiments

(E1): SimplePIR performance [5 compute-minutes]: This experiment benchmarks (1) the communication and (2) the throughput of SimplePIR, when running on a 1 GB database consisting of 2^{33} 1-bit entries.
How to: Run the command

```
LOG_N=33 D=1 go test -bench SimplePirSingle
 ↪ -timeout 0 -run=^$
```

from the `simplepir/pir` directory. The command will run SimplePIR 5 times on a database consisting of 2^{33} 1-bit entries, and print logging information including the communication and the per-core throughput of each run.
Results: For each run of SimplePIR, this experiment prints logging information that look as follows:

- `Offline download: 123572 KB`, indicating that SimplePIR's offline download consists of roughly 121 MB.
- `Online upload: 120.000000 KB`, indicating that SimplePIR's offline upload consists of 120 KB.
- `Rate: 10177.282855 MB/s`, indicating that SimplePIR's throughput was 10,177 MB/s/core.
- `Online download: 120.000000 KB`, indicating that SimplePIR's offline download consists of 120 KB.

(E2): DoublePIR performance [5 compute-minutes]: This experiment benchmarks (1) the communication and (2) the throughput of DoublePIR, when running on a 1 GB database consisting of 2^{33} 1-bit entries.
How to: Run the command

```
LOG_N=33 D=1 go test -bench DoublePirSingle
 ↪ -timeout 0 -run=^$
```

from the `simplepir/pir` directory. The command will run DoublePIR 5 times on a database consisting of 2^{33} 1-bit entries, and print logging information including the communication and the per-core throughput of each run. The logging results are interpreted the same way as in (E1).

(E3): Throughput with batching [1.5h compute-hours]: This experiment benchmarks SimplePIR's and DoublePIR's effective, per-core throughput, when run on batches of queries of increasing size, on a 1 GB database consisting of 2^{33} 1-bit entries.
How to: Run the command

```
go test -bench PirBatchLarge -timeout 0
 ↪ -run=^$
```

from the `simplepir/pir` directory. The command will run both SimplePIR and DoublePIR 5 times on a database consisting of 2^{33} 1-bit entries, with batch sizes ranging from 1 to 1024, and print logging information including the communication and the per-core throughput of each run. The logging results are interpreted the same way as in (E1).
Results: This command produces the output files `simple-batch.log` and `double-batch.log` in the `simplepir/pir` directory. From the `simplepir/eval` directory, run the command

```
python3 plot.py -p batch_tput -f
 ↪ ../pir/simple-batch.log
 ↪ ../pir/double-batch.log -n SimplePIR
 ↪ DoublePIR
```

This command plots SimplePIR and DoublePIR's effective, per-core throughput for various batch sizes, and writes this plot to the file `throughput_with_batching.pdf`. This command was used to generate Figure 9.

(E4): Application evaluation [40 compute-minutes]: This experiment benchmarks (1) the communication and (2)

the per-core throughput of DoublePIR, when running on an 8 GB database consisting of 2^{36} 1-bit entries.

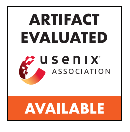
How to: Run the command

```
LOG_N=36 D=1 go test -bench DoublePirSingle
 ↪ -timeout 0 -run=^$
```

from the `simplepir/pir` directory. The command will run DoublePIR 5 times on a database consisting of 2^{36} 1-bit entries, and print logging information including the communication and the per-core throughput of each run. The logging results are interpreted the same way as in (E1).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: Duoram: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation

Adithya Vadapalli
avadapal@uwaterloo.ca
University of Waterloo

Ryan Henry
ryan.henry@ucalgary.ca
University of Calgary

Ian Goldberg
iang@uwaterloo.ca
University of Waterloo

A Artifact Appendix

A.1 Abstract

This artifact contains DUORAM's source code and the necessary scripts to run the experiments and reproduce the major claims of our paper, DUORAM: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation. Our major claims are reflected in Figures 7, 8, 9, and 10 in Section 6.2 of our paper. Along with the source code of DUORAM and the DUORAM replication scripts, the artifact also provides the scripts to replicate the experiments for Doerner and shelat's FLORAM and Jarackei and Wei's 3-party Circuit ORAM, the two implementations the paper compares DUORAM to. The experiments are run in Docker containers under different simulated network conditions. We simulate these conditions by using `tc qdisc add dev eth0 root netem delay Xms rate Ymbit`, to set the latency to `X` ms, and restrict the bandwidth capacity to `Y` Mbit/s. In our experiments, standard network conditions refer to a latency of 30 ms and a bandwidth capacity of 100 Mbit/second. Similarly, colocated network conditions refer to 30 μs of Internet latency and 100 Gbit/s of bandwidth capacity.

A.2 Description & Requirements

A.2.1 How to access

The artifact can be accessed at `https://git-crysp.uwaterloo.ca/avadapal/duoram/src/usenixsec23_artifact`.

To download the artifact:

- `git clone https://git-crysp.uwaterloo.ca/avadapal/duoram`
- `cd duoram`
- `git checkout usenixsec23_artifact`

A.2.2 Hardware dependencies

The hardware dependencies to run our artifact are as follows:

- A system with an x86 processor that supports `AVX2` instructions. This instruction set was released in 2013, so most recent processors should be fine. We have tested it on both Intel and AMD processors. On Linux, `grep avx2 /proc/cpuinfo` to see if your CPU can be used (if the output shows you CPU flags, then it can be; if the output is empty, it cannot).
- At least 16 GB of available RAM. To run the optional "large" tests, you will require at least 660 GB of available RAM (an atypical machine, to be sure, which is why the large tests are optional and not essential to our major claims).

A.2.3 Software dependencies

The Software dependencies to run our artifact are a basic Linux installation, with `git` and `docker` installed. We have tested it on Ubuntu 20.04 and Ubuntu 22.04, with `apt install git docker.io`.

A.3 Setup

Detailed setup instructions are outlined in the `README.md` file in the artifact. We briefly summarize it here.

A.3.1 Installation

The following will download and build the dockers for the DUORAM, FLORAM, and Circuit ORAM systems (approximate compute time: 15 minutes).

```
cd repro && ./build-all-dockers
```

A.3.2 Basic test

A simple "kick the tires" test can be run using `./repro-all-dockers test` from the `repro` directory (approximate compute time: 1 minute). The expected output looks as follows:

```
2PDuoramOnln readwrite 16 1us 100gbit 2
0.86099545 s
2PDuoramOnln readwrite 16 1us 100gbit 2
```

```
22.046875 KiB
2PDuoramTotl readwrite 16 1us 100gbit 2
1.48480395 s
2PDuoramTotl readwrite 16 1us 100gbit 2
177.7138671875 KiB
3PDuoramOnln readwrite 16 1us 100gbit 2
0.012897 s
3PDuoramOnln readwrite 16 1us 100gbit 2
0.104166666666667 KiB
3PDuoramTotl readwrite 16 1us 100gbit 2
0.225875 s
3PDuoramTotl readwrite 16 1us 100gbit 2
12.7916666666667 KiB

Floram read 16 1us 100gbit 2 0.879635 s
Floram read 16 1us 100gbit 2 3837.724609375 KiB

CircuitORAMOnln read 16 1us 100gbit 2 0.313 s
CircuitORAMOnln read 16 1us 100gbit 2 710.625
KiB
CircuitORAMTotl read 16 1us 100gbit 2 0.753 s
CircuitORAMTotl read 16 1us 100gbit 2 4957 KiB
```

What you see here are the four systems (2-party DUORAM, 3-party DUORAM, 2-party FLORAM, 3-party Circuit ORAM), with all except FLORAM showing both the online phase and the total of the preprocessing and the online phase. (FLORAM does not have a separate preprocessing phase.) Each of those seven system/phase combinations shows the time taken for the test run, as well as the average bandwidth used per party.

The output fields are:

- system and phase
- mode (reads, writes, or interleaved reads and writes)
- $\log_2$ of the number of 64-bit words in the ORAM
- one-way latency between the parties (specifying "1us" really means not to add artificial latency, so you end up with the natural latency between dockers, which turns out to be $30\,\mu$s)
- bandwidth between the parties
- number of operations (number of reads or number of writes; interleaved reads and writes do this many reads interleaved with the same number of writes)
- the time in seconds or the bandwidth used in KiB, as indicated

You should see the same set of 14 lines as shown above, though the exact times of course will vary according to your hardware. The bandwidths you see should match the above, however.

If you run the test more than once, you will see means and stddevs of all of your runs.

A.4 Evaluation workflow

A.4.1 Major Claims

Our primary claim is this:

(C1): Under realistic Internet networking conditions, DUORAM outperforms FLORAM (which itself outperforms Circuit ORAM) over a range of ORAM sizes, because it is primarily CPU-bound, while the other schemes are primarily network-bound. Our observation is that it is easier to deploy machines with more local computational power and memory than it is to increase bandwidth or reduce latency between the multiple independent parties running the protocol.

We support this primary claim with the following major claims:

(C2): DUORAM's wall-clock performance changes much less as network conditions change than does FLORAM's.

(C3): DUORAM uses much less bandwidth than FLORAM or Circuit ORAM, and 3P-DUORAM's online bandwidth usage is in fact independent of the ORAM size.

(C4): Even in the less realistic setting where the independent parties running the protocols are colocated, DUORAM's wall-clock performance is better than FLORAM's and Circuit ORAM's, but for a smaller range of ORAM sizes.

(C5): 2P-DUORAM's online performance improves noticeably with increased CPU core availability, unlike FLORAM. (Under our standard network conditions, 3P-DUORAM's online wall-clock time is much smaller than that of FLORAM, regardless of the number of cores.)

A.4.2 Experiments

We provide three sets of experiments: the "small", the "large", and the "scaling" experiments.

The "small" experiments. These experiments support most of our major claims.

(E1): Compare the wall-clock time of 2P-DUORAM, 3P-DUORAM, and FLORAM to do 128 interleaved operations under standard network conditions while varying the ORAM size from 2^{16} to 2^{26} (64-bit items); the results of this experiment appear in Figure 7(a). Supports claim (C1).

(E2): Compare the wall-clock time of 2P-DUORAM, 3P-DUORAM, and FLORAM to do 128 interleaved operations for a 2^{20}-sized ORAM and 30 ms latency, while varying the bandwidth from 10 to 110 Mbit/s; the results of this experiment appear in Figure 7(b). Supports claim (C2).

(E3): Compare the wall-clock time of 2P-DUORAM, 3P-DUORAM, and FLORAM to do 128 interleaved operations for a 2^{20}-sized ORAM and 100 Mbit/s bandwidth, while varying the latency from 10 to 70 ms; the results of this experiment appear in Figure 7(c). Supports claim (C2).

(E4): Compare the bandwidth used by 2P-DUORAM, 3P-DUORAM, and FLORAM to do 128 operations (read, write, or interleaved reads and writes) under standard network conditions while varying the ORAM size from

2^{16} to 2^{26}; the results of this experiment appear in Figures 8(a), 8(b), and 8(c). Supports claim (C3).

(E5): Compare the wall-clock time of 3P-DUORAM, FLORAM, and Circuit ORAM to do 128 read operations under standard network conditions while varying the ORAM size from 2^{16} to 2^{26}; the results of this experiment appear in Figure 9(a). Supports claim (C1).

(E6): Compare the wall-clock time of 3P-DUORAM, FLORAM, and Circuit ORAM to do 128 read operations under colocated network conditions while varying the ORAM size from 2^{16} to 2^{26}; the results of this experiment appear in Figure 9(b). Supports claim (C4).

(E7): Compare the bandwidth used by 3P-DUORAM, FLORAM, and Circuit ORAM to do 128 read operations while varying the ORAM size from 2^{16} to 2^{26}; the results of this experiment appear in Figure 9(c). Supports claim (C3).

To run all seven small experiments:

Preparation: `cd repro`

Execution: `./repro-all-dockers small` *`numops`*

Here, *`numops`* is the number of read, write, or interleaved operations to run in each experiment; the default of 128 is what we used in the paper. Using 128 will require about 10 hours of compute time.

Results: The above command will output the data (up to ORAM sizes of 2^{26}) for Figures 7(a), 7(b), 7(c), 8(a), 8(b), 8(c), 9(a), 9(b), and 9(c), clearly labeled and separated into the data for each line in each subfigure. Running `repro-all-dockers` multiple times will accumulate data, and means and standard deviations will be output for all data points when more than one run has been completed. From this data, one should be able to verify our major claims (though depending on your hardware, the exact numbers will surely vary).

The optional "large" experiments. These experiments do not directly support our major claims, but may optionally be run in case you are curious to see what happens at larger ORAM sizes. These experiments require at least 660 GB of available RAM, which is why they are optional.

(E8): Compare the wall-clock time of 3P-DUORAM, FLORAM, and Circuit ORAM to do 128 read operations under standard network conditions while varying the ORAM size from 2^{28} to 2^{32}; the results of this experiment appear in the rightmost three data points of Figure 9(a).

(E9): Compare the wall-clock time of 3P-DUORAM, FLORAM, and Circuit ORAM to do 128 read operations under colocated network conditions while varying the ORAM size from 2^{28} to 2^{32}; the results of this experiment appear in the rightmost three data points of Figure 9(b).

(E10): Compare the bandwidth used by 3P-DUORAM, FLORAM, and Circuit ORAM to do 128 read operations while varying the ORAM size from 2^{28} to 2^{32}; the results of this experiment appear in the rightmost three data points of Figure 9(c).

To run all three large experiments:

Preparation: `cd repro`

Execution: `./repro-all-dockers large` *`numops`*

Again, *`numops`* is the number of read operations to run in each experiment; the default of 128 is what we used in the paper. Using 128 will require about 40 hours of compute time. Lowering *`numops`* will reduce the time, but not the requirement for 660 GB of available RAM.

Results: The above command will output the data (for ORAM sizes from 2^{28} to 2^{32}) for Figures 9(a), 9(b), and 9(c), similar to the small experiments above.

The "scaling" experiment. This experiment varies the number of cores:

(E11): Compare the online wall-clock time of 2P-DUORAM, 3P-DUORAM, and FLORAM to do 128 read operations under standard network conditions while varying the number of available cores for each party from 4 to 32. The results of this experiment for ORAM sizes of 2^{16}, 2^{20}, and 2^{26} appear in Figures 10(a), 10(b), and 10(c) respectively. Supports claim (C5).

To run the scaling experiment:

Preparation: Reproducing Figure 10 (the effect of scaling the number of cores) is slightly more work because it depends more on your hardware configuration. First, `cd repro`. The top of the script `repro-scaling` in that directory has instructions. Set the variables (in the script, not environment variables) `BASE_DUORAM_NUMA_P0` and `BASE_DUORAM_NUMA_P1` to numactl commands (examples are given in the comments) to divide your system into two partitions as separate as possible: separate NUMA nodes if you have them, otherwise separate CPUs if you have them, otherwise separate cores. If each of your two partitions has n cores, ensure that the elements of `CORESLIST` do not exceed n (of course, you cannot replicate those data points in that case, but the trend should still be apparent). The paper uses values of n up to 32 cores in each partition, so 64 cores in total (P2 can reuse the cores of P0 since P2's primary work is done after P0 and P1's main work has finished).

Execution: `./repro-scaling` *`numops`*, where *`numops`* as before defaults to 128. Using 128 will require about 4 to 5 hours of compute time.

Results: The output will be similar to that described above with clearly labelled data for Figures 10(a), 10(b), and 10(c) (with an additional column for core count).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this Artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: A Peek into the Metaverse: Detecting 3D Model Clones in Mobile Games

Chaoshun Zuo
The Ohio State University

Chao Wang
The Ohio State University

Zhiqiang Lin
The Ohio State University

A Artifact Appendix

A.1 Abstract

The artifact is the source code of 3DSacn. A tool to detect model clones. In particular, it can extract 3D models from Android games, and compute a hash value for a 3D model which can be used to identify the clone ones.

A.2 Description & Requirements

Here we list the requirements to run it on Ubuntu 22.04.

- Wine
- Wine-mono
- python3
- SciPy

A.2.1 Security, privacy, and ethical concerns

It would extract the raw data for the 3D models in Android games. Those models are owned by the developer and please don't distribute them.

A.2.2 How to access

We are actively maintaining the code. We will make sure the latest version on the master branch is stable.
https://github.com/OSUSecLab/3DScan/releases/tag/ae

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

- Wine
- Wine-mono
- python3
- SciPy

A.2.5 Benchmarks

None.

A.3 Set-up

Use the standard way to install python3 and SciPy on Ubuntu 22.04.

A.3.1 Installation

Clone the repo.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: EnigMap: External-Memory Oblivious Map for Secure Enclaves

Afonso Tinoco[1,2] Sixiang Gao[1] Elaine Shi[1]

CMU[1], IST[2]

A Artifact Appendix

A.1 Abstract

These artifacts are meant to compliment the paper ENIGMAP: External-Memory Oblivious Map for Secure Enclaves. Our goal with the artifacts is to provide the motivation on why external memory algorithms are relevant for enclaves, our opensource implementation of ENIGMAP, experiments that allow to easily replicate the results in our paper, and details explanation on how the security goals are achieved by our implementation code. We additionally provide the signal and signal-ht code we used to benchmark signal original code. We provide all our experiment code as single line commands to make results simpler to reproduce, and the commands should be simple to modify to try new sets of experimental parameters. We also provide a section explaining at a high level how our implementation works. Our main goal with the experiments is not to reproduce the exact same numbers as we have in our graphs, as it will vary greatly with hardware used, but to show that the speedup against signal, as well as the assymptotic behavior is on the order of magnitude shown in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Tag usenix-artifacts in this repo: https://github.com/odslib/EnigMap/tree/usenix-artifacts

Listing 1: download artifacts

```
git clone \
https://github.com/odslib/EnigMap.git \
ARTIFACTS60
cd ARTIFACTS60/
git checkout usenix-artifacts
```

A.2.3 Hardware dependencies

In terms of infrastructure we require 3 types of machines to reproduce our code. Machine A is used mostly for benchmarking SGX, machine B is used to generate a single graph in our experiments, machine C is where most of our experiments were run.

(Machine A) - CPU with SGXv1, configured with at least 128MB of EPC and at least 16GB of RAM. In our experiments we used an intel E2200 processor.

(Machine B) - CPU with SGXv2, configured with 4GB EPC, at least 8GB RAM and an SSD available for storage. This is used mostly for benchmarking ocalls.

(Machine C) - CPU with SGXv2, configured with 192GB EPC, at least 256GB RAM and and SSD available for storage. This is used for most of the experiments. We have a server available with 512GB max EPC and 1TB RAM that can be used to reproduce the experiments for artifact evaluation. Please contact us if access is needed (send us a(n annonymized) public key and we will generate a user in our server)

A.2.4 Software dependencies

Our artifacts are meant to be run under docker, we provide the image use to build and run them under tools/docker/buildDockerImage.sh . Alternatively, we also provide a script to setup a vanila ubuntu 22.04 install (tools/docker/setup_sgx.sh) to run the artifacts.

A.2.5 Benchmarks

We ran baseline benchmarks on private contact discovery using signal and signal-ht (signal-icelake) code. We included them in our repo.

A.3 Set-up

A.3.1 Installation

First, build the docker image:

Listing 2: build docker images

```
#!/bin/bash
```

```
cd ARTIFACTS60/
cd tools/docker/
sh buildDockerImage.sh
```

Second, for every test, run docker at the top level of the repo, mounting the ssd for storage and with access to SGX (running with priviledged works for this):

Listing 3: start test environment

```
#!/bin/bash
cd ARTIFACTS60/
docker run -it --rm -m512G \
-v $PWD:/builder -v /mnt/ssd0:/mnt/ssd0 \
--privileged xtrm0/cppbuilder
source /startsgxenv.sh
```

We expect every experiment to be run inside this docker viroment, additionally some experiments modify configuration files, we assume for every experiment that the configuration files start as they are in the artifact code. We note that some modifications to the config might require deleting the build directory and running cmake again.

A.3.2 Basic Test

We include 3 tests here.
(T1) Make sure the code compiles:

Listing 4: compile the code

```
cmake -B build -G Ninja
ninja -C build
```

(T2) Make sure the tests run:

Listing 5: run unit tests on the code

```
ninja -C build test
```

(T3) Make sure the test enclave compiles and runs in both debug and release mode:

Listing 6: compile and run a test enclave

```
cd applications/benchmark_sgx
make clean
make
./benchmark_sgx.elf
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./benchmark_sgx.elf
```

A.4 Evaluation workflow

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] This section should include all the operational steps and experiments which must be performed to evaluate if your your artifact is functional and to validate your paper's key results and claims. For that purpose, we ask you to use the two following subsections and cross-reference the items therein as explained next.

A.4.1 Major Claims

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Enumerate here the major claims (Cx) made in your paper. Follows an example:

(C1): EnigMap achieves assymptotically speedup on point and batched search over signal's implementation, having speedups at any batch size at a realistic database size of 256 million entries. Our main claim is that switching to EnigMap always provides speedup over signal's original implementation. Additionally, we also claim that for external memory, EnigMap's implementation is assymptotically as described in table 1 in our paper, and concretely efficient.
Experiment (E2) shows the speedup for the database in-RAM case, experiment (E3) shows this claim for the disk eviction case, whose results are ilustrated for SGX1 and SGX2 respectively in figures 3 and 4 in our paper. These two experiments prove the C1 claim.
Further more, we show a detailed view of the costs of external memory and how our optimizations affect them in experiment (E4), reflected in figures 5 and 6 in our paper.
We also show the cost of insertion in experiment, reflected in figure 8 in our paper.

(C2): There is an inherent cost to using EWB that can be reduced with the OCALL approach described. EnigMap's ORAM tree and the OCALL eviction approach provide a way to implicitly store the nonces of evicted pages without any computational overhead compared to EWB, thus saving the space of the evicted nonce table for more EPC pages. We show the inherent costs of EWB and OCALL in experiment (E1), reflected in figure 1 in our paper. We also analyse the effect the of page size for search in experiment (E7) and conclude that roughly 4k is the optimal size for ORAM, reflected in figure 9 in our paper.

(C3): Our faster initialization algorithm is inherently faster than the naive approach of doing sequential insertions. Experiment (E5), reflected in figure 9 in our paper shows these results.

(C4): Our implementation of OBST::Get, OBST::Insert are oblivious. We don't have any experiment for this, but we encorage to look at both the code and the generated code for otree and concluding that all branches inside of otree.hpp::OBST::Get and otree.hpp::OBST::Insert do not depend on private data.

A.4.2 Experiments

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Link explicitly the description of your experiments to the items you have provided in the previous subsection about Major Claims. Please provide your

estimates of human- and compute-time for each of the listed experiments (using the suggested hardware/software configuration above). Follows an example: Most of our experiments are run accross exponentially increasing database sizes from a few bytes to 1TB in size. We include both the time it takes to run the experiments to reach the main conclusions above, as well as the time to fully reproduce the graphs in our paper. We also include the machines required to run each experiment.

(E1.1): *[Benchmark SGX] [20 human minutes + 15 compute-minutes] [Machine A or B or C]* The main goal of this experiment is to show the costs of ocall and and EWB in our paper. We defer computing the cost of EWB to experiment E1.2, altough we analyse it here for clarity.

How to: To run this experiment, compile and run applications/benchmark_sgx . These will show different results based on the processor being used. For SGX1 please only use machine A, for SGX2 use machine B or C. Our graphs in the paper used machine B.

Preparation: Pick a machine, clone the repo and go to the folder applications/benchmark_sgx.

Execution: Compile and run the code:

Listing 7: compile and run a test enclave

```
cd applications/benchmark_sgx
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./benchmark.elf > output.txt
```

Copy paste the values from the benchmark at 4096 bytes to the definition of the data_v2 array at tools/-plot_intrinsics_v2.py. The value for EWB should come from (E1.2). Also update the iops based on the specifications of the SSD being used. Run the script, it should generate a png file with figura 1a.

We generated figure 1b in excel by varying the number of bytes in the encryption and ecall inside of applications/benchmark_sgx/bm.edl.

Results: For the first graph, we expect to see results similar to figure 1a in our paper. For the excel graph, similar to figure 1b.

(E1.2): *[Benchmark SGX] [4 human minutes + 20 compute-minutes] [Machine A or B]* The goal of this experiment is to measure EWB time in a generic SGX machine. It envolves picking an EPC larger than the maximum EPC but smaller than the total RAM, and doing sequential reads and writes to force continuos EWB evictions, and measureing the time based on that (see the function ecall_bm_ewb)

How to: To run this experiment, compile and run applications/benchmark_ewb . These will show different results based on the processor being used. For SGX1 please only use machine A, for SGX2 use machine B. Our graphs in the paper used machine B.

Preparation: Pick a machine, clone the repo and go to the folder applications/benchmark_ewb. Adjust the size of maximum EPC (HeapInitSize and HeapMaxSize) in Enclave.config.xml to be larger than the maximum EPC size on the machine, also adjust the variable ARR_SZ in Enclave/TL/Libcxx.cpp to an appropriate value, so that evictions will occur. The default parameters are all already set for Machine B. We run experiments with different sizes to make it explicit what the page size is.

Execution: Compile and run the code:

Listing 8: compile and run a test enclave

```
cd applications/benchmark_ewb
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./benchmark.elf > output.txt
```

The output should be the total execution time. Divide it by ARR_SZ/4096 to get the cost per page to use in (E1.1).

Results: For the first graph, we expect to see results similar to figure 1a in our paper. For the excel graph, similar to figure 1b.

(E2): *[Point query search inram] [30 human-minutes + 30/120 compute-minutes] [Machine C]:* This experiment computes the cost of point query for a scenario that doesn't need to use disk but needs to do ocalls to in RAM storage. We will explain in here thoroughly how to configure parameters that are further used in other experiments. ODS/common/defs.hpp contains most of the configurable parameters. Take a look at it to see the definitions used and their meanings, configuring experiments is mostly changing this file, as well as changing the Enclave.config.xml used by the experiment. This experiment uses applications/signal.

How to: Go to applications/signal.

Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. Additionally set ORAM_USE_INRAM_SERVER to true. Additionally set TEST_SELECTOR to 0 (search).

Execution: compile and run the enclave:

Listing 9: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_search.py

Results: This will plot graphs similar to figure 3 and 4 in the paper, but not the same, as those graphs use E3.

(E3): *[Point query search with disk] [15 human-minutes + 30/120 compute-minutes] [Machine C]:* This experiment

computes the cost of point query for a scenario that needs to use disk via ocalls.
How to: Go to applications/signal.
Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. set ORAM_USE_INRAM_SERVER to false. Set TEST_SELECTOR to 0 (search).
Execution: compile and run the enclave:

Listing 10: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_search.py
Results: This will plot graphs similar to figure 3 and 4 in the paper. Results should be similar if the same EPC size and maximum RAM were used.

(E4): *[Cost of insertion] [30 human-minutes + 30/120 compute-minutes] [Machine C]:* This shows the cost of insertion queries.
How to: Go to applications/signal.
Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. Additionally set ORAM_USE_INRAM_SERVER to true. Additionally set TEST_SELECTOR to 1 (insertion).
Execution: Compile and run the enclave:

Listing 11: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_insertion.py
The results of this experiment are used to compute the cost in E5 for the naive insertion.
Results: This will plot graphs similar to figure 8 in the paper.

(E5): *[Cost of initialization] [30 human-minutes + 120 compute-minutes/ 24 compute-hours] [Machine C]:* This shows the cost of initialization queries.
How to: Go to applications/signal.
Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. Additionally set ORAM_USE_INRAM_SERVER to true. Additionally set TEST_SELECTOR to 2 (initialization).
Execution: Compile and run the enclave:

Listing 12: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_initialization.py
Use the results of E4 to get the costs for naive initialization (cost of 1 insertion * number of elements in the database)
Results: This will plot graphs similar to figure 7 in the paper. We can see that the fast initialization has a speedup over the naive initialization.

(E6): *[Optimization breakdown] [45 human-minutes + 20 compute-minutes] [Machine B]:* This experiment analyses an execution trace of the code to generate figures 5 and 6.
How to: This uses ods outside of enclave to generate flame graphs of the execution profile.
Preparation: Edit buildtype.cmake to change the build type to debug.
Execution: Compile and run the proiling tests:

Listing 13: compile and run a test enclave

```
rm -rf build
cmake -B build -G Ninja
ninja -C build
./build/tests/improv*_none
mv ./q*/f*/flame*-chrome.json \
./q*/f*/improv*_none.json
./build/tests/improv*_packing
mv ./q*/f*/flame*-chrome.json \
  ./q*/f*/improv*_packing.json
./build/tests/improv*_filecache
mv ./q*/f*/flame*-chrome.json \
  ./q*/f*/improv*_filecache.json
./build/tests/improv*_bucketcache
mv ./q*/f*/flame*-chrome.json \
  ./q*/f*/improv*_bucketcache.json
./build/tests/profiling_test
```

Open the flamegraphs in chrome and analyse the total time in each phase. To plot the graphs, one can use plot_relative_performance.py or plot_relative_performance_large.py.
With the default config, we get the plot in figure 5. Run the experiment again with a larger database size (edit the respective .cpp files for the tests) to get the case for figure 6.

Results: This will plot graphs similar to figure 5 and 6 in the paper. We can see the effect of different optimizations.

(E7): *[Optimal page size] [30 human-minutes + 30 compute-minutes] [Machine B or C]:* This experiment is meant to analyse what is the optimal page size used for insertion. We used machine B in our experiments to enforce disk swap at smaller sizes, but this can also be ran on machine C.

How to: Go to applications/signal.

Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. Additionally set ORAM_USE_INRAM_SERVER to true. Additionally set TEST_SELECTOR to 0 (search). For each experiment, configure ORAM_SERVER__LEVELS_PER_PACK to different values (1 corresponds to 296B page, 2 to 824B, 3 to 1880B, 4 to 3993B and 5 to 8216 in our plot).

Execution: Compile and run the enclave:

Listing 14: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_search_pagesize.py

Results: This will plot graphs similar to figure 9 in the paper, we can see that the optimal page size is 4kb.

In all of the above blocks, please provide indications about the expected outcome for each of the steps (given the suggested hardware/software configuration above).

A.5 Notes on Reusability

We provide our code as an opensource Oblivious Data Structure Library project on https://github.com/odslib/EnigMap. We are actively developing it, both in order to do further reserach in oblivious algorithms as well as in order to provide a way for developers to incorporate oblivious algorithms into enclave code. We made our code to be easy integrable into enclave applications. The example inside the folder applications/signal is meant to be easily integrable as a binary search tree or map into any SGX project.

We hope that our artifacts can be used by industry to provide fast private contact discovery in messaging applications such as signal. We also hope that the code in EnigMap can be further improved and serve as baseline for further research in oblivious algorithms, all the oblivious primitives we developed in EnigMap can easily be integrated into other enclave or oblivious algorithms projects.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Controlled Data Races in Enclaves: Attacks and Detection

Sanchuan Chen
Fordham University
schen409@fordham.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Yinqian Zhang
Southern University of Science and Technology
yinqianz@acm.org

A Artifact Appendix

A.1 Abstract

The artifact SGXRACER is a controlled data race detection tool analyzing Intel SGX enclave binary code. It is implemented atop `angr` binary code analysis tool. SGXRACER performs static analysis, particularly data flow analysis, to detect shared variables and lock variables in binary code, and then use a lockset based algorithm to detect data races. To evaluate SGXRACER, we have tested four well-known SGX SDKs and eight widely-used SGX applications. We have open-sourced SGXRACER on GitHub.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Any discovered 0-day vulnerability should be reported to its software vendor and vulnerability databases such as NVD.

A.2.2 How to access

SGXRACER can be accessed via the following GitHub repository: https://github.com/OSUSecLab/SGXRacer.

A.2.3 Hardware dependencies

The suggested hardware configuration is an x86-64 PC with eight Intel Core i7-7700 processors and 32GB memory or better.

A.2.4 Software dependencies

SGXRACER was originally developed and tested on Ubuntu 20.04. SGXRACER requires Python 3 environment, including command line tool `python3` and `pip3`. SGXRACER also requires Python 3 package `angr`.

A.2.5 Benchmarks

(B1:) SDK binaries: In our repository, we have provided our pre-built binary code for four well-known SGX SDKs: Intel SGX SDK, Microsoft Open Enclave SDK, Apache Teaclave Rust-SGX SDK, and Fortanix Rust EDP SDK.

(B2:) Application binaries: In our repository, we have also provided our pre-built binary code for eight widely used SGX Applications: mbedtls-SGX, intel-sgx-ssl, TaLoS, LibSEAL, SGX_SQLite, stealthdb, SGXDeep, and hotcalls.

A.3 Set-up

A.3.1 Installation

(1) Set up an OS environment: Ubuntu 20.04. (2) Install pip3 for Python 3:

```
sudo apt install python3-pip
```

(3) Install binary code analysis framework angr:

```
sudo pip3 install angr
```

(4) Clone `SGXRacer` GitHub repository:

```
git clone https://github.com/OSUSecLab/SGXRacer.git
```

(5) Read the README.md file for `SGXRacer` tool description and usage details.

A.3.2 Basic Test

Run the following command detects the controlled data races in Intel SGX SDK and will test the basic functionality of all software components:

```
python3 sgxrace.py -input \
./enclave_binaries/intel_sgx_sdk/enclave.signed.so  \
-output intel_sgx_sdk_results.txt \
-output1 intel_sgx_sdk_results1.txt \
> intel_sgx_sdk_stdout
```

This command may take minutes to execute. Please ignore warnings. The command should execute successfully without any exception. It will output three files:

```
intel_sgx_sdk_results.txt
intel_sgx_sdk_results1.txt
intel_sgx_sdk_stdout
```

The content of these files should match our corresponding same name result files in the folder `results/sdk_results`. `intel_sgx_sdk_results.txt` is detailed detection results. `intel_sgx_sdk_results1.txt` is concise version detection results. `intel_sgx_sdk_stdout` is detection statistics.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1:) SGXRACER has been used in detecting controlled data race vulnerabilities in four well-known SGX SDKs: Intel SGX SDK, Microsoft Open Enclave SDK, Apache Teaclave Rust-SGX SDK, and Fortanix Rust EDP SDK. This is proven by the experiment (E1) described in Section 6 whose results are reported in Table 2 in our paper.

(C2:) SGXRACER has been used in detecting controlled data race vulnerabilities in eight widely used SGX Applications: mbedtls-SGX, intel-sgx-ssl, TaLoS, LibSEAL, SGX_SQLite, stealthdb, SGXDeep, and hot-calls. This is proven by the experiment (E2) described in Section 6 whose results are reported in Table 3 in our paper.

A.4.2 Experiments

(E1): SGX SDK Data Race Detection [60 human-minutes + 20 compute-hour + 5GB disk]:
How to: Run the commands in README.md file evaluation part 1: *To detect data races in SGX SDKs*.
Results: Each command will generate three files similar to the ones in basic test:

```
xxx_sdk_results.txt
xxx_sdk_results1.txt
xxx_sdk_stdout
```

The content of these files should match our corresponding same name result files in the folder `results/sdk_results`. `xxx_sdk_results.txt` is detailed detection results. `xxx_sdk_results1.txt` is concise version detection results. `xxx_sdk_stdout` is detection statistics.

Table 2 Variables Part: Please refer to the detection statistics file `xxx_sdk_stdout`.
Example: In file intel_sgx_sdk_stdout:

```
sv_r_count: 317
sv_w_count: 119
sv_rw_count: 6
len(info.gv_reverse_map): 143
```

`sv_r_count` is *# Shared Var. Access (R)*. `sv_w_count` is *# Shared Var. Access (W)*. `sv_rw_count` is *# Shared Var. Access (R&W)*. `len(info.gv_reverse_map)` is *# Uniq. Shared Var.*.
Example: In file intel_sgx_sdk_stdout:

```
mutex_count: 7
spin_count: 53
once_count: 0
unique_locks: 9
```

`mutex_count` is *# Lock Var. Access (Mutex)*.
`spin_count` is *# Lock Var. Access (Spinlock)*.
`once_count` is *# Lock Var. Access (Others)*.
`unique_locks` is *# Uniq. Lock Var.*.

Table 2 Lockset and Acquisition History Part: Please refer to the data race detection statistics file `xxx_sdk_stdout`.
Example: In file intel_sgx_sdk_stdout:

```
max_lockset_size: 2
min_lockset_size: 0
average_lockset_size: 0.46113479324725687
max_history_size: 8
min_history_size: 0
average_history_size: 3.3398070205136885
```

`max_lockset_size` is *Ins. Lockset Size (Max.)*.
`min_lockset_size` is *Ins. Lockset Size (Min.)*.
`average_lockset_size` is *Ins. Lockset Size (Ave.)*.
`max_history_size` is *Acquisition History Size (Max.)*.
`min_history_size` is *Acquisition History Size (Min.)*.
`average_history_size` is *Acquisition History Size (Ave.)*.

Table 2 Var. and Func. Distribution (On table right): The variable and function distribution are identified by manually inspecting SDK source code, which are in folder `enclave_source`. The result of variable and function distribution can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_sdk_lib_distribution`.

Table 2 Performance Part: Please refer to the concise detection results file `xxx_sdk_results1.txt` and statistics file `xxx_sdk_stdout`.
Example: In file intel_sgx_sdk_results1.txt:

```
ULx86_64_init_done*_ULx86_64_init*unw_init_local_common
```

Each line is a detected data race, which is separated by * into three parts: The first part is the shared variable name in the race, the second part is the function name in the first thread, and the third part is the function name in the second thread. By counting the number of lines in this file and checking the unique shared varaible names, we can get `# Shared Variables` and `# Data Races`. The false positives are identified by manually inspecting SDK source code, which is in folder `enclave_source`. The result of our false positive analysis can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_sdk_fp` (false positives highlighted).
Example: In file intel_sgx_sdk_stdout:

```
potential racing pairs: 1567
...
phase 1 time:
0.16607975959777832
phase 2 time:
19.74062967300415
```

`potential racing pairs` is *Shared Variable Access Pairs*. `phase 1 time` is *Variable Analysis Time (m)*. `phase 2 time` is *Data Race Detection Time (m)*. The

sum of `phase 1 time` and `phase 2 time` is *Total Time (m)*. Note the phase 1 and phase 2 are using cached file in our repository thus may not reflect the real analysis time. Also note that time may vary due to different software and hardware configuration and work load on the machine.

Heap variables (reported in Section 6.1.2): The heap variable allocations are identified by manually inspecting SDK source code and find out unique heap allocation sites, which are in folder `enclave_source`. The result of heap variable allocations can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_sdk_heap`.

(E2): SGX Application Data Race Detection [60 human-minutes + 10 compute-hour + 5GB disk]:

How to: Run the commands in README.md file evaluation part 2: *To detect data races in SGX applications*.

Results: Each command will generate three files similar to the ones in basic test:

```
xxx_results.txt
xxx_results1.txt
xxx_stdout
```

The content of these files should match our corresponding same name result files in the folder `results/app_results`. `xxx_results.txt` is detailed detection results. `xxx_results1.txt` is concise version detection results. `xxx_stdout` is detection statistics.

Table 3 Detected Data Races Part: Please refer to the concise detection results file `xxx_results1.txt` and statistics file `xxx_stdout`.

Example:

In file 001_mbedtls-SGX_results1.txt:

```
add_count*ecp_add_mixed*ecp_add_mixed
```

Each line is similarly divided by * as in SDK cases. By counting the number of lines in this file and checking the unique shared varaible names, we can get the number of shared variables in the detected data races `Var.` and the number of detected data races `Races`. The false positives are identified by manually inspecting application source code, which are in folder `enclave_source`. The result of our false positive analysis can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_fp` (false positives highlighted).

Example: In file 001_mbedtls-SGX_stdout:

```
potential racing pairs: 7817
potential racing interleavings: 15317
```

`potential racing pairs` is *Acc. Pairs*.
`potential racing interleavings` is *# Total Inter.*.

Table 3 Variables Part: Please refer to the detection statistics file `xxx_stdout`.

Example: In file 001_mbedtls-SGX_stdout:

```
sv_r_count: 234
sv_w_count: 138
sv_rw_count: 0
len(info.gv_reverse_map): 84
```

`sv_r_count` is count of shared variable accesses (R).
`sv_w_count` is count of shared variable accesses (W).
`sv_rw_count` is count of accesses (R&W).
The sum of these three numbers is *# Shared Var. Access*.
`len(info.gv_reverse_map)` is the number of unique shared variables, *i.e.*, *# Var.*.

Example: In file intel_sgx_sdk_stdout:

```
mutex_count: 2
spin_count: 66
once_count: 0
unique_locks: 3
```

`mutex_count` is count of lock variable accesses (mutex).
`spin_count` is count of accesses (spinlock).
`once_count` is count of lock variable accesses (others).
The sum of these three numbers is *# Lock Var. Access*.
`unique_locks` is the number of unique lock variables, *i.e.*, *# Lock Var.*.

Example: In file intel_sgx_sdk_stdout:

```
average_lockset_size: 0.1978053439017108
...
average_history_size: 9.270506565018288e-05
```

`average_lockset_size` is *Ave. Lockset*.
`average_history_size` is *Ave. Acq. History*.

Table 3 Performance Part: Please refer to statistics file `xxx_stdout`.

Example: In file 001_mbedtls-SGX_stdout:

```
phase 1 time:
0.27547693252563477
phase 2 time:
307.8826234340668
```

`phase 1 time` is *Variable Ana. (m)*. `phase 2 time` is *Race Det. (m)*. The sum of `phase 1 time` and `phase 2 time` is *Total Time (m)*. Note the phase 1 and phase 2 is using cached file in our repository thus may not reflect the real analysis time. Also note that time may vary due to different software and hardware configuration and work load on the machine.

Heap variables (reported in Section 6.1.2): The heap variable allocations are identified by manually inspecting application source code and find out unique heap allocation sites, which are in folder `enclave_source`. The result of heap variable allocations can be find in `results/result_cal.xlsx` file corresponding tab: `xxx_heap`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Guarding Serverless Applications with Kalium

Deepak Sirone Jegan
University of Wisconsin-Madison
dsirone@cs.wisc.edu

Liang Wang
Princeton University
lw19@princeton.edu

Siddhant Bhagat
Microsoft
sbhagat3@wisc.edu

Michael Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

A Artifact Appendix

This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of Kalium.

A.1 Abstract

As an emerging application paradigm, serverless computing attracts attention from more and more adversaries. Unfortunately, security tools for conventional web applications cannot be easily ported to serverless computing due to its distributed nature, and existing serverless security solutions focus on enforcing user specified information flow policies which are unable to detect the manipulation of the order of functions in application control flow paths. In this paper, we present *Kalium*, an extensible security framework that leverages local function state and global application state to enforce control-flow integrity (CFI) in serverless applications. We evaluate the performance overhead and security of Kalium using realistic open-source applications; our results show that Kalium mitigates several classes of attacks with relatively low performance overhead and outperforms the state-of-the-art serverless information flow protection systems.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Kubernetes installation requires root privileges on all the Kubernetes nodes and certain ports need to be exposed for the OpenFaas installation. The Kalium controller node requires a TLS issued by Let's Encrypt. All the nodes need to be able to communicate with each other through a DNS name (possibly public to the Internet).

A.2.2 How to access

The artifact including its sub-repositories can be found here: https://github.com/multifacet/kalium_artifact/tree/83110fcfd091d9f8bd164007b1570742e0ad107c

A.2.3 Hardware dependencies

Our testbed uses machines with an Intel Xeon E5-2630 2.40GHz CPU and 64 GB RAM on CloudLab. Each machine is connected to a star topology LAN network with a speed of 25 Gbps. The Kalium controller runs on a separate identical node outside the LAN but on the same datacenter.

Any machines of comparable specifications, connected in a LAN of uniform speed maybe used for the Kubernetes nodes. The Kalium controller should run on the same datacenter to minimize noise as opposed to running it elsewhere on the internet.

All the machines need to be addressable from each other with a hostname. The Kalium controller doubles up as an image server that needs a certificate issued by Let's Encrypt, this mandates that the hostname of the Kalium controller should be visible to the internet.

A.2.4 Software dependencies

All the Kubernetes nodes and the Kalium controller have been testing using Ubuntu 18.04 LTS. We highly recommend using Ubuntu 18.04 LTS for all machines.

The build machine needs docker installed for the build process. Please install docker as per the build machine's distro's instructions https://docs.docker.com/engine/install/. Please do not install docker on any of the Kubernetes nodes or the Kalium controller.

A.2.5 Benchmarks

All the required benchmarks and data is packaged in the artifact.

A.3 Set-Up

[Mandatory] This section should include all the installation and configuration steps required to prepare the environment to be used for the evaluation of your artifact.

Provision 5 machines for running Kubernetes and 1 machine for the Kalium controller as specified in A.2.3 and A.2.4. A possible configuration is:

- node0: Kubernetes Worker
- node1: Kubernetes Worker
- node2: Kubernetes Worker
- node3: Kubernetes Worker
- node4: Kubernetes Controller
- node5: Kalium Controller

Root access is assumed on all the Kubernetes node as well as the Kalium controller node. Kubernetes requires port 6443 for the service API exposed on all the nodes while OpenFaas needs port 31112 for the gateway. The Kalium Controller listens at port 5000 and needs to be exposed to all the Kubernetes nodes. The Image Server listens at port 4443 and needs to be exposed to all the Kubernetes nodes.

Provision a machine for building the artifact binaries. This system should *not* be one of the Kubernetes nodes or the Kalium controller. Clone the artifact repository into the build machine using `git clone https://github.com/multifacet/kalium_artifact && cd kalium_artifact && git submodule update -init -recursive && git checkout 83110fcfd091d9f8bd164007b1570742e0ad107c`.

Please follow `kalium-benchmarks/README.md` to obtain a TLS certificate issued by Let's Encrypt for the Kalium Controller node.

A.3.1 Installation

Please follow the steps in README.md except "Running Benchmarks". By this time, the build machine should have a `build/bin` folder that contains the various artifact binaries, Kubernetes and OpenFaas should be setup in the cluster.

A.3.2 Basic Test

Please refer to README.md for detailed steps to do a basic test.

A.4 Evaluation workflow

A.4.1 Major Claims

The main claims validated by this artifact are related to the main claim of Kalium achieving comparable performance to Valve and Trapeze and being a usable solution due to its low system call overhead. As noted in Section 5.1.1, the semi-automated policy generation from existing applications is not a major contribution of the paper as a lot of the analysis was done manually. Our paper mainly focuses on defining control flow integrity for serverless applications, its challenges and enforcing the same with low overhead.

(C1): *Kalium achieves comparable performance as the state of the art information flow systems Valve and Trapeze. This is proven by the relative latency overhead experiment described in Section 7.4 of the paper whose results are illustrated in Figure 8.*

(C2): *Kalium achieves tolerable per system call overhead of the order of a few milliseconds in the worst case. This is proven by the per-syscall measurements in Section 7.4*

A.4.2 Experiments

All the experiments have been described in detail in the README.md file provided with the kalium-benchmarks sub repository. We omit repeating all the steps here for brevity.

(E1): *Valve Benchmarks: Run the Valve Benchmarks with stock gVisor and Kalium to generate Figure 8 in the paper. The figure shows the relative overheads of Kalium, Valve and Trapeze with respect to stock gVisor baseline. This validates claim C1*

How to: Run steps 1-3 in kalium-benchmarks/README.md

Preparation: Run steps 1-2 kalium-benchmarks/README.md

Execution: Run step 3 in kalium-benchmarks/README.md

Results: Run step 5 in kalium-benchmarks/README.md to generate Figure 8. The graph should show that Kalium has comparable overhead as Valve and Trapeze.

(E2): *Per Syscall Overhead: Run a microbenchmark function to generate per system-call overheads in Kalium. This validates claim C2*

How to: Run steps 4 and 7 in kalium-benchmarks/README.md

Preparation: Run steps 1-2 kalium-benchmarks/README.md only if it has not been run yet

Execution: Run step 4 in kalium-benchmarks/README.md

Results: Run step 7 in kalium-benchmarks/README.md to print out the per-syscall (SendMsg and Write) overheads which include (i) parsing TLS records (ii) TLS record cache lookup (iii) Event construction time (iv) Guard Local Graph Lookup (v) Controller Query Total Time and (vi) Total Syscall Overheads. The overheads should be comparable to that in Section 7.4 in the paper. The total syscall overhead

should be of the order of a few milliseconds in the worst case.

USENIX'23 Artifact Appendix: <PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel>

Zicheng Wang*
wzc@smail.nju.edu.cn
Nanjing University

Yueqi Chen
yueqi.chen@colorado.edu
University of Colorado Boulder

Qingkai Zeng
zqk@nju.edu.cn
Nanjing University

A Artifact Appendix

A.1 Abstract

This artifact is applying for an **Artifacts Available** badge, an **Artifacts Functional** badge, and an **Results Reproduced** badge. It provides two main artifact sets for evaluators to reproduce PET. The first artifact set, detailed in Github, enables constructing PET from scratch, and the second artifact set includes kernel images and a root filesystem which allow the evaluator to reproduce our results in an isolated environment without any destructive steps.

Both artifact sets include Proof of Concept (PoC) programs and exploits of vulnerabilities used as test cases and eBPF programs that enable PET protection. After installing the eBPF program, the evaluator can execute the PoC programs and exploits to access the effectiveness of PET, by observing that the error triggering is prevented.

Besides we include user guidance in the first artifact set to help readers understand the design of PET, and develop their own eBPF programs for more error types that have not been covered in PET so far.

In this appendix, we will provide necessary instructions for evaluators to reproduce PET as well as an example along with screenshots for illustration.

A.2 Description & Requirements

In this section, we first describe whether reproducing our artifacts will risk the evaluator's machine security, followed by approaches to accessing our artifacts. Then, we describe hardware dependencies and software dependencies before listing the benchmarks.

A.2.1 Security, privacy, and ethical concerns

PET aims to protect the OS kernel through the eBPF ecosystem. To enable PET, the kernel needs additional eBPF helper functions before being compiled and installed, which is destructive to some extent. Therefore, to make evaluators feel safe, we prepared a kernel image and a root filesystem for evaluators. As such, evaluators can download the image and reproduce our results in an isolated environment, ensuring the safety and privacy of the host machine. The access for the image can be found in § A.2.2.

Furthermore, it is important to note that all vulnerabilities and proof-of-concept programs included in the artifact are publicly available and have been addressed in the mainstream kernel. Therefore, there are no security, privacy, or ethical concerns regarding the open-source community. The artifact builds upon resolved issues, and its purpose is to contribute to the knowledge and advancement of the field.

A.2.2 How to access

The complete artifacts are available in a public Github Repo https://github.com/purplewall1206/PET, which includes three main components: eBPF programs and corresponding scripts for evaluation, source code developed in PET, manuals and examples for evaluators to quickly understand the key idea of PET. Due to the space limit, we cannot list all details from building kernel to compiling eBPF programs to enable PET protections. Therefore, the repo also includes elaborate instructions for evaluators to follow.

The artifacts provided in the Github Repo are sufficient for evaluators to reproduce. However, as we mentioned in § A.2.1, the reproducing procedure includes destructive steps. To this end, we additionally provide a root filesystem and a compiled kernel image, which are in https://tinyurl.com/2428uac5.

A.2.3 Hardware dependencies

To completely reproduce PET, we recommend the following minimum hardware configurations: ❶ an Intel CPU with VT-X virtualization feature, ❷ 8GB or larger memory, and ❸ at least 100GB disk space.

A.2.4 Software dependencies

It is preferable to perform the evaluation on the Ubuntu Linux distro, especially the 20.04 desktop which is the same OS for PET development. The OS is supposed to include essential packages such as debootstrap, qemu-system-x86_64, open-ssh,

*The work was done while visiting the University of Colorado Boulder.

and `wget`. These packages are necessary for setting up the evaluation environment and conducting runtime evaluations.

Besides, it is advised **not** to utilize Docker for the evaluation process because the artifact necessitates the use of two separate terminals - one for executing the Proof of Concept programs and another for displaying the output of the eBPF programs.

A.2.5 Benchmarks

The Proof of Concept programs for vulnerabilities used as test cases have been collected and provided in the Github Repo. We used Phoronix-benchmark for performance measurement which is publicly available online.

A.3 Set-up

In this section, we focus on the installation and testing of `PET` using the kernel image and a root filesystem we provided for the sake of ethics (§ A.2.2). The evaluator can boot up the kernel using `QEMU`. Due to the space limit, we move the detailed instruction for reproducing `PET` from scratch in the Github Repo.

A.3.1 Functional

For the functional evaluations, we have implemented a `evaluate.sh` script to set up the environments. including: ❶ use `apt` to install required software mentioned in A.2.4, ❷ pop up 2 terminals, *terminal 1* start the virtual machine, and *terminal 2* connect to the virtual machine with `ssh`, and ❸ copy the test scripts into virtual machine.

A.3.2 Reproduce

For the reproducible evaluations, we first present a `phoro-run.sh` script to reproduce all performance results. We also present an instruction for generating new BPF prevention programs from scratch and evaluate it in the also in the functional testing environment.

A.3.3 Installation

None.

A.3.4 Basic Test

After evaluators pull the github repository and run `evaluate.sh`, there will be 2 terminals pop up. In figure 1, the left *terminal 1* boots up the virtual machine, and waits to be logged in, and the right *terminal 2* has already logged in through `ssh`. Evaluators can login the virtual machine with the user name `root` and no password is needed. After that, evaluators can run `ls` command on either terminal, and there will be three directories, including `bpf`, `PoCs`, `scripts`.

The `bpf` directory contains all compiled BPF prevention programs. The `POCs` directory contains all compiled proof-of-concepts programs that can trigger the vulnerabilities. The `scripts` directory contains evaluation scripts that need evaluator to execute in the virtual machine.

A.4 Evaluation workflow

As described in Section 5 (Error-dependent Prevention Policies) of our paper, the PET framework provides support for preventing five distinct types of errors from being triggered. To evaluate the functional of each type of kernel error prevention, the artifact includes five BPF prevention programs.

During the evaluation process, the virtual machine will initiate the execution of these five BPF protection programs immediately after boot-up. Subsequently, the evaluator can proceed to run the proof-of-concept tests for the corresponding five vulnerabilities. The BPF prevention programs are designed to intercept and bypass the error-prone sections within the kernel. As a result, the system will continue to function smoothly, ensuring the stability and integrity of its operations.

After the functional evaluation, evaluators can also reproduce the performance overhead, and try to add new BPF programs to prevent the other vulnerabilities from being triggered.

A.4.1 Major Claims

C1: The 5 types of kernel errors (integer overflow, out-of-bound, use-after-free, uninitialization, data race) will be prevented. **C2:** The system will keep functioning after the errors are prevented from being triggered.

A.4.2 Experiments

Functional: First of all, evaluators need to change directories to `/root/scripts` in both terminals, then execute `start-bpf-progs.sh` to start all 5 BPF programs in the *terminal 2*. After that, output of the 5 BPF programs will be printed in the *terminal 2*. We have also prepared a video to demonstrate the evaluation workflow on Youtube `https://www.youtube.com/watch?v=0BVsULXT0xI`.

(E1): Test if the BPF program can prevent an integer overflow vulnerability CVE-2017-7184 from being triggered.
Execution: execute `test-CVE-2017-7184.sh` in *terminal 1*
Results: There will be `killed` signal in *terminal 1*, and there will be a report `====CVE-2017-7184 is happened====` in *terminal 2*.

(E2): Test if the BPF program can prevent an out-of-bound vulnerability CVE-2016-6187 from being triggered.
Execution: execute `test-CVE-2016-6187.sh` in *terminal 1*
Results: There will be `killed` signal in *terminal 1*, and there will be a report `====CVE-2016-6187 is happened====` in *terminal 2*.

```
terminal1: RUN VM & POC here
[    2.060089] random: 7 urandom warning(s) missed due to ratelimiting
[  OK  ] Finished Load/Save Random Seed.
[  OK  ] Finished Raise network interfaces.
[  OK  ] Reached target Network.
         Starting OpenBSD Secure Shell server...
         Starting Permit User Sessions...
[  OK  ] Finished Permit User Sessions.
[  OK  ] Started Getty on tty1.
[  OK  ] Started Getty on tty2.
[  OK  ] Started Getty on tty3.
[  OK  ] Started Getty on tty4.
[  OK  ] Started Getty on tty5.
[  OK  ] Started Getty on tty6.
[  OK  ] Started Serial Getty on ttyS0.
[  OK  ] Reached target Login Prompts.
[  OK  ] Started OpenBSD Secure Shell server.
[  OK  ] Reached target Multi-User System.
[  OK  ] Reached target Graphical Interface.
         Starting Update UTMP about System Runlevel Changes...
[  OK  ] Finished Update UTMP about System Runlevel Changes.

Debian GNU/Linux 11 syzkaller ttyS0

syzkaller login:
```

```
terminal2: RUN BPF & MONITOR OUTPUT here
Warning: Permanently added '[localhost]:10021' (ED25519) to the list of known ho
sts.
Linux syzkaller 5.15.0ebpf-detector #88 SMP Fri Jun 2 15:51:41 MDT 2023 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Jun 18 16:27:48 2023 from 10.0.2.10
root@syzkaller:~#
```

Figure 1: Gnome terminals, *terminal 1* boot up the virtual machine, *terminal 2* connect to the virtual machine through `ssh`.

```
 poc-355     [001] d...1    49.640095: bpf_trace_printk: ====KMASN 4b28366af7d9 uninitialized happend====
selin-357    [000] d.Z.1    70.170145: bpf_trace_printk: ====CVE-2017-7184 integer overflow happens====
selin-361    [000] d.Z.1   103.822859: bpf_trace_printk: ====CVE-2016-6187 out-of-bound happened====
 poc-363     [000] d...1   119.726617: bpf_trace_printk: ====CVE-2021-4154 pointer in put_fs_context quarantined=
```

Figure 2: Outputs of the BPF prevention programs in *terminal 2*.

(E3): Test if the BPF program can prevent an use-after-free vulnerability CVE-2021-4154 from being triggered.
Execution: execute `test-CVE-2021-4154.sh` in *terminal 1*
Results: Because of the quarantine & sweep policy, the dangling pointer of a use-after-free vulnerability will be quarantined, the vulnerability will not be triggered. Proof-of-concept in *terminal 1* cannot trigger the vulnerability, and *terminal 2* will report dangling pointers are quarantined.

(E4): Test if the BPF program can prevent an uninitialization vulnerability kmsan-4b28366af7d9 from being triggered.
Execution: execute `test-kmsan_4b28366af7d9.sh` in *terminal 1*
Results: The vulnerability is triggered in *terminal 1* under an conservative check policy, it means that the BPF program will catch the uninitialization memory but not kill the process. There will be reports `====kmsan-4b28366af7d9 is happened====` in *terminal 2*.

(E5): Test if the BPF program can prevent a data race vulnerability kcsan-dcf8e5633e2e from being triggered.
Execution: KCSAN does not provide proof-of-concept, so no proof-of-concept is executed in *terminal 1*.
Results: The BPF program `detector_kcsan_dcf8e5633e2e` will be keep checking if the vulnerability kcsan-dcf8e5633e2e is being triggered.

The evaluation results, as depicted in Figure 2, demonstrate the successful prevention of various vulnerabilities. Starting from the top, the artifact effectively prevent uninitialized variables, integer overflows, and out-of-bound vulnerabilities. Additionally, the artifact identifies and quarantines the potential dangling pointers of use-after-free vulnerabilities. It is important to note that evaluators have the freedom to execute additional commands on *terminal 1* during the evaluation process. Despite the execution of these commands, the system remains functional and unaffected by the errors.

Reproducible The reproducible evaluation includes 2 part, the performance test and add new BPF prevention programs.

(E1) : test performance overhead when BPF prevention programs protect the system.
Execution: (about 2 hour for each performance test, 14 hours in total) execute the `phoro-run.sh` scripts in the *terminal 1*, and 7 seperate performance tests are queued to be executed, including the vanilla, system protected by 5 BPF program individually and simultaneously.
Results: execute `phoronix-benchmark` `↪ start-result-viewer`

(E2) : add new BPF program to prevent new
Execution: (about 10 minutes) We present an instruction including a demo(a use-after-free), executor can follow the guidance to extract the sanitizer report and generate a new program. Similar to the functional evaluation, evaluators can also evaluate the effectiveness of the new added BPF program.
Results: After execute proof-of-concept in *terminal 1*, there will be report that dangling pointers are quarantined in *terminal 2*.

The results show that artifact can reproduce the performance overhead and easy to add new BPF programs for upcoming kernel vulnerabilities.

USENIX'23 Artifact Appendix: <Mitigating Security Risks in Linux with KLAUS – A Method for Evaluating Patch Correctness –>

Yuhang Wu
yuhang.wu@northwestern.edu
Northwestern University

Zhenpeng Lin
zplin@u.northwestern.edu
Northwestern University

Yueqi Chen
yueqi.chen@colorado.edu
University of Colorado Boulder

Dang K Le
dang.le@northwestern.edu
Northwestern University

Dongliang Mu
dzm91@hust.edu.cn
Huazhong University of Science and Technology

Xinyu Xing
xinyu.xing@northwestern.edu
Northwestern University

A Artifact Appendix

A.1 Abstract

This artifact is applying for an **Artifacts Available** badge, an **Artifacts Functional** badge, and an **Results Reproduced** badge.

The artifact primarily consists of two parts: the source code of KLAUS, and the Docker runtime environment for KLAUS. These components encompass the specific implementation of the designs in our paper, and also offer a very user-friendly and convenient mode of operation. KLAUS is a framework for verifying the correctness of Linux kernel patches, mainly composed of a static analysis part (identifying AWRPs as proposed in our paper) and a dynamic fuzz testing part (Fuzzing).

Firstly, our open-source source code includes the source code for static analysis, the source code for automatic instrumentation, and the source code for the fuzzer. These source codes have good extensibility and will be beneficial for other researchers to conduct more in-depth research improvements or extensions. Subsequently, we encapsulate the entire KLAUS framework in a Docker image, for the convenience of all researchers and users. In this Appendix, we will provide the necessary explanations and some screenshots to facilitate the evaluation of our academic achievements.

A.2 Description & Requirements

Hardware, for evaluation purposes, it is recommended to use a multi-core CPU environment that supports Kernel-based Virtual Machine technology. Additionally, due to the fuzzing process, it is recommended to have a minimum of 4 CPU cores, at least 32GB of memory, and a minimum of 100GB of hard disk space.

Software, the experiment requires a system with X86/64 architecture that supports running a Docker environment. It is necessary to have a network environment that supports the installation of dependencies and accessing information and code from the syzkaller community and Google's hosted Git website.

A.2.1 Security, privacy, and ethical concerns

All experiments (static analysis/fuzzing) are conducted within Docker containers, but it is necessary to map a shared folder from the local machine to the Docker container to serve as data storage. This might result in the generation of malicious files in the shared folder; however, as long as they are not executed on the local machine, they will not cause any harm. Each instance of the Fuzzer runs inside QEMU within the Docker container and will not pose any threat to the local machine's system.

A.2.2 How to access

All the artifacts are available in `https://github.com/wupco/KLAUS`, the directory

- *1-Docker-env* are the runtime evaluations.
- *2-Syzpatch* are the major portion of the code used in our research.

A.2.3 Hardware dependencies

- CPU: 4 CPU cores with virtualization technology.

- Memory: 32GB or larger.
- Disk space: 100GB or larger.

A.2.4 Software dependencies

- Softwares: `Docker`.

A.2.5 Benchmarks

None.

A.3 Set-up

Please adhere to the instructions provided in the README.md file of our GitHub repository.

A.3.1 Installation

None.

A.3.2 Basic Test

```
root@d046d5fd4767:/# ls
analyzer.zip  gcc-bin      klaus_fuzzer.zip    llvm.zip        root  usr
bin           gcc.zip      lib                 media           run   var
boot          home         lib32               mnt             sbin
data          image        lib64               opt             srv
dev           image.zip    libx32              patch_analyzer  sys
etc           klaus_fuzzer llvm-project-10.0.1 proc            tmp
root@d046d5fd4767:/# cd data/
root@d046d5fd4767:/data# ls
fuzz_cfgs_dir  fuzz_workdir  kernels
root@d046d5fd4767:/data# cd fuzz_cfgs_dir
root@d046d5fd4767:/data/fuzz_cfgs_dir# ls
build_env.py  clang.patch  classmap.patch  config  kernel.patch  res.txt
root@d046d5fd4767:/data/fuzz_cfgs_dir# python3 build_env.py
Usage: python3 build_env.py [commitid] [syzid]
root@d046d5fd4767:/data/fuzz_cfgs_dir#
```

Figure 1: The files in the docker container.

After successfully building the Docker, execute it using the command `docker run -v $(pwd)/data:/data --rm -it --privileged klaus`. Upon executing this command, one should be inside the Docker container where commands can be executed freely. At this point, in the root directory of the container, there should be the directories essential for the experiment, namely `data`, `klaus_fuzzer`, `llvm-project-10.0.1`, `patch_analyzer`, `image`, and `gcc-bin`. Within the subdirectory `fuzz_cfgs_dir` of the `data` directory, one can execute the `build_env.py` file, which requires two arguments: `commitid`, representing the commit id of the buggy patch, and `syzid`, representing the bug report id of the bug that the patch addresses. For instance, to test the correctness of the patch located at https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=730c5fd42c1e, the required commitid is `730c5fd42c1e`, and the bug report https://syzkaller.appspot.com/bug?id=53b6555b27af2cae74e2fbdac6cadc73f9cb18aa id `syzid` is `53b6555b27af2cae74e2fbdac6cadc73f9cb18aa` that this patch fixes. This information can be observed in the Figure 1.

A.4 Evaluation workflow

KLAUS is utilized for verifying the correctness of Linux kernel patches. To achieve this objective, we employ a combination of static analysis and dynamic fuzzing techniques, which are the two critical components of KLAUS. It is important to note that our use of fuzzing technology is solely to validate that the AWRPs identified through static analysis are effective in assessing the correctness of Linux kernel patches; it is merely one implementation approach. Our primary contribution lies in the discovery and identification of AWRPs through static analysis. By successfully executing KLAUS, we anticipate generating information about the identified AWRPs, and also utilizing this information to automatically instrument the code pre-fuzzing. Ultimately, this will enable the successful launch of the fuzzer to evaluate the patch.

A.4.1 Major Claims

(C1): KLAUS will identify the AWRPs corresponding to each case in the ground truth dataset with respect to the patch.

(C2): The Fuzzer component of KLAUS can operate normally, and there is a high probability that it can trigger bugs resulting from errors in the patch.

A.4.2 Experiments

First, please follow the guide in our GitHub repository to properly set up the Docker environment. Subsequently, launch Docker, enter the Docker container, and execute the `build_env.py` file with the specified parameters. It is imperative to note that detailed information regarding our ground truth data is located in `Evaluation_Results.xlsx`. Upon the completion of static analysis and instrumentation, navigate to `/data/fuzz_cfgs_dir/[commitid]` and execute `fuzz_start.sh` to initiate the fuzzer. Information on AWRPs can be found in `prop.txt` and `cond.txt` within the `/data/kernels/[commitid]` directory, while the working directory of the fuzzer is located at `/data/fuzz_workdir`. Additionally, configuration information for running the fuzzer can be found in `/data/fuzz_cfgs_dir/[commitid]/config`. If it is necessary to empty and reset the environment under the `data` folder, `cleardata.sh` can be executed on the local machine. It is important to note that occasionally, when there is an error in applying `clang.patch` or `classmap.patch`, it can be ignored by pressing Enter directly.

(E1): Test whether the ground truth cases can be analyzed correctly.
Execution: execute `build_env.py` file with the specified parameters.
Results: `[commitid] has inst` will be reported in stdout. Information on AWRPs can be found in `prop.txt` and `cond.txt` within the `/data/kernels/[commitid]` directory.

(E2): Test whether the fuzzer can be run normally.

Execution: execute `fuzz_start.sh` file in `/data/fuzz_cfgs_dir/[commitid]`.

Results: The fuzzer will operate normally, and concurrently, the status of the fuzzer will be outputted to the stdout.

```
  CC      arch/x86/boot/compressed/acpi.o
  LD [M]  fs/nfs/flexfilelayout/nfs_layout_flexfiles.ko
  GZIP    arch/x86/boot/compressed/vmlinux.bin.gz
  CC      arch/x86/boot/compressed/misc.o
  MKPIGGY arch/x86/boot/compressed/piggy.S
  AS      arch/x86/boot/compressed/piggy.o
  LD      arch/x86/boot/compressed/vmlinux
ld: arch/x86/boot/compressed/head_64.o: warning: relocation in read-only section `.head.text'
ld: warning: creating DT_TEXTREL in a PIE
  ZOFFSET arch/x86/boot/zoffset.h
  OBJCOPY arch/x86/boot/vmlinux.bin
  AS      arch/x86/boot/header.o
  LD      arch/x86/boot/setup.elf
  OBJCOPY arch/x86/boot/setup.bin
  BUILD   arch/x86/boot/bzImage
Setup is 16540 bytes (padded to 16896 bytes).
System is 50933 kB
CRC bc64e9d2
Kernel: arch/x86/boot/bzImage is ready  (#1)
yes: standard output: Broken pipe
fixing hash poc -> b3d32ffcd753cf76e03f0f14b8fc12ebdf2ee3c7
730c5fd42c1e3652a065448fd235cb9fafb2bd10 has inst
root@e45cc29c95c6:/data/fuzz_cfgs_dir# python3 build_env.py 730c5fd42c1e 53b6555b27af2cae74e2fbdac6cadc73f9cb18aa
```

Figure 2: The expected result of the static analysis part.

```
root@e45cc29c95c6:/data/fuzz_cfgs_dir/730c5fd42c1e3652a065448fd235cb9fafb2bd10# ./fuzz_start.sh
2023/06/21 06:28:42 loading corpus...
2023/06/21 06:28:42 loading from auxiliary file
2023/06/21 06:28:42 serving http on http://127.0.0.1:13924
2023/06/21 06:28:42 serving rpc on tcp://[::]:46809
2023/06/21 06:28:42 booting test machines...
2023/06/21 06:28:42 wait for the connection from test machine...
```

Figure 3: The fuzzer has been successfully executed.

For the results of the static analysis part, as shown in Figure 2, it successfully identified the AWRPs in the patch and instrumented the kernel code. Subsequently, when the fuzzer is executed, information and status during fuzzing will be displayed, as in Figure 3.

USENIX'23 Artifact Appendix: ARGUS: Context-Based Detection of Stealthy IoT Infiltration Attacks

Phillip Rieger Marco Chilese Reham Mohamed

Markus Miettinen Hossein Fereidooni Ahmad-Reza Sadeghi

Technical University of Darmstadt

A Artifact Appendix

A.1 Abstract

The continuous growth and expansion of Internet of Things (IoT) application domains, device diversity, and connectivity is a well-established trend. IoT devices have been implemented to manage and monitor various functions in smart homes, buildings, cities, and factories. However, this surge in adoption has made IoT devices an appealing target for cyber attackers. ARGUS analysis the IoT devices' behavior to detect contextual attacks, e.g., opening a smart lock while the smart home residents are absent. The artifact provides the 5 real-world datasets that were collected as part of the research. Using the datasets provides a benchmark dataset for future works and eases the comparison to future approaches.

A.2 Description & Requirements

The dataset contains the benign status updates of various IoT devices deployed for several months in 5 different real homes. The corresponding folder of a dataset contains for each day one CSV file, each with the devices' status updates of the corresponding day. The CSV files contain three columns, the time stamp, the unique identifier of the device/sensor, and the new status. The column new_status contains the status that the IoT device reported at the given time. Therefore, depending on the device, the value can be boolean (on/off), nominal (e.g., for the weather sunny, cloudy, partlycloudy, etc.) or numerical. The timestamps are given in UTC. However, it should be noted that the local time zone was ECT. Table 1 shows the number of events per dataset. In total, the artifact consists of 2 599 292 events.

Based on these events, attacks can be easily simulated, e.g., by injecting an event that sets the status of "camera.status" to on, while the sensor "person.home" has the status "home.

To obtain the dataset, we gathered data from IoT devices across various smart home environments. These environments, named Home 1- Home 5, were made up of multiple

Table 1: Number of events contained in each dataset

Dataset	#Events
Home1	2 599 292
Home2	362 744
Home3	16 952
Home4	311 111
Home5	459 670

sensors (such as temperature, humidity, and motion sensors) and actors (such as light bulbs and thermostats). To ensure the dataset was diverse and that the individual setups differ from each other, each home also included additional sensors and actors. For instance, Home 1 had a CO_2 sensor, while Home 4 and Home 5 had multiple smart thermostats. The devices were installed in different homes, ranging from single-person apartments to shared homes with four inhabitants. In total, ten male and female participants from different age groups, including teenagers, students, and adults up to around 49 years old, were involved in the experiments. The data was collected using the open-source smart-home control system, Home Assistant.

The smart-home setups include 3 homes with multiple rooms and multiple inhabitants (Home 3, Home 4, Home 5), a one-room apartment (Home 2), as well as, a single room in a shared apartment (Home 1). The experiments included ten different male and female participants (teenagers, students, and adults up to approximately 49 years). We made use of the deployed home automation platform (in our setup HomeAssistant) to automatically trigger events, e.g., turning off the camera when the user comes home, or to turn off the heating when the window is opened.

Table 2 shows for each setup in the dataset, a detailed list of the deployed IoT devices and measured values. The measured values cover different categories of contextual features: i) Sensors/devices that measure ambient or temporal features (e.g., temperature, humidity, and luminosity), ii) user features (e.g., user presence and sleep confidence), and event features (e.g., states of the light bulbs, doors or windows).

Table 2: Deployed devices in the collected real-world IoT dataset. The deployment of a sensor/actor is indicated by •, while the absence is indicated by ○.

Device	Home 1	Home 2	Home 3	Home 4	Home 5
Automation - All lights off	○	○	○	○	•
Automation - All lights on	○	○	○	○	•
Automation - Camera off when at home	○	○	○	•	○
Automation - Dinner lights	○	○	○	•	•
Automation - Dinner table light	○	○	○	○	•
Automation - Gaming mode	○	○	○	○	•
Automation - Heating boost off	○	○	○	○	•
Automation - Light off when no motion	○	○	○	•	•
Automation - Lights off in the evening	○	•	○	○	○
Automation - Lights off when too bright	○	•	○	•	○
Automation - Lights on in the morning	○	•	○	○	○
Automation - Lights on when motion detected	○	•	○	•	○
Automation - Piano Light	○	○	○	○	•
Automation - Sofa Lamp	○	○	○	○	•
Automation - Studio Light off	○	○	○	○	•
Automation - Studio Light on when motion	○	○	○	○	•
Automation: Camera on when user leave	○	○	○	•	○
Camera Status Sensor	○	•	•	○	○
Climate - Control access point 1	○	○	○	○	•
CO_2 Sensor Status	•	○	○	○	○
CO_2 Sensor	•	○	○	○	○
Control Access Room 1 Sensor	○	○	○	•	○
Door Sensor	•	•	•	•	•
Floor lamp	○	○	○	○	•
Heating - heater valve	○	○	○	○	•
Heating Temperature Sensor	•	•	○	•	○
Homematic - Radiator Thermostat Temperature Sensor	○	○	•	○	○
Humidity Sensor	•	•	○	•	•
IKEA Tradfri Roller Blind Sensor	•	○	○	○	○
IP Camera - Light Level	○	○	○	○	•
IP Camera - Motion	○	○	○	○	•
IP Camera - Motion Active	○	○	○	○	•
IP Camera - Pressure	○	○	○	○	•
IP Camera - Sound	○	○	○	○	•
Lamp consumption	○	○	○	○	•
Lamp consumption (daily)	○	○	○	○	•
Lamp consumption (total)	○	○	○	○	•
Lamp current	○	○	○	○	•
Lamp voltage	○	○	○	○	•
Light - Ceiling	•	•	•	•	•
Light - Desk Lamp	•	•	•	•	•
Light - Living Room	○	○	•	○	○
Philips Hue - Light Level Sensor 1	○	•	•	•	•
Philips Hue - Light Level Sensor 2	○	○	•	○	○
Philips Hue - Motion Sensor 2	○	○	•	○	○
Philips Hue - Temperature Sensor 1	○	•	•	•	•
Philips Hue - Temperature Sensor 2	○	○	•	○	○
Philips Hue - White Lamp 2	○	○	•	○	○
Philips Hue - White Lamp 3	○	○	•	○	○
Philips Hue - Motion Sensor 1	○	○	•	•	•
Piano lamp	○	○	○	○	•
Radiator Thermostat Sensor	○	○	○	•	•
Smartphone - Battery Life	○	•	○	○	○
Smartphone - Charging	○	○	○	○	•
Smartphone - Charging Sensor	•	○	○	○	○
Smartphone - Connected to WLAN	○	•	○	•	○
Smartphone - Detected Activity	•	•	○	○	•
Smartphone - Light Sensor	○	•	○	○	○
Smartphone - Locked	○	○	○	○	•
Smartphone - Phone Status	○	•	○	○	○
Smartphone - Sleep Confidence	•	•	○	•	○
Smartphone - Sleep Segment	○	○	○	•	○
Smartphone - Tracker	•	•	○	•	○
Studio lamp	○	○	○	○	•
Sun Sensor	•	•	•	•	•
Temperature Sensor (ESP)	•	•	○	•	•
User Presence	•	•	○	•	○
Weather - Home Location	•	•	•	•	•
Weather - Town	•	○	•	•	○
Window Sensor	•	•	•	•	○

A.2.1 Security, privacy, and ethical concerns

The dataset collection raised ethical concerns, as the recorded behavior of the users might contain sensitive data. We addressed these concerns by ensuring that all affected persons, i.e., the users as well as all guests, were aware of the data collection and gave their consent. Further, we limited the approach to non-privacy-sensitive sensors and excluded the other sensors like the geolocation or the SSID of the WiFi network that the mobile phone is connected to. In addition, all potentially sensitive data items were anonymized. Our experimental set-up has been reviewed and approved by the ethics board of our university.

As the artifact is an anonymized dataset, it does not raise any security, privacy, or ethical concerns for people using this dataset. The license allows the usage of any non-commercial purpose (CC-BY-NC-SA).

A.2.2 How to access

The dataset was uploaded at GitHub and is accessible at: `https://github.com/TRUST-TUDa/argus-data/tree/606d5a5ebe78f602e27b9f2c48ea103348463eeb`. The dataset that was used for the paper is tagged as ArtifactAppendix.

A.2.3 Software dependencies

The git program to check out the dataset from GitHub.

A.3 Set-up

The artifact consists of the datasets that were collected for the paper. The artifact includes contextual events from several real-world smart homes and makes them publicly available for future research. Therefore, the artifact does not include any software/code but is intended as a benchmark for future work on contextual intrusion detection.

A.3.1 Installation

Clone the dataset from GitHub and checkout the specified state via:

```
$ git clone https://github.com/TRUST-TUDa/argus-data.git
$ cd argus-data
$ git checkout 606d5a5
```

A.3.2 Basic Test

Verify via `git status` that the repository is at 606d5a5 to verify that all files were cloned correctly.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix
xNIDS: Explaining Deep Learning-based Network Intrusion Detection Systems for Active Intrusion Responses

Feng Wei
University at Buffalo

Hongda Li
Palo Alto Networks

Ziming Zhao
University at Buffalo

Hongxin Hu
University at Buffalo

A Artifact Appendix

A.1 Abstract

We present xNIDS, a novel framework that facilitates active intrusion responses by explaining DL-NIDS. Our artifact includes the proposed explanation method dedicated to explaining DL-NIDS.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact can be used by users anywhere, but it should be utilized strictly for research purposes and in adherence to good ethical practices.

A.2.2 How to access

This artifact is publicly available at https://github.com/CactiLab/code-xNIDS/releases/tag/v2023.1.0.

A.2.3 Hardware dependencies

The demo code is hardware-independent and can be optimized for execution on Google Colab.

A.2.4 Software dependencies

To run the code, the following software packages are required: Python, TensorFlow, Keras, NumPy, pandas, scikit-learn, Matplotlib, psutil, and asgl.

A.2.5 Benchmarks

The benchmark datasets utilized in this artifact are the NSL-KDD and Kitsune datasets.

A.3 Set-up

A.3.1 Installation

To access the code, kindly download it from the following link https://github.com/CactiLab/code-xNIDS/tree/main.

A.3.2 Basic Test

The demo code is written in Jupyter Notebook and can be executed on Google Colab.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix
Curve Trees: Practical and Transparent Zero-Knowledge Accumulators

Matteo Campanelli
Protocol Labs
matteo@protocol.ai

Mathias Hall-Andersen
Aarhus University
ma@cs.au.dk

Simon Holmgaard Kamp
Aarhus University
kamp@cs.au.dk

A Artifact Appendix

A.1 Abstract

We provide a Rust implementation of Curve Trees instantiated with Bulletproofs. The repository includes an implementation of the *select and rerandomize* primitive, which proofs that a commitment is a rerandomization of a commitment in a committed set. This primitive is then used to construct an accumulator as well as a simple anonymous payment system.

A.2 Description & Requirements

Our results were produced using an AWS C6i.2xlarge instance with 16GB of RAM and 8 vCPUs. This corresponds to 4 physical cores on an Intel Xeon 8375C processor with 2.9 GHz clock speed. The benchmarks were compiled using version 1.68.0 of the rust compiler. We found similar performance on our laptops. The field arithmetic is optimized for newer x86_64 chips, but the code will still work on other architectures.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The code is available at https://github.com/simonkamp/curve-trees/tree/4467be81737732a5b2794b5ad70459681b3bd19c.

A.2.3 Hardware dependencies

Any system with access to the internet and a rust compiler and standard library should be able to run the artifact.

A.2.4 Software dependencies

The only software dependency is the rust compiler, which can be downloaded from https://rustup.rs/.

A.2.5 Benchmarks

None.

A.3 Set-up

Install the rust compiler (https://rustup.rs/). Install jq command for formatting output (https://jqlang.github.io/jq/).

A.3.1 Installation

Clone the repository.

A.3.2 Basic Test

Run `cargo build`.

A.4 Evaluation workflow

After the set-up above, verify the functionality of the artifact by running the tests:

```
cargo test --release
```

A.4.1 Major Claims

(C1): The implementation of the select and rerandomize primitive matches the performance reported in table 1 of our paper. This is proven by the experiment (E1).

(C2): The implementation of the accumulator matches the performance reported in table 2 of our paper. This is proven by the experiment (E2).

(C3): The implementation of the $\mathbb{V}$Cash matches the performance reported in table 3 of our paper. This is proven by the experiment (E3).

A.4.2 Experiments

Each experiment will run benchmarks of the proving and verification time for the set sizes and curves reported in the relevant table of our paper.

(Before experiments): *[All tables] [1 human-minutes + 30 compute-minutes]:* We recommend using the script `gen_fmt_estimates.sh` to run all the benchmarks. The output can be reprinted using `fmt_estimates.sh`.

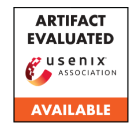
(E1): *[Table 1] [5 human-minutes + 0 compute-minutes]: Demonstrates proof size, proving and verification times for select-and-rerandomize of Curve Trees (batching and non-batching case) in different curves and on different parameters (Table 1). The results are found under "Table 1 (Accumulator)" in the output of* `fmt_estimates.sh`.

(E2): *[Table 2] [5 human-minutes + 0 compute-minutes]: Demonstrates proof size, proving and verification times for accumulators from Curve Trees in different curves on sets of size* 2^{30} *(Table 2). The results are found under "Table 2 (SelectAndRerand)" in the output of* `fmt_estimates.sh`.

(E3): *[Table 3] [5 human-minutes + 0 compute-minutes]: Demonstrates transaction size, proving and verification times for* $\mathbb{V}$*Cash (Table 3). The results are found under "Table 3 (Pour)" in the output of* `fmt_estimates.sh`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: VERIZEXE: Decentralized Private Computation with Universal Setup

Alex Luoyuan Xiong[1], Binyi Chen[2], Zhenfei Zhang[3], Benedikt Bünz[4], Ben Fisch[5], Fernando Krell[6], and Philippe Camacho[7]

[1,2,3,4,5,6,7]Espresso Systems
[1]National University of Singapore
[4]Stanford University
[5]Yale University

April 24, 2023

A Artifact Appendix

A.1 Abstract

We provide the instructions to access and evaluate artifacts for performance of VERIZEXE system. The artifacts contain a `veri-zexe` code base written in Rust with benchmark test suites, and a forked `snarkVM` code base[1] as the state-of-the-art to compare against. We further specify the hardware specifications under which our VERIZEXE can successfully generate transaction in a reasonable time frame thanks to the massive improvements on prover time and memory usage. This demonstrates the practicality of our system even on resource-limited devices like phones and laptops.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There should be no security risk or privacy leakage of any kind for evaluators. Executions of the artifacts have no side-effect outside of the testing folders, nor would the programs require any privileged system permission to run.

A.2.2 How to access

Our artifacts (software only) are hosted as public repositories on GitHub. Our implementation of VERIZEXE system is accessible via:

https://github.com/EspressoSystems/veri-zexe/tree/42657f254c7f1353914b098dc78f5fb97408bfcd.

The primary prior work that we improve on and benchmark against is accessible via:

https://github.com/alxiong/snarkVM/tree/290c05273e3a30523335524fb682ef316cbbf414.

[1]Modified for fair comparison and faithful instantiation of the original DPC scheme

A.2.3 Hardware dependencies

We do not require special hardware, and the evaluation can be run on any Linux machine. To reproduce the same result, we recommend using Amazon EC2 instances:

Instance Type	vCPU	Memory	Arch	Simulating
a1.xlarge	4	8 GB	arm64	Phone
c5a.4xlarge	16	32 GB	x86_64	Laptop
c5a.16xlarge	64	128 GB	x86_64	Server

Table 1: AWS EC2 instance type and hardware spec

A.2.4 Software dependencies

Any Linux distribution will work, and we use Ubuntu 20.04 across all experiments. The only software prerequisite is: `Rust` : https://www.rust-lang.org/tools/install.

A.2.5 Benchmarks

None. No external data-set or benchmark model required.

A.3 Set-up

A.3.1 Installation

1. Install software prerequisites listed in ??.
2. Git clone both repos, `veri-zexe` at https://github.com/EspressoSystems/veri-zexe.git and `snarkVM` at https://github.com/alxiong/snarkVM.
3. For both repos (same procedure), `cd` into the repo folder, git checkout to `paper-benchmark` branch, run `cargo build` .

A.3.2 Basic Test

Ensure you can compile the source code and test/bench code by running:

```
cargo check && cargo test --no-run
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): VERIZEXE improves the state-of-the-art (snarkVM) by 9x in transaction generation time and by 3.4x in memory usage with small variability across different transaction dimensions. This is proven by the Experiment (E1 + E2) described in **??** whose results are reported in Table. 2.

(C2): VERIZEXE is the first DPC scheme to make transaction generation possible and practical in resource-limited hardware environments, such as mobile phones or consumer-grade laptops. Furthermore, we exhibit a trade-off between prover time and peak memory usage. This is proven by the Experiment (E3) desribed in **??** whose results are reported in Table. 4.

A.4.2 Experiments

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Link explicitly the description of your experiments to the items you have provided in the previous subsection about Major Claims. Please provide your estimates of human- and compute-time for each of the listed experiments (using the suggested hardware/software configuration above). Follows an example:

(E1): *[2 human-minutes + 1.2 computer-minutes + c5a.16xlarge EC2]:* We run benchmarks on `veri-zexe` across different transaction dimensions (`2x2,3x3,4x4`) and measure all major metrics among which total transaction generation time and peak memory usage are the main targets.
Preparation: Enter into your AWS c5a.16xlarge EC2 instance, or environments of the same hardware spec (see Table. **??**).
Execution and Result: Please follow detailed instructions in `usenix-ae.md` file in the `veri-zexe` project root.

(E2): *[5 human-minutes + 15 computer-minutes + c5a.16xlarge EC2]* We run benchmarks on `snarkVM` across different transaction dimensions (`2x2,3x3,4x4`) in the same environment and measuring the same metrics.
Preparation: Enter into your AWS c5a.16xlarge EC2 instance.
Execution and Result: Please follow detailed instructions in `usenix-ae.md` file in the `snarkVM` project root.

(E3): *[5 human-minutes + 6 computer-minutes + a1.xlarge & c5a.4xlarge EC2]* We try to generate 2-input-2-output DPC transaction across different hardware environments, especially resource-limited environment simulating phones and laptops. This used to be impossible for `snarkVM` due to high memory usage and much slower prover.
Preparation: Enter into your AWS a1.xlarge and c5a.4xlarge EC2 instance.
Execution and Result: Please follow detailed instructions in `usenix-ae.md` file in the `veri-zexe` project root.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Systematic Assessment of Fuzzers using Mutation Analysis

Philipp Görz[1], Björn Mathis[1], Keno Hassler[1], Emre Güler[2],
Thorsten Holz[1], Andreas Zeller[1], and Rahul Gopinath[3]

[1]CISPA Helmholtz Center for Information Security, Germany
[2]Ruhr-Universität Bochum, Germany
[3]University of Sydney, Australia

A Artifact Appendix

A.1 Abstract

We provide the source code of our benchmarking framework, which is written in Python. This includes the code to create the plots, written in Python and R. Additionally, we provide the seed corpus used as the initial input for the tool. Furthermore, we provide the artifacts from the evaluation stages, including the final databases from which the plots in the paper are generated.

Additionally, we provide notes and tooling for the two manual analyses performed for the paper.

A.2 Description & Requirements

Recreating the experimental setup requires a Linux system with docker installed. Additionally, the user needs to be in the docker group. To manage the python version and dependencies hatch is used. For details see the readme of the framework.

The evaluation requires a initial minimal seed corpus, which is provided by the zip file in the Zenodo link, it is the 'seeds/minimal' directory. Usage of this seed corpus is described in the readme of the framework.

Additionally, an environment setup script is provided by the framework, again see the readme for details.

This should (hopefully) be all that is required to recreate the experimental setup, while we have tested using the framework on some systems we can obviously not guarantee that it will work on all systems. If there are any issues please contact us.

The minimal hardware requirements are at least 16GB of RAM and 50GB of disk space. The RAM requirements scale with number of running instances, which are more likely limited by the number of cores available. The framework is designed to scale to with the number of cores.

For reference, the evaluation for the paper used four servers with Intel Xeon Gold 6230R CPUs, each with 52 cores and 188 GB RAM.

A.2.1 Security, privacy, and ethical concerns

The framework requires a user that is in the 'docker' group, this should be seen as equivalent to root access, although this way, we avoid running the whole framework as root. Furthermore, the provided setup script will disable ASLR on the system to stabilize the fuzzing results, but it also facilitates exploitation of security vulnerabilities. If this is a concern, comment out the respective line. The docker containers will access '/dev/shm' and '<project root>/tmp' on the host system, this is required for the shared memory and exchanging files. Other than for building the Docker images, no internet access is required.

A.2.2 How to access

The artifact consists of two parts: the main framework and the other artifacts, both are required to reproduce the results. From the 'Other Artifacts', only the seed corpus in the `mua-fuzzer-bench-eval-data.7z` archive under the directory 'seeds/minimal' is strictly needed to reproduce the results. The remaining files can be referenced to support the evaluation process as they contain our intermediate and final results.

'Framework - Source Code': https://github.com/CISPA-SysSec/mua_fuzzer_bench/tree/b3cc3815f9dce9371eb5d461bb5beb888c032327

'Other Artifacts': https://zenodo.org/record/8060560

A.2.3 Hardware dependencies

The evaluation framework runs on commodity hardware, but reproducing every result in the paper will consume a considerable amount of CPU ressources. For reference, the evaluation

for the paper used four servers with Intel Xeon Gold 6230R CPUs, each with 52 cores and 188 GB RAM. Thanks to the modular design, it is also possible to run the evaluation on a subset of fuzzers or programs.

A.2.4 Software dependencies

The evaluation framework depends on Linux with docker and hatch installed. The user needs to be in the docker group. We tested the framework on Ubuntu and Debian, but it should run on any distribution.

A.2.5 Benchmarks

The evaluation requires a initial minimal seed corpus, which is provided by the zip file in the Zenodo link, it is the 'seeds/minimal' directory. To reproduce the exact figures shown in the paper, we provide the result databases.

A.3 Set-up

See the 'Usage' section of the readme in the framework repository.

A.3.1 Installation

See the 'Installation' section of the readme in the framework repository.

A.3.2 Basic Test

See the 'Usage' section of the readme in the framework repository. For a basic test the `–fuzz-time` parameters of the commands shown can be reduced to one minute, `--instances` can also be reduced. Additionally, the command run under 'Basic Evaluation' can be manually aborted early via a keyboard interrupt (`Ctrl+C`) to reduce the number of evaluated supermutants. The commands under 'ASan' and '24 Hours' will only evaluate supermutants that have been tried for the 'Basic Evaluation'.

Expected output for the `coverage_fuzzing` command is the directory containing the coverage seed corpus (see the `–result-dir` argument), for the `eval` command the expected output are the databases placed at the path given by the `–result-path` argument.

A.4 Evaluation workflow

A.4.1 Major Claims

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Enumerate here the major claims (Cx) made in your paper. Follows an example:

(C1) : Computational effort is reduced by (on average) factor 3.8 by using supermutants. This is a side result from experiment (E1) described in section 5.2 and reported in Table 4.

(C2) : Different fuzzers show quite similar results. This is also proven by experiment (E1) described in section 5.2; the results are reported in Table 5 and illustrated in a Venn diagram (Figure 3).

(C3) : Coverage accounts for most mutants detected (97.5%) in our evaluation. This is the share of all mutants that were killed by the ensemble of all fuzzers during coverage fuzzing, as explained in Section 5.2 (paragraph *Results*). This number can be calculated from the data in Table 5, which is generated in experiment (E1).

(C4) : ASan moderately increases the number of killed mutants. In Section 5.3, we calculate this number per evaluated fuzzer. This is based on comparing the results from experiment (E2) shown in Table 6 with the results from experiment (E1) shown in Table 5.

(C5) : One hour of fuzzing after the seed coverage stage is sufficient to evaluate a supermutant. In experiment (E3), we re-run a random subsample for 24 hours and see that almost no additional mutants are killed (Table 7). This is described in Section 5.2.

(C6) : Most of the remaining mutants (84%) introduce a semantic change (*theoretically* detectable with a perfect oracle). This is based on manual analysis of non-killed mutants (E4). The result is described in Section 5.2.1.

(C7) : Mutations induced by our mutation operators are coupled to real faults, since 71% of the studied recent vulnerabilities in experiment (E5) can be re-introduced with our mutation operators. We explain this result in Section 5.4.

A.4.2 Experiments

(E1): [Basic Experiment] [16.36 CPU core years] The initial experiment as described in Section 5.2. Includes Phase I and Phase II.
How to: All setup and preparation is explained in the readme of the framework. Everything should be explained when following the instructions up to 'Basic Evaluation'.
Note that the `--seed-dir` should point to the extracted content of the `seeds/minimal` directory of the eval data archive.
Results: The resulting plots for experiments E1 to E3 can be are produced as described in the readme of the framework in the section 'Getting the Results'.

(E2): [ASan Experiment] [15.16 CPU core years] This experiment depends on E1, keep following the process as described in the readme of the framework to the section 'ASan'.

(E3): [24 Hours Experiment] [7.42 CPU core years] This experiment depends on E2, keep following the process as described in the readme of the framework to the section '24 Hours'.

Note that either the rerun json file can be adapted to contain 100 mutations or a manual interruption can be done once the 100 mutations are reached.

Now, finally, the results can be produced as described in the readme of the framework in the section 'Getting the Results'.

(E4): [Manual Analysis of Mutations] [8 human hours]: This is a manual analysis of mutations that are not killed to see if the created mutations are useful to evaluate fuzzers.

How to: Examine covered mutations that are not killed even after the 24 hour experiment, see Section 5.2.1. Just for reference, we provide our notes of the manual analysis in the Zenodo repository, under the `not_killed_24.xlsx` file.

Preparation: This experiment depends on the result of the previous experiment (E3), the following SQL query should be run on the database produced. Note that the `prepare_db` command needs to be run on the database, see 'Getting the Results' in the readme of the framework. The list of mutants that are still not killed after 24 hours is obtained with the following SQL query:

```
select completed_runs.prog, completed_runs.
    mut_id, directory, file_path, line,
    column, instr, funname, pattern_name,
    description, procedure from
    completed_runs
join mutations on mutations.prog =
    completed_runs.prog and mutations.
    mutation_id = completed_runs.mut_id
join mutation_types using (mut_type)
where num_confirmed == 0
order by random()
limit 120;
```

Note that some of the mutants can be in system libraries, which we have skipped during our manual analysis, this is also the reason why the limit is set to 120 instead of 100.

Note that the source code of the programs can be found under the `<project root>/tmp/programs` directory.

Execution: This is a manual experiment, where for each mutation the corresponding line and surrounding code is examined to decide if the mutation does not cause a semantic change, or if it does, whether it is detectable when using ASan or a simple crash oracle.

(E5): [Manual Analysis of Vulnerability Coupling] [8 human hours] This is a manual analysis to see if the mutations that are created simulate real vulnerabilities. This is described in Section 5.4.

How to: We regard a vulnerability to be reintroduced if the mutation causes the patched program to reintroduce the vulnerability. We provide the list of CVEs we analyzed in the Zenodo repository, under the `CVEs.xlsx`. The list of CVEs that we analyzed was obtained using the code in the file `cve-script.7z`, which uses the official CVE list as source. The script is written in Rust, though we would recommend to just re-examine the CVEs we analyzed.

Preparation: Required is the list of CVEs to analyze and a description of the mutation operators. Which can be found in the framework repository under `<project root>/mutation_doc.json`.

Execution: For each CVE, examine the patch if it would be introduced by a mutation operator or a combination of mutation operators. If so, the CVE is regarded as reintroduced.

A.5 Notes on Reusability

The framework is modular and allows to run on specified sets of fuzzers and programs for a chosen time. For details, consult the readme and help for the evalutation script (accessible with `-h`). The readme of the framework repository contains a section 'Extending the Tool', describing how the tool can be used to evaluate on other programs, fuzzers, and mutations.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: HECO: Fully Homomorphic Encryption Compiler

Alexander Viand[1], Patrick Jattke[2], Miro Haller[3], Anwar Hithnawi[2]

[1]*Intel Labs* [2]*ETH Zurich* [3]*UC San Diego*

A Artifact Appendix

A.1 Abstract

HECO is a compiler for Fully Homomorphic Encryption built using the MLIR compiler framework. It translates imperative programs (defined in a high-level intermediate representation) into the SIMD-like paradigm required for most FHE schemes. It uses Microsoft SEAL as the underlying FHE implementation, generating C++ code that is then compiled and linked against the SEAL library. HECO uses xDSL (a Python-based "sidekick" to the C++ based MLIR framework) for its frontend, which features a simple embedded Domain Specific Language (DSL) that allows developers to specify FHE computations in a straight-forward manner.

The artifact also contains reference C++ implementations written directly against SEAL. For each of the programs we evaluate, we provide two implementations: one representing a "naive" non-expert baseline, and one "optimal" implementation based on the batching approaches generated by the synthesis-based Porcupine tool, which HECO is compared against in the paper.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

HECO requires evaluators to download, compile and run a variety of open-source software on their system. Beyond this, HECO should not impact the security or privacy of the system. HECO is a purely local application that does not initiate network connections. HECO itself does not interact with the filesystem, using `stdin`/`stdout` for input and output; the evaluation utilities write to but do not read from the filesystem.

A.2.2 How to Access

HECO is available as open-source software at `github.com/MarbleHE/HECO`. The evaluated artifact, specifically, is available at `github.com/MarbleHE/HECO/tree/artifact`.

A.2.3 Hardware Dependencies

HECO does not have specific hardware requirements. Note, however, that the "naive" versions of some of the evaluation workloads require at least 10 GB of free memory.

A.2.4 Software Dependencies

The HECO artifact has been tested on Ubuntu 20.04 LTS. Evaluating HECO requires `git`, `cmake` and a C/C++ compiler and linker (e.g., `clang` and `lld`). In addition, the LLVM/MLIR framework that HECO depends on requires the `ninja` build system. The HECO `README.MD` provides instructions on how to satisfy these requirements on debian-like systems. On other distributions, equivalent packages should exist, while on macOS, package managers such as `brew` should be able to provide these requirements. Note that the Python frontend (which is not part of this Artifact) additionally requires Python 3.11 or newer, with the `pip` package manger. A plotting script is included with the artifact for convenience, this also requires Python and, additionally, LaTeX to be installed.

A.2.5 Benchmarks

The runtime and memory benchmarks require the SEAL library, which is included as a git submodule. The optional plotting scripts also require Python and LaTeX to be installed.

A.3 Set-up

HECO should be cloned using git (`git clone https://github.com/MarbleHE/HECO.git`). After cloning, it is necessary to initialize the git submodules that are used to provide HECO's external dependencies, which are the LLVM/MLIR framework and the Microsoft SEAL library: `git submodule update -init -recursive`.

A.3.1 Installation

Before HECO can be built, the MLIR framework needs to be built. For evaluation, it is recommended to build MLIR in `Release` configuration. Note that compiling MLIR can require significant time, ranging from around 20 min on a powerful desktop or server, to up to two hours on weaker laptops. Assuming the current working directory is the HECO repository root, execute the following to build MLIR:

```
mkdir dependencies/llvm-project/build
cd dependencies/llvm-project/build
cmake -G Ninja ../llvm \
  -DLLVM_ENABLE_PROJECTS=mlir \
  -DLLVM_BUILD_EXAMPLES=OFF \
  -DLLVM_TARGETS_TO_BUILD=X86 \
  -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_C_COMPILER=clang \
  -DCMAKE_CXX_COMPILER=clang++ \
  -DLLVM_ENABLE_LLD=ON \
  -DLLVM_INSTALL_UTILS=ON \
  -DMLIR_INCLUDE_INTEGRATION_TESTS=OFF
```

In order to compile and run the generated C++ code, the Microsoft SEAL library needs to be installed. This can be done (from the HECO repository root) as follows:

```
cd ../../seal
cmake -S . -B build
cmake -build build
sudo cmake -install build
cd ../..
```

HECO, like its dependencies, uses the `cmake` build system. The dependencies are not automatically included in the HECO build structure (i.e., not added via `add_subdirectory`) and `cmake` will search for the dependencies during configuration. While SEAL's installation will be automatically detected, MLIR requires providing a `MLIR_DIR` path. Assuming SEAL and MLIR have been built as indicate above, execute the following (from the repository root) to build HECO:

```
mkdir build
cmake -S . -B build \
  -DMLIR_DIR=dependencies/llvm-project\
   /build/lib/cmake/mlir \
cmake -build build -target heco
```

A.3.2 Basic Test

After building HECO, you can call `heco` without any parameters and feed in any `*.mlir` file as input, which should round-trip and output the unmodified input program:

```
./build/bin/heco < test/example.mlir
```

You can also test the full compilation flow, by first compiling the High-Level Intermediate Representation (HIR) into a low-level C-friendly Intermediate Representation (emitC):

```
./build/bin/heco -full-pass\
   < test/example.mlir > test/out.mlir
```

This can then be translated into C/C++ source code:

```
./build/bin/emitc-translate -mlir-to-cpp\
   < test/out.mlir > test/out.cpp
```

A.4 Evaluation workflow

A.4.1 Major Claims

The paper makes the following major claims about the artifact

(C1): *HECO produces code that achieves significant speedup compared to naive/non-batched FHE implementations (i.e., up to several orders of magnitude faster). This is demonstrated by (E1) which compares the performance of naive implementations with HECO-produced code and is described in Section 6.1 of the paper, with results highlighted in Figure 5.*

(C2): *HECO produces code that matches the performance of "optimally' batched code. This is shown by the second half of (E1) which compares HECO to the synthesis-based Porcupine tool and is described in Section 6.2 of the paper, with results highlighted in Figure 6.*

(C3): *HECO's solution scales to real-world problem sizes (which synthesis-based tools fail to do). This is shown by (E2), which shows HECO's compile time for various problem sizes and is described in Section 6.1, with results shown in Table 1.*

A.4.2 Experiments

In the following, we describe the experiments. Note that all time estimates assume the set-up process (which includes the potentially lengthy compilation of the LLVM/MLIR dependency) has been completed. Detailed instructions can also be found in `evaluation/README.MD`.

(E1): *Speedup* (`evaluation/benchmark`)
In order to reproduce the results of Figure 5, the inputs need to be first compiled using HECO and then run using the Microsoft SEAL library. In addition, the naive baseline implementations need to be run using SEAL, too. This requires the vast majority of the (compute-)time, as the naive baseline implementations quickly become significantly less efficient (i.e., over 15min for a single problem, compared to fractions of a second for the HECO optimized version). Running this experiment should require around 15 human-minutes and no more than 2 compute-hours.
Preparation: *In order to compile the programs from the high-level intermediate representation (HIR) form given here to `*.cpp`, you can use a helper script that does this for all files in the `heco_input` folder (assuming your current working directory is the repository root):*
`./evaluation/benchmark/heco_helper.sh`
You can then compile and build the `benchmark` target (assuming your current working directory is the repository root): `cmake -build build -target benchmark`
Execution: *Execute the generated binary (`./build/bin/benchmark`). This will*

create a set of `*.csv` files of the format `<workload>_HECO_<size>.csv` in `evaluation/plotting/data/benchmark`.

Results: *The `*.csv` files report one iteration on each line, reporting key generation time, encryption time, evaluation time, and decryption time (in this order) in microseconds. In `evaluation/plotting/plot_all.py` a rough plotting script is provided, including a pipfile that defines the necessary dependencies. In addition, the plotting requires LaTeX to be installed.*

(E2): *Comparison* (`evaluation/comparison`)

In order to reproduce the results of Figure 6, the procedure is similar to that for Experiment 1. However, in addition to the HECO versions and naive baseline implementations, there are also Porcupine reference implementations. As in the previous experiment, the naive baselines consume the vast majority of the compute time. Running this experiment should require around 15 human-minutes and no more than 2 compute-hours.

Preparation: *In order to compile the programs from the high-level intermediate representation (HIR) form given here to `*.cpp`, you can use a helper script that does this for all files in the `heco_input` folder (assuming your current working directory is the repository root):* `./evaluation/comparison/heco_helper.sh`

You can then compile and build the `comparison` target (assuming your current working directory is the repository root): `cmake -build build -target comparison`

Execution: *Execute the generated binary (`./build/bin/comparison`). This will create a set of `*.csv` files of the format `<workload>_HECO_<size>.csv` in `evaluation/plotting/data/comparison`.*

Results: *The `*.csv` files report one iteration on each line, reporting key generation time, encryption time, evaluation time, and decryption time (in this order) in microseconds. In `evaluation/plotting/plot_all.py` a rough plotting script is provided, including a pipfile that defines the necessary dependencies. In addition, the plotting requires LaTeX to be installed.*

(E3): *Compile Time* (`evaluation/compile_time`)

In order to reproduce the results of Table 1, the programs need to be compiled with the `mlir-timing` option. Running this experiment should require around 30 human-minutes.

Preparation: *No additional preparation is required.*

Execution: *Compile each of the provided HIR inputs with the timing flags `–mlir-timing –mlir-timing-display=list` added. You can use a helper script that does this for all files in the `heco_input` folder (assuming your current working directory is the repository root):* `./evaluation/compile_time/heco_helper.sh`

Results: *The execution time report includes the Total Execution Time, which can be compared to the results in the paper. Note that a significant fraction of the compile time is usually spend in the `Canonicalizer` pass, which is a built-in pass from MLIR. As a result, compile times might vary significantly as MLIR updates and changes the underlying framework. Please also note that MLIR must be built in Release configuration in order to achieve acceptable compile time performance.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: A Verified Confidential Computing as a Service Framework for Privacy Preservation

Hongbo Chen[1], Haobin Hiroki Chen[1], Mingshen Sun[2], Kang Li[3],
Zhaofeng Chen[3], and XiaoFeng Wang[1]

[1]*Indiana University Bloomington, {hc50,haobchen,xw7}@iu.edu*
[2]*Independent Researcher, bob@mssun.me*
[3]*CertiK, {kang.li, zhaofeng.chen}@certik.com*

A Artifact Appendix

A.1 Abstract

We propose a security protection principle for confidential computing, Proof of Being Forgotten (PoBF). It has two requirements: NOLEAKAGE and NORESIDUE. These properties are formalized and proven under an abstract model for Trusted Execution Environment (TEE) in Coq. On the other hand, we implement a prototype PoBF-Compliant Framework (PoCF), which provides a framework to conduct Confidential Computing as a Service (CCaaS). These prototypes come with a verifier that can prove some properties specified in PoBF are satisfied. Besides, PoCF can support various real-world applications and the protections introduced in PoCF incur minor runtime performance overhead.

A.2 Description & Requirements

Reproducing the exact experiment results needs special hardware support: processors with Intel SGX and AMD SEV instruction extension support.

A.2.1 Security, privacy, and ethical concerns

Our artifacts come with no security, privacy, or ethical concerns. However, since building it requires plenty of dependencies and runtimes, we suggest reproducing the experiments in a non-production environment. We also provide access to an experiment virtual and/or physical machine.

A.2.2 How to access

Our code is published on GitHub and can be accessed via the link: `https://github.com/ya0guang/PoBF/tree/usenix-sec-ae`.

A.2.3 Hardware dependencies

The SGX-related experiments require an Intel processor with SGX instruction extension, and SEV-related experiments require an AMD processor with SEV instruction extension. Running the multi-threading test requires at least 32GB RAM (EPC Size) and we recommend using servers with at least 64GB RAM.

A.2.4 Software dependencies

PoCF requires dependencies from the system software and many dependencies for different platforms, so we recommend following the build instructions in our `README.md` in the GitHub repository or **just running the script located under the root directory, named `setup.sh`**. We only list the general software dependencies here.

Common Dependencies.

- Linux OS, preferably `Ubuntu 20/22.04 LTS`
- Rust `nightly-2022-11-01`
- `python3`: $\geq$ v3.8, $<$ v3.11
- `jupyter` notebook
- `mirai` abstract interpreter (version recorded in the script)
- `prusti` verifier (version recorded in the script)
- `tvm` v0.12.0
- `llvm` $\geq$ v10.0
- `Coq` proof assistant $>$ v8.13

SEV Specific Dependencies.

- Azure's SEV Guest attestation library. See `https://github.com/Azure/confidential-computing-cvm-guest-attestation`.

SGX Dependencies.

- SGX Driver, not needed if kernel > 5.11
- SGX SDK v2.17.101
- Teaclave Rust SGX SDK and Intel SGX SDK for Linux
- Attestation dependencies, including `aesm`, `dcap` or `epid`
- (*Optional*) other TEE frameworks: `enarx`, `gramine`, `occlum`

A.2.5 Benchmarks

Our benchmark data or the build scripts are included in the repository. We develop several confidential computing tasks that may require specific data and/or models. Workloads are all in `cctask/` folder, and the corresponding data (generators) are `data/`.

- `tvm` task requires `resnet152` and a picture input.
- `db` requires a `ycsb` client for generating, loading, and querying the database. This client is a submodule in our repository.
- `fasta` and `fann` requires generated sequence and an input number (serialized as a little-endian byte array) respectively.
- Other tasks require a dummy data payload.

A.3 Set-up

A.3.1 Installation

Please follow the `README.md` at https://github.com/ya0guang/PoBF/tree/usenix-sec-ae to set up the software environment. If you use the (virtual) machine provided by us, you can ignore this step.

A.3.2 Basic Test

- To make sure SGX functions correctly, check `sudo service aesmd status` and confirm it is successfully serving. Also, you should see SGX-related devices when using `ls /dev | grep sgx`.
- If TVM is installed and configured properly, one should be able to compile the TVM library under `cctasks/evaluation_tvm/model_deploy`. It can be checked by simply executing `make -j`.
- Rust programs should work if compilation does not fail.
- Coq works if `coqc` command can be executed.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The Coq proof is machine-checked.
(C2): PoCF verifier works on the sample project.
(C3): The protections in the PoCF introduce minor runtime performance overhead compared to NATIVE setting.
(C4): The tasks are indeed supported by PoCF.

A.4.2 Experiments

Note: For the comprehensive instruction, please check `scripts/README.md` in our repository.

E1 and E2 are verification of PoCF, and we expect the experiment to be passing the verification. E3 and E4 are performance evaluations that can be performed on Intel SGX platform and/or AMD SEV platform. We have scripts for both single- and multi-threaded experiments. We expect that the protection introduced by POCF is minor compared to NATIVE.

(E1): [Coq Proof Checking] [3 minutes]: successfully compilation implies successful proof verification.
How to: Just compile the Coq source code.
Execution: Run `coqc *.v` at `pobf_proof/`
Results: Successful compilation.

(E2): [PoCF Verification] [5 human-minutes + 30-60 compute-minutes]: Verify the implementation of PoCF. The verifier invokes `mirai` and `prusti` to conduct NOLEAKAGE and NORESIDUE checkings.
How to: Please follow the steps in `Verification towards PoCF` in our `README.md`.
Preparation: Install `mirai` and `prusti` using the scripts. This takes some time to compile from the source code.
Results: There are two cases for a negative case and a positive case: one contains threats and one does not. One can try to remove the `verified_log!` in the source file `src/userfunc.rs` to see the difference

(E3): [Overhead Analysis (microbenchmarks)] [15 human-minutes + 1 compute-hour + 1GB disk]: Confirm the overhead introduced by PoCF protections is minor.
How to: First, compile and run the microbenchmarks. Then analyze the data and generate the figure. This evaluation is performed in single- and multi-threading scenarios.
Preparation: Compile the microbenchmark tasks.
Execution: Run the evaluation scripts that can be found under `scripts/evaluation.sh`. Remember to set the `task` variable to the task `polybench`. It will perform 10 repetitions (or more if you need). The cost breakup results would be automatically printed to the console if you are evaluating the PoBF task. For the stack page number microbenchmark, please refer to `README.md`.
Results: First execute the code in the Python notebook

under `scripts/figure.ipynb`. The script draws the figures that show the performance of PoCF and NATIVE on different tasks. We expect the performance (execution time) of PoCF to be better. For other microbenhmarks, please copy-and-paste the results to the Jupyter notebook and visualize them.

(E4): [Real-world application (macrobenchmarks)] [30 human-minutes + 1 compute-hour + 5GB disk]: Confirm the overhead introduced by PoCF protections is minor.
How to: First compile and run the confidential computing tasks. Then analyze the data and generate the figure. This evaluation is also performed in single- and multi-threading scenarios.
Preparation: Compile the macrobenchmark tasks.
Execution: For the KVDB task: Please execute the YCSB client that is included as a submodule in the repository. Follow the instruction in the `README.md`. You may need to first load the data by the workload `workload/load.toml` and then execute the corresponding workload A and C. For other tasks: Please execute the evaluation script `scripts/evaluation.sh`.
Results: Execute code in the Python notebook `scripts/db.ipynb`, `scripts/figure.ipynb` The script draws the figures that show the performance of PoCF and NATIVE on different tasks. We expect the performance (execution time) of PoCF to be better. For the results collected from YCSB, please copy-and-paste the results to the plotting script.

A.5 Notes on Reusability

- If you want to modify the state transitions, edit `pobf_state/src/task.rs`.
- If you want to add/modify the CC Task, follow the examples in `cctasks`. You may also want to modify the build script.
- You could also change the verification options by modifying `pobf_verifier/pobf-verify`

Note that our artifact is an academic project. Any use of the code should adhere to the license.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Precise and Generalized Robustness Certification for Neural Networks

[1,2]Yuanyuan Yuan, [1]Shuai Wang, and [2]Zhendong Su
[1]The Hong Kong University of Science and Technology, [2]ETH Zurich
yyuanaq@cse.ust.hk, shuaiw@cse.ust.hk, zhendong.su@inf.ethz.ch

A Artifact Appendix

A.1 Abstract

We provide code and data of our paper in this artifact. Our artifact is publicly available at `https://github.com/Yuanyuan-Yuan/GCert` with detailed documents. Using our tool, users can certify neural network robustness towards various semantic-level mutations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None

A.2.2 How to access

An archived copy of the initial version is available at: `https://zenodo.org/record/8062051`.

Our artifact is actively maintained at: `https://github.com/Yuanyuan-Yuan/GCert`.

A.2.3 Hardware dependencies

We do not have any particular requirements for the hardware. Our artifact may need GPUs to speed up the certification; we suggest evaluators having at least one GPU.

A.2.4 Software dependencies

Our tool is built based on Pytorch; evaluators need to first install Pytorch. See detailed instructions in our documents.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Users only need to install Pytorch first. See details in our documents.

A.3.2 Basic Test

To test the basic functionality, evaluators can first run `cd experiments` to change the current directory. Then run `python augment_geometrical.py`. This script will start training a generative model with regulation proposed in our paper.

Detailed instructions are provided in our documents.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: HOLMES: Efficient Distribution Testing for Secure Collaborative Learning

Ian Chang
UC Berkeley

Katerina Sotiraki
UC Berkeley

Weikeng Chen
Discreet Labs & UC Berkeley

Murat Kantarcioglu
University of Texas at Dallas & UC Berkeley

Raluca Ada Popa
UC Berkeley

A Artifact Appendix

A.1 Abstract

We present HOLMES, a protocol for performing secure distribution testing efficiently. Our artifact includes:

1. the efficiency comparison of HOLMES against various baselines;
2. the efficiency evaluation of HOLMES in real-world datasets;
3. the accuracy evaluation for HOLMES' statistical tests against corruptions to simulated and real-world data.

This artifact reproduces the tables and figures in our paper.

A.2 Description & Requirements

HOLMES allows efficient distribution testing in a multiparty setting without revealing the dataset of any of the parties. A distribution test is a predicate over an individual or joint (i.e., from multiple parties) dataset. Examples include well-known statistical tests, such as mean equality z-test (when the variance of the dataset is known) and t-test (when the variance is unknown), variance equality F-test, and Pearson's χ^2-test. These tests check a property between two populations, or between a population and a public distribution.

We support major statistical tests including the t-test, z-test, F-test, and the chi-squared test for single and multiple dimensions. We also provide support for computing and checking dataset properties essential in distribution testing; specifically, we support computing mean, trimmed mean, variance, histogram, random linear combination, and range check.

HOLMES integrates zero-knowledge proofs and secure multiparty computation with a lightweight consistency check. Specifically, HOLMES uses QuickSilver as a framework for zero-knowledge proofs and SCALE-MAMBA for the MPC computation. The codebase also includes integration tests, unit tests, and individual benchmarks.

A.2.1 Security, privacy, and ethical concerns

We assume that t parties want to participate in secure collaborative learning based on a t-party MPC protocol (e.g., SCALE-MAMBA). Before they engage in the learning protocol, the parties wish to check the quality of the dataset using distribution tests. HOLMES offers privacy in the dishonest and malicious majority setting, where at most $t-1$ out of t parties can collude and arbitrary deviate from the protocol.

Note that any distribution testing leaks one-bit information, i.e., whether the test passed or failed. Hence, it is important that a party does not participate in distribution tests that may leak sensitive information.

A.2.2 How to access

HOLMES is available (at a stable URL) here. The repository includes instructions for compiling HOLMES and reproducing our results.

The artifacts for reproducing our experiments and graphs are available (at a stable URL) here. We have also published the AMIs as public, and prepared scripts to automatically launch the clusters, so users can launch their own cluster on their own AWS account at here.

A.2.3 Hardware dependencies

HOLMES does not require any specialized hardware. Our experiments were performed on AWS c5.9xlarge instances, each with 36 cores and 72 GB memory. Different hardware configurations will affect the performance of HOLMES, but will result in a similar performance gain over the baselines.

A.2.4 Software dependencies

We provide the software dependencies for the plots and creating the AMI cluster in https://github.com/holmes-inputcheck/holmes-artifacts/. To install HOLMES only on a local machine, a user has to perform the following steps:

1. Install GMP https://gmplib.org/
2. Install MPFR https://www.mpfr.org/

3. Install FLINT https://www.flintlib.org/
4. Install emp-tool https://github.com/emp-toolkit/emp-tool
5. Install emp-ot https://github.com/emp-toolkit/emp-ot
6. Install emp-zk https://github.com/holmes-inputcheck/emp-zk
7. Clone and compile HOLMES https://github.com/holmes-anonymous-submission/holmes-library

A.2.5 Benchmarks

We provide benchmarks for the following tasks.

- Efficiency comparison for the histogram, mean, variance, and trimmed mean of HOLMES with the generic MPC baseline;
- Overhead comparison of range checks and ZK-friendly sketching with three baselines; these are the two most expensive gadgets supported in HOLMES;
- Overhead of running sample distribution tests on a real-world dataset from bank marketing. This dataset is provided in https://github.com/holmes-inputcheck/holmes-library, and is pre-cleaned and provided as CSV files in https://github.com/holmes-inputcheck/holmes-library/blob/master/bench/dataset[1-3].csv;
- Computing the accuracy of HOLMES' distribution tests against specific types of corruptions on simulated and real-world dataset.

A.3 Set-up

In this section, we provide information about setting up and running the artifacts.

A.3.1 Installation

We provide instructions on how to install the dependencies and necessary configuration steps in https://github.com/holmes-inputcheck/holmes-artifacts/.

A.3.2 Basic Test

We provide instructions for running unit tests on all the statistical tests of HOLMES in your AMI cluster located here, or on your local machine, given that you have all of the prerequisites installed, located here.

The instructions for running the integration tests, which measure the overhead of the dataset testing workflows, for your AMI cluster are provided in here, or on your local machine, given that you have all of the prerequisites installed, located here.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): HOLMES achieves a speedup of up to 10x for classical distribution tests over the generic MPC baseline with $t = 2$ parties. This is proven by experiment (E1). This result is described in Section 4.4 of the full version of paper and is illustrated in Figures 7a-f.

(C2): HOLMES achieves a speedup up to 10000x for its ZK-friendly sketching multidimensional tests over the strawman one-hot encoding multidimensional tests with $t = 2$ parties. This is proven by experiment (E2). This result is described in Section 4.4.2 of the full version of paper and is illustrated in Figures 7g-h.

(C3): The generic MPC baseline is 10–256x and 35–198x slower than HOLMES (i.e., QuickSilver, which is the underlying IZK protocol in HOLMES) for the range check and the ZK-friendly sketching, respectively, with $t \in \{2,6,10\}$ parties. The pairwise 2PC baseline is 4–32x slower for the range check and 13–36x slower for the ZK-friendly sketching than HOLMES. $\text{Spartan}_{\text{NIZK}}$ is 1–16× slower for the range check and 4–45x slower for the ZK-friendly sketching than HOLMES. This is proven by experiment (E3). This result is described in Section 4.4.1 of the full version of paper and is illustrated in Table 1.

(C4): HOLMES' chi-squared test has approximately the same accuracy as the naive normalized and unnormalized chi-squared test. This is proven by experiment (E4). This result is described in Section 4.3 of the full version of paper and is illustrated in Figure 5.

(C5): HOLMES's approach outperforms the generic MPC baseline by 77–264x for a real-world testing workflow on the bank marketing dataset for $t \in \{2,6,10\}$ parties. This is proven by experiment (E5). This result is described in Section 4.4.3 of the full version of paper and is illustrated in Figure 6 and Table 2.

A.4.2 Experiments

(E1): Classic distribution tests for two parties (20 human-minutes + 2 compute-hours + 72GB disk): This experiment measures the overhead of classic distribution tests, i.e., the naive histogram check, the trimmed mean check, the mean check, and the variance check, on a fake dataset of all ones. We compare the overhead between the two setups: HOLMES and SCALE-MAMBA.

Preparation: Perform the set up as described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#setup.

Execution: Use the designated scripts described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#misc-bench-scripts-experiment-e1–e2 to run the benchmarks and retrieve

the results.

Results: To interpret the results, run the designated scripts described in https://github.com/holmes-inputcheck/holmes-artifacts#classical-distribution-tests-experiment-e1. These scripts produce six figures, one for the cost in HOLMES and one for the cost in SCALE-MAMBA, for the following tests: histogram, trimmed mean, mean and variance. The cost of histogram check is plotted for 10 buckets with varying range sizes and input sizes. The cost of trimmed mean is plotted for datasets with 100k and 200k entries with varying threshold θ. The cost of mean and variance is summed and plotted for varying dataset sizes from 1 million entries to 5 million entries. This experiment supports claim (C1).

(E2): HOLMES' Multidimensional Test vs. Naive Multidimensional Test in HOLMES for two parties (20 human-minutes + 2 compute-hours + 72GB disk): This experiment measures the overhead of multidimensional tests on a fake dataset of all ones. We compare the overhead between two approaches for the multidimensional χ^2-test for the canonical two-party case with HOLMES (QuickSilver). We show that using our ZK-friendly sketching approach to compute the χ^2-test is much more efficient than the standard strawman approach of naively computing the one-hot encoding for each multidimensional input.

Preparation: Perform the set up as described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#setup.

Execution: Use the designated scripts described in https://github.com/holmes-inputcheck/holmes-artifacts#misc-bench-scripts-experiment-e1–e2 to run the benchmarks and retrieve the results.

Results: To interpret the results, run the designated scripts described in https://github.com/holmes-inputcheck/holmes-artifacts#holmes-multidimensional-test-vs-naive-multidimensional-test-experiment-e2. These scripts produce two figures showing the computational cost with respect to the number of dimensions and with respect to the number of individual labels in each dimension. In the first figure, the baseline is the naive multidimensional χ^2-test, whereas HOLMES uses the ZK-friendly sketching multidimensional χ^2-test. We plot the cost for datasets with 100k, 200k, 500k entries and 10 individual labels per dimension with varying number of dimensions. In the second figure, the cost of the multidimensional χ^2-test in SCALE-MAMBA and in HOLMES is plotted for datasets with 100k, 200k, 500k entries and four dimensions with varying number of labels. This experiment supports claim (C2).

(E3): Efficiency comparison of range checks and ZK-friendly sketching against the baselines (30 human-minutes + 20 compute-hours + 72GB disk): This experiment measures the overhead of range check and ZK-friendly sketching on a fake dataset of all ones. We compare the overhead between the following setups: HOLMES (i.e., QuickSilver), t-party SCALE-MAMBA, pairwise 2-party SCALE-MAMBA, $\text{Spartan}_{\text{NIZK}}$ for datasets with 100k, 200k, and 500k entries. For experiments that take too long or require more memory than available, e.g. 500k entries of ZK-friendly sketching for SCALE-MAMBA and $\text{Spartan}_{\text{NIZK}}$, we perform the ZK-friendly sketching for a smaller number of entries and extrapolate to larger entries using Euler's method. $\text{Spartan}_{\text{SNARK}}$ is only plotted for up to 100k entries.

Preparation: Perform the set up as described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#setup.

Execution: Use the designated scripts described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#range-checks-and-zk-friendly-sketching-against-the-baselines-experiment-e3 to run the benchmarks and retrieve the results.

Results: To interpret the results, run the designated scripts described in https://github.com/holmes-inputcheck/holmes-artifacts#range-checks-and-zk-friendly-sketching-against-the-baselines-experiment-e3. These scripts compute the cost for each setup. We produce a csv file, which can be compared with Table 2 and Table 3 in the paper. For each setup, we compute the cost for datasets with 100k, 200k, 500k entries for $t \in \{2,6,10\}$ parties. This experiment supports claim (C3), and as expected, QuickSilver runs the fastest in all setups.

(E4): Accuracy of distribution tests against corrupted datasets (5 human-minutes + 3 compute-hours): This experiment gradually corrupts up to 30% of the input dataset, and plots the statistical p-value of various tests as a function of the percentage of input corruptions.

Preparation: Perform the set up as described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#setup.

Execution: Use the designated scripts described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#statistical-corruption-accuracy-graphs-experiment-e4 to run the benchmarks and retrieve the results.

Results: To interpret the results, run the designated scripts described in https://github.com/holmes-inputcheck/holmes-artifacts#statistical-p-value-accuracy-graphs-experiment-e4. These scripts produce two plots: one for a simulated dataset and one for the bank marketing dataset. The corruption model is described in Section 4.3 of the full version of the paper. Due to randomness in sampling the dataset before corruption, the initial p-values might vary; in expectation, the chi-squared tests hit the p-value of 0.05

before all other tests, the z-test and t-test hit the p-value of 0.05 next followed by the F-test. This experiment supports claim (C4).

(E5): Marketing dataset overhead and cost breakdown (5 human-minutes + 1 compute-hour): This experiment measures the overhead of HOLMES and SCALE-MAMBA on the bank marketing dataset for $t \in \{2, 6, 10\}$ parties.

Preparation: Perform the set up as described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#setup.

Execution: Use the designated scripts described in the following link https://github.com/holmes-inputcheck/holmes-artifacts#marketing-dataset-testing-workflow-benchmarking-experiment-e5 to run the benchmarks and retrieve the results.

Results: To interpret the results, run the designated scripts described in https://github.com/holmes-inputcheck/holmes-artifacts#marketing-dataset-graphs-holmes-vs-mpc-baseline-experiment-e. These scripts produce a figure for the computational overhead as a function of the number of parties and a file with the breakdown of the cost. This experiment supports claim (C5).

A.5 Notes on Reusability

HOLMES' assumes that the highest degree of security (malicious security) is required and is best applied when all but one of the parties are untrusted. The parties most practically represent a powerful entity with lots of data and computing power. In example, competing bank conglomerates might want to jointly train their data over a specific model but do not trust each other, and are reasonably confident that other competing banks will collude. In this setting, HOLMES can securely perform distribution tests and securely compute aggregate statistics and analytics in a much faster speed than previous multiparty computation techniques.

HOLMES is flexible such that any future developer who chooses to use their own dataset, add their own custom checks, or add their own distribution tests can do so easily. An exciting future direction of HOLMES is that parties who wish to jointly train a model over their data can implement checks that prevent data poisoning attacks, such as algorithms from robust statistics. We encourage future users and developers to implement their own checks through HOLMES and use the existing checks to expedite the secure computation over their own selection of data.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

Exploring the Unknown DTLS Universe: Analysis of the DTLS Server Ecosystem on the Internet

Nurullah Erinola[1], Marcel Maehren[1], Robert Merget[2], Juraj Somorovsky[3], and Jörg Schwenk[1]

[1]Ruhr University Bochum
[2]Technology Innovation Institute
[3]Paderborn University

A Artifact Appendix

A.1 Abstract

TLS-Scanner is an open-source tool to assist pentesters and security researchers in evaluating TLS server implementations. It automatically scans a TLS server and provides a report of supported features like protocol versions, cipher suites, extensions, and potential security issues.

In our work, we extended TLS-Scanner with support for DTLS and implemented additional tests specifically designed to evaluate DTLS-specific features. Subsequently, we evaluated twelve open-source DTLS server implementations and uncovered eleven security vulnerabilities. We then proceeded to scan publicly available DTLS servers to gain detailed insights into the publicly accessible DTLS server landscape.

Artifact users can reproduce the results of our lab evaluation by running TLS-Scanner against the respective DTLS server implementations.

A.2 Description & Requirements

Upon completion of the scan, TLS-Scanner provides a comprehensive report containing detailed information regarding the server's configuration and its security-relevant properties. We provide the source code of the extended TLS-Scanner and Docker files of OpenSSL and Mbed TLS as artifacts, enabling the testing of TLS-Scanner.

A.2.1 Security, Privacy, and Ethical Concerns

We are not aware of any exploitable issues in TLS-Scanner. TLS-Scanner only establishes multiple DTLS connections to the server under test. However, depending on the number of threads, it is possible to overwhelm the server. Therefore, by default, the scan is performed with one thread. Note that such a scan will inevitably reveal your IP address to the tested server.

TLS-Scanner can also be used to scan servers owned by other people. Depending on your local jurisdiction, it may be illegal for you to do so. Additionally, conducting scans on public servers should follow the best practices for Internet-wide scanning setup by Durumeric et al. [1].

A.2.2 How To Access

Our artifact can be found on GitHub at `https://github.com/tls-attacker/Exploring-the-Unknown-DTLS-Universe/tree/563b9ca12920eed26b00f518fe7465b2b833024e`. The repository includes the source code of the extended TLS-Scanner and example Docker files which build open-source DTLS server implementations (OpenSSL and Mbed TLS).

A.2.3 Hardware Dependencies

None.

A.2.4 Software Dependencies

TLS-Scanner is written in Java. This requires Maven and Java 11 to be installed. To run the example servers in the docker containers, Docker is required. We tested this artifact on Kubuntu 22.04, but any Linux system should work. TLS-Scanner should also run on Windows (but the docker examples will not).

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Downloading the Artifact. Clone the GitHub repository using *git*:

```
https://github.com/tls-attacker/\
Exploring-the-Unknown-DTLS-Universe/tree/\
563b9ca12920eed26b00f518fe7465b2b833024e
```

Installing Java and Maven. Install Java 11 and Maven using *apt*:

```
sudo apt install openjdk-11-jdk
sudo apt install maven
```

To verify if the dependencies are installed and set up correctly, run the following commands:

```
java -version
mvn -version
```

If everything works correctly, both commands should display their respective versions. Additionally, it is important to ensure that the correct version of Java is specified for Maven.

Installing Docker. Install Docker by following the instructions at https://docs.docker.com/engine/install/.

Setting up TLS-Scanner.

a) Using the provided Dockerfile:

1. Navigate to `tls-scanner/`
2. Run

   ```
   docker build --tag tls-scanner --file \
   dockerfile-tls-scanner .
   ```

3. Run `docker images`

   If everything works correctly, the last command should print the names of all available docker images on your system. The output should contain *tls-scanner*.

b) Building TLS-Scanner yourself:

1. Navigate to `tls-scanner/TLS-Attacker/`
2. Execute `mvn clean install`
3. Navigate to `tls-scanner/TLS-Scanner/`
4. Execute `mvn clean package`

   If everything compiles correctly, the `apps\` folder should now contain the `TLS-Server-Scanner.jar` file.

Building the Server Implementations.

1. Navigate to `libraries/`
2. Execute `setup.sh`
3. Run `docker images`

If everything works correctly, the last command should again print the names of all available docker images on your system. The output should contain the names of the server implementations (e.g., *openssl-dtls-server* or *mbedtls-dtls-server*).

A.3.2 Basic Test

Testing TLS-Scanner. To verify that the TLS-Scanner can be executed correctly, run the following command:

```
docker run --rm --network="host" --name \
tls-scanner tls-scanner -help
```

If everything works correctly this should print the parameter list of TLS-Scanner with usage instructions.

Testing the Server Implementations. To verify that the docker images can be used, run the following command:

```
docker run --rm --network="host" --name \
mbedtls-dtls-server mbedtls-dtls-server \
server_port=4433 dtls=1
```

If everything works correctly this should start the example server of Mbed TLS.

A.4 Evaluation workflow

Running TLS-Scanner on a DTLS server implementation is straightforward. Typically, the DTLS server is started, then TLS-Scanner is started. TLS-Scanner will then perform the scan without interaction from the user. After completing the scan, TLS-Scanner will output the results which the user can analyze.

A.4.1 Major Claims

We claim to be able to evaluate various DTLS-specific features of a given server, including potential DoS vulnerabilities. Specifically, we claim to evaluate the properties of twelve open-source DTLS server implementations. In the following, we exemplary describe our claims for OpenSSL and Mbed TLS:

(C1): OpenSSL issues 20 byte long cookies and deviates from the recommended cookie computation.

(C2): OpenSSL does not perform a cookie exchange upon renegotiation.

(C3): OpenSSL allows users to implement *no* cookie exchange, a *stateful* cookie exchange, or a *stateless* cookie exchange. The example server uses the stateful cookie exchange by default, but stateless mode can be requested through command line parameters. To mitigate DoS attacks, the stateless mode should be used when deploying OpenSSL in production.

(C4): Mbed TLS issues 32 byte long cookies and deviates from the recommended cookie computation.

(C5): Mbed TLS supports the fragmentation of messages after the cookie exchange is successfully completed.

The experiment (E1) will demonstrate (C1)-(C3) while (C4)-(C5) are demonstrated by the experiment (E2). The results of the two experiments are summarized in our paper in Table 1 with a detailed explanation in Section 5.

A.4.2 Experiments

Since TLS-Scanner is designed to scan only a single server, we define one scan per experiment.

(E1 - OpenSSL) *(5 human-minutes + 10 compute-minutes)* In this experiment, the OpenSSL example server is scanned with TLS-Scanner. TLS-Scanner will then output a report.

1. Start the OpenSSL server

```
docker run --rm -v [absolute path to \
libraries/certs/]:/certs/ \
--network="host" --name \
openssl-dtls-server openssl-dtls-server \
-key /certs/private_key.pem -cert \
/certs/certificate.pem -accept 4433 -dtls
```

2. Start the evaluation with TLS-Scanner

```
docker run --rm --network="host" --name \
tls-scanner tls-scanner -connect \
localhost:4433 -dtls -timeout 100
```

TLS-Scanner will now perform a series of DTLS handshakes. After that, the report should be visible.

Results.
(C1): The report should contain a section `DTLS Hello Verify Request`. It summarizes the behavior the server showed against the implemented cookie exchange tests. There you can see which client parameters influence the cookie computation. For OpenSSL, it should contain:

```
DTLS Hello Verify Request

HVR Retransmissions           : false
Cookie length                 : 20
Checks cookie                 : true
Cookie is influenced by
-ip                           : not tested yet
-port                         : true
-version                      : false
-random                       : false
-session id                   : false
-cipher suites                : false
-compressions                 : false
```

To confirm (C1), the `Cookie length` field should have the value `20`. In addition, the `-version`, `-random`, `-session id`, `-cipher suites`, and `-compressions` fields should have `false`. Please note that evaluating if the IP influences the cookie computation requires a proxy running on a host with a different IP.

(C2): The report should contain a section `Renegotioation`. It summarizes whether the server supports renegotiation and whether cookie exchange is performed there. For OpenSSL, it should contain:

```
Renegotioation

Secure (Extension)            : true
Secure (CipherSuite)          : true
Insecure                      : false
DTLS cookie exchange in renegotiation : false
```

To confirm (C2), the `DTLS cookie exchange in renegotiation` field should have `false`.

(C3): The report should contain a section `DTLS Fragmentation`. It summarizes the behavior the server showed against the implemented fragmentation tests. For OpenSSL, it should contain:

```
DTLS Fragmentation

Supports fragmentation        : true
Supports fragmentation with individual transport packets :
    ↪ true
```

To confirm (C3), both fields should have `true`.

(E2 - Mbed TLS) *(5 human-minutes + 10 compute-minutes)* In this experiment, the Mbed TLS example server is scanned with TLS-Scanner. TLS-Scanner will then output a report.

1. Start the Mbed TLS server

```
docker run --rm --network="host" --name \
mbedtls-dtls-server mbedtls-dtls-server \
server_port=4433 dtls=1
```

2. Start the evaluation with TLS-Scanner

```
docker run --rm --network="host" --name \
tls-scanner tls-scanner -connect \
localhost:4433 -dtls -timeout 100
```

TLS-Scanner will now perform a series of DTLS handshakes. After that, the report should be visible.

Results.
(C4): Similar to (C1). For Mbed TLS, the `DTLS Hello Verify Request` section should contain:

```
DTLS Hello Verify Request

HVR Retransmissions           : false
Cookie length                 : 32
Checks cookie                 : true
Cookie is influenced by
-ip                           : not tested yet
```

```
-port                          : false
-version                       : cannot be tested
-random                        : false
-session id                    : false
-cipher suites                 : false
-compressions                  : false
```

To confirm (C4), the `Cookie length` field should have the value `32`. In addition, the `-port`, `-random`, `-session id`, `-cipher suites`, and `-compressions` fields should have `false`. Please note that evaluating if the version influences the cookie cannot be executed for Mbed TLS because it supports only one protocol version (DTLS 1.2).

(C5): Similar to (C3). For Mbed TLS, the `DTLS Fragmentation` section should contain:

```
DTLS Fragmentation

Supports fragmentation       : partially
-After cookie exchange
Supports fragmentation with individual transport packets :
    ↪ partially
-After cookie exchange
```

To confirm (C5), both fields should have `partially`.

A.5 Notes on Reusability

Our extensions to TLS-Scanner have been merged in the project and released with v5.2.5. It can be found on GitHub at https://github.com/tls-attacker/TLS-Scanner/releases/tag/v5.2.5. In addition, our extensions to TLS-Attacker which is used by TLS-Scanner have been merged in the project and are contained in the latest release v5.2.1. It can be found on GitHub at https://github.com/tls-attacker/TLS-Attacker/releases/tag/v5.2.1.

To perform large-scale scans with TLS-Scanner, TLS-Crawler can be used. It utilizes multiple TLS-Scanner instances to scan a large number of servers in parallel and write the results to a database. The latest release v1.0.1 can be found on GitHub at https://github.com/tls-attacker/TLS-Crawler/releases/tag/v1.0.1.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

References

[1] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *22nd USENIX Security Symposium*, 2013.

USENIX'23 Artifact Appendix: We Really Need to Talk About Session Tickets: A Large-Scale Analysis of Cryptographic Dangers with TLS Session Tickets

Sven Hebrok[1], Simon Nachtigall[1,3], Marcel Maehren[2], Nurullah Erinola[2], Robert Merget[4,2], Juraj Somorovsky[1], and Jörg Schwenk[2]

[1]Paderborn University
[2]Ruhr University Bochum
[3]achelos GmbH
[4]Technology Innovation Institute

A Artifact Appendix

A.1 Abstract

We performed a large-scale analysis of TLS session tickets. To this end, we extended TLS-Scanner with further tests which detected a variety of weak keys being used in the wild. To support developers and server administrators when evaluating their session ticket configurations, we publish our extension for TLS-Scanner. This extension is able to detect the vulnerabilities discussed in our paper. Within this artifact evaluation, we show that our extension is capable of detecting the vulnerabilities from our paper.

A.2 Description & Requirements

TLS-Scanner is an established tool to scan TLS servers for weaknesses. We implemented new probes to scan for weaknesses related to session tickets. These perform the tests outlined in our paper. Additionally, we provide a test server to test the scanner against (not part of the artifact, but used to evaluate it). For both tools we provide the source code, Dockerfiles, and Docker images. We recommend using the Docker images, as they ensure a reproducible environment.

A.2.1 Security, privacy, and ethical concerns

We are not aware of any exploitable issues in the tool. It should be secure to run on your machine. To evaluate a TLS server, TLS-Scanner needs to connect with that server, effectively revealing your IP address. The scanner also sends an HTTP request to the server which includes TLS-Attacker as the user agent.

The scan initiates multiple TLS connections to the server. Depending on the number of threads (option `-threads`) you could overwhelm the server. In our scans, we ensured this does not happen by scanning multiple servers at once with a shared limited threadpool (done by TLS-Crawler[1]). When using the scanner, use a low number of threads. The test server we provide is single threaded, so only a single (connection) thread should be used there.

Any data our probes exfiltrate from the ticket is already known to the scanner. However, if it is able to exfiltrate data from a ticket, this indicates a severe issue (as outlined in our paper) that you should report to the server administrator.

Test Server The test server uses an outdated version of BoringSSL and adds further parameters to the included server. The ticket decryption is implemented such that it is vulnerable to padding oracle attacks if the authenticity of the ticket is not ensured (e.g., no MAC). We recommend running it inside an isolated environment.

A.2.2 How to access

We provide a repository summarizing our artifact at https://github.com/tls-attacker/We-Really-Need-to-Talk-About-Session-Tickets/tree/ad64fe34f41894f1aa5bbec65cf0446cdb0ad3f8. This includes the TLS-Scanner source code[2] and a testserver. Further, this also contains the Dockerfiles to build Docker images of both components yourself. Alternatively, you can get the Docker image from Docker Hub.[3] Within this document, we describe how to run the scanner using maven and the server using docker. Additional topics, such as using the scanner with Docker or the server from source, are covered in the artifact repo's readme.

[1]https://github.com/tls-attacker/TLS-Crawler

[2]https://github.com/tls-attacker/TLS-Scanner/commit/a0d4c1a910f0eb3e5ee3e28c9435818820c67919

[3]snhebrok/tls-scanner-ae and snhebrok/vulnerable-bssl

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

For TLS-Scanner a Java 11 development kit and maven are required.[4] To run the server using Docker, Docker needs to be installed. You can also build the image yourself using the Dockerfile contained in our repository or pull it from Docker Hub.

A.2.5 Benchmarks

None.

A.3 Set-up

We provide multiple ways to set up the scanner and test server. Within this document we cover building the scanner from source using maven.

A.3.1 Installation

Using Source and Maven You need to clone the code and then compile it using maven. This automatically fetches all dependencies.

```
git clone --recurse-submodules
 ↪ https://github.com/tls-attacker/
 ↪ We-Really-Need-to-Talk-About-Session-Tickets
cd We-Really-Need-to-Talk-About-Session-Tickets
git checkout c71fb839bd4ad2dc00cbb1a578d7d4254f8aec17
mvn clean package
```

A.3.2 Basic Test

To verify that the scanner is initialized correctly run the scanner without any arguments:

```
cd TLS-Scanner/apps
java -jar TLS-Server-Scanner.jar
```

If everything works correctly this should print an error stating that the provided parameters could not be parsed (including a stack trace). This error message should state that the option `-connect` is required and also include a stack trace. The available options should be printed afterward.

If you pass the `-connect` flag followed by a host, the specified host should be scanned.

If an error occurs, ensure you are using Java 11 to build and run the project. Other versions usually fail.

[4] More information about the TLS-Attacker projects and their requirements can be found under https://github.com/tls-attacker/TLS-Attacker-Description.

A.3.3 Test Server

The Docker image is available at Docker Hub as `snhebrok/vulnerable-bssl`[5] with the tag `sessionticket-ae`. Running the image will automatically pull it. You can still pull it explicitly with `docker pull snhebrok/vulnerable-bssl:sessionticket-ae`.

A.4 Evaluation workflow

A.4.1 Major Claims

To evaluate the ecosystem in our paper, we used TLS-Scanner. We claim to be able to detect different vulnerabilities related to TLS session tickets:

(C1): We can detect a variety of default keys for encryption and authentication. This is shown in experiment (E1).

(C2): We can detect padding oracle vulnerabilities. This is shown in experiment (E2).

(C3): We can detect missing ticket authentication. This is shown in experiment (E2).

A.4.2 Experiments

TLS-Scanner scans a single server at a time. For our experiments we propose to scan our test server. You can also scan other servers, but these might not be vulnerable to our proposed vulnerabilities. All time estimates were created using a laptop with an i7-1165G7 and 32G of RAM. Each scan should take about five to ten minutes with the parameters we recommend below.

Basic Test Execution For each experiment we describe which parameters to pass to the test server for this experiment. After the test server is started, you need to run TLS-Scanner against the server. The scan might take some minutes and finishes by outputting the results. This also contains results not related to session tickets. The results related to session tickets are located in the section with the heading `SessionTicketEval`. We describe what this section should contain for each experiment. For all experiments, there are some parameters that you should always set, which we outline below.

Running the Scanner For all tests, you need to execute the scanner against the testserver as follows:

```
java -jar TLS-Server-Scanner.jar -connect
 ↪ [host] -scanDetail NORMAL
```

`[host]` is the host to scan (including port). When using the docker test server as specified, this should be `172.17.0.1:8000`.[6] The detail affects how many test vectors

[5] https://hub.docker.com/r/snhebrok/vulnerable-bssl

[6] Check whether docker assigns this IP to one of your network interfaces. You can also use any other IP assigned to your device.

are tested against the server. For our paper we used `DETAILED`, but all experiments work with `NORMAL` (the default value) as well.

Running the Test Server For the test server we recommend running it as follows:

```
docker run --rm  -it -p8000:8000
↪  snhebrok/vulnerable-bssl:sessionticket-ae
↪  s_server -accept 8000 -loop -www
```

Depending on the experiment, further parameters need to be added.

(E1) Detecting default keys:
[5 human-minutes + 5 compute-minutes]
Scan a server whether it uses one of our proposed weak keys. The scanner outputs the detected key and format of the ticket.
Test Server Preparation: The test server needs to use a weak key. Example parameters are

```
-ticketEnc AES-128-CBC
-ticketEncKey 00
-ticketHMac SHA256
-ticketHMacKey 00
-ticketHMacKeyLen 32
```

Results: The summary should contain the following lines:

```
Ticket use default STEK (enc) : true
Ticket use default STEK (MAC) : true
```

Further down is a section `Default STEK` which contains details about the detected keys for encryption and HMAC (if you set both groups of parameters). This includes the detected format, algorithm, and key. For encryption, this also contains which secret is included in the ticket.
No padding oracle or MAC check issues should be found as an Encrypt-then-Mac scheme is used (albeit with a weak key).
Within the repository's readme we describe how to manually verify this attack using OpenSSL.

(E2) Missing Authentication and Padding Oracles:
[5 human-minutes + 15 compute-minutes]
Scan a server whether it is not properly authenticating its tickets and even has a padding oracle vulnerability.
Test Server Preparation: The test server must not use authentication (HMAC) for the ticket. To detect a padding oracle vulnerability, a CBC cipher must be used. Example parameters are

```
-ticketEnc AES-256-CBC
-ticketHMac None
```

Results: The summary should contain the following lines:

```
No (full) MAC check : true
Vulnerable to Padding Oracle : true
```

Further down is a section `Manipulation`. This summarizes the behavior the server showed when inducing `bitflips` into a ticket. Several behaviors are preclassified:

- **A** The modified ticket was accepted. The authenticity of the ticket was hence not verified (completely).
- **#** The modified ticket was accepted, but key material unknown to the scanner was used. That is, the server recovered some corrupted key material from the modified ticket.
- **_** The modified ticket was rejected, and a normal handshake was performed. This should be the case if the authenticity of the ticket is properly ensured.
- Other characters are explained in the results.

Further down is a subsection `Padding Oracle` which contains details stating at which position the oracle was found. This also includes the recovered plaintext, as well as what value was XOR-ed at which position to recover the plaintext. Further down is a summary of the observed behavior difference per offset (when modifying the last byte). Note that multiple offsets might show different behavior, but not all are necessarily caused by a valid padding oracle vulnerability. This is internally verified by trying to recover the second byte. As the `Overall Result` is `TRUE`, this second byte was found.

A.5 Notes on Reusability

We believe the source code can help other researchers to more easily detect default keys in encrypted blobs with a possibly unknown format (classes `SessionTicketEncryptionFormat`, `SessionTicketMacFormat`, and `DefaultKeys`). This approach could be applied to other protocols where one party chooses key material to protect a payload.

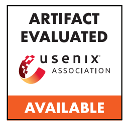
The code is currently under internal review and will be merged into the main versions of TLS-Scanner and TLS-Anvil[7] in the future. This allows researchers to more easily scan for issues related to session tickets. In combination with TLS-Crawler[8], this also allows for performing larger scans.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

[7] Maehren et al. at Usenix 22

[8] We used https://github.com/tls-attacker/TLS-Crawler/commit/d0c6e1e3d6a7168da2181ab74fa0d33b13b426f2 in our scans.



USENIX'23 Artifact Appendix
DAFL: Directed Grey-box Fuzzing Guided by Data Dependency

Tae Eun Kim, KAIST
Jaeseung Choi, Sogang University
Kihong Heo, KAIST
Sang Kil Cha, KAIST

A Artifact Appendix

This artifact appendix is a self-contained document which describes a roadmap for the evaluation of DAFL.

A.1 Abstract

DAFL is a directed grey-box fuzzer that leverages data dependency to guide the fuzzing process. DAFL's artifact provides a framework to run DAFL as well as the baseline fuzzers. The framework comprises an environment provided as a Docker image to run the fuzzers, the source code of DAFL, and the scripts to run the fuzzers and evaluate the results. This document describes how to set up the framework and replicate the experiment conducted in our paper.

A.2 Description & Requirements

In this section, we describe how to obtain our artifact, along with the hardware and software requirements to run our artifact.

A.2.1 Security, privacy, and ethical concerns

None

A.2.2 How to access

Our artifact is composed of two components: the Docker image and the framework to build and utilize it. The Docker image provides the environment to run individual fuzzing sessions by supporting all the necessary tools and dependencies. The framework builds the Docker image, orchestrates the fuzzing experiments, and evaluates the results.

The framework is accessible via a Zenodo link (`https://zenodo.org/record/8219904`). You can also download the same framework from the GitHub repository (`https://github.com/prosyslab/DAFL-artifact`). For the Docker image, we provide a pre-built Docker image via Docker Hub `https://hub.docker.com/r/prosyslab/dafl-artifact`. Nonetheless, you can also build the Docker image from scratch using the Dockerfile provided in the framework.

A.2.3 Hardware dependencies

We ran the experiment on the machines equipped with Intel(R) Xeon(R) Gold 6226R CPU (2.90GHz) with 64 cores and 192 GB of RAM. Each fuzzing session was run on a Docker container assigned with a single CPU core and 4GB of memory. As we repeated the experiment 40 times, each fuzzing session was run in parallel, utilizing 40 CPU cores at a time.

It is possible to run the experiment on a machine with fewer CPU cores and smaller RAM, but be sure to assign enough resources to each fuzzing session (e.g., 1 CPU core and 4GB of RAM for each fuzzing session).

The disk space requirement will vary depending on the volume of the experiment. However, we recommend at least 70 GB of disk space, since the provided Docker image alone is 25 GB and our main experiment results in 45 GB of data.

A.2.4 Software dependencies

Ubuntu 20.04, Docker, and Python 3.8 are required to run the artifact. The required Python dependencies can be installed by running the following command in the `DAFL-artifact` directory.

```
yes | pip3 install -r requirements.txt
```

A.2.5 Benchmarks

As described in our paper (Section 5.1), we used 41 vulnerabilities from the Beacon [1] paper.

A.3 Set-up

In this section, we describe how to set up the artifact for DAFL.

A.3.1 Installation

The following is the steps to install the artifact for DAFL.

1) Download the file `DAFL-artifact.tar.gz` from the provided Zenodo link. This file is the aforementioned

framework where you can build the Docker image, orchestrate the fuzzing experiments, and evaluate the results.

2) Extract the file to a directory of your choice.

```
tar -zxvf DAFL-artifact.tar.gz
```

3) Obtain the Docker image by:

1) pulling from Docker Hub

```
docker pull prosyslab/dafl-artifact
```

2) or building from scratch

```
docker build -t prosyslab/dafl-artifact -f Dockerfile .
```

This Docker image is the environment to run individual fuzzing sessions.

A.3.2 Basic Test

For a single fuzzing experiment, run the following command in the `DAFL-artifact` directory:

```
python3 scripts/reproduce.py run [target] [time budget]
[iterations] "[list of fuzzers]"
```

where `[target]` is the name of the target, `[time budget]` is the time budget in seconds, `[iterations]` is the number of fuzzing iterations, and `[list of fuzzers]` is the list of fuzzers to run.

To run a simple functionality test, we recommend using the target `lrzip-ed51e14-2018-11496` with 60 seconds of time budget and 10 fuzzing iterations. For the list of fuzzers, use `"AFL AFLGo WindRanger Beacon DAFL"`. The command will look like the following:

```
python3 scripts/reproduce.py run lrzip-ed51e14-2018-
11496 60 10 "AFL AFLGo WindRanger Beacon DAFL"
```

If set up was successful, a CSV file will be generated in the output directory with the name `output/lrzip-ed51e14-2018-11496-60sec-10iters`. This CSV file contains the results of the experiment, which is the median TTE of each fuzzer. In case of running 10 fuzzing sessions in parallel, this small experiment will take about 10 minutes.

A.4 Evaluation workflow

This section describes the operational steps and experiments which must be performed to evaluate DAFL's artifact.

A.4.1 Major Claims

Our paper makes the following claims:

(C1): DAFL is more effective in reproducing target crashes compared to the baseline fuzzers. This is proven by the experiment (E1) described in Section 5.2 of our paper whose results are illustrated in Table 2 and Figure 5.

(C2): Thin slicing shows better fuzzing performance compared to the naive approach. This is proven by the experiment (E2) described in Section 5.3 of our paper whose results are illustrated in Figure 7.

(C3): Two major components of DAFL, selective coverage instrumentation and semantic relevance scoring, both contribute to the fuzzing performance. This is proven by the experiment (E3) described in Section 5.4 of our paper whose results are illustrated in Figure 8.

A.4.2 Experiments

We have used a vast amount of resources to conduct the experiments for our paper. For example, we ran a 24-hour fuzzing session with 6 fuzzers on 41 targets, each repeated 40 times for the main experiment (E1) described in Section 5.2 of our paper. If run on a single machine capable of running 40 fuzzing sessions in parallel, this experiment would take 246 days of fuzzing time. We believe that it is not an easy task to replicate the experiments in our paper at a full scale.

Thus, apart from the instructions to exactly replicate our paper's experiments, we additionally provide instructions to run a scaled down version for each of the experiments to enable a feasible evaluation. The scaled down version of the experiments comprises fewer fuzzers, fewer targets, fewer iterations, and shorter time limit. For example, it excludes the targets where all the fuzzers failed to produce a median TTE within 24 hours. It also early terminates the fuzzing session based on the previously observed median TTE of each target. We also provide a minimal version of each experiment, which runs a small subset of targets. For more details on the scaled down version and the minimal version, please refer to the `README` file.

We believe that the scaled down version of the experiments is sufficient to validate the claims made in our paper. However, please note that the results of the alternative versions of the experiments are more prone to fluctuations due to the reduced number of iterations.

The expected time for each of the experiments is calculated under the assumption of running the experiment on a machine where 40 fuzzing sessions can be run in parallel. Each experiment results in a CSV file which contains the median TTE of each fuzzer for each target and a bar plot in the same format as in the paper.

(E1): [Effectiveness of DAFL] [246 compute-day + 70GB disk]: This is the main experiment described in Section

5.2 of our paper, which compares the effectiveness of DAFL with the baseline fuzzers.
How to: In the `DAFL-artifact` directory, run the following command:

```
python3 scripts/reproduce.py run tbl2 86400 40
```

Results: The result is in the form of a CSV file, which is located in the output directory (refer to the `README` file for the exact location of this file). This CSV file contains the median TTE of each fuzzer for each target. Additionally, a bar plot will be generated from the CSV file.

(E1: scaled down): [8 compute-days + 30GB disk]: The scaled down version of (E1).
How to: In the `DAFL-artifact` directory, run the following command:

```
python3 scripts/reproduce.py run tbl2-scaled 86400 10
```

Results: Same as (E1), with fewer targets and fuzzers.

(E2): [Effectiveness of thin slicing] [93 compute-day + 60GB disk]: This is the experiment described in Section 5.3 of our paper, which compares the effectiveness of thin and naive slicing approaches.
How to: In the `DAFL-artifact` directory, run the following command:

```
python3 scripts/reproduce.py run fig7 86400 40
```

Results: In same format as in (E1).

(E2: scaled down): [6 (or 2) compute-days + 30GB disk]: The scaled down version of (E2).
How to: In the `DAFL-artifact` directory, run the following command:

```
python3 scripts/reproduce.py run fig7-scaled 86400 10
```

If you have already run (E1: scaled down), the results of AFL and DAFL will be automatically reused as long as you have the results of (E1: scaled down) under the expected output directory, `output/tbl2-scaled-86400sec-10iters`. Thus, the expected runtime will be reduced to 1 compute-day.
Results: Same as (E2), with fewer targets.

(E3): [Effectiveness of DAFL's components] [124 compute-day + 60GB disk]: This is the experiment described in Section 5.4 of our paper, which evaluated the effectiveness of DAFL's components.
How to: In the `DAFL-artifact` directory, run the following command:

```
python3 scripts/reproduce.py run fig8 86400 40
```

Results: In same format as in (E1).

(E3: scaled down): [8 (or 4) compute-days + 30GB disk]: The scaled down version of (E3).
How to: In the `DAFL-artifact` directory, run the following command:

```
python3 scripts/reproduce.py run fig8-scaled 86400 10
```

As so in (E2: scaled down), if you have already run (E1: scaled down), the results of AFL and DAFL will be automatically reused as long as you have the results of (E1: scaled down) under the expected output directory, `output/tbl2-scaled-86400sec-10iters`. Thus, the expected runtime will be reduced to 2 compute-days.

Results: Same as (E3), with fewer targets.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

References

[1] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed greybox fuzzing with provable path pruning. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 36–50, 2022.

USENIX'23 Artifact Appendix: BoKASAN: Binary-only Kernel Address Sanitizer for Effective Kernel Fuzzing

Mingi Cho, Dohyeon An, Hoyong Jin, and Taekyoung Kwon, Yonsei University

A Artifact Appendix

A.1 Abstract

This artifact contains the source code of BoKASAN and the necessary data for the experiments. For Artifact Functional, two experiments are proposed, POC test and kernel fuzzing. The recommended environment for the experiment is Ubuntu 20.04 running on the machine with a multi-core x86_64 CPU and >=8GB of RAM. Using more CPU cores can reduce compile time. We expect artifact evaluation to require three human-hours and 12 compute-hours.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Access the Github repo at https://github.com/seclab-yonsei/bokasan/tree/usenix-ae

A.2.3 Hardware dependencies

- Multi-core x86_64 CPU (e.g., Intel i5, i7)
- >=8GB RAM
- >=100GB HDD/SSD

A.2.4 Software dependencies

- Ubuntu 16.04, 18.04, or 20.04
- gcc-6 or 7
- qemu-system-x86
- Syzkaller

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To install BoKASAN, download the target kernel and build it with the *'CONFIG_FUNCTION_TRACER'* flag set. Then, compile the loadable BoKASAN module according to the target kernel version. Finally, create a Linux image and insert the compiled BoKASAN module. Detailed installation instructions are described in README.md.

A.3.2 Basic Test

When the BoKASAN module is successfully compiled and the *.ko* file is created, run the target kernel using QEMU and load the module. If the installation is successful, we can find that the BoKASAN module is loaded on the target kernel using the *lsmod* command. We provide *script/qemu.sh* to run the QEMU.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *BoKASAN detects out-of-bounds and use-after-free bugs in kernels to which KASAN is not applied. This is proven by the experiment (E1) described in Section 5.1 whose results are reported in Table 5.*

(C2): *BoKASAN detects out-of-bounds and use-after-free bugs when fuzzing kernels to which KASAN is not applied. This is proven by the experiment (E2).*

A.4.2 Experiments

(E1): *[POC test] [30 human-minutes + 1 compute-hour]: Reproduce Table 5 of the paper. Execute 15 POC codes on the target kernel.*
How to: Execute 15 POCs that trigger out-of-bounds and use-after-free bugs on the target kernel running on QEMU. The detailed process is described in *README.md*.
Preparation: Download and compile Linux kernel v4.19 without 'CONFIG_KASAN' flag. Make a Linux image including the BoKASAN module. This is prepared during the installation mentioned above. Then,

execute *compile.py* and *mount.sh* in the *poc/syz* directory. As a result, a Linux image containing POC binaries is created.

Execution: Run *scripts/qemu.sh* to boot the target kernel. After the target kernel boots, executes the POC binaries under the *poc_syz* directory. Each POC is located in the *"vuln type"_"vuln name"* directory as the executable binary named *repro_setpid*. Execute one of the POC, wait for about 15 seconds, and then terminate QEMU using the *ctrl a+x* command. Repeat the above steps to test all of the POCs.

Results: When BoKASAN successfully detects a bug, the message *"BUG: KASAN: ..."* is printed as *dmesg*.

(E2): *[Fuzzing test] [30 human-minute + 6 compute-hour]: Performing fuzzing on the kernel to which KASAN is not applied using Syzkaller.*

How to: Fuzzing the kernel to which KASAN is not applied using Syzkaller. The detailed process is described in *README.md*.

Preparation: Download Syzkaller and patch it using *syzkaller/syz.diff*. Then build Syzkaller following their guidelines. Compile target kernel with KCOV and without KASAN. Fuzzing Linux kernel 4.19 with the Linux image including BoKASAN.

Execution: Run the *scripts/run_fuzz.sh* to perform fuzzing.

Results: We can see the KASAN log in Syzkaller's web interface or in the results directory when BoKASAN detects OOB and UAF bugs.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules

Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele
Boston University
{jaggel, gian, megele}@bu.edu

A Artifact Appendix

A.1 Abstract

Our artifacts include the source code and instructions about how to install the FirmSolo prototype and use it to analyze the binary kernel modules of IoT firmware images. We provide two examples of firmware images that can be analyzed with FirmSolo. We also packaged FirmSolo within a docker image along with the two downstream analysis systems (i.e., Firmadyne and TriforceAFL) that we used to demonstrate FirmSolo's utility. The docker image can run on any system that has docker installed.

In this appendix, we describe the steps to analyze a sample firmware image using FirmSolo. The analysis process involves extracting metadata information from the kernel modules within the firmware image, reverse engineering the original firmware kernel and building a custom kernel capable of loading said modules. Finally, we demonstrate how downstream analysis tools can take advantage of FirmSolo to analyze the kernel modules within firmware images for bugs and vulnerabilities.

A.2 Description & Requirements

A.2.1 How to access

You can access the artifacts at https://github.com/BUseclab/FirmSolo/tree/v1.0.0

A.2.2 Hardware dependencies

None

A.2.3 Software dependencies

Python, Docker and Java (for Ghidra).

A.2.4 Benchmarks

For the artifact evaluation we use an example Netgear firmware image as a benchmark ($id = 1$).

A.3 Set-up

A.3.1 Installation

Using the Docker:

Download the docker image from:
https://doi.org/10.5281/zenodo.7865451

Install the docker image:

```
docker load < firmsolo.tar.gz
git clone https://github.com/BUseclab/FirmSolo.git
cd FirmSolo
docker build -t firmsolo .
```

Spawn a docker container:

```
docker run -v $(pwd):/output --rm -it
--privileged firmsolo /bin/bash
```

Manual Installation:

To manually install and run FirmSolo you first need to install these dependencies:

```
sudo apt-get install build-essential
zlib1g-dev pkg-config libglib2.0-dev
binutils-dev libboost-all-dev autoconf libtool
libssl-dev libpixman-1-dev libpython3-dev
python3-pip python3-capstone python-is-python3
virtualenv sudo gcc make g++ python3 python2
flex bison dwarves kmod universal-ctags fdisk
fakeroot git dmsetup kpartx netcat-openbsd
nmap python3-psycopg2 snmp uml-utilities
util-linux vlan busybox-static postgresql wget
cscope qemu qemu-system-arm qemu-system-mips
qemu-system-mipsel qemu-utils
```

Install these python packages:

```
pip3 install ply anytree sympy requests
pexpect scipy
```

Within the FirmSolo installation directory run:

```
git submodule init
git submodule update
```

Install Ghidra:

Follow instuctions in https://ghidra-sre.org/InstallationGuide.html

Download TriforceAFL:

```
git clone https://github.com/BUseclab/TriforceAFL.git
cd TriforceAFL && make
```

Download TriforceLinuxSyscallFuzzer:

```
git clone https://github.com/BUseclab/Triforce-LinuxSyscallFuzzer.git
cd TriforceLinuxSyscallFuzzer && ./compile_harnesses.sh
```

Note: The `compile_harnesses.sh` script will make use of legacy (and unavailable) compiler toolchains that are currently only installed within the Docker image.

Download Firmadyne:

```
git clone --recursive https://github.com/BUseclab/firmadyne.git
```

Download the buildroot filesystems:

https://drive.google.com/file/d/11GiU8N1U4Nkhv-kurkoGgwmp38CM_Umg/view?usp=share_link and download the buildroot_fs.tar.gz file within FirmSolo's installation directory.

Execute:

```
tar xvf builroot_fs.tar.gz
```

Finally, specify the toolchain to be used by FirmSolo. Go into the installation directory of FirmSolo and edit the `custom_utils.py` script. Within the `get_toolchain` function edit the `cross` variable with the path(s) to your toolchain(s).

A.3.2 Basic Test

Our basic test for FirmSolo includes statically analyzing the kernel modules within a firmware image to extract metadata about the original firmware kernel. Please proceed as follows:

Spawn a docker container according to Section A.3.1 and run:

```
mkdir -p /output/images/
```

On your host download the images from this link: https://drive.google.com/file/d/1xzdTAz3PexQD8YWWAg7KYyQ8dQiVTGiR/view?usp=share_link

Execute:

```
tar xvf examples.tar.gz
cp -r ./examples/* <work_dir>/images/
```

The `work_dir` is your work directory (e.g., `./`)

Then inside your container execute:

```
cd /FirmSolo
python3 firmsolo.py -i 1 -s 1
ls /output/Image_Info/
```

After running the `ls` command you should see a file named `1.pkl` within the `/output/Image_Info/` directory.

A.4 Evaluation workflow

A.4.1 Major Claims

FirmSolo is a framework that exposes Linux-based binary IoT kernel modules to downstream analysis. Below we list and prove the claims related to the evaluation of our artifact.

(C1): *FirmSolo reverse engineers the original kernel of a firmware image and builds a new kernel supported by QEMU that can load the kernel modules within the firmware image. This claim is proven by experiment (E1), described in Section 5.2, Table 2 and Figure 2 in our paper.*

(C2): *Downstream analysis systems can use FirmSolo to analyze binary firmware kernel modules for bugs and vulnerabilities. This claim is proven in Experiments (E2) and (E3), described in Section 5.4, Table 5 and Table 6 in our paper.*

A.4.2 Experiments

We assume that you use the docker image to perform the artifact evaluation.

(E1): *[Reverse Engineering] [5 human-minutes + 10 compute-minutes + 5GB disk]: In this experiment FirmSolo extracts metadata from the binary kernel modules of a firmware image, reverse engineers the original firmware kernel and builds a new kernel capable of loading the kernel modules within the firmware image.*

Preparation: *Copy the extracted file-system and kernel of the target firmware image in the work directory.*

Execution: *After you install and connect to the docker container as described in Section A.3.1, proceed to analyze an example firmware image:*

Within the docker execute:

```
mkdir -p /output/images/
```

On your host download the images from this link (if you have not implemented the basic test): https://drive.google.com/file/d/1xzdTAz3PexQD8YWWAg7KYyQ8dQiVTGiR/view?usp=share_link

Execute:

```
tar xvf examples.tar.gz
cp -r ./examples/* <work_dir>/images/
```

The `work_dir` is your work directory (e.g., `./`)

To analyze image 1 with FirmSolo, inside your container execute:

```
cd /FirmSolo
python3 firmsolo.py -i 1 -a
```

FirmSolo will analyze firmware image 1, reverse

engineer the original firmware kernel and build a new kernel that is capable of loading the kernel modules within image 1. FirmSolo will also find which kernel modules can actually load and also which kernel modules crash during emulation (if any). If any kernel modules crashed during emulation, FirmSolo will try to address the error by running stage 2c.

Results: *To get information about the analyzed image 1, such as the kernel modules within the firmware image, the kernel modules that loaded successfully using the FirmSolo kernel, the kernel modules that crashed during emulation and the kernel module substitutions implemented by FirmSolo, run this command:*

```
python3 firmsolo.py -i 1 -d
```

In the output you should see:

```
Image: 1 Total Modules: 16 Loaded Modules:
5 Crashing Modules: 0 Substitutions: 0
```

along with specific information about which modules were successfully loaded, crashed, and substituted.

Note: Depending on the metadata information extracted/processed in this step (e.g., kernel symbols) this step can take longer. However, FirmSolo caches data about each kernel version used by the firmware images it analyzes, which renders future runs faster.

(E2): *[TriforceAFL] [1 human-minute + 1.5 compute-hour]: In this experiment you use the TriforceAFL kernel fuzzer to analyze the kernel modules within the target firmware image.*

Preparation: *None*

Execution: *Setup TriforceAFL with these commands:*

```
echo core >/proc/sys/kernel/core_pattern
cd /sys/devices/system/cpu
echo performance | tee
cpu*/cpufreq/scaling_governor
```

These commands are needed by AFL for improving performance. To fuzz the kernel modules within image 1 for 30 minutes each run:

```
python3 ./triforceafl/triforce_run.py -i 1
-t 30m
```

The `triforce_run.py` script will analyze the binary kernel modules within image 1 which expose an IOCTL interface. The script will find potential IOCTL command numbers that can be used to access these IOCTL interfaces and will use the command numbers found as seeds for TriforceAFL. Image 1 has two kernel modules that can be fuzzed (*acos_nat.ko* and *ipv6_spi.ko*). The kernel module *acos_nat.ko* exposes two IOCTL interfaces and each will be fuzzed separately. Thus the total fuzzing time for both kernel modules will be around 90 minutes.

Results: *The fuzzing results will be available in the* `/output/Fuzz_Results_Cur/1` *directory. To be able to quickly test for a crash found by the fuzzer run this command:*

```
python3 ./triforceafl/get_fuzzing_cmd.py 1
```

If the fuzzer triggered any crashes for any of the IOCTL interfaces fuzzed, the `get_fuzzing_cmd.py` script will output commands that can be copy/pasted into the terminal and executed to quickly test a crash. The commands will be available under the `CRASHES:` section (for each IOCTL interface) else this section will be blank.

(E3): *[Firmadyne] [1 human-minute + 30 compute-minutes]: In this experiment you use the Firmadyne dynamic analysis system to analyze the kernel modules within the target firmware image.*

Preparation: *None*

Execution: *Run the Firmadyne analysis for image 1 as follows:*

```
cd /firmadyne && ./experiment.sh 1
```

The `experiment.sh` script will run a full analysis with Firmadyne; creating a file-system, detecting a network configuration and testing the firmware kernel modules of image 1 against exploits from ExploitDB and the bugs found by TriforceAFL as explained in our paper.

Results: *The results will be available in the* `/output/firmadyne_results/1` *directory.*

To check if any of the exploits from ExploitDB and the TriforceAFL bugs triggered a crash, manually inspect the serial logs under `/output/firmadyne_results/1/[remote,local]/` and `/output/firmadyne_results/1/afl/`, respectively.

The `qemu.final.serial.log_2694_ipv6_spi_1` file in `/output/firmadyne_results/1/afl/` should contain an “Oops” message. It might be the case though that the crash message is not present because the kernel hangs before printing it. In this case you may need to re-run the analysis.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: UNCONTAINED: Uncovering Container Confusion in the Linux Kernel

Jakob Koschel†
Vrije Universiteit Amsterdam
j.koschel@vu.nl

Pietro Borrello†
Sapienza University of Rome
borrello@diag.uniroma1.it

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

† Equal contribution joint first authors

A Artifact Appendix

A.1 Abstract

In this artifact we provide the means to reproduce our main results. Specifically, we show that our framework, UNCONTAINED, finds container confusion, both dynamically while fuzzing and statically with dataflow tracking. We have evaluated our artifact on an Ubuntu 22.04.1 (stock Linux kernel v.5.15) with 16 cores @2.3GHz (AMD EPYC 7643) using a total of 16 QEMU-KVM virtual machines with 4GB RAM. Our source code is available at: github.com/vusec/uncontained.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Since UNCONTAINED is only used for bug finding either statically or dynamically but running within VMs it does not impose any machine security, data privacy or other ethical concerns.

A.2.2 How to access

The files for the artifact evaluation are available at: https://github.com/vusec/uncontained/releases/tag/ae.

A.2.3 Hardware dependencies

UNCONTAINED does not impose any strict hardware requirements but we assume a recent x86_64 machine with enough RAM (minimum 64GB, or enough swap) to compile LLVM/Linux and run virtual QEMU machines for fuzzing with syzkaller.

A.2.4 Software dependencies

We expect certain packages from the Ubuntu package manager to be installed, which are required to compile LLVM, Linux, syzkaller, etc. We describe the necessary packages in the Set-up section.

If you use a different distribution you need to make sure to fulfil the necessary dependencies using your dedicated package manager.

A.2.5 Benchmarks

None.

A.3 Set-up

In general, we recommend using a bare-metal desktop system running Ubuntu 22.04. Make sure that you have KVM support and your user is allowed to use KVM. The following packages are required:

```
# go-task
sh -c "$(curl -ssL https://taskfile.dev/install.sh)" \
  -- -d -b ~/.local/bin
# llvm-project
sudo apt install build-essential clang-12 lld-12 ninja-build \
    ccache cmake
# linux
sudo apt install bison flex libelf-dev libssl-dev coccinelle
# syzkaller
sudo apt install debootstrap
# install golang 1.20.5
GO_VERSION=go1.20.5.linux-amd64
wget https://go.dev/dl/$GO_VERSION.tar.gz
sudo rm -rf /usr/local/go
sudo tar -C /usr/local -xzf $GO_VERSION.tar.gz
rm -f $GO_VERSION.tar.gz
# qemu
sudo apt install qemu-system-x86
# evaluation
pip3 install scipy pandas
```

Then make sure that `~/.local/bin` and `/usr/local/go/bin` are in your `PATH` to find `go` and the `task` binaries:

```
export PATH=$HOME/.local/bin:/usr/local/go/bin:$PATH
```

A.3.1 Installation

1. Obtain the artifact source and necessary dependencies:

```
git clone --recurse-submodules \
    https://github.com/vusec/uncontained.git
```

2. Create the `kernel-tools/.env` file with the following content (replace `/patch/to/uncontained` with the actual absolute path):

```
REPOS=/path/to/uncontained
LLVMPREFIX=/path/to/uncontained/llvm-project/build
KERNEL=/path/to/uncontained/linux
ENABLE_KASAN=1
ENABLE_DEBUG=1
ENABLE_SYZKALLER=1
ENABLE_GDB_BUILD=1
ADDITIONAL_LLVM_VARIABLES=-DLLVM_ENABLE_EH=ON -DLLVM_ENABLE_RTTI=ON
```

3. Compile all the necessary dependencies (this will take a while to compile llvm-project and Linux with fullLTO):

```
scripts/compile.sh
```

A.3.2 Basic Test

To test if the sanitizer and the static analyzers work as intended you can use the tests by running the following:

```
LLVM_DIR=$PWD/llvm-project/build tests/test.sh
LLVM_DIR=$PWD/llvm-project/build tests/testDF.sh
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The* UNCONTAINED *sanitizer finds new types of container confusions. This is proven by the experiment (E1).*

(C2): *The* UNCONTAINED *sanitizer comes with an acceptable performance runtime overhead. This is proven by the experiments (E2) and (E3).*

(C3): *The* UNCONTAINED *static analyzer has been used to uncover new bugs in the Linux kernel. This is proven by the experiments (E4).*

A.4.2 Experiments

(E1): *[fuzzing evaluation] [2 human-hours + 24 compute-hours]: This is the fuzzing experiment using the sanitizer while fuzzing with syzkaller. Expected results are a range of bugs reported.*

How to: `kernel-tools` *is responsible for starting the fuzzer with the kernel that has been instrumented with the sanitizer.*

Preparation: *Make sure you setup everything from the Installation step, including building syzkaller and create the syzkaller image (should be done by the* `./scripts/compile.sh` *script).*

Execution: *You can compile the kernel with instrumentation and start the fuzzer with executing* `./scripts/compile.sh && ./scripts/run.sh`*. Then let it run for at least 24 hours to get some results.*

Results: *The result will be the crashes in the* `kernel-tools/out/syzkaller-workdir/crashes` *directory. We need to manually filter out bugs that are not triggered by* UNCONTAINED *(all that do not have three lines of* `[UNCONTAINED]` *before the* `BUG:` *line).*

(E2): *[2 human-hours + 30 compute-hours]: This is the fuzzing performance experiment using the sanitizer while fuzzing with syzkaller. Expected results are the overhead in terms of throughput of executed testcases.*

How to: *We need to run syzkaller 10 times for one hour for the baseline (stock syzkaller), with KASAN and with* UNCONTAINED.

Preparation: *Make sure you setup everything from the Installation step, including building syzkaller and create the syzkaller image (should be done by the* `./scripts/compile.sh` *script).*

Execution: *You can compile the kernel with instrumentation and start the fuzzer with executing* `./scripts/run-fuzzing-performance-evaluation.sh`*. Then let it run for the 30 hours to get the results.*

Results: *The result will be the percentage of decreased executed testcases when running syzkaller. You can now look at the results with executing:*

```
./scripts/evaluation/syzkaller-bench.py --prefix \
    'evaluation/syzkaller/results/syzkaller-bench-'
```

(E3): *[1 human-hour + 1 compute-hour]: This is the LMBench experiment using the sanitizer while running the benchmarking suite to verify performance overhead.*

How to: *We need to run LMBench 10 times for the different configurations (baseline,* UNCONTAINED, *KASAN).*

Preparation: *Make sure you setup everything from the Installation step, including building syzkaller and create the syzkaller image (should be done by the* `./scripts/compile.sh` *script).*

Execution: *You can compile the kernel with instrumentation and start LMBench with executing* `./scripts/run-lmbench-performance-evaluation.sh`*. Then let it run to get the results.*

Results: *The result will be the overhead numbers of the different configurations on top of the baseline for the LMBench testcases. You can now look at the results with executing:*

```
./scripts/evaluation/lmbench.py --prefix \
    'evaluation/lmbench/results'
```

(E4): *[1 human-hour + 3 compute-hours]: This is the static analyzers experiment using the static analyzer to find the necessary reports with static analysis.*

How to: *Compile the kernel with our static analyzers*

enabled to extract all the bug reports.

Preparation: *Make sure you setup everything from the Installation step, including building syzkaller and create the syzkaller image (should be done by the* `./scripts/compile.sh` *script).*

Execution: *You can generate all the reports with* `./scripts/run-static-analyzer.sh`*. Then let it run to get the results.*

Results: *The result will be the reports for the different rules. The results from the LLVM passes are in YAML and are not yet grouped by the source line (to remove duplicates). The results from the coccinelle script are text based and are already filtered based on uniqueness. You can load the YAML reports into the* `vscode-extension` *to look at them in a more convenient way and do the grouping based on the source code line.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Educators' Perspectives of Using (or Not Using) Online Exam Proctoring

David G. Balash, Elena Korkes, Miles Grant, and Adam J. Aviv
The George Washington University

Rahel A. Fainchtein and Micah Sherr
Georgetown University

A Artifact Appendix

A.1 Abstract

In our paper we explore how educators balance the security and privacy of their students with the requirements of remote exams. We developed a survey and we had n=125 educator responses. In our archive we make available functional artifacts that can be used to reproduce our qualitative and quantitative study results. The artifact includes the survey, R-scripts, Python, and shell scripts with detailed instructions on how to run this software. A single PC, Mac, or Linux machine should be sufficient hardware. Software requirements include RStudio, Python, and a terminal to run a shell script. The artifact can be evaluated by running the R-programming files, and evaluating the qualitative coding results.

A.2 Description & Requirements

This archive contains the survey questionnaire, and the data obtained from an online survey conducted during our study. The archive includes qualitative open coding analysis of open-ended survey results, as well as the R-scripts used to processes the quantitative results. We have included all of the software that we created to deploy the online survey. We have provided instructions for running the analysis.

A.2.1 Security, privacy, and ethical concerns

All datasets have been sanitized of any personally identifiable information. ResponseId variables are randomized alphanumeric strings.

A.2.2 How to access

The artifact can be accessed at the following URL:
https://github.com/gwusec/2023-USENIX-Educator-Perspectives-of-Exam-Proctoring/tree/10b55097bd807eb0cf3e6a41b154fe4e4e235f43
Please read the provided README.md file for full details: https://github.com/gwusec/2023-USENIX-Educator-Perspectives-of-Exam-Proctoring/tree/10b55097bd807eb0cf3e6a41b154fe4e4e235f43/README.md

A.2.3 Hardware dependencies

A single PC, Mac, or Linux machine should be sufficient hardware.

A.2.4 Software dependencies

Software requirements: RStudio, Python, shell.

A.2.5 Benchmarks

None.

A.3 Set-up

1. Install RStudio https://support--rstudio-com.netlify.app/products/rstudio/download/ and use the R command install.packages() to install the following R packages which are required to run the R scripts: ordinal, MASS, tidyverse, ggalluvial, ggrepel, ggfittext, cowplot, scales, lubridate, broom, xtable, rstatix, Hmisc

2. Install Python 3+ https://www.python.org/downloads/

3. Clone or download the git repository https://github.com/gwusec/2023-USENIX-Educator-Perspectives-of-Exam-Proctoring/tree/10b55097bd807eb0cf3e6a41b154fe4e4e235f43

4. Run the scripts/qualitative-analysis/runirr.sh shell script to validate the inter-rater reliability scores

5. Load the scripts/quantitative-analysis/2021-educator.Rmd into RStudio and run the script to generate the figures and regression tables

6. Create a free Qualitrics account https://www.qualtrics.com/free-account/
7. Load the survey questionnaire Qualitrics file survey-instruments/Online_Proctoring_Educator_Survey.qsf into Qualtrics. To do this on Qualtircs you should select "Create a survey using a file."

A.3.1 Installation

https://github.com/gwusec/2023-USENIX-Educator-Perspectives-of-Exam-Proctoring/blob/10b55097bd807eb0cf3e6a41b154fe4e4e235f43/survey-instruments/Readme.md

A.3.2 Basic Test

Open the https://github.com/gwusec/2023-USENIX-Educator-Perspectives-of-Exam-Proctoring/blob/10b55097bd807eb0cf3e6a41b154fe4e4e235f43/scripts/quantitative-analysis/2021-educator.Rmd R-script in the RStudio program and run the script.

A.4 Evaluation workflow

We created and deployed an online survey. We collected data from the survey and used qualitative open coding to analyse the qualitative results, and R-programming to analyse the quantitative results. Both the qualitative spreadsheets and R-scripts are provided, along with the raw data from the surveys.

A.4.1 Major Claims

The key results of the paper are the online survey result data and the detailed analysis of this data. The descriptive figures and regression analysis as described in the paper is our next key results, and those can be validated using the raw data along with the provided R-scripts.

A.4.2 Experiments

The figures and regression tables can be rebuilt using the provided R-scripts.

A.5 Notes on Reusability

The survey can be loaded in Qualtrics and new data obtained. The R-Scripts can be resused with new data. The irr.py script can be used to check new qualitative data.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: No more Reviewer #2: Subverting Automatic Paper-Reviewer Assignment using Adversarial Learning

Thorsten Eisenhofer*†, Erwin Quiring*†‡, Jonas Möller§, Doreen Riepel†,
Thorsten Holz¶, Konrad Rieck§

†Ruhr University Bochum
‡International Computer Science Institute (ICSI) Berkeley
§Technische Universität Berlin
¶CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

The number of papers submitted to academic conferences is steadily rising in many scientific disciplines. To handle this growth, systems for automatic *paper-reviewer assignments* are increasingly used during the reviewing process. These systems use statistical topic models to characterize the content of submissions and automate the assignment to reviewers. In this paper, we show that this automation can be manipulated using adversarial learning. We propose an attack that adapts a given paper so that it misleads the assignment and selects its own reviewers. Our attack is based on a novel optimization strategy that alternates between the feature space and problem space to realize unobtrusive changes to the paper. To evaluate the feasibility of our attack, we simulate the paper-reviewer assignment of an actual security conference (IEEE S&P) with 165 reviewers on the program committee. Our results show that we can successfully select and remove reviewers without access to the assignment system. Moreover, we demonstrate that the manipulated papers remain plausible and are often indistinguishable from benign submissions.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no expected risks or others ethical concerns when executing the artifact.

A.2.2 How to access

The code for the artifact is available on GitHub github.com/RUB-SysSec/adversarial-papers with commit hash `01fc915612c7ca72481b50ab7700dde1e0fa6188`.

*Shared first authorship

The main experiments require the following files

```
evaluation
|-- models
|   |-- overlap_0.70
|   |-- victim
|-- problemspace
|   |-- bibsources
|   |-- llms
|   |-- synonyms
|-- submissions
|   |-- oakland_22
|-- targets
    |-- budget-vs-transformer.json
    |-- featurespace-search.json
    |-- surrogates
        |-- surrogate_targets_4.json
```

Targets files and bib sources are included in the repository. Pre-trained models and required target submissions are provided at *blinded*. Due to licensing issues, we cannot make these submissions publicly available. We do, however, publish all of our crawling scripts, dummy examples, and pre-trained models. Refer to the repository for more information.

A.2.3 Hardware dependencies

The evaluation does not require special hardware. All experiments can be performed on a "regular" server using only CPUs. We performed the experiments on a server with 256 GB RAM and two Intel Xeon Gold 5320 CPUs.

A.2.4 Software dependencies

We provide a docker container that setups the required environment and which can be used to run the experiments.

A.2.5 Benchmarks

There are no benchmarks required to evaluate this artifact.

A.3 Set-up

A.3.1 Installation

For ease of use, we include a Dockerfile with all necessary tools to reproduce the results from the paper. It can be build via

```
git clone
 ↪  https://github.com/RUB-SysSec/adversarial-papers.git
 ↪  adversarial-papers
cd adversarial-papers; ./docker.sh build
```

After building the container, it is possible to spawn a shell with

```
./docker.sh shell
```

All containers get automatically deleted after the shell exits (cf. the `--rm` flag from `docker run`). To make the evaluation results both easily accessible and persistent, we map subdirectories of the evaluation folder `evaluation` from the host inside the container. To setup all paths correctly, it is therefore necessary to invoke the `docker.sh` in the base directory of the project.

A.3.2 Basic Test

After building the docker container, you can test your setup by running the following command

```
./docker.sh run "python3
 ↪  /root/adversarial-papers/src/attack.py --targets_file
 ↪  /root/adversarial-papers/evaluation/targets/test.json
 ↪  --format_level --workers 1 --trial_name basic-test"
```

This will start the attack for the target described in `evaluation/targets/test.json`. If everything is working properly, the attack should run for one iteration and immediately return successful. Results are stored in `evaluation/trials/basic-test`.

The main entry point for the attack is in the `src/attack.py` file. There are options provided to configure almost every aspect of the attack grouped into general, feature-space and problem-space specific configurations. See the documentation in the repository for further details.

When setting the number of workers to 1, the attack produces verbose output for debugging. For larger numbers, this output is *not* send to `stdout` but stored only as a log file in the respective result directory.

A.4 Evaluation workflow

The full evaluation consists of ten experiments, which requires about 6.5 CPU years to fully execute. In the following, we therefore describe only the subset of experiments we think are necessary to reproduce the major claims in the paper. Refer to the documentation in the repository for a complete description of all ten experiments include the hyperparameter search, and how to train your own models.

Each experiment is configured with a file describing all considered targets (`--targets_file`). These files are located at `evaluation/targets`. The scripts to re-generate these files are located at `scripts/targets`. Each target is optimized to run on a single-core and the experiments are therefore highly amneable for parallelization across CPU cores and machines. Note, that depending on the experiments more or less computer memory might be required (e.g., the black-box experiments require more memory per instance to store the surrogate models). Depending on the machine, this might limit the number of parallel executions. To get a good estimate, we will additionally report an approximated (!) maximal memory per instance (e.g., with 100 workers the experiment requires $100\times$ the amount of this value).

Finally, for almost any experiment, it is possible to continuously check the current results which might allow to stop experiments early if the numbers have sufficiently converged (see the expected results for each experiment). Sending the interrupt signal (w/ CTRL+C) should usually stop all processes, but sometimes the scripts need a bit more persuasion. In this case, stopping the container proved to be an effective strategy (i.e., `docker kill <container-id>` with the container id either being autocomplete with pressing TAB or using `docker ps`).

A.4.1 Major Claims

In the paper, we investigate three major claims:

(C1): First, we show that the proposed attack is effective in crafting adversarial papers in a white-box setting. This is investigated with experiment (E1) described in the *feature-space search* paragraph in Section 4.1. The results of the experiment are reported inline in the text as well as Table 2.

(C2): Second, we demonstrate that the attack extends to different classes of transformations. This is described in the *all transformations* paragraph in Section 4.1 and is part of experiment (E2). The results of the experiment are reported in Figure 4.

(C3): Finally, we analyze if the attack remains viable in a black-box scenario as described in Section 4.2. We consider this in experiment (E3) and simulate the attack with varying numbers of surrogate models.

A.4.2 Experiments

(E1): *Feature-space search* [800 CPU hours + 31GB disk]: We start our evaluation by examining the feature-space search of our attack. For this experiment, we consider format-level transformations that can realize arbitrary changes. Other transformations are evaluated as part of experiment (E2).

The experiment can be executed with:

```
WORKERS=100
./docker.sh run "python3
↪  /root/adversarial-papers/src/attack.py
↪  --targets_file
↪  /root/adversarial-papers/evaluation/
targets/featurespace-search.json --reviewer_window 6
↪  --reviewer_offset 2 --no_successors 256
↪  --beam_width 4 --step 64
↪  --problem_space_block_features
↪  --feature_problem_switch 8 --format_level
↪  --workers ${WORKERS} --trial_name
↪  featurespace-search"
```

Per worker, roughly 850MB of memory are expected. Adjust the number of parallel executions accordingly. Raw results are stored in `evaluation/trials/featurespace-search` and can be analyzed with

```
./docker.sh run "python3 /root/adversarial-papers/
evaluation/scripts/00_featurespace_search.py"
```

Expected output (cf. Table 2 and inline in text)

```
FEATURE-SPACE SEARCH
[+] Overall success rate
    -> 99.67%

[+] Overall run-time
    -> median: 7m 12s

[+] Overall L1
    -> min   : 9
    -> max   : 22621

[+] Ratio between modifications and original content
    -> selection: 9.42%
    -> rejection: 13.37%

[+] Modifications per objective
        Selection Rejection Substitution
    L1         704      1032         2059
    Linf        17        43           62
```

(E2): *All transformations* [1200 CPU hours + 32GB disk]: In experiment (E1), we have focused on format-level transformations to realize manipulations. These transformations exploit intrinsics of the submission format, which effectively allows us to make arbitrary changes to a PDF file. In experiment (E2) we consider different classes of transformations as introduced in Section 3.2.

The experiment can be executed with:

```
WORKERS=100
./docker.sh run "python3
↪  /root/adversarial-papers/src/attack.py
↪  --targets_file
↪  /root/adversarial-papers/evaluation/
targets/budget-vs-transformer.json
↪  --problem_space_block_features --reviewer_window 6
↪  --reviewer_offset 2 --no_successors 256
↪  --beam_width 4 --step 64 --workers ${WORKERS}
↪  --trial_name budget-vs-transformer-1"
```

Per worker, roughly 2300MB of memory are expected. Adjust the number of parallel executions accordingly. Raw results are stored in `evaluation/trials/budget-vs-transformer` and can be analyzed with

```
./docker.sh run "python3 /root/adversarial-papers/
evaluation/scripts/04_all_transformations.py"
```

Expected output (cf. left part of Figure 4)

```
[+] Switches
    found no trials

[+] Budget
                  0.25   0.50   1.00   2.00   4.00
    Text      :  22.00  28.00  40.00  52.00  68.00
    + Encoding:  24.00  31.00  45.00  53.00  69.00
    + Format  : 100.00 100.00 100.00 100.00  99.00

[+] Saved plot @
↪  evaluation/plots/all-transformations.pdf
```

Note that the full plot in Figure 4 aggregates eight of such runs.

(E3): *Surrogates* [1000 CPU hours + 46GB disk]: In practice, an attacker typically does not have unrestricted access to the target system. We therefore also assume a black-box scenario and consider an adversary with only limited knowledge.

The experiment can be executed with:

```
WORKERS=50
./docker.sh run "python3
↪  /root/adversarial-papers/src/attack.py
↪  --targets_file
↪  /root/adversarial-papers/evaluation/targets/
surrogates/surrogate_targets_4.json --reviewer_window
↪  2 --delta -0.16 --reviewer_offset 1
↪  --no_successors 128 --beam_width 4 --step 256
↪  --problem_space_block_features
↪  --feature_problem_switch 8 --format_level
↪  --workers ${WORKERS} --trial_name surrogates-4"
```

Per worker, roughly 2000MB of memory are expected. Adjust the number of parallel executions accordingly. Raw results are stored in `evaluation/trials/surrogates-4` and can be analyzed with

```
./docker.sh run "python3 /root/adversarial-papers/
evaluation/scripts/05_surrogates.py"
```

Expected output (cf. Figure 5 with ensemble size 4)

```
[+] Saved plot @ evaluation/plots/surrogates.pdf
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: WaterBear: Asynchronous BFT with Information-Theoretic Security and Quantum Security

Haibin Zhang, Sisi Duan, Boxin Zhao, and Liehuang Zhu

A Artifact Appendix

A.1 Abstract

This document provides a tutorial on how to use the Waterbear codebase. In particular, five protocols are evaluated in the paper: BEAT-Cobalt; WaterBear-C; WaterBear-Q; WaterBear-QS-C; WaterBear-QS-Q. The results are evaluated on Amazon EC2 instances, reproducing the results of which requires an account on AWS. As reproducing all of our results in the paper is time-consuming (which takes us a few weeks), this document only focuses on how to use the codebase. If one is interested in reproducing the results in the paper, please refer to README under the waterbear/ec2 folder of our codebase for details.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There is no security, privacy, or ethical concerns.

A.2.2 How to access

Our code can be obtained from https://github.com/fififish/waterbear, and the stable version can be found at https://github.com/fififish/waterbear/releases/tag/usenixsec.

To prevent testers from being able to download dependencies, we provide a complete code base including all dependencies, which can be obtained from https://github.com/fififish/waterbear/tree/waterbear-with-dependencies. All dependencies are included in "waterbear/src/".

A.2.3 Hardware dependencies

All experiments are deployed on EC2 of Amazon Web Services. We use both t2.medium and m5.xlarge instances for our evaluation. The t2.medium type has two virtual vCPUs (Intel Xeon expandable processor with maximum frequency of 3.3GHz) and 4GB memory and the m5.xlarge has four vCPUs (Intel Xeon Platinum processor with maximum frequency of 3.1GHz) and 16GB memory. Please refer to https://aws.amazon.com/cn/ec2/instance-types/ for more information about the EC2 instance.

Most results we reported in the paper are conducted on m5.xlarge instances. We recommend m5.xlarge for reproducibility of our results.

A.2.4 Software dependencies

We ran our experiments using Ubuntu 20.04. More specifically, we choose "ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20220610" on AWS.

To compile our code of protocols, we require go1.15.14 linux/amd64. We require the following libraries.

Additionally, several open source libraries are required. One can download the libraries using the following commands:

- go get -u google.golang.org/grpc
- go get -u golang.org/x/net
- go get -u golang.org/x/text
- go get -u golang.org/x/crypto/...
- go get -u golang.org/x/sys
- go get -u google.golang.org/genproto/
- go get -u github.com/klauspost/reedsolomon
- go get -u github.com/klauspost/cpuid
- go get -u github.com/cbergoon/merkletree
- go get -u github.com/golang/protobuf

Alternatively, one can also use the following command to get the dependencies:

make go

make install

The dependencies have tested as of Aug 2023. In case of failures of executing the commands above, one can alternatively download the dependencies and place them under waterbear/src/.

A.2.5 Benchmarks

We provide a script for reproducing our results. One can use python 3.x to run the script, we used python 3.8.10 in the experiment. The script can be found under the waterbear/ec2 folder. We provide a README on how to start the experiments. Alternatively, one can directly ssh to the servers to start the experiments manually.

For each experiment, we vary two major parameters: n and b, where n is the number of servers and b is the batch size. Given a fixed n, we vary b from 1 to a sufficiently large number (e.g., 30,000) to generate the results. For each experiment, we obtain the results from all servers, exclude ones with outstanding results (e.g., extremely large throughput), and obtain the average result of the servers. We repeat each batch size in experiment five times to obtain the results reported in the paper.

A.3 Set-up

A.3.1 Installation

1. Pull the code from https://github.com/fififish/waterbear or download the stable version from https://github.com/fififish/waterbear/releases/tag/usenixsec. Or pull the complete code base including all dependencies from https://github.com/fififish/waterbear/tree/waterbear-with-dependencies, which can be compiled directly without downloading any dependencies.

2. Set the environment by:

$$export\ \text{GOPATH} = \text{\$PWD}$$

$$export\ \text{GOBIN} = \text{\$PWD}/bin$$

$$export\ \text{GO111MODULE} = off$$

3. Download the dependencies by running the following commands:

$$make\ go$$

$$make\ install$$

4. Compile the code by running the following commands:

$$make\ build$$

The compiled file will be created in "$PWD/bin". We recommend setting up $PWD as ./waterbear.

A.3.2 Basic Test

1. Modify the configuration file "$etc/conf.json$" to choose which protocol to execute. Details about the protocols are included in "$etc/node.txt$". The *id* of each server should be unique. By default, we use monotonically increasing ids, 0, 1, 2, $\cdots$. If one tests the code locally, modify IP addresses and port numbers of all servers manually. The IP addresses and port numbers of all servers can be modified by script when testing on AWS.

2. To run BEAT-Cobalt, generate keys for threshold PRF first by running the following command:

$$keygen\ [n]\ [k]$$

Here, n is the number of servers, and k is the threshold to generate the common coin. We set up $n = 3f+1$ and k to $f+1$ for most of our experiments.

If the keys are successfully generated, they are located under waterbear/etc. Make sure that the generated keys are placed under the repository of *all* servers.

3. For all the servers, run the command below to start the servers:

$$server\ [id]$$

Here, $[id]$ is configured in conf.json and is different at each server.

4. Start a client to send transaction to start the protocol by running the following command:

$$client\ [id]\ 1\ [b]\ [msg]$$

Here, $[id]$ is the identifier of the client. We do not require the client to be registered. One can use any id that is unique, e.g., 1000. $[b]$ is the batch size. $[msg]$ can be any message. One can ignore the $[msg]$ field and a default message is included in the codebase.

5. All servers will print text like " *****epoch ends". This means the success of the epoch. One can repeat the operation of client after the epoch ends to start a new epoch.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Practical Asynchronous High-threshold Distributed Key Generation and Distributed Polynomial Sampling

Sourav Das[1] *Zhuolun Xiang*[2] *Lefteris Kokoris-Kogias*[3] *Ling Ren*[1]
[1]*University of Illinois at Urbana-Champaign,* [2]*Aptos,* [3]*IST Austria & Mysten Labs*
souravd2@illinois.edu, xiangzhuolun@gmail.com, lefteris@mystenlabs.com, renling@illinois.edu

A Artifact Appendix

A.1 Abstract

Our artifact is built using docker, and can be run on a single machine with multi-threaded or multi-processes emulation, and in a geo-distributed setting with multiple amazon-web service virtual machines.

There are four important parameter choices of our artifact: (i) `num`: number of nodes in the ADKG protocol, (ii) `ths`: maximum number of faulty nodes, (iii) `deg`: the degree of the ADKG polynomial, and (iv) `curve`: the choice of the elliptic curve, which is either `bls12381` or `ed25519`.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

N/A

A.2.2 How to access

Our entire artifact is packaged as a single library and can be downloaded from:

- https://github.com/sourav1547/htadkg

We recommend using the version specified by the commit:

- https://github.com/sourav1547/htadkg/commit/0d221e8965c5cf6b18823d894ef48c0fabc34b6e

A.2.3 Hardware dependencies

The basic tests require a stable internet connection. The main experiments require `num` amazon web services instances.

A.2.4 Software dependencies

- Docker https://www.docker.com/
- Docker compose https://docs.docker.com/compose/
- Python `3.7.x` or higher https://www.python.org/

A.2.5 Benchmarks

None.

A.3 Set-up

Our artifact requires docker and docker compose. To install docker, see https://docs.docker.com/get-docker/. To install docker-compose, see https://docs.docker.com/compose/install/. Check that docker and docker compose are installed correctly by running the `$docker` and `$docker-compose` commands in the terminal. Upon successful installation, both of these commands will display the available options.

Start the docker daemon. In case of docker desktop start the docker daemon by starting the docker desktop application or by running the `$open -a Docker` command in the terminal.

A.3.1 Installation

Our entire artifact is packaged as a single library and can be downloaded from:

- https://github.com/sourav1547/htadkg

Once the docker and docker compose are installed, and docker daemon is running, is installed build the code using the following instructions. Note that building the `adkg` docker image will take approximately 10 minutes, possibly longer depending upon the internet connection.

1. `cd` to `htadkg` folder
2. Build using `$docker-compose build adkg`.
3. Run a docker image of adkg `$docker-compose run --rm adkg bash`

Upon successful installation, the last command will open a terminal with `root@fb0991941061:/usr/src/adkg#`.

Remark. If the docker daemon is not running, expect the following error message while building the `adkg` docker image.

```
docker.errors.DockerException: Error while
   fetching server API version: 500 Server
   Error for http+docker://localhost/version:
   Internal Server Error ("b'dial unix
   docker.raw.sock: connect: connection
   refused'")
[71057] Failed to execute script
   docker-compose
```

A.3.2 Basic Test

There are two modes to perform basic tests, and both of these tests can be run locally. We elaborate on each of these tests below.

- Emulating `num` threads inside a docker image.
- Emulating `num` processes inside a docker image

Emulating multiple threads. After completing the steps mentioned in §A.3.1, run this test using the following command inside the `adkg` docker.

```
$pytest tests/test_adkg.py -o log_cli=true
   --num 4 --ths 1 --deg 2 --curve ed25519
```

The command runs an ADKG protocol with four nodes where at most one node is corrupt, and we want to secret share the ADKG secret key using a polynomial of degree two. Note that `num>3*ths` and `ths-1<deg<num-ths`. It is possible to change these values arbitrarily as long as they satisfy these constraints. We recommend running this basic test with less than 10 nodes for quicker results.

Emulating multiple processes. This approach creates multiple processes within a single docker image. Each process corresponds to one ADKG node and these processes communicate using an Inter-process communication (`ipc`) channel. To start the experiment, run the following command after following the instructions in §A.3.1.

```
$sh scripts/launch-tmuxlocal.sh
   scripts/adkg-tutorial.py [NUM_NODES]
```

For this basic test, our artifact supports 16, 32, and 64 nodes. To evaluate with arbitrary `num`,`ths` and `deg`, first, generate the corresponding configuration files using `gen_config.py`. We recommend testing with 16 and 32 nodes for quicker results.

Remark 1. Although this process runs `NUM_NODES` number of ADKG nodes, our artifact only displays the log of the first four nodes. All remaining logs are available at `dump.log`.

Remark 2. Since each node in this basic test communicates using `ipc`, the bandwidth usage of this basic test approximates the bandwidth usage we report in Table 3 of the paper.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) *The ADKG protocol improves the average runtime and bandwidth usage per node compared to the state-of-the-art, as summarized in Table 3 of the paper.*

A.4.2 Experiments

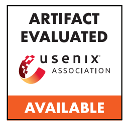
(E1) [Setup AWS machines] [30 human-minutes]. Here are the steps to set up the AWS machines:

1. Create an AWS account and sign in to the AWS Management Console.
2. Open the EC2 console and then choose Launch Instance.
3. Choose an Amazon Machine Image (AMI), which is a pre-configured virtual machine image that contains the operating system, docker and docker compose.
4. Select the region where you want to launch your instances, and then choose an instance type, which specifies the hardware of the host computer.
5. Configure the instance details, such as the number of instances, network settings, and IAM roles.
6. Choose the storage and add any additional storage volumes, if required.
7. Configure the security group, which acts as a virtual firewall for your instance to control inbound and outbound traffic.
8. Review and launch the instances.
9. Create a key pair to securely access the instances remotely over SSH. Download and save the private key file (.pem extension) on your local machine.

We automate steps 4, 5, and 8 using the AWS CLI and the configuration file https://github.com/sourav1547/htadkg/blob/main/aws/aws-config.json. Note that the configuration file specifies the regions, security group IDs, image IDs, key file path, key name, instance type, and other parameters needed to launch and configure the instances. Make sure to fill in these fields with the appropriate values. We describe the configuration file in more detail in https://github.com/sourav1547/htadkg/tree/main/aws

(E2) [ADKG experiments][1 human hour + 3 compute hours]: Follow the instructions on https://github.com/sourav1547/htadkg/blob/main/aws/README.md.

1. `cd /path/to/htadkg/aws`
2. Update the config with appropriate parameters. Run `python3 -m aws.run-on-ec2` to start the AWS instances and run the adkg command specified in the config. This command creates a current.vms file which consists of instance ids of the VMs created during this run. Subsequent runs of this command will reuse the same VMs.
3. Upon completion, the raw data from each ADKG node will be available in the `/path/to/htadkg/data/` folder in your local machine.
4. After you are done testing you can delete the VMs using `python3 -m aws.delete_vms`.



USENIX'23 Artifact Appendix: TVA: A multi-party computation system for secure and expressive time series analytics

Muhammad Faisal
Boston University

Jerry Zhang*
University of California San Diego

John Liagouris
Boston University

Vasiliki Kalavri
Boston University

Mayank Varia
Boston University

A Artifact Appendix

A.1 Abstract

TVA is a multi-party computation (MPC) system for secure analytics on secret-shared time series data. Our work introduces new secure time series protocols that compute tumbling and session window operators. We implement these operators in the semi-honest setting using the *3PC replicated secret-sharing* protocol and in the malicious setting using the *4PC Fantastic Four protocol*. We run experiments to evaluate the performance of the newly introduced protocols using queries from real-world applications. The experiments require the deployment of two MPI clusters on AWS: (i) using machines in the same region (*LAN setting*) and (ii) using machines in different regions (*WAN setting*).

A.2 Description & Requirements

TVA is a typical MPC system that consists of *input parties*, *computing parties*, *output parties*. Our experiments are designed to evaluate the computation performed by the machines that host the computing parties. Before starting any experiment, the software needs to be deployed and initialized on each machine hosting a computing party to form a MPC cluster.

A.2.1 Security, privacy, and ethical concerns

There are no such concerns for our artifact deployment and evaluation. We emphasize that TVA is an academic proof-of-concept prototype and has not received careful code review. This implementation is NOT ready for production use.

A.2.2 How to access

We host our artifact on Github.

*Work completed at Boston University.

A.2.3 Hardware dependencies

TVA does not require special hardware and can operate on general-purpose CPU-based machines. For the experiments, we use two types of machines which are available on AWS: EC2 `r5.8xlarge` instances (32 vCPUS and 256GB RAM) and EC2 `r5n.16xlarge` instances (64 vCPUs and 512GB RAM). We use the `r5.8xlarge` machines when evaluating end-to-end latency for queries in the LAN and WAN settings (experiments E1-E4 in Section A.4.2). We use the `r5n.16xlarge` machines in the experiments that compare TVA with Waldo (experiments E5, E6 in Section A.4.2).

We provide SSH access to our AWS clusters so that the reviewers can reproduce the results using the same hardware and settings we used.

A.2.4 Software dependencies

Our artifact is implemented in `C++14`. The artifact requires two dependency packages: `libsodium` (1.0.18) and `MPICH` (3.3.2). We used Linux Ubuntu (20.04.4), which also requires installation of `CMake` (>=3.15.0) and `pkg-config` (>=0.29.2). The installation may require other packages based on the operating system version's pre-installed packages. For more details, check our dependency installation script `./scripts/setup.sh`.

A.2.5 Benchmarks

Our main performance results are based on two set of experiments. The first set evaluates the end-to-end latency of the real-world queries described in Section 6.2 in the paper. The results are shown in Figure 4. The second set of experiments is used to compare TVA's performance with the Waldo time series database. We report the comparison results in Table 2 and Figure 3.

TVA's workload consists of both local computation and communication among the parties. Depending on the network bandwidth and latency characteristics of the cluster, we categorize the experimental setting as follows:

- **Same Machine**: In this scenario, computing parties are deployed on the same machine. This is useful for testing the framework setup and protocol correctness. The experiments can be run with the following command.

```
cd build && cmake .. && make cloud
mpirun -np 3 ./cloud
```

- **LAN**: In this scenario, parties are deployed on different machines but in the same data center. For these experiments, we use the AWS `us-east-2` region. Network latency in this setting is less than `1ms`. The experiments can be run using the following command, where `machine-i` represents the machine of the corresponding computing party.

```
mpirun -np 3 -hosts \
machine-1,machine-2,machine-3 ./cloud
```

- **WAN**: In this scenario, we deploy computing parties on machines across different regions. Specifically, we use the following four regions: `us-east-2` (Ohio), `us-east-1` (Virginia), `us-west-1` (California), and `us-west-2` (Oregon). Network latency between machines in these experiments should be around `40ms`.
- **WAN-Simulated**: For simulating the same conditions as those Waldo has been evaluated in, we use a simulated WAN network where latency between computing parties is fixed at `20ms`. To achieve this configuration, deploy a LAN setting as described above and run the script *"./scripts/waldo_wan.sh"* to simulate the WAN.

A.3 Set-up

A.3.1 Installation

There are two phases to correctly set up TVA. Since the system consists of multiple computing parties (either 3 or 4 parties), we first need to install the framework on each party independently and then set up the MPI cluster.

Computing Party setup: In this step, we install the TVA source code and its dependencies. First, install the following dependencies:

- **building tools**: `cmake` and `pkgconfig`.
- **libsodium**: We use it for random number generation.
- **MPICH / OpenMPI**: We use it for establishing communication among the computing parties.

For instructions on how to install these dependencies on a linux distribution, see the script "`./scripts/setup.sh`".

After completing the above step, you should be able to use the system in the "Same Machine" setting, i.e., where all computing parties run in the same machine. At this point, you can run the tests to make sure everything is correct.

```
cd ./scripts
./run_tests.sh
```

Cluster setup: To use the framework in the LAN or WAN settings, we need to replicate the previous steps for each machine in the cluster. Once we have TVA working on every machine, we can start building the cluster as follows:

1. Make sure that machines have pair wise SSH access to each other. This step depends on the cloud service provider, as firewall settings and defaults are different.
2. Modify the `/etc/hosts` file on each machine to include other computing parties with names in the format `machine-i` for LAN, and `machine-wan-i` for WAN.
3. Build the code using either the semi-honest (**semi**) or the malicious protocol (**mal**) on each machine with the following script.

```
cd scripts
./build_experiments.sh semi  # mal
```

4. Run one of the experiments under `scripts` such as `cloud.sh` depending on which results you need to reproduce.

```
cd scripts
./cloud.sh semi lan
```

A.3.2 Basic Test

After finishing setting up a machine, test the framework setup using the following command:

```
cd scripts
./run_tests.sh
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): TVA's can successfully compute online queries with rigid time constraints and evaluate complex analytics on millions of input rows with modest use of resources. For this part, we need to reproduce the results in Figure 4 and Table 3.

(C2): For multi-predicate queries, TVA provides lower latency compared to Waldo both in the malicious setting and in the semi-honest setting. For this, we need to reproduce the results in Table 2.

(C3): For window queries, TVA is up to two orders of magnitude faster than Waldo, which becomes competitive only when the ratio of the window length over the whole time domain is relatively small. The results of this experiment are shown in Figure 3 and exact numbers are reported in Section 6.1.2.

A.4.2 Experiments

The following steps are required if using a private cluster. However, we can provide SSH access to our AWS machines and, in this case, you can start from the execution step directly.

For experiments E1 and E2, use the LAN setting described in Section A.2.5. For experiments E3 and E4, use the WAN setting. For experiments E5 and E6, use the WAN-Simulated. Use machines similar to machines `5.8xlarge` for experiments E1 through E4, and machines similar to machines `5n.16xlarge` for experiments E5 and E6.

(E1): *[Semi-honest LAN] [15 human-minutes + 1.5 compute-hour]*: In this experiment, we reproduce the results in Figure 4-a, which represents the latency for the 3 application queries in the semi-honest 3PC protocol when the cluster is deployed in the LAN setting.

Preparation: Prepare a cluster with three machines in the LAN setting as described in Sections A.2.5 and A.3.1.

Execution: Follow these steps:

1. On each machine, execute the following command to build the experiments files:

```
cd scripts
./build_experiments.sh semi
```

2. Use the corresponding bash file on just one of the 3 machines to start the experiment execution.

```
# On main machine of the cluster
cd scripts
./energy.sh semi lan
./cloud.sh semi lan
./medical.sh semi lan
```

Results: The results will be appended to the files in the results directory and you can compare the new results with the old ones at the beginning of the file.

(E2): *[Malicious LAN] [15 human-minutes + 3 compute-hour]*: Follow the same steps as in E1, except deploy 4 machines and replace the argument `semi` with `mal`.

(E3): *[Semi-honest WAN] [15 human-minutes + 5 compute-hour]*: Follow the same steps as in E1, except deploy the cluster in the WAN setting and replace the argument `lan` with `wan`. Make sure to update the hosts file using `machine-wan-i` as the host name.

(E4): *[Malicious WAN] [15 human-minutes + 10 compute-hour]*: Follow the same steps as in E2, except deploy the cluster in the WAN setting and replace the argument `lan` with `wan`. Make sure to update the hosts file using `machine-wan-i` as host the name.

(E5): *[Semi-honest Waldo] [15 human-minutes + 1 compute-hour]*: In this experiment, we reproduce the results in Figure 3 and Table 2.

Preparation: Prepare a cluster with 3 machines in the LAN setting as described in Sections A.2.5 and A.3.1. Use the script specified in WAN-Simulated to configure the network latency.

Execution: Follow these steps:

1. On each machine, execute the following command to build the experiments files:

```
cd scripts
./waldo_wan.sh S
./build_experiments.sh semi
```

2. Use the corresponding bash file on just one of the 3 machines to start the experiment execution.

```
# On main machine of the cluster
cd scripts
./waldo_energy_query.sh semi wan
./waldo_table_equality.sh semi wan
./waldo_table_greater.sh semi wan
```

Results: The results will be appended to the files in the results directory and you can compare the new results with the old ones at the beginning of the file.

(E6): *[Malicious Waldo] [15 human-minutes + 2 compute-hour]*: Follow the same steps as in E5 except use 4 machines and replace the argument `semi` with `mal`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Powering Privacy: On the Energy Demand and Feasibility of Anonymity Networks on Smartphones

Daniel Hugenroth
University of Cambridge

Alastair R. Beresford
University of Cambridge

A Artifact Appendix

A.1 Abstract

In the submitted paper we present an energy measurement setup for Android smartphones and use it to perform measurements of both individual operations (micro studies) and entire protocols runs (macro studies). In this appendix and our repository we provide instructions for building the required custom hardware, software to perform and analyze measurements, and sample traces. The latter can be used to verify the functionality of the analysis tools without the custom hardware.

Due to the length of the experiments, reproducing all results is not feasible within the scope of the artifact evaluation process. Therefore, we have not applied for the "Results Reproduced" badge. Instead, we chose representative experiments that demonstrate the functionality of all involved components.

Due to the particularities of the hardware prototype the artifact evaluation was performed via an interactive online meeting. In this online meeting we executed all experiments E0, E1, and E2 while sharing our computer screen and capturing the hardware setup with a camera.

A.2 Description & Requirements

We have developed and tested all software on a system running Ubuntu 20.04. The individual programs are written using Rust (serial port logger), Python (data analysis, web service), Android Studio (various apps), and Java (interactive live plot GUI). For all components we provide instructions in the repository for building them from source and give the precise version numbers of the build tools we use. Where possible and helpful, we include Docker files.

Building the hardware requires basic skills in both 3D printing and soldering. Please make sure to follow all necessary safety measures that arise from working with the mentioned equipment and smartphone batteries. If you are interested in testing the software components independently, you can use the provided sample traces in the repository.

A.2.1 Security, privacy, and ethical concerns

HEALTH & SAFETY: While accidents are rare, working with tools and smartphone batteries carries significant risk. Before starting you must familiarize yourself with all applicable safety and compliance requirements. This includes, but is not limited to, departmental, local, federal, and international policies, laws, and regulations. We strongly recommend that you talk to a designated person in your institution that can provide you with the required training and information.

A.2.2 How to access

The repository containing all documentation, software, and other files is available here: `https://github.com/lambdapioneer/powering-privacy/tree/aec-final`. The link points to the tag `aec-final` which references the version used in the artifact evaluation process. We suggest you follow the included `walkthrough.md` file which covers all components and experiments mentioned in this appendix.

A.2.3 Hardware dependencies

The evaluation of the software requires a modern Linux computer with at least 16 GiB RAM, at least 200 GiB of free disk space, a free USB-2.0 port, and a local WiFi network.

For sharing log data from the smartphone with the local computer, the setup requires to run a web service with a IPv4 address that is reachable both from the smartphone and the local computer. The web service can be run on the local computer if its reachable by the smartphone over WiFi.

For building the hardware, you will need a Motorola Moto E6 Plus, a 3D printer, soldering equipment, and more electronic parts as per bill of material. The total costs of all parts including the smartphones are not more than 500 USD.

A.2.4 Software dependencies

The local computer should run a modern Linux distribution with support for the latest versions of Rust, Python, Java, Android Studio, and Docker. Language support might be installed via the package manager or directly using the install scripts from the respective websites. Detailed instructions are included in the repository. We used Ubuntu 20.04 for developing and testing our software.

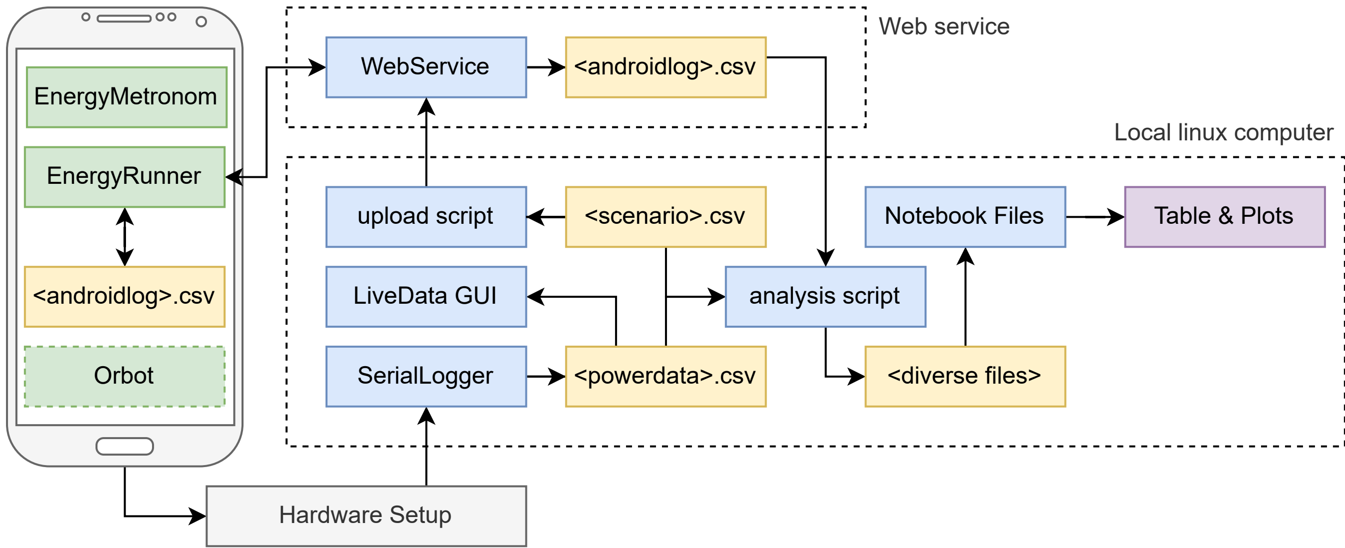


Figure 1: Overview of all components relevant to this artifact appendix. The green boxes are Android app, the blue boxes are Linux executable, the yellow boxes are files, and the purple box is the final result. The arrows indicate the flow of data. The EnergyRunner app both downloads scenario files and uploads its log data.

A.2.5 Benchmarks

Our artifact requires no external data sets. However, we provide sample traces in case the hardware setup is not available to the reviewers.

A.3 Overview of components

In Figure 1 we show all involved components relevant to the experiments described in this document.

A.4 Set-up

We recommend to read all documentation including the walk-through documentation included in the repository before starting.

A.4.1 Installation

Ensure that you are running a compatible Linux distribution, such as Ubuntu 20.04, on your local computer. Clone the repository, checkout tag `aec-final`, and follow the walk-through documentation and the linked `README.md` files to install and build all artifacts. You can skip some of these steps where you already have the required dependencies installed.

A.4.2 Hardware

You can skip this section if you have been provided with a hardware set or plan to only use the sample traces. If you build the hardware yourself, we recommend that you order all required material as early as possible to avoid disappointment due to slow shipping. Building the hardware can be done independently of setting up the software.

A.4.3 Basic Test

The experiment E0 as described in the walk-through document within the repository. It allows to verify that all components, and in particular the hardware setup, are working correctly.

A.5 Evaluation workflow

Our claims cover the functionality of the documentation, hardware setup, and software for both micro and macro studies. While the claims and experiments do not cover all results from our paper, they representatively demonstrate the functionality of the involved components. This is because running all experiments would be unreasonably time consuming for the artifact evaluation.

As described in the paper, many steps cannot be easily automated in order, and require manual operation of the device. For instance, we cannot run a persistent service on the Android device to automate start and stopping the apps, as it would prevent it from reaching its low-power states. We arranged our claims and experiments such that claim there is a one-to-one relationship between C0 and E0, C1 and E1, and so on.

A.5.1 Major Claims

(C0): The documentation and software is complete and allows skilled researchers to perform their own measure-

ments. Users can observe live plots and use data for further analysis. This includes the aforementioned based test in Section A.4.3.

(C1): Using the hardware setup one can perform a micro study that shows that the energy costs for EC operations is lower than for RSA. This reproduces Table 2 on page 8.

(C2): Using the hardware setup one can perform a macro study that shows for access to `.com` using Tor requires more power than doing so directly, but is still feasible in terms of energy costs. This reproduces parts of Figure 9 on page 11.

A.5.2 Experiments

All experiments require the aforementioned hardware and software requirements. If the hardware is not available, the provided sample traces can be used. The given *experiment-hours* provides the blocks of time that experiment runs on the smartphone device and which cannot be interrupted. They are in additional to the *human-hours* that cover software setup and analysis. If the provided sample traces are used, the experiment-time does not apply and the overall required time is reduced drastically. The `walkthrough.md` file in our repository guides you through all experiments step by step.

(E0): *[2-20 human-hours +* 1×1 *experiment-hour]: Build and test the hardware setup. Building is optional if a hardware kit is provided or the sample traces are used.*
Preparation: Acquire all hardware listed in the documentation. Compile all software as described in documentation. Build and connect the hardware as described.
Execution: Start the serial logger. Start the live logger interface. Turn the screen on and off. Navigate the live plot using keyboard and mouse actions. Stop the experiment. Alternatively, use the `sample-traces/e0/...` files.
Results: The users sees live data changing with the operations on the smartphone. In particular, turning the screen on and off are very visible events.

(E1): *[5 human-hours +* 1×2 *experiment-hours]: Use the hardware kit to measure individual cryptographic operations.*
Preparation: Start the log collecting web service running on IP *ip*. Update the source code of the Android apps and the analysis scripts with *ip* as per documentation. Upload the scenarios using the Python script. Connect the hardware as described. Install the `EnergyRunner` app and use it to download the scenario.
Execution: Select the scenario file Connect the USB-dongle for clock synchronization. Start the serial logger with the scenario file name. When instructed, turn off the screen and disconnect the USB-dongle. Wait until all operations finished. Stop the serial logger. Stop the app. Alternatively, use the `sample-traces/e1/...` files. Execute the respective analysis Notebook file.
Results: The analysis notebook outputs results that are comparable to those in Table 2.

(E2): *[5 human-hours +* 2×1 *experiment-hour]: Use the hardware kit to measure longer protocol runs.*
Preparation: Install the Orbot app and enable *Full Connection-Padding* as per documentation. Install the `EnergyMetronom` app and enable “Display over other app” as per documentation. Connect the hardware as described.
Execution: Force-stop Orbot. In the `EnergyMetronom` app select a regular execution of a `.com` website visit and set the interval to every 60 seconds. Start and observe at least one successful website load. Immediately afterwards turn off the screen and start the serial logger. If you like, observe the ongoing experiment using the live logger tool. After 10 minutes stop the serial logger and stop the app. Repeat above steps with Orbot enabled in VPN mode. Alternatively, use the `sample-traces/e2/...` files. Execute the respective analysis Notebook file.
Results: The analysis notebook outputs results that are comparable to the corresponding bars in Figure 9.

A.6 Notes on Reusability

We hope that our hardware and setup will be used by many researchers and fosters new activity in designing, implementing, and evaluating anonymity networks on smartphones. For this we include detailed information on how to measure individual operations as well as longer protocol executions in the linked repository. We are happy to assist others in building copies of our hardware setup and planning experiments. For this please reach-out to us via email.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

Acknowledgements

We would like to thank the chairs and reviewers for the granted special accommodations that allowed our hardware prototype and software to be included in the artifact evaluation process.

USENIX'23 Artifact Appendix: "The OK Is Not Enough: A Large Scale Study of Consent Dialogs in Smartphone Applications"

Simon Koch
TU Braunschweig
simon.koch@tu-braunschweig.de

Benjamin Altpeter
Datenanfragen.de e.V.
benni@datenanfragen.de

Martin Johns
TU Braunschweig
m.johns@tu-braunschweig.de

A Artifact Appendix

Our artifact submission contains our used tool chain to intercept network traffic of smartphones and analyze any present privacy dialog. This entails a set of three Scala programs and their dependencies. We are applying for the artifacts available badge.

A.1 Description & Requirements

While we only apply for the artifacts available badge we do list anticipated concerns when running our software before describing the access and depdenencies.

A.1.1 Security, privacy, and ethical concerns

Simply checking out the availability of the artificats does not entail any concerns. However, using our artificats has minor privacy and security concerns.

Ethical Concerns: There are no ethical concerns as our programs runs on self owned devices and evaluates software in a non-intrusive fashion.

Privacy Concerns: There are minor privacy concerns as using the software to evaluate the privacy violations of apps implies running the apps and observing transmitted traffic. We have seen traffic containing location data as well as limited network information (local IP addresses). However, our analysis of the observed traffic is not complete and unevaluated parts of the traffic may contain more sensitive data. We strongly advice using test accounts as well as test devices for both Android and iOS if the reviewer wants to test our software to limit the impact of possible data leaks.

Security Concerns: App measurement studies entail the execution of large amount of apps on real device within the network. Depending on the sourcing of the apps there is always the possibility of maleware being installed and executed. Using the official App Stores as a source mitigates that risk but neither Google nor Apple have perfect security checks. We strongly advise using a separate network as well as using research devices that are not intended for any personal use.

A.1.2 How to access

Our artifact has three major components with sub dependencies that need to be installed. Each linked repository contains a `README.md` that goes into details concerning installation and usage. Furthermore, we grouped our artifacts using a GitHub org that also references the paper and has a intruductory `README.md` at https://github.com/the-ok-is-not-enough[1].

app-downloader: A tool to download the current rankings as well as APKs/IPAs from the Goolge and Android App Store available at https://github.com/the-ok-is-not-enough/app-downloader[2].

However, two tools are required for the downloader to work properly which we forked and made available as a stable artifact: https://github.com/the-ok-is-not-enough/googleplay[3] and https://github.com/the-ok-is-not-enough/ipatool-py[4].

scala-appanalyzer: A tool to run apps on either an Android Smartphone or iPhone collect traffic as well avaiable at: https://github.com/the-ok-is-not-enough/scala-appanalyzer[5].

scala-plotalyzer: A tool to analyze, aggregate, and summarize data collected by the scala-appanalyzer available at: https://github.com/the-ok-is-not-enough/scala-plotalyzer[6].

[1]commit:32b904b4e21c45b345bc1b9cbfd84f6661177b6b url:https://github.com/the-ok-is-not-enough/.github/tree/32b904b4e21c45b345bc1b9cbfd84f6661177b6b/profile
[2]commit:0d41a37e4e1c5c2f4e6be19837f758f8eae98fc6
[3]commit:4c178c10bc3cc5ab2e6895016e7161296777dca0
[4]commit:a8b2d37bba40ed427420f6a2a8fa9a89c4844256
[5]commit:b618948c0d24b917b3a46a88f5c1cf6ff84571cd
[6]commit:d89a76985b20d140f949b0a86438c38de09388bd

A.1.3 Hardware dependencies

Depending on the targeted smartphone eco system different hardware is required. While it might be possible to replace some requirements (e.g., the measurement machine, or Android Smartphone to the Android Emulator) we cannot guarantee functionality or anticipate the impact on the results. In case no OpenWRT router is available it is also possible to directly use the test machine as a proxy by changing the corresponding Smartphone OS configuration, however, it is not guaranteed that apps will always adhere to this configuration and some traffic might be missed.

Always:

- WLAN Router able to run OpenWRT
- Network/Internet connection

iOS

- MacMini connected to the Internet via Cable and also to the WLAN
- rooted iPhone 8s
- Lightning Cable

Android

- rooted Samsung Galaxy A13
- Micro USB/USB Cable
- any recent computer able to run Arch Linux connected to the Internet via Cable and also to the WLAN

A.1.4 Software dependencies

Depending on the targeted smartphone eco system different software is required. While it might be possible to replace some requirements (e.g., the OS of the measurement machine) we cannot guarantee functionality or anticipate the impact on the results.

We only list the general requirements here as each of our tools contains a `README.md` with more detailed instructions that would exceed the available space limit.

Always:

- OpenWRT (on the router)
- Scala 2.13
- Go
- Python 3
- Objection
- Frida
- Postgres
- MitMproxy
- Appium

iOS

- MacOS (on the measurement machine)
- xCode
- rooted iOS 14.X (on the iPhone)
- cydia/checkRa1n (to root the iPhone)

Android

- rooted Android (on the Galaxy A13)
- ArchLinux (on the measurement machine)
- Android Studio

A.1.5 Benchmarks

None.

A.2 Set-up

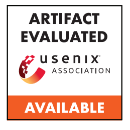
We are only applying for the artifacts available badge. However, each repository contains a `README.md` with instructions on installation and usage.

A.3 Notes on Reusability

While the provided URLs reference the Artifacts as used in our published paper we are continuing the development of our measurement framework and made some major changes significantly improving adaptability and usability. We are excited by the posibility that our tools might be of use for other researchers and strongly advise checking out development branches available at https://github.com/simkoc/scala-appanalyzer and https://github.com/simkoc/scala-plotalyzer as those represent the current state of the art as used by us. For example the new versions provide a plugin system making it easier to extend the functionality without having to do major changes on the main program. We also added emulator support for Android.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This artifact implements static code analysis for detecting prototype pollution vulnerabilities and gadgets in server-side JavaScript libraries and applications, including the Node.js source code. The analysis builds on GitHub's CodeQL framework to identify prototype pollution sinks and gadgets. We evaluate precision and recall metrics for prototype pollution detection in comparison with existing CodeQL analysis as well as the tool ODGen. Further, we evaluate the capabilities of our tool, in combination with dynamic analysis, to detect gadgets in a range of popular applications, including the Node.js source code. Finally, we evaluate the prevalence of detected gadgets on a dataset of popular libraries. All of the artifact evaluation results refer to Section 6 of the paper and the Appendix. The artifact evaluation aims for the three badges: available, functional, and reproducible.

A.2 Description & Requirements

Here we describe hardware and software requirements to run the artifact, as well as an overview of the benchmarks.

A.2.1 Security, privacy, and ethical concerns

There are no risks for the reviewers relating to security and privacy of their machines. The artifact has been used to detect 8 remote code execution vulnerabilities in production-ready applications and these vulnerabilities have been responsibly disclosed to the vendors. We do not provide any details on exploits that are yet to be fixed by the developers. Moreover, exploit generation is a manual process, hence it is not part of this artifact evaluation.

A.2.2 How to access

The artifact is accessible on GitHub at address https://github.com/yuske/silent-spring/tree/2c7cfab. The reproducibility of the results is supported by two modes: (1) a prepackaged docker container and (2) detailed instructions on how to set up the environment on own machine.

A.2.3 Hardware dependencies

We perform the experiments on an Intel Core i7-8850H CPU 2.60GHz, 16 GB RAM, and 50 GB of disk space. No specific hardware features are required for the artifact evaluation.

A.2.4 Software dependencies

We originally run our experiments (except for the experiment **E2** of ODGen evaluation) on Windows OS and presented these results in the paper. However, CodeQL and our evaluation scripts support Linux and provide similar results.

A.2.5 Benchmarks

We provide five benchmarks for our experiments. The root directory of the artifact repository contains folders with benchmark names from the list below. Clone the repository with its Git submodules and follow the instructions of Appendix A.3 to download all code of benchmark-silent-spring and benchmark-npm-packages.

(benchmark-silent-spring): We compile an open-source dataset of 100 vulnerable Node.js packages to evaluate the recall and precision metrics of our static analysis. We refer to Section 6.1 and Table 3 of the paper for details of the benchmark and our experiments against this set of packages.

(benchmark-odgen): We consider the dataset of 19 packages provided by the tool ODGen to compare our static analysis approach with the state-of-the-art results of ODGen. The paper presents the details of the dataset and analysis results in Section 6.1 and Table 3 as well.

(benchmark-popular-apps): We evaluate our approach on popular Node.js applications from GitHub. The benchmark contains exact versions of 15 analyzed applications.

The evaluation results are presented in Section 6.3 and Table 2.

(benchmark-nodejs): We run our gadget detection analysis against Node.js version 16.13.1. The source code of the analyzed Node.js is located in a folder of the benchmark. Table 1 of the paper reports all the detected gadgets and their summary.

(benchmark-npm-packages): We estimate the prevalence of the gadgets in an experiment with the 10,000 most dependent-upon NPM packages. This benchmark contains these NPM packages. We describe the results of the experiment in the last paragraph of Section 6.2.3.

A.3 Set-up

We provide two modes for testing the artifacts (1) a docker image with the prepared environment and (2) detailed instructions on how to set up the environment on own machine. To use the docker image, pull the docker image `yu5k3/silent-spring-experiments:latest` from Docker Hub, launch a docker container, and run `/bin/bash` into the container to get access to the pre-configured environment. In this mode, the reviewers may skip the setup and installation steps, and move directly to the folder `~/projs/silent-spring` in the docker container and follow the instructions from Appendix A.3.2.

The following steps describe how to set up a required environment on own machine.

(S1): Clone the ODGen repository https://github.com/Song-Li/ODGen.git and checkout commit `306f6f2`. Follow its README file to set up the tool.

(S2): Clone the Silent Spring repository with its submodules https://github.com/yuske/silent-spring.git and checkout commit `2c7cfab`.

(S3): Move to the scripts by `cd silent-spring/scripts/`. Further, it is important to run any setup and evaluation scripts using the `scripts` as a working directory.

(S4): Run the script `./benchmark-silent-spring.install-dependencies.sh` to install dependencies for benchmark-silent-spring.

(S5): Install NPM dependencies for the scripts by `npm i`.

A.3.1 Installation

The experimental evaluation requires the following software:

(I1): Node.js v.16.13.1. Follow the instruction on the official website to install Node.js.

(I2): Cloc. We use `cloc` application to count lines of analyzed code. Use in the official repository to download and install the latest version.

(I3): CodeQL v.2.9.2. Download and unzip an asset for your platform of the version 2.9.2 from the official repository. Add the path of the `codeql` folder to `PATH` environment variable.

A.3.2 Basic Test

We recommend a basic test for 1-2 NPM packages with our CodeQL queries to check that all required components function correctly. The execution of command `node ./benchmark-silent-spring.codeql.js -l 1` from directory `scripts` performs the analysis of only one NPM package from benchmark-silent-spring and stores the results at `../raw-data/benchmark-silent-spring.codeql.limit.md`. The analysis should be completed in about 3 minutes. We provide a reference file for comparison with the basic test results. The easiest way to compare the evaluation results with the reference is to execute `git diff -- ../raw-data/benchmark-silent-spring.codeql.limit.md`. The count of detected cases in the table should be the same.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our static analysis tool, built on top of CodeQL, achieves higher recall (up to 97%) for prototype pollution detection as compared to existing CodeQL analysis and the state-of-the-art tool ODGen. At the same time, it achieves moderate precision (on average 39%). This is evaluated by the experiments **(E1)** and **(E2)** described in Section 6.1 of the paper with results reported in Table 3.

(C2): Our tool has been used to uncover 8 new critical vulnerabilities in popular Node.js open-source applications. This is evaluated by the experiment **(E3)** and described in Section 6.3 and Table 2 of the paper.

(C3): We use static and dynamic analysis to detect 11 new gadgets in Node.js code that may lead to Remote Code Execution attacks. The gadget detection is evaluated by the experiments **(E4)** and **(E5)** described in Section 6.2 and summarized in Table 1 of the paper.

(C4): We estimate the prevalence of the detected gadgets on 10,000 most dependent-upon NPM packages. The measurement of the prevalence is shown by the experiment **(E6)** and described in Section 6.2.3 of the paper.

A.4.2 Experiments

All experiments should be run in the `scripts` folder to match the relative paths in the script files. All scripts collect the results of experiments in the folder `raw-data`. This folder already contains our results which can be used as reference for comparison.

(E1): Prototype pollution detection with CodeQL [1 human-hour + 3 compute-hours]: evaluate the existing CodeQL analysis and our analysis framework on benchmark-silent-spring and benchmark-odgen.

Execution: Run the following scripts:

```
>node ./benchmark-silent-spring.codeql.js
>node ./benchmark-silent-spring.baseline.
    codeql.js
>node ./benchmark-odgen.codeql.js
```

Results: The file names of the analysis results correspond to the file names with `.md` extension. The files consist of tables where *columns* contain the detected cases for the executed CodeQL queries. The last row calculates the total number of True Positives (TP) and False Positives (FP), as well as the recall and precision metrics. The result for benchmark-odgen contains only detected *sinks* that should be matched to code locations from `.PoC*.expected` files (including `.PoC.ext.expected`), e.g., `benchmark-odgen/asciitable.js@1.0.2/asciitable.PoC.expected`. We summarized benchmark-silent-spring results in Table 3 in the paper. The experiment should yield the recall and precision metrics that correspond to the metrics of *Total* row in Table 3. The results of benchmark-odgen are discussed in the last paragraph of Section 6.1.

(E2): Prototype pollution detection by ODGen [1 human-hour + 11 compute-hours]: evaluate ODGen analysis on benchmark-silent-spring and benchmark-odgen.
Preparation: Set the absolute paths to ODGen (variable `odgenDir`) and the silent-spring folder (variable `ppStuffDir`) in `benchmark-odgen.odgen.js` and `benchmark-silent-spring.odgen.js` files. This is already done for the provided docker image.
Execution: Run the following scripts:

```
>node ./benchmark-silent-spring.odgen.js
>node ./benchmark-odgen.odgen.js
```

Results: The scripts create two reports for benchmark-silent-spring and benchmark-odgen that are structured as the results of **(E1)**. The results in `benchmark-silent-spring.odgen.md` have worse metrics than we reported. This is because ODGen makes random choices and, in our experiments, we ran the ODGen tool several times and merged their best results from all runs in Table 2 (in order to compare with their best configuration).

(E3): Vulnerability detection in applications [1 human-hour]: evaluate our analysis to detect prototype pollution in Node.js applications.
Execution: Run the following script: `node ./benchmark-popular-apps.codeql.js`
Results: File `benchmark-popular-apps.codeql.md` contains the count of the detected prototype pollution cases and links to the source code of the detected sinks. The number of the detected cases corresponds to the column *Total - Cases* of Table 2 in the paper. The provided script reports two extra cases for one *parse-server* and one *sails* due to the usage of earlier version of CodeQL in the original experiments.

(E4): Gadget detection (dynamic analysis phase) [1 human-hour]: evaluate the dynamic analysis of three Node.js APIs for prototype pollution gadgets.
Execution: Run the following scripts:

```
>node ./gadgets.infer-properties.js
>node ./gadgets.dynamic-analysis.js
```

Results: The scripts report undefined properties subject to prototype pollution in the file `gadgets.dynamic-analysis.csv`. We detected 37 undefined property reads in `child_process`, `require`, and `vm` APIs, and described this experiment in Section 6.2.1. The property `TERM` can be reached on Windows but not Linux. The list of the reported properties contains *universal properties* of the identified gadgets that we describe in Table 1 in the paper.

(E5): Gadget detection (static analysis phase) [1 human-hour]: evaluate the data flow analysis for the detected properties in **(E4)**.
Execution: Run the following script: `node ./gadgets.static-analysis.js`
Results: We implement a CodeQL-based analysis to detect flows from polluted properties to sinks, and validate the results manually, as described in Section 6.2.2. The provided script summarizes the results and reports *sources* that are the exported functions triggering a reading of polluted properties and *sinks* that are the internal functions taking the read values. The report `gadgets.static-analysis.md` counts *sources* and *sinks* to show feasibility of the manual analysis. The folder `gadgets.static-analysis.tmp` contains the detected function names.

(E6): Gadgets prevalence estimation [1 human-hour]: analyze the most dependent-upon NPM packages to estimate potential exploitability of detected gadgets.
Preparation: Script `./gadgets.download-packages.sh` downloads NPM packages for analysis (execution takes 40 mins). Skip this step if you use the docker image.
Execution: Run the script (takes about 15 minutes): `node ./gadgets.prevalence-analysis.js`
Results: The last line of the script's output contains analysis results, reporting *Packages with no main - 2041; packages have relative 'require' - 4393; packages have 'child_process' methods - 350.* We report the results of our experiment in the last paragraph of Section 6.2.3 in the paper. The slight discrepancy is due to the use of different versions of the NPM packages for the analysis.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Cookie Crumbles: Breaking and Fixing Web Session Integrity

Marco Squarcina
TU Wien

Pedro Adão
Instituto Superior Técnico, ULisboa
Instituto de Telecomunicações

Lorenzo Veronese
TU Wien

Matteo Maffei
TU Wien

A Artifact Appendix

A.1 Abstract

This artifact is provided to support the evaluation of all the results presented in the paper. In particular, (i) the cross-browser testing suite used to validate the results presented in Table 2, (ii) the toolchain developed to automatically test server-side cookie parsers (Section 4.2.2), (iii) the dataset and processing code of our cookie measurement study (Section 4.4), (iv) reproducible proof-of-concept attacks against vulnerable Web frameworks (Section 6), as well as (v) the ProVerif models and scripts (Section 7).

A.2 Description & Requirements

We provide in this section all the information necessary to download the artifact and recreate the same experimental setup used to run the analysis and experiments.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is available at https://doi.org/10.5281/zenodo.8220368.

A.2.3 Hardware dependencies

The artifact does not require any specific hardware features. Notice that the repository includes a copy of the dataset used in the measurement study (Section 4.4), which is about 3GB in size.

A.2.4 Software dependencies

Most software dependencies of the artifact are packaged as Docker containers, hence we require a working Docker Engine installation before testing the artifact. When some components are not packaged as Docker containers, we provide instructions to execute them on the host machine. The readme files of each subfolder describe the specific requirements for each component. All the components of the artifact have been tested on Linux.

A.2.5 Benchmarks

The source dataset for the measurement study (Section 4.4) is the Archive dataset[1] using the optimized tables from Web Almanac.[2] Since this dataset is in the public domain, we include a copy of the processed dataset via Google BigQuery in the artifact. The resulting dataset is about 3GB in size and is provided in the `measurement` folder of the artifact. All processing queries are also included in the artifact.

A.3 Set-up

We describe in the following the steps required for the installation and the basic functionality test of the artifact. We split each of the following subsections in 5 paragraphs, each detailing the specific steps required for each subfolder. We advise reviewers to install and evaluate one component at a time.

A.3.1 Installation

Browser Tests. The browser test suite can be installed via Docker.

```
1 cd browser-tests
2 docker compose up --build
```

Ensure that ports 80 and 443 are available on the local machine and that cookies.localtest.me and sub.cookies.localtest.me resolve to 127.0.0.1. If this is not the case, follow the instructions in the readme file. Detailed information on how to install a specific version of Firefox is also included in the readme file.

Reflectors. This component includes the toolchain developed to automatically test server-side cookie parsers. Reflectors are minimal programs implemented in one of the

[1] https://httparchive.org/
[2] https://almanac.httparchive.org/

tested backends that parses HTTP requests containing a Cookie header and returns a JSON dump of the cookie names and values. All supported reflectors for PHP, ReactPHP, and Werkzeug can be installed using Docker.

```
cd reflectors
docker compose up --build
```

Please ensure that ports 1700, 1701, and 1702 are free on the local machine. The fuzzer runs on the host machine and has been tested on Python 3.10.6. The only dependency is the `requests` library, which can be installed via pip.

```
pip install --user requests
```

Measurement. The dataset for the cookie measurement study is provided in the `measurement` folder. The script used to analyze the dataset (`analyze.py`) is written in Python 3 and requires no third-party modules. The other Python script (`draw.py`) is used to generate the plot in the paper and requires the `matplotlib` and `numpy` library. The scripts have been tested on Python 3.10.6.

```
pip install --user matplotlib numpy
```

Web Frameworks. Each vulnerable framework can be installed via Docker and requires the usage of some environment variables (detailed in the readme file), for instance:

```
cd web-frameworks/express-pre-login
export VERSION="v0.5.3"; docker-compose --env-file
    ../testing.env up -d --build
```

Ensure that ports 80 and 443 are available on the local machine and that localtest.me and attack.localtest.me resolve to 127.0.0.1. The automatic testing script runs on the host machine and has been tested in Python 3.11.3. The dependencies are the `requests` and `bs4` libraries, which can be installed via pip.

```
pip install --user requests bs4
```

ProVerif. We provide an exact copy of our testing environment in the docker image `wert310/proverif:a2e281f`.

```
docker pull wert310/proverif:a2e281f
```

A.3.2 Basic Test

Browser Tests. Point your browser to `http://cookies.localtest.me` and `https://cookies.localtest.me` to ensure that the test suite is running. Accept the self-signed certificates for the HTTPS test. Both URLs should display the "Cookie Integrity Evaluator" page.

Reflectors. Ensure that the reflectors are running by executing a simple request to each of them.

```
for port in 1700 1701 1702; do curl -H "Cookie: foo=bar"
    "http://localhost:${port}"; echo; done
```

The output should be `{"foo":"bar"}` repeated 3 times. Notice that the presence of an additional whitespace character in the last row is not an issue.

Measurement. The measurement study can be executed by running the `analyze.py` script. Ensure that the script is working by running it without arguments.

```
./analyzer.py
Usage: python3 ./analyzer.py <csv_directory>
```

Similarly, the plot can be generated by running the `draw.py` script.

```
./draw.py
```

Web Frameworks. Point your browser to `http://localtest.me` and login with credentials `alice:alice`. Transfer 1 credit to `bob` and ensure your final balance is 999 credits. Access `http://attack.localtest.me`. The attacker's site is running if you obtain information in the debug session after pressing `Set-Pression`.

ProVerif. The functionality of the test can be checked by running ProVerif on one of the models without applying the fix. Run a shell of the testing environment:

```
docker run --rm -ti -v$PWD:/mnt --workdir /mnt
    wert310/proverif:a2e281f bash
```

Then execute the Flask model without fix:

```
stdbuf -o0 make -B run-flask
```

The output should include:

```
Query event(app_action_successful(cp_9,token_6)) ==>
    event(app_action_begin(b_9,token_6)) cannot be proved.
```

showing that our invariant does not hold for Flask without applying our proposed mitigation. We provide technical details on the formalization and instructions on how to verify all frameworks in the `README.md` file in the `proverif` folder.

A.4 Evaluation workflow

Below we describe the steps to reproduce the evaluation of the paper.

A.4.1 Major Claims

(C1): We performed a thorough cross-browser evaluation of known cookie integrity attacks and introduced new attack vectors classified along 4 different categories: serialization collisions due to nameless cookies (Section 4.2.1), server-side parsing vulnerabilities (Section 4.2.2), cookie jar desynchronization issues (Section 4.2.3), and broken composition of (compliant) parsers (Section 4.2.4). The artifact follows the methodology presented in (Section 4.3). Browser-side experiments can be reproduced using the provided test suite (**E1.A**), while server-side experiments can be reproduced using the fuzzer and reflectors (**E1.B**). The vulnerability affecting the AWS Lambda Proxy integrations has been fixed by Amazon and cannot be reproduced.

(C2): We also conducted a cookie measurement study aimed at assessing the prevalence of cookie name prefixes, secure cookies and nameless cookies in the top 100K websites (Section 4.4). The results of the study can be reproduced using the provided dataset and scripts (**E2**).

(C3): We performed a systematic security analysis of the top 13 Web frameworks, exposing CORF token fixation and session fixation vulnerabilities in 9 of them. Experiment **E3** reproduces these experiments as well as the results of our disculosure process.

(C4): We formally verified of the correctness of our proposed mitigation to the synchronizer token pattern using the ProVerif protocol verifier (**E4**).

A.4.2 Experiments

(E1.A): *Browser Test Suite [15 human-minutes + 2 compute-minute + 100MB disk].* Execute the test suite on Firefox-104 to match relevant findings presented in Table 2. This experiment is functional to reproduce browser-side cookie issues (C1). Due to space constraints, details on how to install Firefox an understand the output of the test suite are provided in the `browser-tests/README.md`.

(E1.B): *Server-Side Reflectors [15 human-minutes + 5 compute-minutes].* This experiment is meant to reproduce server-side cookie issues (C1). A simple fuzzer generates variations of the Cookie header, sends the same request to all reflectors, and records any differences in the JSON dumps. The provided `reflectors/README.md` file explains in detail how to interpret the obtained CSV file and match it to the 3 CVEs assigned to the discovered vulnerabilities in PHP, ReactPHP, and Werkzeug.

(E2): *Cookie Measurement [15 human-minutes + 3 compute-minutes + 3GB disk].* Reproduce the results of the cookie measurement study on the top-100K websites (C2), including the output of Table 3, Figure 4, and Table 4. From the `measurement` folder, execute the analyzer script on the two provided datasets:

```
1 python3 ./analyzer.py data-2021-07-01
2 python3 ./analyzer.py data-2022-06-01
```

A detailed explanation of the output of the script is provided in the `measurement/README.md`. Notice that the queries to obtain the datasets from Web Archive are also available in the same folder.

(E3): *Web Frameworks [10 human-minutes + 15 compute-minutes].* Each framework is provided with an automatic testing script that can be used to test the application.

```
1 cd express-pre-login
2 export VERSION="v0.5.3"; docker-compose --env-file
      ../testing.env up -d --build
3 echo "Should␣be␣vulnerable␣to␣pre-login"
4 python3 test-express-pre-login.py
```

For convenience, we also provide a script `test_all.sh` that builds and tests all the applications in sequence. Applications can also be manually tested. For the example above, the following tests can be performed: (i) Access `http://attack.localtest.me/` in a browser and press `Set Pre-session`. (ii) Open a new tab in the browser and access `http://localtest.me/`. (iii) Login as one of the users, `alice`, `bob`, or `john_doe`. The password is equal to the name of the user. You shoud start with 1000 credits. (iv) Return to the tab `http://attack.localtest.me/` and execute a transfer of 1 credit to `attacker`. (v). Return to the tab `http://localtest.me/` and refresh. Your balance should now be 999 credits. Further details in `web-frameworks/README.md`.

(E4): *ProVerif Verification [30 human-minutes + 7 compute-minutes].* Verification of the correctness of our proposed fix, i.e., refreshing the token upon login, to the synchronizer token pattern for all 7 frameworks vulnerable to the CORF token fixation (pre-login) attack. To run the experiment, follow the instructions in the `proverif/README.md` file by executing the for loop listed under the "Verifying all frameworks" section. The output of the above loop will contain, for each of the 7 frameworks, the checked properties and the ProVerif results. The expected output of each framework should contain 6 reachability queries with result `cannot be proved`, showing that all events in the model are reachable. As the last line it should contain

```
1 RESULT event(app_action_successful(cp_18,token_6)) ==>
      event(app_action_begin(b_9,token_6)) is true.
```

proving that our expected invariant is true after applying the fix to the framework.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Minimalist: Semi-automated Debloating of PHP Web Applications through Static Analysis

Rasoul Jahanshahi
Boston University
rasoulj@bu.edu

Babak Amin Azad
Stony Brook University
baminazad@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
nick@cs.stonybrook.edu

Manuel Egele
Boston University
megele@bu.edu

A Artifact Appendix

A.1 Abstract

Our artifact facilitates building and running Minimalist for phpMyAdmin 4.0.0. We packaged the artifact in a set of Docker containers. There is no restriction on the CPU architecture or the operating system to build and run the Docker containers.

In this appendix, we describe the workflow of analyzing a PHP application (e.g., phpMyAdmin) using Minimalist, identifying the set of unnecessary functions according to prior user interaction, and debloating the PHP applications. Finally, we demonstrate that debloating PHP applications leads to reducing the size of the application as well as removing security vulnerabilities.

A.2 Description & Requirements

A.2.1 How to access

Download the Artifacts from: https://github.com/BUseclab/Minimalist/releases/tag/v1.0.1

A.2.2 Software dependencies

Docker and Docker compose

A.2.3 Benchmarks

In our artifact evaluation of Minimalist, we use phpMyAdmin v4.0.0 as the benchmark for our artifact evaluation.

A.3 Set-up

A.3.1 Installation

Our instructions are based around Docker containers. Please install Docker and Docker compose to run these containers:

- Docker https://docs.docker.com/get-docker/
- Docker-compose https://docs.docker.com/compose/install/

A.3.2 Prepration

In order to run our artifact, you need to download the required packages for Minimalist as well as Less is More artifact. To do so, please run the following command to download the required packages.

```
$ cat prepare.sh  # Examine the script

$ ./prepare.sh    # Run the prepare script
```

A.3.3 Basic Test

Our basic test involves building and running Minimalist on a sample PHP web application inside a Docker container. In order to run the basic test, please run the following command in the main directory of the artifact.

```
$ cat init.sh # Examine the init script...

$ ./init.sh   # Build and run the initial test
```

In case of a successful initial test, you should see the following message.

```
Basic Test was successful.
```

A.4 Evaluation workflow

A.4.1 Major Claims

Minimalist is a debloating mechanism for PHP web applications. According to our paper, we prove the following claims regarding the evaluation of our artifact and its results:

(C1): *Minimalist reduces the size of a given PHP web application (e.g., phpMyAdmin) in terms of lines of code. This claim is proven by the reduction in size of phpMyAdmin*

in experiment (E1), which is described in Section 4.4.1 as well as Figure 7 of our paper.

(C2): *Minimalist removes the security vulnerabilities in PHP applications by removing unnecessary features. This claim is proven in experiment (E2), which is described in Table 1 of our paper.*

A.4.2 Experiments

(E1): *[Debloat] [20 human-minutes + 1 compute-hour + 5GB disk]: In this experiment, Minimalist statically analyzes phpMyAdmin 4.0.0 and generates the call-graph. Next, Minimalist debloats the web application using Less is More interface.*

Preparation: None

Execution: The first step is running the Minimalist analysis on phpMyAdmin 4.0.0. To do so, run the following commands to download phpMyAdmin, create a Docker container, prepare the Docker environment, and run the analysis.

```
$ cat step_1.sh  # Examine step_1 script

$ ./step_1.sh    # Run step_1 script
```

At the end of this step, Minimalist generates the call-graph for phpMyAdmin 4.0.0. You can compare the results regarding the number of different function calls shown in the terminal with the numbers in the first three columns of Table 1 for phpMyAdmin 4.0.0.

The next step includes debloating phpMyAdmin using the Less is More (LIM) container. To do so, run the following commands to run the LIM container.

```
$ cat step_2.sh  # Examine step_2 script

$ ./step_2.sh    # Run step_2 script
```

After running the LIM container, you can import Minimalist results to LIM either manually or automatically. You can follow our tutorial in our Github repository to import the results manually. Run the following command to import the results automatically. Note that this process takes up to 20 minutes.

```
$ cat auto_import.sh #Examine import script

$ ./auto_import.sh   #Run import script
```

Before debloating, run the following command to measure the lines of code (LoC) for phpMyAdmin 4.0.0.

```
$ ./phploc.sh   # Run the phploc Docker
```

Furthermore, run the following command to perform a SQLi attack on phpMyAdmin 4.0.0. For a successful attack, the server takes more than five seconds to respond.

```
$ ./exploit.sh   # Run exploit script
```

In the last step, visit the following link to start the debloating process.
http://localhost:8086/admin/software_file/description

In the following link, click on the `add` button in the top right corner, fill out the form using the following inputs, and click `populate database`.

```
Software: phpMyAdmin
Version: 4.0.0
Web App Directory: /var/www/html/4.0.0/
Description: Artifact
```

After finishing the above process, click on the `Debloat functions` to start the debloating process. This process takes up to five minutes to complete. For more information, you can follow the visual tutorial in our Github repository.

After finishing the debloating process for phpMyAdmin 4.0.0, you can run the `phploc` script again to calculate the LoC for the debloated web application. As shown in Figure 7 in our paper, you can observe the reduction in LoC for phpMyAdmin 4.0.0 before and after debloating.

(E2): *[Attack] [10 human-minutes + 10 compute-minutes]: In this experiment, you perform a SQLi attack to examine the removed vulnerability from Minimalist-debloated phpMyAdmin.*

Prepration: To perform this experiment, do not stop the LIM container.

Execution: In order to perform the SQLi attack, run the following command in a separate terminal.

```
$ cat exploit.sh   # Examine exploit script

$ ./exploit.sh    # Run exploit script
```

Results: In the case of a failed attack, the response from phpMyAdmin is immediate. Otherwise, the server takes more than five seconds to respond.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: How Effective is Multiple-Vantage-Point Domain Control Validation?

Grace H. Cimaszewski[†] Henry Birge-Lee[†] Liang Wang[†] Jennifer Rexford[†]
Prateek Mittal[†]
[†] Princeton University

A Artifact Appendix

A.1 Abstract

We model the security of multiple-vantage-point domain control validation (more briefly, multiVA) by performing quantitative Internet-level simulations of the full-graph DNS resolution of domain names included in Let's Encrypt certificates. At a high level, the submitted artifact consists of 3 parts:

1. the *Internet topology simulator*, which calculates the effects of equally-specific prefix length BGP hijacks by selected attacker ASes;
2. the *DNS resolver*, which performs full-graph DNS lookups of domain names to record all IP addresses vulnerable to BGP hijacks (i.e., not DNSSEC-signed);
3. the *resilience processor*, which combines the output of (1) and (2) to compute a resilience value in the range of 0-1.0 to describe how likely a domain name may be attacked by a random attacker AS using BGP hijacks during the domain control validation process to gain a fraudulent certificate.

This artifact aims to reproduce the results in sections 7 and 8 of our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We extract only the domain names listed in the subjects of Let's Encrypt-issued certificates, which are publicly available in certificate transparency logs. The DNS lookup tool performs a default number of 10 lookups per domain name, which is assumed to be a manageable request volume for the domains' nameservers. BGP simulations do not entail any real hijacks or announcement of prefixes, and do not leak information about private routing policies. Running our code does not require any admin/sudo privileges or elevated access.

A.2.2 How to access

The artifact can be accessed by downloading our tagged public Github project Github project. All the requisite data, containers, and code are contained within the Git repository. Git clone the artifact access URL provided in HotCRP submission.

A.2.3 Hardware dependencies

For Experiment E1, simulation load is CPU-intensive: to be able to run the end-to-end experiments in a practicable amount of time, access to a many-cored (e.g., 64+) computing system with ample memory (approximately 2GB per core). Performing DNS lookups will require Internet access and corresponding firewall rules to allow inbound/outbound traffic.

A.2.4 Software dependencies

The main software dependency needed to run the artifact is Docker (available for install at https://www.docker.com/). Other required dependencies (Python 3.8, libraries, etc.) are packaged within the containerized environment.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Git clone the artifact access URL: https://github.com/inspire-group/routing-aware-dns/commit/23194fc824633122cbfb79206a62ac662389f63c.
`cd` into the cloned repository directory. From here, build the Docker container:

```
docker build --tag full-graph-dns-resolver .
```

After image build successfully completes, begin running container in the background:

```
docker run --name dns-resolver -d
full-graph-dns-resolver
```

From here, enter the container with interactive shell to execute the subsequent commands for the artifact:

```
docker exec -it dns-resolver bash
```

More detailed instructions for setup are included in the README of the artifact repository.

A.3.2 Basic Test

Validate that the DNS full-graph resolver tool properly executes (can send/receive DNS queries, local Unbound stub resolver is live):

```
python3 log_processor_artifact.py -d
data/domains_random_samp_small.txt
```

This runs lookups for a sample of 1397 domains (0.1% of our dataset) for validation purposes.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Resilience of domain names takes a noticeable hit when including the DNS nameserver in the BGP hijack attack surface. Considering the current level of RPKI deployment in the Internet counterbalances some of this resilience hit. This is illustrated by experiment (E3), which reproduces results described in Section 7 of the paper.*

(C1): *multiVA deployments with only one or two additional vantage points in diverse public cloud providers can strengthen resilience values to above 90%. This is reproduced by experiments (E3,4) (corresponding to Section 8 of the paper).*

A.4.2 Experiments

(E1): *[Full Internet-scale topology simulations]*
How to: Please see the README of the pki-topology-simulator submodule for instructions.
Approximate runtime: 192 CPU hours.

(E2): *[DNS full-graph resolution of Let's Encrypt domain names]*
How to: Please see the README of the routing-aware-dns repo for instructions and commands.
Approximate runtime: 1.5 hours

(E3): *[Calculation of domain name-level resilience]*
How to: See instructions for `resilience.py` in `princeton-letsencrypt/resilience-computation`. lease see documentation for the resilience.py script in the pki-resilience-processing submodule.
Approximate runtime: 2 hours

(E4): *[Results analysis]*
How to: Please see documentation for the interpret_results.py script in the pki-resilience-processing submodule.

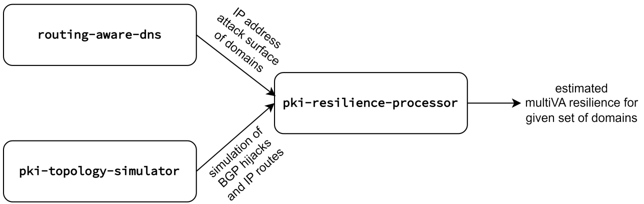


Figure 1: Integration of the artifact submodules.

Approximate runtime: <5 minutes.

A diagram showing the interconnection between the above listed experiments/submodules is given in Figure A.4.2.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Bypassing Tunnels: Leaking VPN Client Traffic by Abusing Routing Tables

Nian Xue
New York University

Yashaswi Malla, Zihang Xia, Christina Pöpper
New York University Abu Dhabi

Mathy Vanhoef
imec-DistriNet, KU Leuven

A Artifact Appendix

A.1 Abstract

In this artifact, we present the instructions for conducting two novel attacks (i.e., called LocalNet Attacks and ServerIP Attacks) that are described in our paper, causing VPN clients to leak traffic outside the protected VPN tunnel. In our paper, we claim that the traffic to the local network and to the VPN server itself can be manipulated by abusing routing tables such that it will be sent in plaintext outside the VPN tunnel. Through extensive experimentation with various VPN clients, we have identified that these attacks pose a general vulnerability across multiple OSs.

A.2 Description & Requirements

Our attacks manipulate the client's routing table such that traffic will be sent outside the VPN tunnel, i.e., without encryption. Normally, when the VPN is not enabled, a client's routing table might look like the following:

```
[tester@zbook ~]$ ip route
default via 192.168.1.1 dev wlp0s20f3
192.168.1.0/24 dev wlp0s20f3 scope link
```

The IP address of the client in this example is 192.168.1.101. The two output lines mean:

- The first line says that by *default* all outgoing IP packets are forwarded via 192.168.1.1. Here 192.168.1.1 is the router. The rule also specifies "dev wlp0s20f3", meaning that the packets are sent over the wlp0s20f3 Wi-Fi network card. All combined, all outgoing IP packets are, by default, sent to the router using the Wi-Fi network card.
- The second line is an exception [2] to the above rule: all IP packets to 192.168.1.0/24, so to IP addresses between 192.168.1.0 and 192.168.1.255, are sent over "dev wlp0s20f3", specifically over the Wi-Fi network card. Moreover, "scope link" means these IP addresses are directly reachable: the packets can directly be sent to their destination instead of first being forwarded to the router.

When a VPN is enabled, a client's routing table might look like this:

```
[tester@zbook ~]$ ip route
default via 10.8.0.1 dev tun0
76.26.140.111 via 192.168.1.1 dev wlp0s20f3
192.168.1.0/24 dev wlp0s20f3 scope link
```

Here, the IP address of the VPN server is 76.26.140.111. The first rule says that by default, all outgoing IP packets are sent over "dev tun0". Here tun0 is a virtual network card representing the encrypted VPN tunnel. In other words, by default, all packets are sent through the VPN tunnel. There are two exceptions:

1. The second rule says that packets with as destination the VPN server must be sent to the router using the Wi-Fi network card. This exception avoids a rooting loop where already-encrypted VPN packets would otherwise get encrypted again.
2. The third rule is the same as when the VPN wasn't enabled: all packets to the local network (notice the "scope link") are directly transmitted over the Wi-Fi network card to the destination (so not through the VPN tunnel). This assures that local devices in the network, such as printers and file servers, remain accessible when using the VPN.

A.2.1 Security, privacy, and ethical concerns

During the experiment, two types of traffic (see two claims) may be sent outside the VPN tunnel. It is best not to enter sensitive information while doing the experiments.

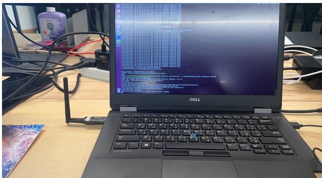
A.2.2 How to access

We have compiled a GitHub repository that provides a readme and additional necessary files.

A.2.3 Hardware dependencies

Table 1 displays the list of required soft- and hardware equipment. In terms of hardware, to conduct the experiment, a malicious AP (Access Point) is necessary to create an AP on any channel. Often the built-in Wi-Fi network card of a laptop can be used. Alternatively, an an external wireless USB adapter can be used, such as the Panda Wireless PAU06 300Mbps Wireless N USB (see Figure 1). The test platform we used is shown in Figure 1.

(a) A Panda PAU06 300Mbps Wireless USB Adapter.



(b) A laptop with an external wireless USB Adapter.

Figure 1: Our example test platform running Ubuntu.

Table 1: Hardware and Software requirements to conduct the experiment.

Class	Item name
hardware	Laptops or cellphones, e.g., iPhones or Androids
hardware	Wireless network card (built-in or USB dongle)
software	Wireshark
software	`create_ap` script
software	VPN clients

A.2.4 Software dependencies

The `create_ap` script to start and configure the AP is required [3]. Wireshark can be used to check leaks outside the VPN tunnel by inspecting traffic. Sections A.4.2 and A.4.3 provide detailed instructions on which commands to run.

The targeted commercial VPN clients are available on Apple App Store and Google Play Store, or on the vendor's website, and can be directly downloaded from these stores. The paid VPN apps require a subscription.

A.2.5 Benchmarks

None.

A.3 Set-up

This section includes all the installation and configuration steps required to prepare the environment to be used for the evaluation of the attacks.

A.3.1 Installation

We used the `create_ap` tool to create a Wi-Fi network for the tests. The generic installation instructions are available here: create_ap. On certain Linux distributions, it can be installed using the package manager. On Ubuntu, it requires to install the following dependencies:

```
sudo apt install hostapd wireshark
```

A standard AP can be created using the command:

```
sudo create_ap wlan1 wlan0 testnetwork abcdefgh
```

```
garuda@Superluminal:~$ sudo create_ap wlx9cefd5fcf304 wlo1 testnetwork abcdefgh
[sudo] password for garuda:
WARN: Your adapter does not fully support AP virtual interface, enabling --no-virt
Config dir: /tmp/create_ap.wlx9cefd5fcf304.conf.3FaIBAhI
PID: 3793
Sharing Internet using method: nat
hostapd command-line interface: hostapd_cli -p /tmp/create_ap.wlx9cefd5fcf304.conf.3FaIBAhI/hosta
pd_ctrl
Configuration file: /tmp/create_ap.wlx9cefd5fcf304.conf.3FaIBAhI/hostapd.conf
```

Figure 2: An example of creating an AP.

Figure 2 shows an example of how to create a Wi-Fi network called *testnetwork* with password *abcdefgh*. The arguments `wlan1` and `wlan0` depend on the machine used for the test:

- The argument `wlan0` refers to the built-in network card and may be different depending on the machine. Find out this name by executing `ip addr` and picking the interface that is assigned an IP address.
- The argument `wlan1` refers to the Wi-Fi dongle/wireless USB adapter plugged in. Find out its name on the machine by executing `ip addr` before and after plugging in the Wi-Fi dongle and seeing which interface was added.

One should now be able to connect to the created Wi-Fi network. To inspect the traffic of any client connect to the AP start Wireshark and listen for packets on the 'ap0' interface (or on the interface of the Wi-Fi dongle in case it does not support virtual interfaces).

Errors and warnings:

- If the error "ERROR: Failed to initialize lock" occurs, then execute: `sudo rm /tmp/create_ap.all.lock`
- The warning "Your adapter does not fully support AP virtual interface" means the Wi-Fi dongle cannot simultaneously act as a client and AP. If creating the Wi-Fi network fails, then try a different USB dongle.

A.3.2 Basic Test

Install the VPN app and connect to the Wi-Fi hotspot. Open Wireshark on the platform to monitor packets. Then enable the VPN and perform the following tests.

A.4 Evaluation Workflow

This section includes all the operational steps and experiments which must be performed to evaluate our attacks.

A.4.1 Major Claims

(C1 – LocalNet): *Traffic to local IP addresses is not sent through the VPN tunnel.*

(C2 – ServerIP): *Traffic sent to the IP address of the VPN server is not (again) sent through the VPN tunnel.*

A.4.2 Testing LocalNet Attacks

A quick method to test for this vulnerability is to make the router hand out non-RFC1918 IP addresses for the local network, e.g., using 207.241.237.0/24 for the local network. Then enable the VPN and try to visit visit http://207.241.237.3/. In Wireshark, this should result in ARP requests for the IP address 207.241.237.3, indicating that the tested VPN is vulnerable to the LocalNet traffic leak attack.

Alternatively, start the `create_ap` script to hand out public IP addresses. For example, if we want to intercept traffic to web.archive.org, which has IP address 207.241.237.3 at the time of writing, the hotspot has to hand out IP addresses from a subnet that contains this IP address. This can be done by starting `create_ap` as follows:

```
sudo create_ap wlan1 wlan0 testnetwork abcdefgh
 -g 207.241.237.3
```

Now connect with the created AP and enable the VPN client. Open Wireshark. Then try to visit http://207.241.237.3 in a browser. If TCP SYN packets can be seen to 207.241.237.3, this means that the VPN app is vulnerable: by using the Wireshark filter `tcp.flags.syn == 1`, it is easy to filter for plaintext TCP SYN packets. One successful example is shown in Figure 3.

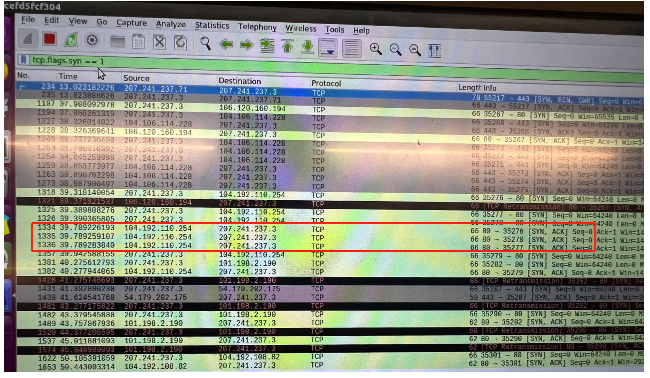


Figure 3: An example of LocalNet Attacks. The target IP address is 207.241.237.3.

A.4.3 Testing ServerIP Attacks

Start the `create_ap` script and then connect with the device being tested:

```
sudo create_ap wlan1 wlan0 testnetwork abcdefgh
```

Now start capturing frames on the test platform. After starting to capture frames, connect to the VPN server, and then identify the IP address of the VPN server based on the transmitted traffic in Wireshark. Then visit "http://$VPN_SERVERIP". If there are no plaintext TCP SYNs in Wireshark, then the VPN client is not vulnerable (we can use the Wireshark filter `tcp.flags.syn == 1` to filter for plaintext TCP SYN packets). If the VPN protocol is using TCP or UDP then you can also try to visit "http://$VPN_SERVERIP:$PORT" where we add the port that is also used by the VPN server. One successful attack is shown in Fig. 4.

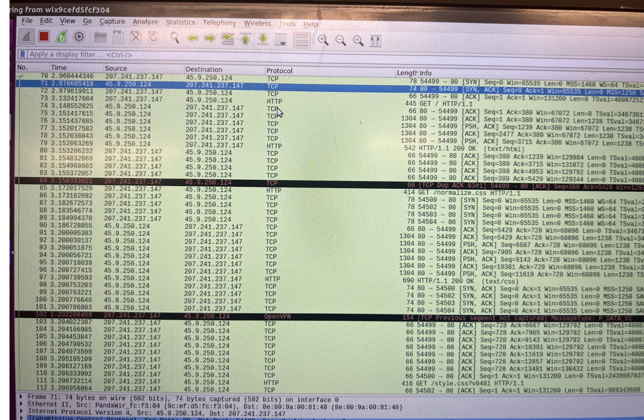


Figure 4: An example of ServerIP Attacks. The IP address of the VPN server is 45.9.250.124.

In case there are plaintext TCP SYN packets, the next step is to test whether the VPN client used plaintext DNS to find the VPN server's IP address. To determine this, we can use the Wireshark filter '`dns.a == $VPN_SERVERIP`'. If there are any results, then the VPN client is highly likely vulnerable.

A.5 Stable URL

We provide a stable URL where the community can find the final copy of our artifact in order to achieve replicability of the experiments [1].

References

[1] https://github.com/vanhoefm/vpnleaks, 2023.

[2] Martin A. Brown. Guide to IP Layer Network Administration with Linux. http://linux-ip.net/html/routing-selection.html, 2013. Accessed June 12, 2023.

[3] Yiannis M. create_ap. https://github.com/oblique/create_ap, 2013. Accessed September 12, 2022.

USENIX'23 Artifact Appendix: Greenhouse - Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation

Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy,
Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, Adam Doupé,
Tiffany Bao, Yan Shoshitaishvili, Ruoyu Wang

Arizona State University

A Artifact Appendix

A.1 Abstract

The primary artifact provided is a Docker image containing an implementation of the Greenhouse prototype presented in our paper, along with the complete dataset of 7,140 firmware images referenced in our evaluation. Greenhouse is a Python3 framework that implements *user-space single-service rehosting*, using various *interventions* detailed in our paper to enable the emulation of a specific web-facing service for a given firmware image. Unlike previous rehosting works, the rehosted firmware service is executed via user-space emulation (qemu-user) instead of a full-system emulation environment.

We evaluated Greenhouse on a dataset of 7,140 firmware images from nine different vendors to demonstrate its scalability and generalizability. Greenhouse successfully rehosts 2,841 HTTP web-services, and an additional 685 web-services to partial connectivity. Our experiment demonstrates the usability of these images by finding 717 N-day vulnerabilities using the open-source framework Routersploit and 26 zero-day vulnerabilities through fuzzing with AFL++.

A.2 Description & Requirements

Greenhouse was evaluated using a kubernetes cluster containing 42 nodes and over 2,000 CPU cores in order to complete our analysis on 7,140 firmware images. The Docker image packaged in this artifact contains an entrypoint script that the cluster pods use to run Greenhouse on each firmware target. As this script is designed for automation with our kubernetes cluster setup, we have also provided a *run.sh* wrapper script for the purposes of manual evaluation. Our submitted artifacts consist of the following items:

- *greenhouse-ae.tar*, a prepackaged Docker image containing our experiment setup for manual evaluation
- *greenhouse-rehosted.csv*, a file detailing the rehosting success of each sample in our dataset
- *gh2routersploit.csv*, a file mapping the 717 N-days found to their respective rehosted targets
- source code and instructions for building the Greenhouse docker image, fuzzer component and minikube setup on GitHub

Within the prepackaged Docker container are the following:

- a standalone version of Greenhouse for manual evaluation + a run script */gh/run.sh*
- a modified routersploit framework used to find the 717 N-days on our rehosted images + a run script */routersploit/run_routersploit.sh*
- 2 crashing input files that demonstrate two of the 26 zero-day vulnerabilities discussed in our paper that have since been publicly released by D-Link

A.2.1 Security, privacy, and ethical concerns

Greenhouse itself has a low risk of causing issues on the machine it is run on. However, many of its functions require control of device mounts and network interfaces that necessitate that the Docker image is run in *privileged* mode. While Greenhouse itself does not perform any malicious activity, the firmware it is emulating may execute commands that affect the host machine. It is thus recommended to run Greenhouse within the provided container to minimize the impact of such behavior on the host machine.

A.2.2 How to access

A copy of our artifact is available on Zenodo (https://doi.org/10.5281/zenodo.8217895). We also open-sourced the code used to build our Greenhouse artifacts on GitHub[1].

[1] https://github.com/sefcom/greenhouse/tree/08f7caf45 6f31de4f9c25325302705f7881a5e39

A.2.3 Hardware dependencies

Greenhouse requires at least 1 CPU core and 8GB of RAM. As the amount of storage needed varies based on the firmware and scale of the job, we recommend having at least 50GB of disk space available. Our full dataset of all 7,140 firmware images requires at least 125GB of disk space. The total disk space of our experiment, including all rehosted images and log files, is approximately 655GB. To perform large-scale evaluation of Greenhouse, a kubernetes cluster is necessary. Running on a local minikube instance is possible, but is unstable compared to kubernetes and may have reduced rehosting performance.

A.2.4 Software dependencies

Greenhouse was tested on a host machine using Ubuntu 20.04 and Python 3.7. Greenhouse is dependent on *qemu-user, docker, angr* and *binwalk*, and makes use of another rehosting tool, *FirmAE*, as a supplementary component. The Docker image provided as part of the artifact contains a stable, working version of Greenhouse for evaluation and all third-party software needed for it to run. The artifact Docker image provided must be run in *privileged* mode for optimal results. It is recommended that the host machine be running Ubuntu 20.04 or later, and have Docker and *docker-compose* installed.

A.2.5 Benchmarks

Greenhouse was run on a dataset of 7,140 firmware images crawled from nine different vendors. This dataset is hosted privately as part of our artifact submission. Please contact the authors for access to the dataset if necessary.

A.3 Set-up

Install Docker and *docker-compose* on the host machine.

- *Docker version 24.0.2, build cb74dfc*
- *docker-compose version 1.29.2, build 5becea4c*

To manually validate the rehosted images, we recommend installing *curl* and a web-browser such as Firefox.

Greenhouse and Docker use the network ip addresses in the range 172.17.0.0 and 172.21.0.0 by default. We recommend keeping these network ranges open. A significant number of firmware web services were observed to make use of addresses in the range 192.168.0.0. Thus, we recommended ensuring that ip addresses in this range are available during the rehosting and testing process.

A.3.1 Installation

- Load the Docker image with *'docker load -i greenhouse-ae.tar'*
- Check that the image *greenhouse:usenix-eval-jul2023* is present *'docker image list -a'*
- Start the container in privileged, interactive mode:

  docker run --privileged -v /dev:/host/dev -it greenhouse:usenix-eval-jul2023 bash
- Copy a firmware image file from the dataset into the Docker container:

  docker cp <externalpathtoimage> <container-name>:/<imagepath>
- Inside the Docker container, run the setup script:

  bash /gh/docker_init.sh
- The container and target are now ready.

A.3.2 Basic Test

The provided Docker image comes with a simple bash script */gh/test.sh* that can be run from within the Docker container. One run, the script should exit with the message 'All tests pass!' after about a minute of execution.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Greenhouse is able to perform user-space rehosting of single-services with a success rate of 39.7%. This is proven by experiments described in Section 7.2 of our paper. The results of our evaluation are reported in Table 2 of our paper, and reflected in the greenhouse-rehosted.csv table provided as part of this artifact evaluation. Note that due to the non-deterministic nature of rehosting firmware, the exact number of rehosted services may fluctuate, but should average out to our experimental numbers over a sufficiently large dataset.*

(C2): *Firmware images rehosted by Greenhouse are of sufficient fidelity to be used with dynamic analysis to find real-world vulnerabilities. Greenhouse found 717 N-day vulnerabilities on images it rehosted using routersploit, and 26 zero-day vulnerabilities via fuzzing with AFL++. This is described in Section 7.4 and 7.5 of our paper, with results specified in Table 7 for the routersploit N-days and Table 9 for the crashing inputs found*[2]*. A breakdown of these numbers is reflected in the routersploit.csv table provided.*

[2] We do not provide all the crashing inputs mentioned in Table 9 as not all have been made public. The 2 crashing inputs provided with the artifact were publicized here: https://supportannouncement.us.dlink.com/announcement/publication.aspx?name=SAP10313

A.4.2 Experiments

(E1): [Rehosting a single firmware] *[30 human-minutes 2 to 8 compute-hours + 50GB disk]*: This section describes how to run Greenhouse to rehost a single firmware image using the provided Docker image artifact. This experiment validates C1.

Preparation: Make sure to setup the Docker container provided and the target sample as discussed in the Installation section above.

Execution: To rehost a target firmware image with Greenhouse, run

/gh/run.sh <brand> <image-path-in-container>

from inside the Docker container. The script performs the rehosting from start to finish, printing logs to *stdout* and */tmp/gh.logs*. A step-by-step can be found in the README file with the rest of our Zenodo artifact. A larger scale, parallelised approach to running Greenhouse can be done with a kubernetes cluster or minikube setup. Instructions on how to do so can be found in the MINIKUBE.md file on our GitHub.

Results: Greenhouse takes approximately 2 to 8 hours to rehost an image. When it completes, the script will print 'GHREHOST COMPLETE'. The rehosted image itself can be found inside the container under */gh/results/<sha256hash>*.

The file *config.json* describes the results of the rehosting. A 'SUCCESS' result corresponds to the Interact column of Table 2, which contributes to our total of 2,841 rehosted images. More detailed instructions on how to manually run and evaluate an individual rehosted image are in the README file.

Note that the Greenhouse rehosted services may superficially deviate from the original, though functionality is usually unaffected. As emulating firmware images tends to come with a degree of non-determinism, results observed may also vary on a sample-to-sample basis. This should average out over larger sets for a given brand.

(E2): [Exploit replay with routersploit] *[30 human-minutes + 4 compute-hours + 50GB disk]*: This section describes how to use the rehosted image created in E1 with routersploit and crashing scripts to validate C2.

Preparation: Make sure that a rehosted image is available inside the container with a folder named debug (e.g. */gh/results/<sha256hash>/debug.*)

Execution: Routersploit can be run inside the Docker artifact via a script given the path to a rehosted image (e.g. */gh/results/<sha256hash>.*)

/routersploit/run_routersploit.sh <path-to-rehosted-image>

Results: The script takes approximately 4 hours to run all 125 N-days that are built into the routersploit framework used for our evaluation. Results should be automatically consolidated inside */routersploit/results/vulnerable.csv*. 'gh2routersploit.csv' is a breakdown mapping each routersploit N-day to the rehosted sample on which it found the vulnerability.

(E3): [Validating crashing PoCs] *[10 human-minutes + 0.2 compute-hours + 50GB disk]*: This section describes how to use the two crashing inputs provided on their corresponding Greenhouse rehosted services to validate C2. Only two of the 26 vulnerabilities discovered are provided as the rest have to yet to be made public at the time of this report.

Preparation: Make sure that a rehosted image is available inside the container with a folder named debug (e.g. */gh/results/<sha256hash>/debug.*) The crashing input files are available inside the folder */crashing_inputs*.

Execution: Follow instructions in the README on starting up a Greenhouse rehosted image manually using docker-compose. Once the rehosted firmware is fully up, emit the crashing input to its target using the built-in netcat client:

cat <crashing-input-file> | nc -w2 <ip> <port>

Results: The two crashing inputs provided are for two zero-days found for the *DIR-601_REVA_1.02* and *DIR-825_REVB_2.03* firmware samples in Table 9 and Table 11 of our paper. Emitting them to the rehosted web service should cause it to crash, with a segmentation fault in the terminal output of the artifact container.

A.5 Notes on Reusability

Greenhouse can be extended to run on other types of webservices. We provide two such extensions in the image for UPNP and DNS services.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: *ACFA*: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation

Adam Caulfield
Rochester Institute of Technology

Norrathep Rattanavipanon
Prince of Songkla University, Phuket Campus

Ivan De Oliveira Nunes
Rochester Institute of Technology

A Artifact Appendix

A.1 Abstract

The artifact of *ACFA* is a hybrid (hardware/software) architecture to enable secure auditing of vulnerability sources and guaranteed remediation when compromise is detected on a remotely deployed MCU. *ACFA* prototype is written in C and Verilog. It is designed alongside an open-source TI MSP430 (openMSP430) and evaluated on a Basys3 FPGA. The artifact includes Python scripts to execute an end-to-end active *CFA* protocol between a remotely deployed MCU Prover ($\mathcal{P}$rv) equipped with *ACFA* and a Verifier ($\mathcal{V}$rf) who manages the MCU and verifies reports from $\mathcal{P}$rv. This appendix aims to assist evaluators in verifying the following *ACFA* major claims: low hardware cost of the hybrid *CFA* design, the ability to audit periodic runtime reports containing fixed-size control flow logs ($\mathcal{CF}_{Log}$), and the ability to execute a guaranteed remediation action as soon as a compromise is detected.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Commit tree `9cf6550` of the *ACFA* Github Repository.

A.2.3 Hardware dependencies

The Basys3 Artix-7 FPGA development board is required.

A.2.4 Software dependencies

The current version was evaluated using the 64-bit Ubuntu 18.04 OS. Xilinx Vivado Toolset 2021.1 or higher is required for synthesizing Verilog files and generating a bitstream for the Artix-7 FPGA. *ACFA* build scripts install Ubuntu packages dependencies in **Part 1** of Sec. A.3.1. A minimum Python version of 3.6.9 is required, and Python dependencies are specified in **Part 3** of Sec. A.3.1.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Part 1: Download *ACFA* source code and Ubuntu libraries:

1. Clone the *ACFA* Github Repository
2. `cd` into the directory `scripts`
3. Run `sudo make install`

Part 2: Download and install Xilinx Vivado

1. Visit Xilinx Vivado Download page.
2. Select the latest version of Vivado that supports Ubuntu.
3. Download and follow directions in the installer.

Part 3: Install `pyserial` using:

- `sudo apt install python3-serial`.

. Verify Python required packages from standard distribution:

- `time, hmac, hashlib, argparse, pickle, dataclasses, os, collections`.

Part 4: Create *ACFA* project in Vivado

- Follow the instructions from `README.md` in the *ACFA* Github Repository to *Create ACFA project in Vivado*.

A.3.2 Basic Test

A simple functionality test includes running a Vivado behavioral simulation of a basic application, an Ultrasonic Sensor, on *ACFA* equipped MCU.

1. Open `openMSP430_defines.v` to set *ACFA* configurations. For the basic test, everything will be simulated in Vivado. Therefore, the flag `ACFA_EQUIPPED` should be set. However, the flag `ACFA_HW_ONLY` should not be set for any simulation. Therefore, "comment-out" this flag by adding "//" to the start of line 58.

2. Now we are ready to synthesize openmsp430 with *ACFA* hardware. On the left menu of the PROJECT MANAGER, click "Run Synthesis", and select execution parameters (e.g., number of CPUs used for synthesis) according to your PC's capabilities. This step takes 2-10 minutes.
3. If synthesis succeeds, a window to "Run Implementation" will appear. Do not "Run Implementation" for the basic test, and close this prompt window.
4. In Vivado, click "Add Sources" (Alt-A), then select "Add or create simulation sources", click "Add Files", and select everything inside `openmsp430/simulation`.
5. Open the `tb_openMSP430_fpga.v` file and find lines 193-202. These lines open `*.cflog` files to simulate the transmission of $\mathcal{CF}_{Log}$ slices for the basic test. Therefore in lines 193-202, replace `<LOGS_FULL_PATH>` with the full file path of the `logs` subdirectory of the *ACFA* directory.
6. Now, navigate to the "Sources" window in Vivado. Search for `tb_openMSP430_fpga`, and in the "Simulation Sources" tab, right-click `tb_openMSP430_fpga.v` and set its file type as the top module.
7. Go back to the Vivado window, and in the "Flow Navigator" tab (on the left-most part of Vivado's window), click "Run Simulation," then "Run Behavioral Simulation."
8. On the newly opened simulation window, select 8ms as the time for the simulation to run. Then press "Shift+F2" to run.
9. The simulation waveform will show two *ACFA* triggers occur during the execution due to the device boot and the program ending. In the `logs` sub-directory of the *ACFA* directory, two `*.cflog` files were generated. If two `*.cflog` files are generated and match the contents of `logs/expected_cflogs_basic_test/`, the basic test has completed successfully.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) Hardware Cost: *ACFA* incurs an additional hardware cost of 275 Look-up Tables (LUTs) and 202 Flip-Flop registers (FFs). This is proven by the experiment (E1). The results of this experiment reflect the results illustrated in Fig. 11 and discussed in Sec. 6.1 of our paper.
(C2) Secure Auditing via Active *CFA*: *ACFA* generates a series of control flow logs of a maximum size which allow for a continuous and active control flow attestation protocol. This is proven by the experiment (E2) and demonstrates the offline and online phases described in Sec. 6.2 of our paper when $\mathcal{P}$rv is not compromised.
(C3) Compromise Detection & Guaranteed Healing: When a control-flow violation has been detected by $\mathcal{V}$rf, *ACFA* immediately executes a remediation action. This is proven by the experiment (E3) and demonstrates the effect of the remediation phase described in Sec. 6.2 of our paper when $\mathcal{P}$rv is compromised.

A.4.2 Experiments

(E1): *Verifying (C1) [10-15 minutes]*

This experiment determines additional LUTs and FFs required by *ACFA* hardware on top of the openMSP430 verilog project in order to estimate the hardware cost. To account for the cost of *ACFA* and all interconnections between *ACFA* and the openMSP430, we determine the cost by taking the difference between the cost of openMSP430+*ACFA* and openMSP430 alone.
[Preparation:] To prepare for this experiment, open *ACFA* Vivado project, as the Vivado toolset will be used to synthesize the Verilog design files into hardware. In addition, open the file `openMSP430_defines.v`. For both of these experiments, the flag `ACFA_HW_ONLY` will always be enabled (ensure no "//" precedes the `` `define ACFA_HW_ONLY `` on line 58). This ensures that only *ACFA* hardware is measured, and additional registers to simulate/emulate memory, which is not part of *ACFA* hardware, are not included in the measurement of hardware cost.
[Execution:] The first phase is to determine the cost of *ACFA* + openMSP430. Ensure `ACFA_EQUIPPED` is enabled (no "//" preceeds the `` `define ACFA_EQUIPPED `` on line 54 of `openMSP430_defines.v`). On the left menu of the PROJECT MANAGER, click "Run Synthesis" as performed in the *Setup*. After Synthesis completes, scroll down to "Utilization" in the "Project Summary" window. Press "Post-Synthesis" and "Table" to see a table of the hardware cost utilized by the synthesized Verilog files. The "Utilization Column" shows the total count of each resource in the "Resource" column. Therefore, this table will show the total LUT (row 1, column 2) and FF (row 3, column 2) required for *ACFA* + openMSP430. Take note of these values (referred to as $LUT_{ACFA+MSP430}$ and $FF_{ACFA+MSP430}$, respectively) since they are required to determine the final result. To get the cost of openMSP430 alone, open `openMSP430_defines.v`) and disable `ACFA_EQUIPPED` (add "//" to the beginning of line 54). Next, save all changes and rerun Synthesis. After it completes, check the LUT and FF utilization using the previous steps. This time, the listed LUT and FF specify the cost of openMSP430 without *ACFA*. Note these values (referred to as LUT_{MSP430} and FF_{MSP430}, respectively) since they are required to determine the final result.
[Results:] The cost of *ACFA* is determined by taking the difference between the cost of *ACFA* + openMSP430 and the cost of openMSP430 alone. Below are the expected results.
Look-Up Tables (LUTs):

- $LUT_{ACFA+MSP430} = 12373$

- $LUT_{MSP430} = 12098$
- $LUT_{ACFA} = LUT_{ACFA+MSP430} - LUT_{MSP430}$
- $LUT_{ACFA} = 12373 - 12098 = 275$

Flip-Flop Registers:

- $FF_{ACFA+MSP430} = 1844$
- $FF_{MSP430} = 1642$
- $FF_{ACFA} = FF_{ACFA+MSP430} - FF_{MSP430}$
- $FF_{ACFA} = 1844 - 1642 = 202$

(E2): *Verifying (C2): [45-75 minutes]*

This experiment demonstrates *ACFA* ability to provide secure auditing of periodic reports through enabling active *CFA*. In this experiment, *ACFA* executes a simple program that receives a remote user's password input and compares it with an expected password. After receiving a correct password, the Prover ($\mathcal{P}$rv) records six readings from an ultrasonic sensor. The Verifier ($\mathcal{V}$rf) has configured *ACFA* to have a maximum $\mathcal{CF}_{Log}$ size of 256B, and the timeout period is set as the maximum value to effectively deactivate triggers due to a timeout. This experiment demonstrates *ACFA* ability to halt execution and a series of fine-grained and timely reports. In addition, this experiment demonstrates the effectiveness of the end-to-end demo's offline and online phases.

[Preparation:] Add "//" to the start of line 59, remove any "//" from the start of line 54 in `demo_prv/main.c`, and save changes. Then, open a terminal window and `cd` into `scripts`. Run `make demo` to compile the software. After this, open `openmsp430_defines.v` and make sure `ACFA_EQUIPPED` is enabled and `ACFA_HW_ONLY` is disabled. Save all changes, then run Synthesis as performed in the Basic Test. Once Synthesis completes, select "Run Implementation." This process takes 30mins-1hour. After Implementation completes, select "Generate Bitstream," which will take 1-2mins. $\mathcal{P}$rv will execute on the FPGA using the bitstream that was just generated. $\mathcal{V}$rf will execute using a Python script during offline and online phases. During the online phase, $\mathcal{V}$rf and $\mathcal{P}$rv connect through a USB-UART interface. Connect the Basys3 FPGA to the machine using the USB cable included with the board. Then, determine which serial port the device is connected to (using `dmesg` command or some other means). After determining the serial port, update lines 16-17 in `demo_vrf/vrf_online.py` to reflect the correct port. The openMSP430 design shares a port between the GPIO and UART, and the GPIO port is selected by default. Therefore to select (and enable) the UART, turn on the physical switch `SW1` on the Basys3 FPGA board.

[Execution]: First complete the offline phase of $\mathcal{V}$rf. During this phase, $\mathcal{V}$rf computes the control flow graph (CFG) of the application software and computes an HMAC over the expected application binary. This is completed by the Python script `vrf_offline.py`. Open a terminal and execute this script by running `python3 vrf_offline.py`, and the binary objects from the offline phase are seen in `demo_vrf/objs`. Next, start the online phase of $\mathcal{V}$rf by running `python3 vrf_online.py`. $\mathcal{V}$rf will start running and wait for a report. Next, in Vivado, click "Open Hardware Manager" and then click "Auto-Connect". The FPGA should now be displayed on the hardware manager menu. Right-click the FPGA and select "Program Device ."After this, the *ACFA*-equipped MCU is programmed onto the FPGA, and $\mathcal{P}$rv will start running.

[Results]: During the online phase, $\mathcal{V}$rf receives reports from $\mathcal{P}$rv which contain a $\mathcal{CF}_{Log}$. The directory `/logs` will be populated with four $\mathcal{CF}_{Log}$ slices during the execution of the online phase. During each iteration of the active *CFA* protocol, $\mathcal{V}$rf will authenticate and verify CFlog slices by comparing them to the CFG and maintaining a shadow stack. In `demo_vrf/objs`, a binary object of the shadow stack is stored and modified during the online phase. *ACFA* generates and sends each report because of an *ACFA* trigger; `0.cflog` and `3.cflog` are generated due to the boot/end of program trigger; `1.cflog` and `2.cflog` are generated due to $\mathcal{CF}_{Log}$ reaching maximum size. The seven segment display of the FPGA board will show the current instruction address: the end of program (`0xe24a`).

(E3): *Verifying (C3) [45-75 minutes]*

This experiment executes the same example application as (E2). However, a buffer overflow is purposefully introduced in this experiment to cause a control flow attack. Therefore, this experiment shows that after a control flow violation has occurred, *ACFA* guarantees the remediation mechanism executes immediately.

[Preparation:] First add "//" to the start of line 54, remove any "//" from the start of line 59 in `demo_prv/main.c`, and save changes. This is the reverse of the first step in (E2) and enables the buffer overflow. After this, follow all remaining steps in the *Preparation* of (E2).

[Execution]: Follow the same steps of *Execution* of (E3).

[Results]: $\mathcal{V}$rf detects the buffer-overflow during the first intermediate report (`1.cflog`). Because of this, $\mathcal{V}$rf sends a command to execute the healing mechanism: shut down $\mathcal{P}$rv. On MSP430, this is achieved by setting a bit in the status register. After doing so, $\mathcal{P}$rv does not continue executing and pauses in TCB. This demonstrates that the compromised $\mathcal{P}$rv could not continue executing due to *ACFA*. In addition, because *ACFA* triggers sent $\mathcal{V}$rf an intermediate report, $\mathcal{V}$rf could find the vulnerability before the adversary could complete their attack. The seven segment display shows that the software is contained at TCB-Heal (`0xa606`).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artefact Appendix: The Impostor Among US(B): Off-Path Injection Attacks on USB Communications

Robert Dumitru
The University of Adelaide &
Defence Science and Technology Group
robert.dumitru@adelaide.edu.au

Daniel Genkin
Georgia Institute of Technology
genkin@gatech.edu

Andrew Wabnitz
Defence Science and Technology Group
andrew.wabnitz1@defence.gov.au

Yuval Yarom
The University of Adelaide
yval@cs.adelaide.edu.au

A Artefact Appendix

A.1 Abstract

The artefact is an instance of a USB device capable of injecting transmissions that a USB host will attribute to a neighbouring connected device, as described in the paper. It is configured to present itself as a USB mouse and can inject keystrokes on behalf of an adjacently connected USB keyboard, while optionally also blocking genuine input from the keyboard victim.

The artefact is in the form of a bitstream to be programmed onto an FPGA training board (ported for a Basys 3). The RTL source is also provided for modification and re-generation of the bitstream for use on other boards. A 1.5 kΩ resistor and basic cable splicing/rewiring is required.

With reference to the paper, the basis of this artefact is described in Section 5.1, and the claims to be reproduced are described in Section 7.1.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There is no risk of destructive behaviour from evaluation of this artefact.

A.2.2 How to access

Most recent source available at:
https://github.com/0xADE1A1DE/USB-Injection/

Stable tag:
https://github.com/0xADE1A1DE/USB-Injection/releases/tag/PosSec23AE

A.2.3 Hardware dependencies

Target hardware: A Digilent Basys 3 FPGA development board is required. This is a cheap (149 USD) and commonly used FPGA board. If you are at a university with an Elec Eng department, they are likely to have some of these available. (Other FPGA boards can be used if they have 3.3V IO, modification of constraints file and generation of a new bitstream from RTL source would be required).

Supplementary:
- A USB A to Micro B cable is needed to program the FPGA board (this should come in the Basys 3 box)
- A secondary USB cable must be spliced (exposing internal connector wires) while leaving the side of the type-A male connector intact (part that plugs into computer USB ports)
- 1.5 kΩ resistor
- Wires / connectors / breadboard

For testing:
- Any PC with USB ports
- A LS (Low-Speed) keyboard (majority are LS)
- USB hub(s)

A.2.4 Software dependencies

Any version of Xilinx's Vivado software, including free versions, can be used to configure the injection platform. See https://www.xilinx.com/support/download.html for latest versions.

[Windows users] We highly recommend USB Device Tree Viewer (https://www.uwe-sieber.de/usbtreeview_e.html) by Uwe Sieber for viewing the complete hierarchy of USB devices connected to your computer along with their descriptor sets. This will help to confirm the device is working.

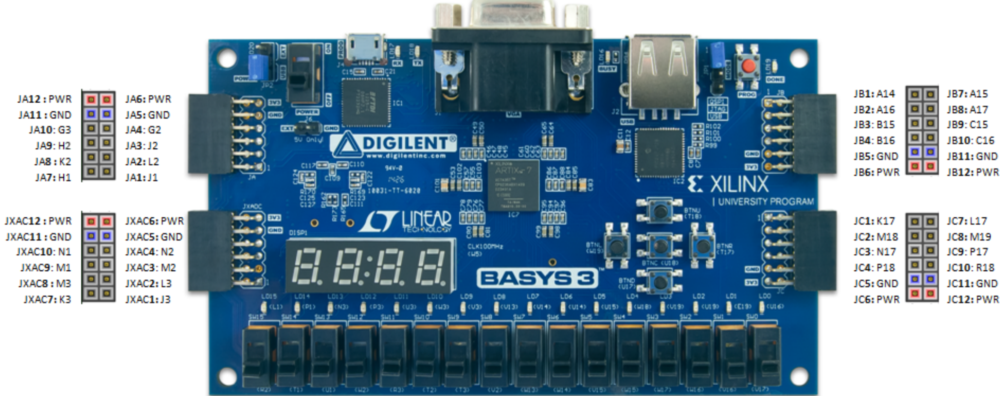


Figure 1: Basys 3 Pmod pinout

A.3 Set-up

A.3.1 Installation

1. Connect Basys 3 board to a computer running Vivado and program the target FPGA with the bitstream file from the repository (OS is according to the host computer the injection platform will be plugged into – this can be the same PC to which the Basys 3 is connected for programming):
   [*Windows*]
   `LS Keystroke Injector > USB_Demo.bit`
   [*Linux*]
   `LS Keystroke Injector > USB_Demo_Linux.bit`
2. Connect wires from a spliced USB cable to the Basys 3 board with the pin correspondence described in Table 1. See Figure 1 and Figure 2.

USB pin	USB wire colour	Basys 3 JB Pmod pin
D+	Green	JB1
D-	White	JB3
Gnd	Black	JB5
Vs	Red	Leave unconnected

Table 1: USB wire to FPGA pin correspondence

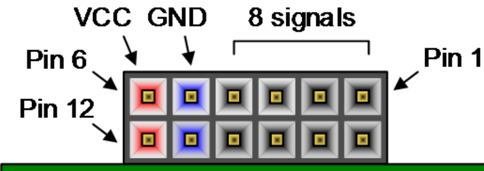


Figure 2: Pmod connector header pin numbering

3. Pull up the D- line to 3.3V across a 1.5kΩ resistor (as in Figure 3). To do this, you can connect one side of the resistor through JB6 (Vcc at 3.3V) on the same Basys 3 Pmod header, and connect the other side to the junction of JB3 and D- from the spliced cable.

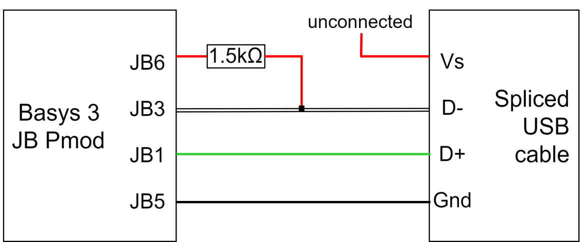


Figure 3: D- pullup resistor wiring

4. Continue to 'Basic Test'

A.3.2 Basic Test

With the Basys board programmed and connected to the spliced cable as instructed, plug the USB connector from the spliced cable into a PC (this can be the same PC to which the Basys 3 is connected for programming its FPGA). This should connect as a new mouse device. Confirm artefact functional as follows:
[*Windows*] Can either use device manager or preferably the USB Device Tree Viewer software previously mentioned in Appendix A.2.4.
[*Linux*] the `lsusb` command can be used to display the entire USB connection hierarchy.
Check the connection hierarchy with the device unconnected vs plugged in to confirm it appears.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The injector device can inject keystroke data on behalf of an adjacently connected victim keyboard when both the injector and victim are connected to the same vulnerable single-TT standard hub. This is proven by experiment (E1) described in Section 7.1 of the paper.

(C2): The injector device can effect a Denial-of-Service to the adjacently connected victim keyboard when both the injector and victim are connected to the same non-vulnerable, single-TT standard hub. This is also proven as an alternative outcome of experiment (E1) described in Section 7.1 of the paper.

A.4.2 Experiments

(E1): Keystroke injection *[10 human-minutes]:*

How to: Connect both the injection platform and victim keyboard to a common USB hub which is connected to a host PC. This can be the same host machine that the Basys board is connected (or any other machine). Connect no other devices through the hub. The hub is the device under test here, if it is vulnerable to injection the injection platform will be able to inject keystroke data on behalf of the connected victim keyboard [**Results**]. If the hub is not vulnerable, the injector should still be able to effect a Denial-of-Service against the victim keyboard [**Alternative Results**].

Preparation: Ensure the injection platform and keyboard are both logically connected to the same USB 2.0 hub and are both operating at LS. This will require using USB Device Tree Viewer (Windows) or the `lsusb` command (Linux).
The common hub must not be the computer root hub, injection will not work against root hubs.
Ensure the USB 2.0/2.1 hub is single-TT.
Ensure the Reset switch (SW0) is off (down), this is furthest right of the switches lining the bottom of the board (Figure 1).
Configure board switches into State 3 as in Table 2.

State	inj (SW1)	DoS (SW2)	Behaviour
0	0	0	NAKs being injected
1	0	1	No injection - victim works
2	1	0	NAKs being injected
3	1	1	Data being injected

Table 2: Injection platform switch configurations

[*Linux*] Open a document in a text editor.
[*Windows*] Same as above or do nothing.

Execution: Push the buttons on the Basys 3 board (5 buttons arranged in + shape on the bottom right side of the board) as follows (orientation as shown in Figure 1):
[*Windows*] **Left**, **Left + Right**, release both, **Up**, release, **Centre**, release, **Down**, release, **Right**
[*Linux*] Push and release any of **Up**, **Centre**, **Down**, or **Right**.

Results: If the hub under test is vulnerable to injection, you should see the following behaviour:
[*Windows*] The sequence of injected keystrokes opens a command prompt.
[*Linux*] The following keys are typed on the text editor document: c (**Up**), m (**Centre**), d (**Down**), and enter (newline) (**Right**).
If possible try with various hubs. USB 2.0 hubs are likely to be vulnerable, whereas USB 3.0 hubs are unlikely as a very small portion have been found vulnerable.

Additional Evidence of Results: *[Optional – if injection working and hub vulnerable]*
Wireshark's USBPcap function can be used to view injected traffic.
Unplugging the victim keyboard and pressing the same buttons on the still-connected injector will result in no keystrokes being fed.
Install any software-based USB authorisation policies and attempt injection with the victim keyboard allowed while the injector is blocked – as in Section 8. Injection will still work.

Alternative Results: Denial-of-Service against the keyboard victim should still be evident with hubs that are not vulnerable to injection. With the injector connected, open a text document and attempt to type keystrokes through the victim keyboard. No keystrokes should pass through. Change the injector switches to State 1 and the victim should then be able to type keystrokes.

A.5 Notes on Reusability

We have made the source RTL available so the injector device can be modified with generation of new bitstreams. Note, this source is for the Windows-compatible injector. Compatibility issue is just a bug from how different OS drivers process descriptors which we have not yet resolved. Some possible alterations for reuse are as follows:

Changing injected input. To change what data is injected, modify what is written to `PCIn` in `USBF_Demo.vhd`. See HID keyboard scan codes for data corresponding to keystrokes.

Changing injector device descriptors. To change any of the descriptor fields (ID, device type, etc.), modify values in `USBF_Declares.vhd`. Injection function is agnostic to the injector platform device type.

Use injector as FS device to target gaming keyboards. Repeat experiment with files under `FS Keystroke Injector` and pull up D+ instead of D-.

Use with different boards. The injector configuration should work with various FPGA boards, all that is needed is 3.3V IO and a different set of constraints (`.xdc` file).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220912. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenix%20sec2023/.

USENIX'23 Artifact Appendix: A comprehensive, formal and automated analysis of the EDHOC protocol

Charlie Jacomme
Inria Paris*

Elise Klein
Inria Nancy
Université de Lorraine

Steve Kremer
Inria Nancy
Université de Lorraine

Maïwenn Racouchot
Inria Nancy
Université de Lorraine

A Artifact Appendix

A.1 Abstract

This artifact permits to reproduce the formal verification of the LAKE EDHOC protocol. It comes as a docker image containing

- the software needed (SAPIC$^+$, TAMARIN, PROVERIF, DEEPSEC);
- the models of the LAKE EDHOC protocol;
- the scripts to batch run the verification of the models using the SAPIC$^+$ platform.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is hosted on docker hub: https://hub.docker.com/r/protocolanalysis/lake-edhoc/.

A.2.3 Hardware dependencies

There are two main sets of experiments, one relying on the PROVERIF tool and one on the TAMARIN tool. The PROVERIF experiments can be executed in a couple of hours on a modern laptop (8 threads at 2.8Ghz), while the Tamarin one while take multiple days. Having access to a server with 48 threads ensures that everything will run under a day.

A.2.4 Software dependencies

Docker is the only required dependency.

A.2.5 Benchmarks

[Mandatory] None.

*This work was partly done while Charlie Jacomme was at the CISPA Helmholtz Center for Information Security.

A.3 Set-up

[Mandatory] Installation instruction for Docker are provided at https://docs.docker.com/engine/install/.

A.3.1 Installation

The artifact is fetched with

```
docker pull protocolanalysis/lake-edhoc:draft-14
```

A.3.2 Basic Test

A bash should be opened inside the docker when running:

```
docker run -it protocolanalysis/lake-edhoc:draft-14 bash
```

A.4 Evaluation workflow

Once inside the Docker, there are two subfolders `lake-draft12` and `lake-draft14`. Each folder contains a README, bash scripts to run the exhaustive verification as well as the tool chain to generate the models from jinja2 templates. To ease the artifact evaluation, we automated all the relevant verifications under two main scripts:

- `./run-proverif.sh`
- `./run-tamarin.sh`

A.4.1 Major Claims

The LAKE EDHOC protocol draft 12 and draft 14 can be analyzed automatically using the SAPIC$^+$ platform under different scenarios. This yields the analysis results provided in Table 7 and 8 of Appendix B of our paper. Those tables where manually produced by formatting in LaTeX the results stored in the two csv files located in the `expected-results` folder.

A.4.2 Experiments

(E1): PROVERIF results: 5 human-minutes + 5.8 single-threaded (2.8Ghz) compute hours. With 7 threads at 2.8Ghz and 31 GB of RAM the verification runs in 62 minutes, and can go down to 31 minutes with additional cores.

Preparation: Enter the docker image. Check the number of available threads, e.g. with `htop`.

Execution: Run `./run-proverif.sh i`, where `i` is the number of available threads, and wait for full completion. The script displays which verification are started (mainly for debugging purposes).

Results: The script produces a `res-proverif.csv` file. To compare the obtained results with the one stored in `expected-results`, the `compare-diff-proverif.sh` script highlights any differences. The diff should only highlight timing differences, or additional timeouts in case of different hardware.

(E2): TAMARIN results: 5 human-minutes + 64 single-threaded (2.8Ghz) compute hours. With parallelization over 48 threads at 2.8Ghz and 756GB, the experiments takes 14 hours.

Preparation: Enter the docker image. Check the number of available threads, e.g. with `htop`.

Execution: Run `./run-tamarin.sh i`, where `i` is the number of threads **divided by 4** (as we allocate 4 threads to each instance of TAMARIN), and wait for full completion. The script displays which verification are started (mainly for debugging purposes).

Results: The script produces a `res-tamarin.csv` file. To compare the obtained results with the one stored in `expected-results`, the `compare-diff-tamarin.sh` script highlights any differences. The diff should only highlight timing differences, or additional timeouts in case of different hardware.

In addition, the privacy analysis of Table 6 can also be verified, running `./lake-draft12/run-anonimity.sh` and `./lake-draft14/run-anonimity.sh`.

A.5 Notes on Reusability

The docker also contains a `README.md` meant for users familiar with the underlying TAMARIN/PROVERIF/SAPIC$^+$ tool chain, which explains how to either reuse our general structure or update the models.

To build the docker, the following actions can be performed:

```
git clone https://github.com/charlie-j/tamarin-prover/
cd tamarin-prover;
git checkout e59304fb8f51e1e25118362daeb3fc008a6e292d;
./etc/docker/build.sh
./etc/docker/build-platform.sh
cd ../
git clone https://github.com/charlie-j/edhoc-formal-analysis
git checkout e2e3f7407e9eb331a8112614fe9e116e57a25e51
cd edhoc-formal-analysis
./Docker/build.sh
```

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Automated Analysis of Protocols that use Authenticated Encryption: How Subtle AEAD Differences can impact Protocol Security

Cas Cremers[1], Alexander Dax[1,3], Charlie Jacomme[2], and Mang Zhao[1,3]

[1]CISPA Helmholtz Center for Information Security, Germany
[2]Inria Paris, France
[3]Saarland University

A Artifact Appendix

A.1 Abstract

This artifact appendix presents a description of the experiments and case studies conducted in the research paper *Automated Analysis of Protocols that use Authenticated Encryption: How Subtle AEAD Differences can impact Protocol Security*. One of the core objectives of this research is to analyze the impact of subtle differences in AEADs on the security of a variety of protocols.

We provide means to reproduce our case studies of 8 distinct security protocols, namely YubiHSM, Facebook's Message Franking, SFrame, WebPush, Whatsapp Group Messaging, GPG, saltpack, and Scuttlebutt. These protocols are analyzed under different various AEAD models. We structured and defined those models in a *library* file. All models in the library, along with the case studies, are implemented using the Tamarin Prover, a symbolic analysis tool for security protocols.

The AEAD models and case studies are made available in a public Github repository with detailed instructions and automated means to replicate the experiments discussed in the original research paper. Additionally, a Docker image with the necessary software is made available for easy setup and execution.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

All our files are publicly available and can be accessed in the following GitHub repository https://github.com/AutomatedAnalysisOf/AEADProtocols/tree/V1.

A.2.3 Hardware dependencies

Our artifact does not require any specific hardware. However, as the used software (e.g. the Tamarin Prover [1]) does also scale with computation power and memory, we recommend to at least use a modern notebook or similar modern computing devices. GPUs are not required.

A.2.4 Software dependencies

We provide access to a docker[2] image which has all the necessary software dependencies pre-installed.

(Optional) Dependencies for manual installation In case the reviewers choose to manually install the dependencies, they should install the following

1. Tamarin Prover[3] (depends on `haskell-stack`, `graphviz`, and `maude`.) Note that Tamarin **does not run on Windows** systems and a virtual machine/WSL may be needed. Additionally, we added a .zip file with the correct Tamarin version to the GitHub repository.
2. Python3 - install `pip`, and use it to install `tabulate` and `matplotlib`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Clone the repository using:

```
$ git clone https://github.com/
 ↪ AutomatedAnalysisOf/AEADProtocols.git
```

[1]https://tamarin-prover.github.io
[2]https://docs.docker.com/engine/install/
[3]https://tamarin-prover.github.io/manual/book/002_installation.html

```
| whatsapp.spthy | consistency | False | 19 | ['collkeys'] |
| whatsapp.spthy | consistency | False | 16 | ['collmmax'] |
| whatsapp.spthy | consistency | True  | 58 | ['collnmax', 'colladmax', 'n_reuse_1', 'n_reuse_2', 'reveal_nad'] |
```

Figure 1: Exempt of the terminal output produced by tamarin_wrapper.py.

and navigate inside the repository.

After installing `docker`, pull the docker image using

```
$ docker pull aeads/tamarin
```

Now, you can run the image using:

```
$ docker run -it -v $PWD:/opt/case-studies
 ↪  aeads/tamarin bash
```

(Optional) Hints for manual installation

In the case you want to set up the experiments **without** docker here are some hints:

- Make sure to use the *tamarin prover* version provided in the GitHub repository.
- Follow the instruction on https://tamarin-prover.github.io/manual/book/002_installation.html to install the tamarin prover. Common problems are missing Haskell dependencies or outdated versions of Maude
- Make sure that the *tamarin-prover* executable is in the *$PATH*.

A.3.2 Basic Test

Execute

```
$ tamarin-prover test
```

to see whether the docker started successfully (or whether your manual installation worked).

You should see a message containing

- a check for `maude`,
- a check for `Grapviz`, and
- a test for the unification structure (0 errors and 0 failures).

In the end you should see the following:

```
All tests successful.
The tamarin-prover should work as intended.
        :-) happy proving (-:
```

A.4 Evaluation workflow

A.4.1 Major Claims

One of the primary objectives of the case study analysis is to identify the most robust AEAD model that preserves the desired security property for each protocol.

The provided models in the artifact appendix include formal representations, so-called *lemmas*, expressed in the input language of the Tamarin Prover, which capture the desired security properties of the protocols.

Through automated execution of Tamarin with different AEAD models using a provided Python script, the lemma results are checked to determine if they hold or provide counterexamples, facilitating efficient analysis. Further details on the Python script and the core idea can be found in the original paper.

Table 1 shows an output of the Python script for the Whatsapp group messaging protocol model. Here, for instance, the lemma marked as *consistency* does not hold under the `collkeys` or the the `collmmax` AEAD models. The name tags are explained in the `README.md` file and defined in the original paper.

Our focus lies on identifying the weakest AEAD model that ensures the security property proven by the lemma, as well as determining the strongest AEAD models that lead to potential attacks. However, it is important to note that certain models may not terminate within the specified timeout. In such cases, we still identify AEAD models that demonstrate both secure properties and attack possibilities. In this case, we do not claim that these models represent the strongest or weakest models where the property still holds or yields a counterexample.

We give details on the reported results in the original paper and provide the concrete results in the GitHub https://github.com/AutomatedAnalysisOf/AEADProtocols/tree/V1.

A.4.2 Experiments

Instead of running all models independently, we provide a python program to run all of them at once. For that we used a computing cluster with Intel® Xeon® Gold 6244 CPUs and 1TB RAM. In case you do not have access to a computing cluster, you may need to increase the timeout in the `case_studies.tamjson` file. Open the file using the editor of your choice and navigate to the line `"timeout": 60,` and increase the number slightly. The number after the *timeout* is defined as seconds.

In the GitHub repository we also provide the results of our case studies when running them on our machine. We had a total evaluation time of 17 hours and 29 minutes with a total of 1404 tamarin prover calls. While we tested the case studies

also on a modern notebook, we cannot guarantee the same precise result, as specific lemmas using certain AEAD models may not terminate within the timeout. This mostly concerns the protocol models if YubiHSM and SFrame. All others should finish rather fast (< 1hour), also on normal notebooks.

Preparation: After following the installation instruction in Section A.3.1, enter the `Models` folder within the cloned repository/docker image.

Execution: Execute

```
$ python3 tamarin_wrapper.py -f
 ↪  case_studies.tamjson
```

Note, that depending on your machine the results may differ. You can increase the timeout in the `case_studies.tamjson`

Results: While the results will be printed into the terminal (see Figure 1), *.csv* files of the results are also stored within the newly created `results` folder. They can be compared to our provided results in the `results_precomputed` folder in the `Models` directory. We also refer to Table 3 and Table 4 in the original paper to confirm that your run did find the same attacks

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Credit Karma: Understanding Security Implications of Exposed Cloud Services through Automated Capability Inference

Xueqiang Wang
University of Central Florida

Yuqiong Sun
Meta

Susanta Nanda
ServiceNow

XiaoFeng Wang
Indiana University Bloomington

A Artifact Appendix

A.1 Abstract

The submission pertains to PrivRuler, a tool utilized in the research paper titled "Credit Karma: Understanding Security Implications of Exposed Cloud Services through Automated Capability Inference." PrivRuler comprises two key components. The first one, AppAnalysis, is a static app analysis component that extracts cloud service credentials and usages from mobile applications. The second component, CloudProbe, takes the output of AppAnalysis as input and probes the associated cloud services to infer the additional capabilities granted to mobile applications.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The AppAnalysis component runs as a Java program on the local machine, and it does not alter any system settings that might impact security, nor does it transmit any data to backend servers.

The CloudProbe component operates within an Android emulator or a test Android device, allowing it to probe cloud services and infer the capabilities associated with a specific cloud credential. In this submission, we established a test credential on our AWS accounts, ensuring that running the CloudProbe component for functional testing would not compromise the privacy of any third parties. Additionally, we minimized the risks associated with cloud service probing using multiple strategies, as detailed in Section 3.6 of the paper.

A.2.2 How to access

URL: https://github.com/privruler/PrivRuler-Public
Commit: 8ff0ae9c8d2611072fde0b112e71b8f662fb2507

A.2.3 Hardware dependencies

There is no hardware dependencies to run PrivRuler.

A.2.4 Software dependencies

- PrivRuler runs on basically all operating systems, including Windows, MacOS, and Linux. We recommend MacOS or Linux as these are the operating systems we test more often.
- Recommend JDK Version 8
- Android Studio
- Android platform tools (e.g., adb)
- Android emulator or device of API Level < 30 (Recommended 28).

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Install AppAnalysis. AppAnalysis is written in Java, and we have created a script to automate the compiling and dependency management process. Users can follow the below steps to compile it.

- `cd $dir/PrivRuler-Public/AppAnalysis`
- `./compile.sh`

Install CloudProbe. CloudProbe is delivered as an Android app. Therefore, we need an Android emulator and Android Studio to install it.

- Create an Android Emulator with API Level 28.
- Import CloudProbe into Android Emulator, and hit "`Run app`" button to install CloudProbe on the emulator.

A.3.2 Basic Test

We created a test app folder at "$dir/PrivRuler-Public/AppAnalysis/apks/" for basic testing.

Run AppAnalysis. Use the below commands to analyze test apps using AppAnalysis:

- `cd $dir/PrivRuler-Public/AppAnalysis`
- `./analyze apks apks`

The output will be stored in `output/app-debug.output` file. A successful run of AppAnalysis will generate a line that contains "`cloudAPIs`" keyword in the output file.

Transfer AppAnalysis result to Android emulator. Launch the Android emulator, and run the below command to transfer the output of AppAnalysis to the emulator.

- `grep -Rh 'appName.*cloudAPIs' $dir/PrivRuler-Public/AppAnalysis/output » summary`
- `adb shell mkdir /sdcard/cloudassets`
- `adb push summary /sdcard/cloudassets`

Run CloudProbe. In Android Studio, click the "`Run app`" button to run CloudProbe. Once the app is launched in the emulator, click the three buttons on the app UI. A successful run will generate analysis results under emulator folder `/sdcard/AWSSummaries/`, with each app has a file named `summary_⟨packagename⟩.json`. Users may check presence of this file by running: `adb shell ls -alh $file_path`.

A.4 Evaluation workflow

We do not request for a complete evaluation since it will require analyzing over 1.3M apps (over 30TB) in storage, and re-probe the cloud backends of 12K apps.

A.5 Notes on Reusability

In addition to inferring over-privileges in cloud services, the artifact has the potential to be used in several other ways:

- **Analyzing mobile apps for sensitive information beyond cloud service credentials.** While the artifact's primary focus is detecting cloud service credentials within mobile apps, it can be customized to scan and identify other sensitive information present within mobile apps with ease.
- **Identifying obfuscated APIs.** As a part of the AppAnalysis component, the code extracts fingerprints for obfuscated APIs by examining invariant information like the number of arguments. The obfuscated APIs fingerprinting module can be employed to analyze other obfuscated APIs aside from cloud APIs.
- **Enhancing mobile app security.** By employing the PrivRuler tool, app developers can evaluate the security of their mobile apps and identify any potential vulnerabilities in regard to cloud services, thereby enhancing their app's security posture and safeguarding against cyber attacks such as data leaks.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Remote direct memory introspection

Hongyi Liu Jiarong Xing Yibo Huang Danyang Zhuo† Srinivas Devadas‡ Ang Chen
Rice University †*Duke University* ‡*MIT*

A Artifact Appendix

A.1 Abstract

This artifact appendix describes the workflow to setup and run RDMI. It includes an artifact check-list, description of hardware/software dependencies to install RDMI as well as setup instructions and experiment workflows. Please refer to the GitHub repository for further installation and execution details.

A.2 Description & Requirements

We provide a check-list for meta-information here.

- **Compilation:** GCC v7.5.0, Tofino SDE v8.4.0.
- **Binary:** Source code included to generate binaries
- **Run-time environment:** End host codes are tested on x86 servers with Ubuntu18.04 OS.
- **Hardware:** Intel/Barefoot Wedge 100BF-32X Tofino switch ×1, x86 server with Mellanox ConnectX-4 RNICs ×2.
- **Metrics:** Throughput, latency, CPU utilization, defense effectiveness.
- **Output:** The compiler will output configuration files used for configuring the programmable switch to enforce policies. Latency and traffic volume can be measure by tools like `tcpdump` or using in-switch telemetry. CPU utilization can be measurement by tools like `top`.
- **Experiments:** DSL compilation, connection establishment, switch reconfiguration and policy execution.
- **How much disk space required (approximately)?:** 1GB (dependencies not included)
- **How much time is needed to prepare workflow (approximately)?:** Compiling all programs needs about 1 hour (installation of software dependencies and hardware is not included)
- **How much time is needed to complete experiments (approximately)?:** About 2 hours to see the effect of all defenses.
- **Publicly available?:** Yes, code is available on GitHub.
- **Code licenses:** MIT license

A.2.1 Security, privacy, and ethical concerns

There is no security, privacy, and ethical concerns.

A.2.2 How to access

Our artifact and guidelines for installing and evaluating RDMI are publicly available at the following GitHub repository: commit: 7b8b15cf9a.

A.2.3 Hardware dependencies

To run RDMI, it requires two x86 servers connected by an Intel/Barefoot Tofino switch through Mellanox ConnectX-4 RNICs.

A.2.4 Software dependencies

Our experiments are performed on x86 servers running 64-bit Ubuntu 18.04, but similar Linux distributions should also work. To enable RDMA, Mellanox MLNX_OFED driver must be installed on the servers. RDMI's P4 code is compiled by proprietary toolchains provided by the switch vendors.

A.2.5 Benchmarks

None.

A.3 Set-up

To run RDMI, user needs to install all the dependency listed in check-list as well as install the NIC driver. We provide more details in the GitHub repository.

A.3.1 Installation

We list the main steps to install RDMI here. More details can be found in our GitHub repository.

- Install RNIC drivers to enable RDMA on end hosts.
- Install and setup the programmable switch following the vendor instructions.

A.3.2 Basic Test

To test compiler, run `make & python parse.py & ./RDMI 1000 100 1` inside *compiler* directory. It should result in a generated `cmd` file used for configuring the switch. To test the connections, run `sudo ./rdmatry_server -a SERVER_IP`

`-n 10 -m 1 -M 1000000000 -d 0` in the introspected machine side and `./rdmatry_client -a SERVER_IP -n 10 -M 1000000000 -r 10000000 -c 1 -t 99999999999 -p 1` in the remote side inside *switch* directory, and follow the instructions to establish the connections. The program should print out connection success information if the connection is setup correctly. To test the switch, run `./run_switchd.sh -p master` on the corresponding Tofino SDE environment. The load success information will be printed if the switch environment and program is correct. Then the user can follow the vendor provided instructions to configure the switch with the generated configuration files.

A.4 Evaluation workflow

We listed detailed workflows to conduct the experiments of the system in the GitHub repo. Here we provide three key steps below for the evaluation workflow. Please refer to our GitHub repository for further details:

- Establish the RDMA connections.
- Compile the policy and generate the corresponding configuration files.
- Configure the switch and run the program.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

A Artifact Appendix

A.1 Abstract

This artifact contains the codebase to run SQIRL, a novel approach to detecting SQL injection vulnerabilities using deep reinforcement learning with multiple worker agents. Each worker intelligently fuzzes the input fields discovered by an automated crawling component. It also includes all the code required to run the different versions of SQIRL including its random (RAND-SQIRL), and federated (FED-SQIRL) variants. The requirements to create the SQLiMicroBenchmark (SMB) are also included. We further detail how to run SQIRL on the SMB in order to reproduce the results found in the main body of the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

To reduce the effect of possible security concerns the SMB runs inside a docker container. The SMB should *not* be exposed on externally available ports as the SMB contains deliberately vulnerable samples.

SQIRL has the potential to find zero-day SQLi vulnerabilities in web applications. However, we have designed SQIRL to be a greybox tool that requires privileged access to the webapplication-under-test. This prevents malicious user from using this on targets that are unware they are being tested.

A.2.2 How to access

We release the artifact as a repository, a stable version of which can be found at: https://github.com/ICL-ml4csec/SQIRL/tree/5a444ee7782a33a097f345fca837125ac2505ee0

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

- Python >= 3.8.16
- Python pip >= 23.0.1
- Docker Engine >= Docker version 23.0.5, build bc4487a
- Docker Compose version >= v2.17.3

A.2.5 Benchmarks

The SQLiMicroBenchmark or SMB is provided in the artifact in the directory `SQLiMicroBenchmark/`. This requires Docker in order to be run, and tested on.

A number of baselines are used as a point of comparison to SQIRL. We do not document here how to install and setup these baselines, we refer the reader to the relevant baselines if they wish to compare against these. Please see the Appendix in the main paper for the configuration used for the baselines.

Baselines:

- OWASP ZAP v2.11.1
- Sqlmap v1.6
- BurpSuite Pro v2022.6.1
- Arachni v1.6.1.3
- Wapiti 3.1.2

If user wish to experiment with the production grade web applications we provide here the list of web applications and plugins where relevant:

1. WordPress core v6.0 and plugins (Download Monitor WordPress V 4.4.4, WP User Frontend 3.5.25, Sliced Invoices 3.8.2, Plugin Photo Gallery 1.5.34, Supsystic Ultimate Maps 1.1.12, WP Statistics 13.0.7, JoomSport)
2. B2evolution v7.2.3-stable
3. BBpress v2.6.9
4. Big tree CMS v4.4.16
5. Drupal v9.3.18
6. Joomla v4.2.0
7. Admidio v4.0
8. Gila CMS
9. Media wiki v1.38.2
10. Pbboard v3.0.3
11. Impresscms v1.4.4
12. WackoWiki v6.0.31
13. Sourcecodester E-learning System v1.0,
14. Sparks Hotel Management System v1.0

A.3 Set-up

A.3.1 Installation

Clone Repository:

```
git clone https://github.com/ICL-ml4csec/SQIRL
```

Install docker: If the docker engine and docker compose requirements are not already met, then install them. Instructions for this can be found here.

Setup SMB: From the `SQLiMicroBenchmark/` directory create the required docker containers: `docker-compose up -d`. This can take a minute or two to install and configure. Note that one of the containers 'db-seeder' will sleep for 40 seconds before configuring the 'db' docker container, and will then exit. This is the intended behaviour, after the configuration containers 'db' and 'php-apache' should be running.

The log file required by SQIRL must be edited to provide read and write access `chmod +rw mysql/general.log`.

SQIRL dependencies: We recommend running the SQIRL framework from a virtual Python environment. We recommend using conda, which can be installed following the guide here. Note conda is not required and other frameworks such as poetry or venv can be used. After install create the conda environment:

- `conda create -name sqirl python=3.9`

```
Usage: sqirl_cmd.py [options]

Options:
  -h, --help            show this help message and exit
  -u URL, --url=URL     Full URL to crawl
  -i AGENT_UNIQUE_ID, --agent_unique_id=AGENT_UNIQUE_ID
                        ID of the agent used for logging
  --level=LEVEL         Depth for the crawler to traverse
  --db_type=DB_TYPE     Type of Database e.g. mysql
  --log_file=LOG_FILE   Path to the log file of the SQL database
  --learning=LEARNING   does the agent learn
  -e EPISODES, --episodes=EPISODES
                        Maximum number of episodes per input found
  --max_timestamp=MAX_TIMESTAMP
                        Maximum timesteps per episode
  --win_criteria=WIN_CRITERIA
                        Minimum number of vulnerabilities found before
                        switching inputs
  --loss_criteria=LOSS_CRITERIA
                        Maximum number of episodes before switching inputs
  -v VERBOSE, --verbose=VERBOSE
                        Set verbose level 0, 1, 2
  --input_selection=INPUT_SELECTION
                        Method to select next input: 1 FIFO queue, 2 is random
  --agent=AGENT_TYPE    SQIRL Variant: 0 for Random, 1 for DQN, 2 for DQN_RND,
                        3 for One_Hot_Encoder_DQN_RND, 4 for Worker_DQN_RND
  --training=TRAIN      Boolean to set if SQIRL is in learning mode
  --login_function_name=LOGIN_FUNCTION
                        Function name for the authentication module in
                        --auth_file_path
  --auth_file_path=MODULE_PATH
                        Path to the .py file used for authentication
```

Figure 1: Expected result for basic test of SQIRL, showing help options.

```
[+] Running 4/4
 ✔ Network sqlimicrobenchmark_default         Created
 ✔ Container db                               Started
 ✔ Container sqlimicrobenchmark-db_seeder-1   Started
 ✔ Container php-apache                       Started
```

Figure 2: Expected result for basic test of SMB, showing all containers have started.

SQIRL packages: Installing the SQIRL packages with pip can be done using the following command: `conda activate sqirl && pip install -r requirements.txt`.

A.3.2 Basic Test

SQIRL: `python sqirl.py -help`

This should startup SQIRL and then display the flags that can be used when running, as shown in Figure 1.

SMB: `cd SQLiMicroBenchmark && docker-compose up -d && cd ..`

This will start the containers required by the SMB, resulting in the expected result shown in Figure 2

A.4 Evaluation workflow

We provide here the framework and instructions for using SQIRL and the SMB.

A.4.1 Major Claims

(C1): SQIRL is able to find more vulnerabilities than existing state-of-the-art-scanners and achieve 0 false positives.

```
[!] reached win criteria
[!] payload: 131149265861 AND  SLEEP (0)
[!] URL: http://localhost:8000/functions_external/sqli1.php
[!] Parameter: name
```

Figure 3: Example of a vulnerability identified by SQIRL and shown in the log files `results_stats_X.stats`, where `X` is the worker agent that found the vulnerability.

(C2): SQIRL is able to find vulnerabilities in a lower number of requests than other scanners.

A.4.2 Experiments

(E1): [10 human minutes + 20 compute hours + 16GB disk + 6CPU]: train SQIRL on the SMB to ensure training functionality.

How to: First the SMB and python environment must be set up. Then SQIRL can be run to start training. After training has finished the log files and trained model can be seen in a new sub-directory in `stats_logs`.

Preparation: Activate the docker container `docker-compose up -d`. Then activate the environment for SQIRL: `conda activate sqirl`.

Execution: Run from a terminal window Example **A** in Table 1. Note this should run as a single command and copying directly from the Table may cause an error due to a newline.

Results: Each agent should finish running, after which the worker server can be closed. There will be a new directory in `stats_logs` that will contain a new model in addition to log data.

(E2): [10 human minutes + 1 compute hour + 16GB disk + 6CPU]: Test SQIRL on the SMB to ensure get test results functionality.

How to: First the SMB and python environment must be set up. Then the SQIRL can be run to test for the SQLi in the SMB. SQIRL will create a new sub-directory in `stats_logs` containing log files and the model check-point that resulted from the new run.

Preparation: Activate the docker container `docker-compose up -d`. Then activate the environment for sqirl: `conda activate sqirl`.

Execution: Run from a terminal window Example **B** in Table 1. Note that the `dir_from_training` should be changed to that from training in Experiment **E1**. Where the `save_dir` is the resulting directory from the first experiment.

Results: In the new sub-directory in `stats_logs` the log file '`results_stats_1.stats`' will contain the vulnerabilities found by SQIRL. These are identified by the pattern shown in Figure 3. The number of requests used to find these vulnerabilities can then be found by accessing `http://localhost:8000/server-status` and is identified by the total number of accesses.

Table 1: Example commands used to run SQIRL. **A**: Training SQIRL with 4 worker agents, **B**: Testing SQIRL with 1 worker agent.

```
A: python3 sqirl.py -u http://localhost:8000/training.php --log_file ./SQLiMicroBenchmark/mysql/general.log \
        --loss_criteria 200 --win_criteria 14 --agent 4 -i 4
B: python3 sqirl.py -u http://localhost:8000/no_feedback.php --log_file  ./SQLiMicroBenchmark/mysql/general.log \
        --agent 4 --model_dir ./stats_logs/dir_from_training/Checkpoint_Worker_Server/
```

A.5 Notes on Reusability

SQIRL is designed to be independent of the SQL database that is being tested. We have developed SQIRL to work with `mysql v5.X`, this can be extended by adding in the ability to parse the required logs to the SQL Proxy (`SQL_Proxy.py`). Any tokens specific to the database syntax would also be required by the environment in order to generate syntactically correct payloads.

Note that for newer versions of `mysql` the error logging functionality changed to include malformed queries in the general log. This can lead to SQIRL producing false positives so it is advised to use `mysql v5.X` when testing SQIRL.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Hiding in Plain Sight: An Empirical Study of Web Application Abuse in Malware

Mingxuan Yao[1], Jonathan Fuller[2], Ranjita Pai Sridhar[1], Saumya Agarwal[1],
Amit K. Sikder[1], Brendan Saltaformaggio[1]
[1]*Georgia Institute of Technology* [2]*United States Military Academy*

A Artifact Appendix

A.1 Abstract

The artifact is a code repository (with supporting documentation) for Marsea, an automated concolic analysis pipeline used to perform a scalable and retroactive study of malware that abuses web applications. Marsea consists of the backend for malware's symbolic execution and the hook project for dynamic binary instrumentation.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact should not pose any inherent security, privacy, or ethical concerns. No preexisting data is read or transmitted. If the user decides to install and run active malware for the purpose of verifying Marsea's claims, they do so at their own risk. For security and ethical reasons, the artifact does not include any active malware.

A.2.2 How to access

The artifact is a code repository and well-documented tutorial thta can can be accessed on GitHub: https://github.com/CyFI-Lab-Public/MARSEA/tree/fc53c4629065eeaad78258a11d950265cb059c5d

A.2.3 Hardware dependencies

Marsea requires a Linux machine and a Windows Machine.

A.2.4 Software dependencies

The preferred environment for running Marsea is Ubuntu 22.04 LTS (Long Time Support). However, Marsea *should* work with any recent version of Ubuntu. Given the fact that Ubuntu is Debian based operataion system. Marsea *should* also work on Debian 11 and up (64 bit).

Another important component of Marsea is its customized DLL (Dynamic Linked Library), which enables Marsea to instrument the malware and introduce symbolic value amid the execution The maintaining and the building of this DLL project requires a Windows machine (Windows 7 or above). The preferred environment for the DLL project is Microsoft Visual Studio (2017 or above), with Windows 8.1 SDK and version 141 Platform Toolset. However, the newer version of SDK and Platform Toolset *should* works as well.

A.2.5 Benchmarks

The primary bench mark used in the paper is a collection of Web-App-Engaged (WAE) malware, such as information stealer, dropper, and other types of malware that target web applications. because it represents a common and important threat to web applications and it allows for a thorough evaluation of the effectiveness of the Marsea. The benchmark was run on the Ubuntu 22.04 LTS operating system with Marsea deployed, and the results are show in Table 2 of the paper. We also performed the baseline concrete execution comparison using Marsea with no instrumentation.

A.3 Set-up

A.3.1 Installation

Users should follow the Setup section of the README to deploy Marsea.

A.3.2 Basic Test

Users should follow the Usage section of the README, which covers includes a step-by-step tutorial of running Marsea against malware, *Razy*, which abuses the Twitter to resolve the C&C server address. Notably, to avoid the possible security risk, we verified that the resolved C&C server has already been mitigated.

Running Marsea against *Razy* should reveal:

1. The context-rich execution trace of the target malware. For *Razy*, Marsea explores different paths and reveals the its connection to VirusTotal, Twitter, and the backend C&C server.

2. Reconstructed network sessions initiated by the target malware. For *Razy*, Marsea reveals (a) a connection to

the VirusTotal with an API key in the request and the malware itself as payload; (b) a connection to Twitter with an user specified.

3. The malicious vectors the malware performed given the abused web apps. For *Razy*, Marsea reveals (a) the flooding attack the malware performed towards VirusTotal; (b) the backend C&C server resolving by abusing the tweet posted on the Twitter.

The README contains step-by-step instructions for deployment and provides running examples to assist users in verifying the successful completion of each phase.

A.4 Evaluation workflow

This subsection serves to illustrate the assertions made in our paper. However, due to ethical considerations, we are unable to release the malware dataset utilized in our research at this time. Consequently, users are required to obtain their own malware dataset for analysis.

A.4.1 Major Claims

(C1): Marsea is able to identify 40 abuse vectors across 20 malware samples. This is proven by experiment (E1) described in Section 4.1 of the paper and illustrated in Table 2.

(C2): Marsea identified 86% of vectors compared with concrete execution. This is proven by experiment (E2) described in Section 4.1 of the paper and illustrated in Table 2.

(C3): Marsea revealed a 226% increase in malware only relying on web apps since 2020, showing malware's growing adoption of web app abuse. This is proven by experiment (E3) described in Section 5.1 of the paper and illustrated in Table 3.

(C4): Marsea revealed 893 WAE malware in 97 families abusing 29 web apps. This is proven by experiment (E4) described in Section 5.2 of the paper and illustrated in Table 4.

(C5): Marsea found that 48% of 893 WAE malware remained actiev until the day of our study. This is proven by experiment (E5) described in Section 5.3 of the paper and illustrated in Table 5.

(C6): Marsea revealed that WAE malware could have infected up to 909,788 victims from 33 abused web app content. This is proven by experiment (E6) described in Section 5.4 of the paper and illustrated in Table 6.

A.4.2 Experiments

(E1): [10 human-days + 3 compute-days + 300GB storage]: Evaluate the performance of Marsea in ground truth dataset.
Preparation: Collected malware are manually reverse-engineered to derive ground truth.
Execution: Run Marsea against the malware in the ground truth dataset and collect the generated results, such as the execution trace, malicious vectors, and reconstructed network sessions.
Results: Marsea should be able to identify the web apps that have been abused and detect most malicious vectors.

(E2): [5 human-days + 3 compute-days + 400GB storage]: Compare the performance of Marsea with the concrete execution.
Preparation: Prepare for concrete execution analysis and set up Marsea.
Execution: Analyze the malware using both Marsea and concrete analysis techniques, and compare the malicious vectors identified by Marsea and the concrete analysis.
Results: Marsea should be able to identify more malicious vectors and abused web apps than concrete analysis.

(E3): [10 human-days + 10 compute-days + 5TB storage]: Execute Marsea on the large scale to evaluate the prevalence of WAE malware.
Preparation: Collect malware from online resources. To ensure an unbiased dataset, the collection should be random and normalized across the timeline (i.e., the same number of samples should be taken for each evaluated time slot). Use domain reputation resources to evaluate the maliciousness of the communication targets.
Execution: Run Marsea against the malware and collect the communication targets. Extract the effective Second-Level Domain (eSLD) for each communication target and measure its maliciousness using domain reputation resources.
Results: Marsea should be able to identify the increase of WAE malware in the last three years.

(E4): [3 human-days + 20 compute-days + 10TB storage]: Run Marsea on large-scale WAE malware to evaluate the capabilities the abused web apps provide attackers.
Preparation: Collect WAE malware.
Execution: Run Marsea on a large-scale WAE malware collection, collect the execution trace, and identify the vectors.
Results: Marsea should be able to a wide range of vectors the malware could perform using web apps.

(E5): [10 human-days + 2 compute-days + 50GB storage]: Examine the effectiveness of the current mitigation of WAE malware by evaluating the activity of the malicious web app content.
Preparation: The communication targets as the intermediate results from E4. VirusTotal access is required to identify the first seen date and the last seen date of the malware.

Execution: Query the first seen and last seen date of the analyzed malware on VirusTotal. Then, run the web app harvest component of Marsea to extract the first creation time and the last update time of abused web app content.
Results: Marsea should be able to show that some web app content can remain on the platform for a long time, even after the corresponding malware has been identified. Note that users may need to write their own harvest components to support additional web apps.

(E6): [3 human-days + 2 compute-days + 20GB storage]: Use the engagement data on the web app platform to prove the large scope of infection caused by WAE malware.
Preparation: The communication targets results intermediate results from E4.
Execution: Run Marsea's web app harvest component to extract the engagement data from the abused web app platforms.
Results: Marsea should be able to identify a significant scope of infection on specific web apps.

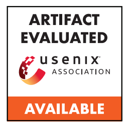
A.5 Notes on Reusability

Marsea has a wide range of in-house scripts that can be used directly or easily extended for other research purposes.

- **`custom-hook.cpp`**: A general framework to inject the DLL into the target program.
- **`utils.cpp`**: An in-house extension built on top of S2E symbolic analysis framework. It supports symbolic tag extraction, on-demand concretization, VM-to-host dropped file transferring, memory symbolic expression extraction, and taint analysis logic.
- **`forkprofiler.py`**: This is an investigation-oriented analysis tool that iterates through the execution traces generated by Marsea and reports the triaged path explosion source (i.e., system APIs) if there is any.
- **`NewCodeSearcher.cpp`**: This is a code-coverage-driven exploration technique that has been implemented as an S2E plugin. Unlike the default exploration technique, which picks a random module and then a random execution state to explore, our technique prioritizes unexplored code regions in the target module being analyzed.
- **`LibraryCallMonitor.cpp`**: This is a customized built-in S2E plugin that provides detailed execution tracing by logging all system APIs invoked by the target binary during analysis.
- **`CyFiFunctionModels.cpp`**: This is an in-house S2E plugin that provides the backbone functionality for the injected DLL.
- **`pipeline.py`**: This is the pipeline script used to create a Marsea project and terminate the analysis in case of a timeout.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: Device Tracking via Linux's New TCP Source Port Selection Algorithm

Moshe Kol
Hebrew University of Jerusalem

Amit Klein
Hebrew University of Jerusalem

Yossi Gilad
Hebrew University of Jerusalem

A Artifact Appendix

A.1 Abstract

This artifact contains a proof-of-concept implementation of a device tracking technique for Linux-based devices by exploiting the way Linux selects TCP source ports. The Linux TCP port selection algorithm is an adaptation of Algorithm 4 ("Double-Hash Port Selection Algorithm") from RFC 6056. The algorithm is used starting from kernel version `5.12-rc1`.

The artifact contains a tracking server written in Go and a tracking snippet written in HTML+JavaScript, served by the tracking server using HTTP.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our attack generates a device ID for the tested device, that can identify it across browsers, browser privacy modes, networks, containers and VPNs. Therefore, for the sake of maintaining evaluator's privacy, we recommend evaluating our artifact on a local/private network.

In addition, in the "Set-up" section, we instruct the evaluator to use older versions of Ubuntu 22.04 and Google Chrome. Older versions are at risk of security bugs, therefore using local network is preferred.

A.2.2 How to access

The artifact is available on GitHub:
https://github.com/0xkol/rfc6056-device-tracker/tree/09dd6ab68e10566eb6ca7760ef78d4689c7e2b85

A.2.3 Hardware dependencies

8GB of RAM, 4 CPU cores and 50GB free disk space.

A.2.4 Software dependencies

Tracking client requirements

1. **Linux kernel:** The Linux kernel of the client device must be one of the following versions: 5.12.*, 5.13.*, 5.14.*, 5.15–5.15.40, 5.16.*, 5.17–5.17.8.
2. **Google Chrome:** version 96.0.
3. (Optional) **Python:** `Python 3.5` or above.

Tracking server requirements We assume that the tracking server runs on a Linux host.

1. `Go` version `1.18`, the `google/gopacket` library and the `google/gopacket/pcap` library.
2. `libpcap-dev` package (on Ubuntu).
3. `git`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To evaluate our attack, you will need two Linux machines: one for our tracking server and one for the tracking client (that interacts with the server using Chrome). The client machine has specific Linux kernel constraint, so we recommend using a virtual machine (VM) for it. The tracking server can run on any Linux machine that has network connectivity (IPv4 and IPv6) to the client machine. In this document we describe how to run both server and client as (separate) virtual machines.

Configure Oracle VirtualBox: Download and Install Oracle VirtualBox from this URL https://www.virtualbox.org/wiki/Downloads.

Configure Host-Only Network on VirtualBox:

1. **Disable address range control (required on Linux hosts only):** Create the file `/etc/vbox/networks.conf` and write this line to it (including the asterisk): `* 0.0.0.0/0 ::/0`

2. **Open "Host Network Manager":** Open VirtualBox and click on "File" → "Host Network Manager".

3. **Create a New Host-Only Network:** On "Host Network Manager", click on the "Create" button. This will create a new interface on your host with the name `vboxnet0` (or similar).

4. **Configure IPv6:** By default, only IPv4 prefix will be assigned to the new virtual interface (`192.168.56.1/24` or similar). To configure IPv6, use our pre-generated ULA prefix `fd3f:e8d8:3a1f:0::/64`. On the "IPv6 address" field enter `fd3f:e8d8:3a1f:0::1` and on the "IPv6 Prefix Length" enter `64`.

5. Click on the "Apply" button. You should see no errors.

Tracking Client Installation: We describe here how to setup a tracking client machine using Oracle VirtualBox.

1. **Download and Install Ubuntu 22.04:** Download Ubuntu Desktop 22.04 (not 22.04.1) from this URL http://old-releases.ubuntu.com/releases/jammy/ubuntu-22.04-desktop-amd64.iso. Install Ubuntu as a new Virtual Machine on Oracle VirtualBox. Notes:

   - You may follow these instructions for reference: https://brb.nci.nih.gov/seqtools/installUbuntu.html
   - We recommend assigning to the VM 4GB of RAM, 2 CPUs and 20GB of disk space.
   - **Avoid downloading updates during installation.** Otherwise, Ubuntu will auto-update its kernel. Also, make sure your machine is **NOT connected to the Internet** during installation by changing the network adapter from "NAT" to "Not attached" in the VM "Settings" window.
   - The kernel version of your installed Ubuntu should be `5.15.0-25-generic`. You can view it using the command: `uname -a`

2. **Connect to the Internet:** When your VM is up and running, connect it to the Internet by changing the network adapter from "Not attached" to "NAT" in the VM "Settings" window. (Avoid updating Ubuntu if it prompts for an update.)

3. **Download and Install Google Chrome:** Download Google Chrome v96.0 from https://dl.google.com/linux/chrome/deb/pool/main/g/google-chrome-stable/google-chrome-stable_96.0.4664.110-1_amd64.deb

   Install using the following command: `sudo dpkg -i google-chrome-stable_96.*.deb`

4. **Switch to Host-Only Network:** Open the VM "Settings" window. On the "Network" tab change the network adapter to "Host-only adapter" and choose the name of the adapter you created previously (probably `vboxnet0` or similar).

5. **Configure IPv6:** Use the following command to ensure IPv6 connectivity between the VMs:

   ```
   sudo ip address add
       fd3f:e8d8:3a1f:0::10/64 dev IFNAME
   ```

   To find *IFNAME*, list the network adapters on the machine using the `ip address` command and note the interface name whose name is not `lo`. Beware: This command does not survive reboot.

6. (Optional) You can verify that the machine you installed is vulnerable (i.e. uses the un-patched version of Algorithm 4 of RFC 6056) by invoking our Python 3 detection tool: `python3 CVE-2022-32296_tester.py`

   Expected output:

   ```
   Verdict: RFC 6056 Algorithm 4 (Vulnerable)
   ```

Tracking Server Installation:

1. Install Ubuntu Desktop 22.04 on a separate virtual machine, similar to the "Tracking Client Installation".

2. **Install Packages:** Run the following commands:

   ```
   sudo apt update
   sudo apt install git golang-go libpcap-dev
   ```

3. **Clone Repository:** Clone the git repository using the following commands:

   ```
   git clone
       https://github.com/0xkol/rfc6056-device-
       tracker.git
   cd rfc6056-device-tracker
   git checkout 09dd6ab
   ```

4. **Install Go Libraries:** On the repository folder, type the following commands:

   ```
   go get github.com/google/gopacket
   go get github.com/google/gopacket/pcap
   ```

5. **Switch to Host-Only Network:** Similar to what you did for the client machine, change the network adapter to "Host-only adapter" and choose the name of the adapter you created previously. After this step, both the client and server VMs should be up and running, with their network adapter configured to the same Host-Only network created previously.

6. **Configure IPv6:** Similar to what you did on the client machine, type the following command:

```
sudo ip address add
    fd3f:e8d8:3a1f:0::20/64 dev IFNAME
```

   Beware: This command does not survive reboot.

7. **Compile and Run the Tracking Server:** Switch to the git repository folder. Then, compile the tracker using: `go build -o tracker tracker.go`

   The compilation should succeed (no output on the console). Proceed by running the server on the interface you discovered on the previous step. For example (assuming the interface is `enp0s3`): `sudo ./tracker -iface enp0s3`

   You should see the output:

```
RFC 6056 Device Tracker v1.3 start
    (capturing on: enp0s3)
```

A.3.2 Basic Test

Connectivity Test: By now you should have two VMs connected to the same Host-Only network, with IPv4 and IPv6 connectivity. Verify that you can ping from the client VM to the server VM by issuing:

```
ping6 fd3f:e8d8:3a1f:0::20
ping SERVER_IPV4_ADDRESS
```

You can find *`SERVER_IPV4_ADDRESS`* by issuing the `ip address` command on the server machine.

Browser Test: On the client VM, open the Google Chrome browser and browse to the server using both IPv4 and IPv6 (i.e. to URLs `http://[fd3f:e8d8:3a1f:0::20]/` and `http://SERVER_IPV4_ADDRESS/`). You should see a webpage with the title "RFC 6056 Device Tracker Demo".

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The same device ID is obtained in a Google Chrome regular tab and an incognito tab. Section 6.1 in the paper. Proven by experiment (E1).

(C2): The same device ID is obtained when using IPv4/IPv6. Section 6.2 in the paper. Proven by experiment (E2).

(C3): The dwell time on Google Chrome varies between 5-15 seconds, depending on the RTT to the tracking server and the physical machine. Section 6.4 in the paper. Proven by experiment (E3).

A.4.2 Experiments

In all of the experiments, you should verify that the tracking server is up and running, and that it has both IPv4 and IPv6 connectivity from the tracking client.

(E1): Cross browser privacy modes consistency. 30 human-minutes, 30 compute-minutes.
Tracking Client VM Preparation: Open two Google Chrome windows: a regular window and an incognito window. On each window, browse to the tracking server. *Make sure that the "Tracker address" field contains the IP address of the tracking server.*
Execution: On the normal window, hit "Fingerprint me!" to launch the fingerprinting process. Few seconds later, you should see "fingerprint" and "fingerprint hash" on the webpage. Write these down for later. Continue by hitting "Fingerprint me!" on the incognito window and ensure you get the same fingerprint. Avoid running two fingerprinting measurements simultaneously.
Results: The same fingerprint should be generated on each window.

(E2): Cross protocol consistency. 30 human-minutes, 30 compute-minutes.
Tracking Client VM Preparation: Open a Google Chrome window (normal one is enough), and browse to the tracking server (over IPv4 or IPv6, does not matter).
Execution: Fingerprint the client machine once using an IPv4 address of the tracking server. Write down the fingerprint for a later comparison. Fingerprint the client machine again, but now use IPv6 as the tracking server. (On the "Tracker address" field use `[fd3f:e8d8:3a1f:0::20]` (including brackets!).) Verify that you get the same fingerprint on each run.
Results: The same fingerprint should be generated for both IPv4 and IPv6.

(E3): Dwell time. 30 human-minutes, 30 compute-minutes.
Tracking Client VM Preparation: Open a Google Chrome window and browse to the tracking server.
Execution: Fingerprint the client machine using IPv4 or IPv6 (it doesn't matter which at this point) and write down the "total time" reported in the webpage. Repeat the experiment a few times to obtain an average readout.
Results: You should observe an average dwell time of 5-15 seconds, depending on the network RTT and physical machine.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220912. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

A Artifact Appendix

A.1 Abstract

This artifact performs model inference on a programmable switch to complete traffic classification. It specifically includes the transformation of the decision tree model to the flow table and the logic of the control plane and data plane on the programmable switch. The evaluation requires a Intel Tofino 1 programmable switch with SDE version 9.1.0, and two servers to send and receive traffic respectively. In addition, a python execution environment is needed that can perform model transformation and interaction between the control plane and data plane. The verification is done by calculating the accuracy of the received packet class, and the expected result is that the accuracy is basically the same as the result in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** New algorithm
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/IDP-code/NetBeacon
- **Code licenses (if publicly available)?:** MIT License
- **Archived (explicitly provide DOI or stable reference)?:** Yes

A.3 Description

A.3.1 How to access

https://github.com/IDP-code/NetBeacon

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

N/A

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

No

A.4 Installation

Download from the https://github.com/IDP-code/NetBeacon

A.5 Experiment workflow

N/A

A.6 Evaluation and expected results

N/A

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

USENIX'23 Artifact Appendix: SPECTREM: Exploiting Electromagnetic Emanations During Transient Execution

Jesse De Meulemeester
COSIC, KU Leuven

Antoon Purnal
COSIC, KU Leuven

Lennert Wouters
COSIC, KU Leuven

Arthur Beckers
COSIC, KU Leuven

Ingrid Verbauwhede
COSIC, KU Leuven

D Artifact Appendix

D.1 Abstract

This appendix describes our artifacts for SPECTREM, a physical transient execution attack. In our paper, we discuss how the physical effects of transient instructions can be leveraged to extract secret information. In this artifact, we provide the source code for our proof-of-concept implementations and the scripts to take and evaluate the traces. To allow the reproduction of our work, we also provide the side-channel traces that were used to produce the results in our paper.

D.2 Description & Requirements

D.2.1 Security, privacy, and ethical concerns

None.

D.2.2 How to access

Our artifact consists of two repositories. The first one, hosted on GitHub, contains all source code and scripts related to our POC implementations. This repository can be accessed at `https://github.com/KULeuven-COSIC/SpectrEM/tree/1c0207db3d55580b7f31dfb22f57100ea5544707`.

Additionally, to enable the reproduction of our results, we also provide the side-channel traces for our work in KU Leuven's Research Data Repository (RDR). This dataset can be accessed at `https://doi.org/10.48804/AHTI1A`.

D.2.3 Hardware dependencies

To allow the reproduction of our results without requiring an extensive EM side-channel setup, we provide our pre-recorded traces, as discussed above. Therefore, we do not require any specific hardware setup.

To download all pre-recorded traces, at least 520 GB of free disk space is required. Additionally, the provided Python scripts to evaluate these traces use up to 12 GB of RAM. We, therefore, recommend at least 16 GB of RAM.

D.2.4 Software dependencies

The evaluation of the side-channel traces was performed using Python 3.11.4. The exact required Python packages are detailed in our GitHub repository.

D.2.5 Benchmarks

None.

D.3 Set-up

D.3.1 Installation

1. Create a new Python environment and install all dependencies as described in `readme.md` in our GitHub repository.

2. Download the traces from the data repository. We provide a Python script to download these folders in our GitHub repository (`traces/download-traces.py`). Specifically, download the following directories:

   - `0-base-experiments/` (33 GB)
   - `1-additional-experiments/` (39 GB)
   - `2-reducing-assumptions/` (21 GB)
   - `3-case-study/` (2 GB)

   The directory containing the MLP training data (`4-mlp-data/` (89 GB)) may also be downloaded but is not required as we provide pre-trained MLP networks along with the evaluation traces.

D.3.2 Basic Test

Activate the created Python environment and start Jupyter Notebook. Open the notebook `scripts/evaluate/evaluate_extraction_methods.ipynb` and run the first cell. If no errors are displayed, all packages are installed correctly.

To make sure the traces are downloaded to the correct location, step through the notebook. If the traces are downloaded

to a different location, the path pointing to these traces can be changed by modifying the `prerecorded_traces_dir` variable.

If no errors are encountered when stepping through this Jupyter Notebook, everything is set up correctly.

D.4 Evaluation workflow

D.4.1 Major Claims

(C1): Variable-time instructions and control flow dependencies enable physical transient execution attacks. In optimal conditions, both SPECTREM and MELTEMDOWN can achieve low BER, even when only considering a single trace (cf. Section 6 of our paper).

(C2): The simplifications introduced for evaluation can be removed by additional post-processing (cf. Section 7 of our paper). Specifically, we show that SPECTREM attacks can still be carried out when the clock frequency is not locked, when the POC is not pinned to a specific core, and when using cache thrashing.

(C3): The code pattern that forms control flow gadgets can be found in OpenSSH. With only minor changes, the two uncovered gadgets can be exploited through the network interface (cf. Section 8 of our paper).

D.4.2 Experiments

Before running the experiments, we recommend stepping through the following Jupyter notebook: `scripts/evaluate/evaluate_extraction_methods.ipynb` [30 human-minutes + 10 compute-minutes]. This notebook details how the traces are evaluated. For each of the following experiments, we provide Python scripts that use the techniques discussed in this notebook to automatically evaluate the traces.

(E1.1): [Base experiments] [15 human-minutes + 20 compute-minutes + 92 GB disk + 4.2 GB RAM]: This experiment evaluates the baseline performance of the SPECTREM and MELTEMDOWN POCs.
Preparation: Download the traces in folder `0-base-experiments` from the data repository.
Execution: Run the following Python script: `scripts/reproduce/0-base-experiments.py`.
This script will output the BERs for each POC.
Results: The Python script will print the BERs for the 5 different base POCs. The expected outputs are included in `scripts/readme.md`. These results can be compared with the results in our paper in Table 1 and Section 6.2.

(E1.2): [Additional experiments] [15 human-minutes + 20 compute-minutes + 110 GB disk + 4.5 GB RAM]: This experiment evaluates the performance of the POCs under different numbers of training packets and different numbers of `udiv` instructions.
Preparation: Download the traces in folder `1-additional-experiments` from the data repository.
Execution: Run the following Python script: `scripts/reproduce/1-additional-experiments.py`.
This script will output the BERs for each POC.
Results: The Python script will print the BERs for the different conditions and produce two figures. The expected outputs are included in `scripts/readme.md`. These results can be compared with the results in our paper in Figure 5 and Figure 7.

(E2): [Reducing assumptions] [15 human-minutes + 2 compute-hours + 58 GB disk + 12 GB RAM]: This experiment evaluates the effect of removing the evaluation assumptions.
Preparation: Download the traces in folder `2-reducing-assumptions` from the data repository.
Execution: Run the following Python script: `scripts/reproduce/2-reducing-assumptions.py`.
This script will output the BERs for each experiment.
Results: The Python script will print the BERs for the 5 different base POCs. The expected outputs are included in `scripts/readme.md`. These results can be compared with the results in our paper in Table 1 and Section 6.2.

(E3): [Case study] [30 human-minutes + 2 compute-minutes + 5 GB disk + 2.8 GB RAM]: This experiment evaluates the performance of two real-world code patterns in OpenSSH.
Preparation: Download the traces in folder `3-case-study` from the data repository.
Execution: Verify that the two code snippets in Listings 4 and 5 in our paper are indeed taken from the latest version of OpenSSH at the time of submission (9.3). To evaluate the performance of these two gadgets, run the script `scripts/reproduce/3-case-study.py`.
Results: The Python script will print the BERs for the two gadgets. The expected outputs are included in `scripts/readme.md`. These results can be compared with the results in our paper in Section 8.

D.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: ARMore: Pushing Love Back Into Binaries

Luca Di Bartolomeo
EPFL

Hossein Moghaddas
EPFL

Mathias Payer
EPFL

A Artifact Appendix

A.1 Abstract

ARMore's artifact contains the source code necessary to run our static rewriter. This document describes how to set-up our prototype, gives a brief overview of the resource requirements to replicate some of the experiments conducted in our evaluation, along with instructions to run them.

A.2 Description & Requirements

This artifact is shipped as an aarch64 Docker container. The provided script `run.sh` takes care of importing the container and spawning a shell. All the relevant files for the experiment are in the root home folder.

A.2.1 Security, privacy, and ethical concerns

This artifact does not contain any threat to the system's integrity or privacy. However, we still recommend running it inside a sandboxed environment (either a VM or a container).

A.2.2 How to access

ARMore's artifact is available online at https://zenodo.org/record/7707863. ARMore's source code can be found at https://github.com/hexhive/retrowrite. Evaluators can visit commit 4a7193b to reproduce experiments shown in the paper.

A.2.3 Hardware dependencies

The evaluation of ARMorerequires an aarch64 machine with large amounts of RAM (around 64 GB for all the experiments). Since we do not expect the evaluators to have access to such hardware, we provide in this artifact a reduced version of our experiments that should run even on an emulated aarch64 VM running on a x86 host with 8 GB of RAM. We provide instruction to run the experiments on both aarch64 or x86.

Note: if the evaluators consider running the full suite of experiments a necessity, we can provide remote ssh access to the required hardware.

A.2.4 Software dependencies

Since the artifact is shipped as a Docker container, all dependencies are already installed. However, in case evaluators would like to run it outside Docker, those are the required dependencies:

ARMore's evaluation was run on Ubuntu 20.04.2. The requried Ubuntu packages are `python3-pip, tcl-dev, build-essential, make`. The required python libraries are the following:

- archinfo
- pyelftools
- capstone (version >= 4.0.2)

the can be all installed by running from the home folder of the Docker container:
`pip3 install -r retrowrite/requirements.txt`
Finally, a working installation of `AFL++` is required.

Note: if the evaluators want to verify the fourth claim too, please contact us and we can prepare a disk image with the kernel patch already applied.

A.2.5 Benchmarks

ARMore's evaluation in the paper used 3 different benchmarks:

- SPEC CPU 2017: to measure the baseline overhead introduced by ARMore, we use the entire SPEC benchmark. However, this benchmark requires large amount of RAM (64 GB) and considerable compute time (around 12 hours). For this reason, in this artifact evaluation we provide a reduced experiment using small benchmarks taken from https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
- MAGMA: to measure the overhead introduced by ARMore's coverage instrumentation, we run the MAGMA fuzzing benchmark. As before, this benchmark is quite expensive to build and run - we provide another minified version of this experiment to be able to run it in an emulated environment.

- SQLite test suite: we include the source code of the testsuite in the artifact along with scripts to test and run it.

A.3 Set-up

A.3.1 Installation

The artifact is shipped as an aarch64 docker image. Depending on the available hardware, we provide two options:

arm64 host: This is the preferred way, as it will make the experiments considerably faster. All the dependencies are already setup inside the Docker container. If not using the container, the following steps need to be taken: The following ubuntu packages need to be installed: `build-essential`, `python3-pip`, `tcl-dev`, `make`. Afterwards, go inside the `retrowrite` directory and run:
`pip3 install -r requirements.txt`
to install ARMore's dependencies.

x86 host: To run it on an x86 host, install the support for emulation of the arm64 architecture in docker images. The simplest way to do this is to run the following:
`docker run -privileged -rm tonistiigi/binfmt -install arm64`
as explained in https://docs.docker.com/build/building/multi-platform/. To test if multi-architecture support is running, you can try the following:
`docker run -rm arm64v8/alpine uname -a`.
Afterwards, download the artifact and use the `run.sh` script to import the image and spawn a shell inside the container.

A.3.2 Basic Test

To test basic functionality of ARMore, run `run.sh` to spawn a shell, go inside the home folder and run:

```
./retrowrite/retrowrite /bin/ls ls.s
./retrowrite/retrowrite -a ls.s rewritten_ls
./rewritten_ls
```

If the output is exactly the same as when running `/bin/ls`, then ARMoreis set up correctly.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *ARMore's baseline overhead is <1%*. This is proven by experiment (E1) which is a reduced version of the one described in Section 5.2 of the paper.

(C2): *ARMore's rewriting is correct and preserves functionality.* This is proven by experiment (E2), as described in Section 5.1 of the paper.

(C3): *ARMoreenables efficient fuzzing of aarch64 closed-source binaries.* This is proven by experiment (E3), a reduced version of the one described in Section 5.4 of the paper.

A.4.2 Experiments

(E1): *[Baseline overhead] [10 human-minutes + 1 compute-hour]: This experiment shows how rewriting binaries without instrumentation adds negligible overhead (<1%). We took some binaries from the benchmarks-game* [1] *and use them to test the overhead introduced by ARMore.*
How to: Go inside the folder `/claim_one_low_overhead`. The script `run.sh` will compile the benchmarks and store the binaries in the `compiled` folder. Afterwards, the script will rewrite the binaries with ARMoreand store the result in the `rewritten` folder. Finally, the benchmarks will be run and the 2 different set of times will be printed.
Execution: Go inside the folder `/claim_one_low_overhead` and run the script `run.sh`.
Results: While this experiment is certainly not conclusive compared to more heavy-weight benchmarks, the times noted by `Rewritten time` should be around 1% higher than the times noted by `Original time`.

(E2): *[Correctness] [5 human-minutes + 2 compute-hours]: This experiments verifies the correctness claims of ARMore, namely that it exactly preserves the original binaries' behaviour. This is done by rewriting the SQLite binaries and running their relevant testsuites.*
How to: Go inside the folder `/claim_two_correctness`. The script `test_sqlite.sh` inside will build and rewrite the binaries from SQLite, and then run the testsuite on them.
Execution: Go inside the folder `/claim_two_correctness` and run the script `test_sqlite.sh`, and check its output.
Results: The fifth to last line of the script output should report `0 errors out of 252692 tests`, indicating that all tests passed correctly.

(E3): *[coverage instrumentation overhead] [30 human-minutes + 2 compute-hours]: This experiment verifies the claims in Section 5.4, that is fuzzing with ARMore's coverage instrumentation is comparable to fuzzing with source-based instrumentation (`afl-cc`).*
How to: Go inside the folder `/claim_three_fuzzing`. The script inside will build the binaries from the first experiment (E1) and store the result in the `compiled` folder. Then, it will com-

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/

pile them again with source instrumentation and store the result in the folder `source_instrumented`. Finally, the script will instrument the original binaries inside the `compiled` folder, adding coverage instrumentation, and store the result in the `rewritten_instrumented` folder.

Execution: Go inside the folder `claim_three_fuzzing` and run the script `run.sh` to build the instrumented binaries. Then, run the script `fuzz.sh` to fuzz each binary twice for 2 minutes: first, the source-instrumented version compiled with `afl-cc` will be fuzzed — secondly, the binary-instrumented version rewritten with ARMorewill be fuzzed. The AFL UI is disabled, and only the executions per second are reported.

Results: As claimed in the paper, the average difference in executions per second should be slower for ARMore-compared to `afl-cc` by around 25%. We note that this number is very variable, due to the non-deterministic nature of fuzzing.

In all of the above blocks, please provide indications about the expected outcome for each of the steps (given the suggested hardware/software configuration above).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

Prime Match: A Privacy-Preserving Inventory Matching System

Antigoni Polychroniadou, Gilad Asharov[1], Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, Manuela Veloso

J.P. Morgan

A Artifact Appendix

A.1 Abstract

In this work, we introduce secure multiparty computation in financial services by presenting a solution, called *Prime Match*, for matching orders in a stock exchange while maintaining the privacy of the orders. Information is revealed only if there is a match. Our central tool is a new protocol for secure comparison with malicious security, which can be of independent interest. In this artifact, we showcase the major claims of our paper titled "Prime Match: A Privacy-Preserving Inventory Matching System".

A.2 Description & Requirements

Prime Match involves a server/bank and (at least) a client which submits orders to buy or sell a particular stock/symbol, along with the intended quantity (number of shares), to the bank. The bank does not learn any information about what the client is interested in on any stock that is not matched, and likewise, the client does not learn any information on what is available in the bank unless she/he is interested in that as well. Only after matches are found, the bank and the client are notified, and the joint interest is revealed.

A.2.1 Security, privacy, and ethical concerns

No concerns.

A.2.2 Hardware dependencies

Our code can run on any commodity hardware since our implementation is targeted to a real-world application where clients hold conventional computers. For example, for our experiments, one of the two clients runs on an Intel Core i7 processor, with 6 cores, each 2.6GHz, and another one runs on an Intel Core i5, with 4 cores, each 2.00 GHz. Both of them are Windows machines. Our server runs in a Linux AWS instance of type c5a.8xlarge, with 32 vCPUs. However, it is possible for a reviewer to test our system by running both the server and client(s) on the same machine. If two clients are selected for a test on the same machine, one of the two clients needs to be executed in incognito mode from the browser.

[1]Currently at Bar-Ilan University, based on work that was conducted while at J.P. Morgan.

A.2.3 Software dependencies

For the purposes of practical convenience, adoption, and portability, our client module is entirely browser-based and written in JavaScript. Its cryptographically intensive components are written in the C language and compiled using Emscripten into WebAssembly (which also runs natively in the browser). Our server is written in Python and also executes its cryptographically intensive code in C. Both components are multi-threaded—using WebWorkers on the client side and a thread pool on the server's—and can execute arbitrarily many concurrent instances of the protocol in parallel (i.e., constrained only by hardware). All players communicate by sending binary data on WebSockets. Our code is independent of the operating system (MacOS is recommended) and can run on any browser (Google Chrome is recommended).

A.2.4 Benchmarks

None, our code generates random inputs on the fly.

A.3 Set-up

A.3.1 Installation

Our library consists of multiple components. Its client is written in JavaScript, while we provide both Python and JavaScript implementations of the server. Both the Python server and the JavaScript client use C code, which is separated into its own folder. Next, we provide the installation guide to install both the server and the client (the same information is also provided in the Readme files of the repo).

Python Server Instructions:

Prerequisites. Install Python 3.8 and pip. Add ./src to the PYTHONPATH environment variable. Run pip3 install −r requirements.txt.

Installation. In the python folder directory execute:

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_C_FLAGS_
RELEASE="-DNDEBUG -O3" -DCMAKE_SHARED_LINKER_
FLAGS_RELEASE="" ..
make
```

JavaScript Client Instructions:

Prerequisites. Install Yarn (tested with version v1.22.4). To build the C components, download and install emscripten.

Installation. After installing the tools in the Prerequisites, navigate back to the JavaScript folder directory and type yarn. To build the WASM components, then type:

```
mkdir build
cd build
emcmake cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE
_C_FLAGS_RELEASE="-DNDEBUG -O3" -DCMAKE_SHARED_
LINKER_FLAGS_RELEASE="" ..
make
```

To build the webpack components, type yarn run build.

A.3.2 Basic Test

To run an example (1 server and 1 client) the following steps must be followed.

1. Open a terminal at the JavaScript folder, run the Server with the command $python −m http.server 9000 (the server starts listening).

2. Open another terminal at the Python server, run the application with the command $python3 src/app.py <symbol−size> <wait−time> e.g. $python3 src/app.py 100 40
The symbol−size refers to the number of symbols/stocks, and wait−time refers to the time till the server waits (in seconds) to start the matching process. This means that the client has submitted 100 symbols to be matched (or not) with the symbols of the Server.

3. Next in a browser, go to http://localhost:9000/dist/ and answer the prompts with the symbol−size = 100 (should match the symbol−size we entered above) and the 1 (Short) or 0 (Long) direction/side. Short is for selling stock, and Long is for buying a stock. The hard-coded direction for the server is *Long*.

4. Initially, the client browser shows the table of symbols with randomly assigned quantities and with 0 as Matched; however, when the wait−time ends, the Matched will be updated after matching the symbols with the symbols of the server. This will conclude a successful execution.

A.4 Evaluation workflow

A.4.1 Major Claims

In this section, we provide the major claim of our paper. For testing purposes, we have created a UI that differs from the one in Figure 5 of [1], as the UI in effect is found in J.P. Morgan's markets portal, where Prime Match is deployed. The markets portal has been built over many years, and it is not available since it can reveal private information from other applications about the bank which are not relevant to this paper. The same holds for the trade management platform. In particular, our code in production was integrated into the trade management platform of the US bank, as described in detail in the paper (See Figure 6 for our final architecture). Table 2 of our paper is generated by running the repository we have uploaded on the private GitHub. We summarize our claim as follows:

(C1): *Prime Match has a throughput of 10 matches per second. This is proven by the experiment (E1) (See Section A.4.2) described in Section 5 of our paper, whose results are illustrated/reported in Table 2 of our paper.*

A.4.2 Experiments

Our experiment is described in Section 5 in the second paragraph of "Secure Minimum Protocol Performance".

(E1): *[25 human-minutes + 10 compute-minutes + <500MB disk]: This experiment aims to run an experiment with two clients and verify the running times as presented in Table 2 for 100, 200, 500, and 1000 symbols (for bank-to-client). This is done by adjusting the* symbol−size *in the command of Step 2 in Section A.3.2 per client. The code can also run for a larger number of symbols, such as 2000, 4000, etc. If one tests these cases, they should ensure that the* wait−time *is increased to a range of 50 to 100 seconds such that there is enough time to register the symbols of the clients before the matching process starts.*

Preparation: *Install the packages as described in Section A.3.1.*

Execution: *Follow all steps in Section A.3.2 but repeat Step 3 two times (sequentially) to accommodate a second client. Note that both clients must be initiated before the* wait−time *is passed. Moreover, the second client needs to be executed in incognito mode from the browser. Repeat the process for different numbers of symbols.*

Results: *After the completion of the experiment, right-click on the client browser to select* console *and check the running time and the MB sent and received, which are reported in the third, fifth and sixth columns of Table 2, respectively.*

References

[1] Antigoni Polychroniadou, Gilad Asharov, Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. Prime match: A privacy-preserving inventory matching system. Cryptology ePrint Archive, Paper 2023/400, 2023. https://eprint.iacr.org/2023/400.

USENIX'23 Artifact Appendix: "EOS: Efficient Private Delegation of zkSNARK Provers"

Alessandro Chiesa
UC Berkeley & EPFL

Ryan Lehmkuhl
MIT*

Pratyush Mishra
Aleo & University of Pennsylvania†

Yinuo Zhang
UC Berkeley

A Artifact Appendix

A.1 Abstract

We provide EOS, a Rust library that realizes our delegation protocols for zkSNARKs, and includes components of independent interest. EOS relies on, and contributes to, the state-of-the-art arkworks libraries. We generalize the existing arkworks implementations of PIOP and PC schemes of the Marlin zkSNARK to support our new abstractions for secret-shared field elements and polynomials.

A.2 Description & Requirements

EOS can be run on any system with access to a C and Rust compiler. All the experiments we describe in subsequent sections can be run on a single machine, but recreating the experimental setup used in the paper requires the following AWS instances running Ubuntu >18.04:

- *Delegator*: A `r4.xlarge` instance in the `us-west-2` region
- *Worker 1*: A `c5.24xlarge` instance in the `us-west-1` region
- *Worker 2*: A `c5.24xlarge` instance in the `us-west-2` region

This setup emulates the LAPTOPHB delegator setup described in the paper. To emulate the LAPTOPLB delegator setup, see Appendix A.2.4. We don't provide instructions for emulating the MOBILE delegator setup since this assumes access to specific hardware.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

A tarball of EOS can be found at this link.

A.2.3 Hardware dependencies

None.

*Work partially done while at UC Berkeley.
†Work partially done while at UC Berkeley.

A.2.4 Software dependencies

EOS can be run on any system with access to a C and Rust compiler. The Rust compiler can be installed using this link and the C compiler can be installed from here. Parsing the execution traces from the experiments required Python3 which can be installed from here.

To emulate the LAPTOPLB delegator setup described in the paper, the bandwidth of the delegator must be throttled to 350 Mbps downlink and 13 Mbps uplink. On Linux platforms, this can be accomplished via the wondershaper package as follows:

```
sudo wondershaper {interface} 350000 13000
```

This can be reset by running:

```
sudo wondershaper clear {interface}
```

A.2.5 Benchmarks

None.

A.3 Set-up

All machines used should accept TCP traffic on ports 8000-10000.

A.3.1 Installation

See the README contained in the artifact.

A.3.2 Basic Test

To run a simple functionality test, navigate to the `experiments/artifact_evaluation` directory and run the `bench_snark_delegator.sh`, `bench_snark_w1.sh`, and `bench_snark_w2.sh` locally within three separate shells. These scripts should finish within 5 minutes (assuming you've already built EOS) with the following output:

```
Running snark delegator with 2^15 constraints
Running snark delegator with 2^16 constraints
...
```

After it completes, run `python3 parse.py` from the delegator machine and ensure that no errors occured, i.e., you should get text output that looks something like:

```
SNARK (mode 0, constraints 2^15):
Online latency: 3.7720s
Delegator uptime: 1.7731s
----
Online comm.: 11.5403MB
```

and not:

```
Failure when reading trace for SNARK (mode
0, constraints 2^15) -- try rerunning
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): EOS reduces end-to-end latency of proof generation by up to 26× and lowers the delegator's active computation time by up to 1447× with minimal communication overhead. This is proven by the experiment (E1) described in Section 8.3 of our paper whose results are shown in Table 1 and Figures 5, 6, 7, and 8.

(C2): EOS enables proving instances up to 256× larger instances within a memory budget of 3 GB. This is proven by the experiment (E2) described in Section 8.2 of our paper whose results are shown in Table 1.

A.4.2 Experiments

(E1): [10 human-minutes + 10 compute-minutes]: This experiment will confirm the numbers given in Table 1 + Figures 5, 6, 7, and 8 for the LAPTOPHB and LAPTOPLB setups. By default, it will run for instance sizes of $2^{15} - 2^{20}$, but can be easily modified to run up to instances of size 2^{25} at the cost of more compute-time.

Note that, if desired, the latency baselines can be recreated by running benchmarks for the Marlin zkSNARK over the BLS12-381 curve locally on the worker and delegator machines.

How to: See the README contained in the artifact for information on how to configure and run the experiments
Results: The latency + communication results should match the upperbounds given in Table 1 and the numbers given in Figures 5, 6, 7, and 8 for the LAPTOPHB and LAPTOPLB setups.

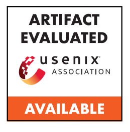
(E2): [10 human-minutes + 10 compute-minutes]: This experiment follows a similar workflow as above, but memory-usage is measured for each protocol execution.

How to: Run the same experiment above, except modify the relevant benchmarks/scripts to also output the memory usage. For example, on Linux we can simply prepend `/usr/bin/time -v` to the relevant command.

Results: After running, inspect the execution traces on the delegator (in `results/delegator`) to retrieve the memory usage. Using the `/usr/bin/time` command described above, this can be obtained by looking for the "Maximum resident set size" field. Ensure that this value is consistent with the upperbounds given in Table 1.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: TAP: Transparent and Privacy-Preserving Data Services

Daniël Reijsbergen[†] Aung Maw[†] Zheng Yang[‡] Tien Tuan Anh Dinh[†] Jianying Zhou[†]

[†]Singapore University of Technology and Design, Singapore, Singapore
[‡]Southwest University, Chongqing, China

A Artifact Appendix

A.1 Abstract

This document describes the code that was used to produce the experimental results in Section 6 of the TAP paper. TAP provides a level of security and privacy that exceeds that of related multi-user approaches (e.g., a trusted server), so the main purpose of the experiments is to demonstrate that TAP still has *practical performance at scale* despite the additional security guarantees. As such, the experiments demonstrate the feasibility (in terms of execution times) of database operations in TAP, e.g., look-up, sum, min, max, and quantile queries.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None – however, if the machine on which the code is run already has MySQL Server installed, then please remember to reset the root password after concluding the experiments.

A.2.2 How to access

The artifact can be found here: https://github.com/tap-group/transparent-data-service/tree/9e97cd42e12fb2941253b0960d4689bf944889a0

A.2.3 Hardware dependencies

The microbenchmark experiments were performed on a MacBook Pro with the following specifications (the code should also work on Linux and Windows systems):

- Processor: 2.4 GHz Quad-Core Intel Core i5
- Memory: 16 GB 2133 MHz LPDDR3
- Operating System: MacOS Monterey Version 12.5.1

The other experiments were performed on Amazon Web Services (AWS) machines: *t2.micro* to represent low-capacity machines, *t2.xlarge* to represent medium-capacity machines, and *m5.4xlarge* to represent high-capacity machines. The latter types have an hourly cost to run, and all types require an AWS account. In the following, we will therefore focus on running the code on a single PC or laptop with the above specifications or similar – we will refer to such a machine as a "medium-capacity machine".

A.2.4 Software dependencies

Working installations of Go, MySQL, and GCC are required (see also the installation guide below). All other dependencies are installed automatically by Go.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Go. The programming code is written in the Go language and therefore requires a working Go installation (version 1.16 or above). To install Go, download it from https://go.dev/doc/install and follow the installation instructions.

MySQL. The TAP implementation requires a working version of MySql Server to simulate the server's back-end, which can be obtained, e.g., through https://dev.mysql.com/downloads/installer/ for Windows. The installer may require that a root password is set. If so, set a temporary password (e.g., '0000'). The TAP code assumes that the root user can access the database without a password. Once MySQL Server has been installed, the password for the root user can be removed as follows: start MySQL from the command line using the following command (which assumes that '0000' is the root user's password)

```
mysql -uroot -p0000
```

then run

```
SET PASSWORD FOR root@localhost='';
```

to remove the password. Finally, create a database named 'tap' by executing the following command in the command line tool:

```
CREATE DATABASE tap;
```

GCC. The TAP implementation uses `go-ethereum` for operations on elliptic curves, and `go-ethereum` in turn requires GCC. To install GCC on Windows, one can use MinGW https://www.mingw-w64.org/downloads/ – make sure that the main executables are accessible via the system path (e.g., by adding `C:\Users\...\mingw64\bin` to the system path variable). On Linux-based systems, run

```
sudo apt install build-essential
```

in the command line.

TAP Code. After downloading the TAP code, use the command line tool to navigate to the main folder and execute

```
go mod tidy
```

to instruct Go to download the required external libraries. It is also helpful to ensure that the *output* folder is empty to avoid confusion with the sample output files that are included with the repository.

A.3.2 Basic Test

To check whether Go was successfully installed, execute the following command on the command line:

```
go version
```

which should return the installed Go version. To check whether MySQL Server was successfully installed, execute the following command:

```
mysql -uroot
```

The above command should start the MySQL command line tool. To check whether GCC was successfully installed, execute the following command:

```
gcc --version
```

This should return the version number of the GCC installation. Finally, to check whether the TAP implementation was successfully built, execute the following command:

```
go run . -experiment1a
```

This starts a basic experiment that tests the time cost of processing data insertions at the server – it should run for less than a minute on a medium-capacity machine. After its completion, it should write "results:" to the command line, followed by a list of simulation results (time and storage costs).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Look-ups, which are essential for clients who monitor their own data, have negligible overhead, and the cost of the other operations is reasonable on a medium-capacity machine.* In particular, a look-up query in TAP takes $\approx$0.005s, a sum query takes $\approx$0.1s, a min/max/quantile query takes $\approx$1s, and an audit takes $\approx$10s in a table with 100–10,000 rows. This is demonstrated by Experiment **E1** described in Section 6.2 whose results are displayed in Figure 8a.

(C2): *For look-up and sum queries, total processing times are dominated by proof generation times at the server. For min and quantile queries, proof generation and verification times are similar (around 10 seconds) at the client and server.* This is demonstrated by Experiment **E2** described in Section 6.3 whose results are displayed in Figure 8b.

(C3): *TAP has a smaller storage requirements for the ADS than Merkle^2, although audits and inserts are faster in Merkle^2.* This is demonstrated by experiment **E3** described in Section 6.4 whose results are displayed in Figure 9. (The IntegriDB implementation[1] requires a different set-up – e.g., a specific version of OpenSSL on Linux – and is hence not integrated into the artifact.)

(C4): *On a high-capacity machine, it is feasible to build TAP's data structure, audit (targeted portions of) the data structure, and perform queries even in real-world settings – i.e., databases to which millions of rows are added every hour.* In particular, the time cost of building TAP's data structure increases from $\approx$0.01s for 100 rows to $\approx$1000s for 10^7 rows regardless of the number of subtrees. Furthermore, the cost of a full audit increases from $\approx$1 second for 100 rows to $\approx$450 seconds for 15 000 rows. Finally, the cost of a quantile query over 1) the entire dataset or 2) a fixed number of subtrees depends at most logarithmically on the total number of rows. This is demonstrated by Experiment **E4** described in Section 6.5 whose results are displayed in Figure 10.

A.4.2 Experiments

(E1): *Microbenchmarks: <1 compute-hour + <1 MB disk*
Execution: This experiment can be reproduced by executing
`go run . -experiment3`
in the repository's main folder. One query type – look-up, sum, average, count, min, max, median, and top 5% queries – is performed in each of experiments 3a–h across 100 epochs.
Results: After each experiment (i.e., 3a–h), a csv-file with a corresponding name will be written to the *output*

[1] https://github.com/integridb/Code

folder – each row in the csv-file corresponds to the result for one epoch. Each csv-file can be used to plot a line in Figure 8a with the epoch counter on the *x*-axis and the query time duration on the *y*-axis. (The line for audits comes independently from a modified version of `experiment7` as discussed under **E4**.) The time costs in the output table should be similar to those mentioned under **C1** on a medium-capacity machine.

(E2): *End-to-end Costs: ≈1 compute-hour + <1 MB disk*

Execution: This experiment can be reproduced by executing

```
go run . -experiment4
```

Results: Each row in the resulting csv-file (*output/experiment4.csv*) corresponds to a unique combination of a) query type (look-up, sum, min, and quantile) and b) number of epochs (10, 30, 100) with 100 new rows per epoch. The following columns contain the relevant data: *prefix_proc_time_server*, *prefix_proc_time_client*, *query_proc_time_server*, *query_proc_time_client*, and *total_time*. The first two correspond to the "prefix tree proof generation" and "prefix tree proof verification" bars in Figure 8b. The "sum tree proof generation", and "sum tree proof verification" bars in Figure 8b represent the difference between the total processing times and the prefix tree processing times. Finally, the "other" bar in Figure 8b represents the difference between the total times and the query processing times. The "other" bar captures network latency, but this is only relevant if the client and server run on different machines.

The claim **C2** can be verified by comparing "query_proc_time_server" to "query_proc_time_client" in the output file: the difference should be more stark in the first six data rows (look-up and sum queries) than in the last six data rows (min and quantile queries)

(E3): *Baselines: 5–10 compute-hours + <1 MB disk*

Execution: This experiment can be reproduced for TAP by executing

```
go run . -experiment5
```

This records the storage cost of building TAP's data structure and the execution cost of the different query types for a varying number of data rows. To reproduce the results for Merkle^2, execute

```
go run .
```

in the `msquare` subfolder of the repository.

Results: After completing the first command, a single output table is created with the data for the graphs of Figure 9 for TAP. The 'storage' column corresponds to Fig 9a, the 'auditor' column to Fig 9b, the 'insert' column to Fig 9c, the 'lookup' column to Fig 9d, the 'sum' column to Fig 9e, and the 'min' column to Fig 9f. The number of rows is not printed by default, but corresponds to 100 times the number in the function call (as per lines 648-654 of `main.go`). The second command will produce a table for Merkle^2 with a similar structure to the one for TAP in the *msquare* folder. **C3** can be verified by comparing the values in the two tables for the same row index.

(Perhaps confusingly, the *x*-axis of Fig. 9a is labeled "epochs" despite there being only one row per epoch.)

(E4): *Scalability: >10 compute-hours + <1 GB disk*

Preparation: The scalability experiments were designed to be run overnight on a *high-capacity machine*, and will take a considerable amount of time to complete with the default settings (i.e., >10 hours on a high-end AWS machine). The range of the experiments can be changed by modifying lines 701 and 704 (the second number in the function call represents the maximum number of epochs, so setting this to a lower number will substantially reduce processing times), and the number of samples can be reduced by setting `nSamples6` and `nSamples7` to 1 in lines 712–713 of `main.go`.

For example, setting `nSamples6` and `nSamples7` to 1, and using

```
getExperimentRange(100, 10000, 3, 10, 100, 2)
```

in line 701 and

```
getExperimentRange(100, 1000, 2, 10, 100, 2)
```

in line 704 should cause both sets of experiments to conclude within 15 minutes on a medium-capacity machine.

Execution: This experiment can be reproduced by executing

```
run . -experiment6
```

for Figures 10a, c, and d, and

```
run . -experiment7
```

for Figure 10b.

Results: The output table consists of one row for each combination of user/sum tree numbers (assuming one sum tree per district), and each cell contains the average result over several queries of the same type. The "quantile_all_total" column contains the results for a quantile query on the entire dataset (Fig. 10c) and "quantile_limited_total" (Fig. 10d) for query over a single subtree. On a medium-capacity machine, **C4** can be verified by observing the trends in the table entries (even if the overall execution times are higher).

A.5 Notes on Reusability

The TAP code can be used to perform queries on data tables with the same format as those generated for the experiments (as specified in *tables/table_factory.go*).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Trojan Source: Invisible Vulnerabilities

Nicholas Boucher & Ross Anderson

A Artifact Appendix

A.1 Abstract

We provide a repository with proofs-of-concept implementing Trojan Source attacks in C, C++, C#, JavaScript, Java, Rust, Go, Python, SQL, Bash, Assembly, and Solidity. When viewed and compiled using vulnerable tools, these short proof-of-concept programs will output different values when executed than would be expected from reading the rendered source code.

A.2 Description & Requirements

Our paper describes a manner of encoding source code that can hide malicious logic in a manner that is not rendered to the user. Our artifact is a collection of proofs-of-concept, for which validation will take the form of verifying that the proofs-of-concept visually match the claimed rendering described in the paper and output the same adversarial logic when executed as described in the paper.

A.2.1 Security, privacy, and ethical concerns

The code provided does not take destructive action. While execution of the provided programs will output different values than expected from reading the rendered source code, viewing, compiling, and executing the provided code is designed to have no negative consequences for the reviewer.

A.2.2 How to access

The artifact is provided as a GitHub repository: `https://github.com/nickboucher/trojan-source/tree/e3dc153fcf465f4a84424ea874ff39be29adb1f7`.

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

Validating this artifact will required opening the proofs-of-concept programs any vulnerable language (Table 2 of the paper) in a vulnerable code viewer (Table 3 of the paper). We recommend C viewed with Visual Studio Code (v1.61) and compiled with `clang` (v12.0.*), and these tools are reasonably cross-platform.

Our experiments were repeated across Window 10 build 19043, MacOS Big Sur, and Ubuntu 20.04, although we anticipate that any modern version of Windows, MacOS, or Ubuntu will work.

A.2.5 Benchmarks

None

A.3 Set-up

Clone the artifact repository.

A.3.1 Installation

Install at least one vulnerable compiler and code viewer as listed Section A.2.4.

A.3.2 Basic Test

Validate that C/commenting-out.c is rendered as shown in Figure 4 of the paper when opened in a vulnerable editor such as Visual Studio Code (v1.61)

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Trojan Source attacks produce different outputs when executing compiled programs written with Trojan Source techniques than would be expected from the rendered source code (absent any defenses).

A.4.2 Experiments

(E1): *[10 human-minutes + .01 compute-hours + <1GB disk]*:
How to: Open each of the proofs-of-concept in the C sub directory of the artifact repository using a vulnerable language and compiler as described in Section A.2.4. Confirm that the code is visualized as shown in Figure 4

of the paper. Compile the code, and confirm that the output matches the output claimed in the paper and differs from what would be expected from reading the rendered source code.

Preparation: Complete the software dependency installations described above.

Execution: Compile and execute the proofs-of concept, with e.g. `clang commenting-out.c && ./a.out`.

Results: The output should match Section 4.2 of the paper.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Cheesecloth: Zero-Knowledge Proofs of Real-World Vulnerabilities

Santiago Cuéllar*
Galois, Inc.

Bill Harris
Galois, Inc.

James Parker
Galois, Inc.

Stuart Pernsteiner
Galois, Inc.

Eran Tromer
Columbia University

A Artifact Appendix

A.1 Abstract

This artifact accompanies the paper, *Cheesecloth: Zero-Knowledge Proofs of Real-World Vulnerabilities*. It contains pointers to the software repository for the work described in the paper, instructions for compiling the software and its dependencies, and instructions for running the benchmarks in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There is no risk for evaluators.

Ethical concerns CHEESECLOTH aids in responsible disclosure by producing zero-knowledge proofs of the existence of vulnerabilities while keeping the vulnerabilities and exploits secret. All vulnerabilities used in our evaluation have been previously disclosed publicly, and fixes are widely deployed. Thus, the work presented in the paper does not constitute an unethical disclosure of potentially harmful information. A black hat researcher could use CHEESECLOTH as part of the process to sell a vulnerability, however CHEESECLOTH's involvement is unlikely to change the fact that the vulnerability will still be sold and abused.

A.2.2 How to access

The source code accompanying this paper is available at `https://github.com/GaloisInc/cheesecloth/tree/usenix-2023-artifact`.

A.2.3 Hardware dependencies

All measurements reported in the paper were performed on a 128 core Intel Xeon E7-8867 CPU with 2 TB of RAM, although our implementation uses considerably less memory. 592 GB of disk space is required to store the outputs of all benchmarks. The OpenSSL benchmark takes up most of this disk space. The GRIT and FFmpeg benchmark outputs combined require less than 35 GB.

A.2.4 Software dependencies

Benchmarks were run on Debian 11. LLVM version 9 is a required dependency. The Haskell (`stack`) and Rust (`cargo`) development tools are also required to build CHEESECLOTH. All other Haskell and Rust dependencies are listed in project configuration files and are automatically retrieved using `stack` and `cargo`.

A.2.5 Benchmarks

Experiments require the vulnerable versions of GRIT, FFmpeg, and OpenSSL. All vulnerable versions are provided as git submodules of the CHEESECLOTH repository and point to the appropriate commit.

A.3 Set-up

A.3.1 Installation

LLVM version 9, `stack`, and `cargo` are dependencies that must be installed. The two main components of the CHEESECLOTH compilation chain are `MicroRAM` and `witness-checker`, which both live in git submodules. For convenience, the scripts `./scripts/build_microram` and `./scripts/build_witness_checker` will compile each tool.

A `Dockerfile` is provided for reproducible builds, however Docker may add too much overhead in practice. You can build and run the Docker container with:

```
docker build --platform linux/x86_64 -f
    cheesecloth/Dockerfile -t
    cheesecloth-image .
docker run --platform linux/x86_64 -it
    cheesecloth-image:latest /bin/bash
```

A.3.2 Basic Test

Correctness tests for `MicroRAM` and `witness-checker` can be run with `stack test` in the `MicroRAM` directory and `cargo test` in the `witness-checker` directory.

*Authors listed alphabetically.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): CHEESECLOTH produces Zero-Knowledge proofs of vulnerabilities for GRIT, FFmpeg, and OpenSSL with multiplication gate counts as described in Table 1 of the paper.

(C2): Turning off the sparsity and public-pc segment optimizations increases circuit size in terms of multiplication gate counts as described in Table 2 of the paper.

A.4.2 Experiments

For each of the experiments, running GRIT and FFmpeg takes hours, while OpenSSL takes days. You may wish to skip the OpenSSL runs.

(E1): Generate the ZK proofs of vulnerabilities for GRIT, FFmpeg, and OpenSSL.
How to: Once all the dependencies are installed, run the following scripts to generate the ZK proofs:

```
./scripts/run_grit && mv out/grit
    out/grit_baseline
./scripts/run_ffmpeg && mv out/
    ffmpeg out/ffmpeg_baseline
./scripts/run_openssl && mv out/
    openssl out/openssl_baseline
```

Results: All of the scripts should finish successfully and the corresponding output directories will contain a `witness-checker.log` file. This file contains the multiplication gate counts at the `"mul_gates"` key (under `"gate_stats"`) which correspond to Table 1.

(E1.5): Run the ZK proofs from E1 through the Diet Mac'n'Cheese ZK proof backend.
How to: While our contribution is agnostic to the ZK backend, you can run the Diet Mac'n'Cheese backend with the following scripts (you will need to update `swanky_dir` in the script to point to the location of Diet Mac'n'Cheese on your file system):

```
./scripts/dmc.sh verifier out/
    grit_baseline &
./scripts/dmc.sh prover out/
    grit_baseline
```

You may want to invoke the prover and verifier in separate terminals.
Results: The scripts will report the backend protocol time for the given ZK proof (replace `prover` with `prover-count` to report protocol communication).

(E2): Regenerate the ZK circuits with the sparsity and public-pc segment optimizations turned off.

How to: Run the following scripts to regenerate the ZK proofs:

```
./scripts/run_grit_no_sparsity && mv
    out/grit out/grit_no_sparsity
./scripts/run_grit_no_publicpc && mv
    out/grit out/grit_no_publicpc
./scripts/run_ffmpeg_no_sparsity &&
   mv out/ffmpeg out/
   ffmpeg_no_sparsity
./scripts/run_ffmpeg_no_publicpc &&
   mv out/ffmpeg out/
   ffmpeg_no_publicpc
./scripts/run_openssl_no_sparsity &&
    mv out/openssl out/
   openssl_no_sparsity
./scripts/run_openssl_no_publicpc &&
    mv out/openssl out/
   openssl_no_publicpc
```

Results: All of the output directories will contain a `witness-checker.log` file again with the multiplication gate counts which correspond to Table 2.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

VulChecker: Graph-based Vulnerability Localization in Source Code

Yisroel Mirsky[1], George Macon[2], Michael Brown[3], Carter Yagemann[4]
Matthew Pruett[3], Evan Downing[3], Sukarno Mertoguno[3] and Wenke Lee[3]
[1]Ben-Gurion University of the Negev
[2]Georgia Tech Research Institute
[3]Georgia Institute of Technology
[4]Ohio State University

A Artifact Appendix

A.1 Abstract

In this document we describe the artifact for our vulnerability detection tool *VulChecker*. The artifact consists of source code, datasets and pre-trained models for reproducing the results from the USENIX'23 paper. This document we only provide the steps requires to demonstrate that the tool is available and functional.

A.2 Description & Requirements

To reproduce the results from the paper you will need to use the source code and assets available on GitHub. There you will find detailed instructions on how to install the tool or acquire the VM which has the tool already installed. You will also find a detailed guide on how to use the the entire pipeline in your own projects.

You can also find our implementation of the baseline `VulDeeLocator` here:
https://github.com/evandowning/VulDeeLocator

A.2.1 Security, privacy, and ethical concerns

There are no risks in executing our tool since it is a vulnerability detector. However, the provided datasets are deviates of projects collected from GitHub. Therefore, users should consider that the projects may contain security risks if executed and users should note the respective software licenses.

A.2.2 How to access

To gain access to the source code an assets, please check out our GitHub repository at https://github.com/ymirsky/VulChecker

`VulChecker` uses a number of components that must be installed in order for it to operate. Here is a list of components of Vulchecker which we maintain in seperate repositories:

- VulChecker: the core library for processing data and training models. All operations with this library are through a command line tool called hector. https://github.com/ymirsky/VulChecker.git
- LLAP: a plugin to LLVM for extracting ePDGs from cmake C/C++ projects. https://github.com/michaelbrownuc/llap
- Structure2Vec: our pyTorch implimentation of the graph-based neural network by Dai et al. https://github.com/gtri/structure2vec
- vulchecker-misc: a collection of helpful (optional) scripts, such as automatic labeling Juliet samples. https://github.com/michaelbrownuc/vulchecker-misc

Information on where the VM and datasets are hosted can be found the main `VulChecker` repo.

A.2.3 Hardware dependencies

To execute the tool on the VM, you will need a host system with at least 16GB RAM. If you intend to preprocess the raw datasets yourself, you will need significantly more RAM (128GB). You can use the VM to train a model on a CPU, but it is highly recommended to use a system with a cuda GPU (we used an NVIDIA TITAN RTX with 24GB RAM). The VM does not come with cuda installed. Therefore, if you want GPU acceleration, you will either need to (1) install the vGPU and cuda libraries on the VM or (2) make a clean installation on a cuda enabled system with the instruction from the repo.

A.2.4 Software dependencies

To make a clean install of the detection tool (instead of using the VM), you will need a Linux system with Ubuntu Ubuntu 20.04 and python 3.8.10.

A.2.5 Benchmarks

Although we provide source code to the baselines used in the paper, we do not provide end-to-end instructions for running them on our datasets at this time. Please follow our VulDeeLocator repo for updates.

A.3 Set-up

For a quickstart, download the VM (link in the README.md of from the `VulChecker` repo). Configure the VM to have at leag 16GB RAM (prefferably more).

A.3.1 Installation

For a clean installation, follow the clean-install steps provided in the repo's README.md. Note that a clean installation can save several hours since LLVM must be compiled.

A.3.2 Basic Test

Once you have booted the VM, you will find three demo scripts on the desktop. These scripts are also available in `demos/` in the repo if you performed a clean-install. These scripts which demonstrate how the tool can be used on a single project provided (Avian) for CWE-121:

1. Convert a C or C++ project into a set of ePDGs
2. Augment datasets (requires large RAM)
3. Train a model
4. Make predictions with model

You can run them to verify that the tool has been installed successfully.

A.4 Evaluation workflow

To use the tool, a separate model and dataset must be prepared for each CWE (190, 121, 122, 415, 416).

There are three ways to reproduce the results from the paper, depending on how far back into the pipeline you want to go: (1) start from raw source code projects, (2) start from preprocessed datasets, (3) start from preprocessed datasets and pretrained models.

The first approaches can take days to perform. Therefore we recommend following the third approach.

(1) For working with the Raw source code files: Follow the instructions in the repo using the diagram provided there. To get the data, follow the link provided in the repo's README. The training data is the 'clean-wild' projects augmented with the Juliet projects. The test data is the 'wild-labeled'. The parameters used to train our models can be found in the `models/trained_on_aug/` directories next to each model.

(2) For working with the preprocessed files, the training data for CWE<id> can be found here: `/CWE<id>/proc_graphs/wild_augmented-labels`.
Use the file that has the format
CWE<id>_*_clean_<N>_<P>.json.gz
which is the dataset after removing a ratio of <N> negative and <P> positive manifestation points.

The test data (projects with CVEs) can be found here: `/CWE<id>/proc_graphs/*/combined`. Use the file that has the format
CWE<id>_*_clean_<N>_<P>.json.gz

(3) To start with preprocessed datasets and pretrained models, follow the instruction in (2) and model files stored in `models/`

A.4.1 Major Claims

(C1): The provided `VulChecker` tool is functional and can be used to detect vulnerabilities/bugs in source code.

A.4.2 Experiments

(E1): *Tool Execution* [30 human-minutes + 3 compute-hour + 200GB disk + 128GB RAM]:
How to: Download the provided VM and allocate the required RAM to the machine. Run the three demo scripts on the VM's desktop. The demos will operate on a single project (Avian) for CWE-121.
Results: The final script should output a csv listing the likelihoods for each potential manifestation point is an actual bug/vulnerability for CWE-121.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Automated Security Analysis of Exposure Notification Systems

Kevin Morio[1] Ilkan Esiyok[1] Dennis Jackson[2] Robert Künnemann[1]

[1]CISPA Helmholtz Center for Information Security
[2]Mozilla

A Artifact Appendix

A.1 Abstract

The artifact consists of the Tamarin model files with the corresponding oracles for the three exposure notification systems ROBERT, DP3T, and the CWA presented in the paper. The artifact is available as a ZIP archive as well as packaged in a Docker container featuring a recent version of Tamarin.

Our analysis results presented in Section 6 of the paper can be revalidated by verifying the lemmas of each model with Tamarin.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact can be directly downloaded from https://doi.org/10.6084/m9.figshare.21304305.

The provided Docker container with the artifact and Tamarin can be obtained by executing

```
docker pull kevinmorio/usenix23-ens
```

A.2.3 Hardware dependencies

The evaluation of the artifact requires about 43.15 GB of memory (peak) and ideally 12 cores.

A.2.4 Software dependencies

Tamarin with its dependencies Maude and Graphviz can be directly installed on the system. Alternatively, for using the provided Docker container, a working installation of Docker is required.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Tamarin and its dependencies can be installed following the instruction from the Tamarin manual: https://tamarin-prover.github.io/manual/book/002_installation.html. The artifact can be downloaded from https://doi.org/10.6084/m9.figshare.21304305.

Installation instruction for Docker are available in the Docker documentation: https://docs.docker.com/engine/install. The Docker container can be obtained by executing

```
docker pull kevinmorio/usenix23-ens
```

A.3.2 Basic Test

To check that Tamarin works correctly, execute

```
tamarin-prover test
```

for a local installation of Tamarin or

```
docker run kevinmorio/usenix23-ens \
  tamarin-prover test
```

when using the Docker container.

The output should be

```
Self-testing the tamarin-prover installation.

*** Testing the availability of the required tools ***
maude tool: 'maude'
 checking version: 2.7.1. OK.
 checking installation: OK.

GraphViz tool: 'dot'
 checking version: dot - graphviz version 7.0.1 (20221109.1506). OK.
 checking PNG support: OK.

*** Testing the unification infrastructure ***
Cases: 55  Tried: 55  Errors: 0  Failures: 0

*** TEST SUMMARY ***
All tests successful.
The tamarin-prover should work as intended.

           :-) happy proving (-:
```

where the reported versions for Maude and Graphviz can differ depending on the system.

A.4 Evaluation workflow

A.4.1 Major Claims

(Claim 1): A1–A4 categorise all attacks against upload authorisation (Def. 3) for ROBERT as described in Section 6.2. X1–X7 categorise all attacks against soundness (Def. 1) for ROBERT as described in Section 6.3. This is verified by experiment (E1).

(Claim 2): B1–B3 categorise all attacks against upload authorisation (Def. 2) for DP3T as described in Section 6.2. Y1–Y7 categorise all attacks against soundness (Def. 1) for DP3T as described in Section 6.3. This is verified by experiment (E2).

(Claim 3): C1–C2 categorise all attacks against upload authorisation (Def. 2) for the CWA as described in Section 6.2. Z1–Z4 categorise all attacks against soundness (Def. 1) for the CWA as described in Section 6.3. This is verified by experiment (E3).

A.4.2 Experiments

(E1): *[Verification of ROBERT] [10 human-minutes + 14 compute-hours]: Verify the lemmas in `robert.spthy` with Tamarin. All lemmas should verify.*

Preparation: Install Tamarin directly and download + extract the artifact or obtain the Docker container as described above.

Execution: For the local Tamarin install, enter the directory where the artifact has been extracted to and execute

```
tamarin-prover --prove robert.spthy
```

Alternatively, for Docker execute

```
docker run kevinmorio/usenix23-ens \
  tamarin-prover --prove robert.spthy
```

The number of cores and the amount of memory used by Tamarin can be configured by adding `+RTS -N<num-cores> -M<gb-mem>g -RTS` directly after `tamarin-prover` in the commands above. The reported compute-hours have been obtained with `-N12` and no memory limit on an Intel(R) Xeon(R) CPU E5-4650L workstation.

Results: When Tamarin terminates, it reports a summary of summaries listing all lemmas and their verification result line-by-line, e.g.,

```
soundness (all-traces): verified (.. steps)
```

The verification result of each lemma should be `verified`. Moreover, the summary of summaries reported should be the same as the summary of summaries at the end of `robert.spthy`

(E2): *[Verification of DP3T] [10 human-minutes + 2 compute-hours]: Verify the lemmas in `dp3t.spthy` with Tamarin. All lemmas should verify.*

This experiment is the same as the one above for ROBERT except for using `dp3t.spthy` instead of `robert.spthy`.

(E3): *[Verification of the CWA] [10 human-minutes + 1 compute-hour]: Verify the lemmas in `cwa.spthy` with Tamarin. All lemmas should verify.*

This experiment is the same as the one above for ROBERT except for using `cwa.spthy` instead of `robert.spthy`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Formal Analysis of SPDM: Security Protocol and Data Model version 1.2

Cas Cremers
CISPA Helmholtz Center for Information Security

Alexander Dax
CISPA Helmholtz Center for Information Security

Aurora Naska
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This document contains the description of the formal models and proofs of the three modes of Security Protocol and Data Model (SPDM) protocol version 1.2.1. We provide four models that capture the main device attestation mechanism, and the three modes of key exchange with (i) preshared symmetric keys, (ii) preshared public keys, and (iii) public key pair, certificates over the public key, and a root of trust. During our analysis we prove the main security guarantees of each of the models, such as Responder Authentication, Measurements Authentication, Handshake Secrecy, etc. Our proofs and models are formalized using the Tamarin Prover's input language.

We provide the artifacts in a public Github repository for inspection and reproduction with instructions on how to replicate each of the proofs. In addition, the repository includes a Python programm to obtain all our results automatically.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

All our files are public and can be inspected and reused by accessing the following GitHub repository `https://github.com/FormalAnalysisOf/SPDM/tree/V1`.

A.2.3 Hardware dependencies

The hardware dependencies are inherited from the Tamarin Prover, although the manual of the latter does not mention any hardware dependencies. To the best of our knowledge, any modern notebook should be sufficient to run Tamarin.

A.2.4 Software dependencies

In the following we list all software dependencies:

1. Tamarin Prover[1] which depends on `haskell-stack`, `graphviz`, and `maude`. Note that Tamarin does not run on Windows systems. To run Tamarin on Windows refer to WSL[2].
2. Python3 - install `pip`, and use it to install `tabulate` and `matplotlib`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Clone the repository using

```
$ git clone
  ↪ https://github.com/FormalAnalysisOf/SPDM.git
```

To install the Python dependencies, please install *Python3, pip, tabulate*, and *matplotlib*. For instance, on an Ubuntu system you can install them using

```
$ apt install python3
$ apt install pip3
$ pip3 install tabulate matplotlib
```

Afterwards install the development branch of Tamarin.

Installing Tamarin Some package managers let you install Tamarin directly. We suggest compiling it from scratch (develop branch) using the manual `https://tamarin-prover.github.io/manual/book/002_installation.html`. for instructions. The exact commit we used to obtain our proofs in the develop branch is:

[1] `https://tamarin-prover.github.io/manual/book/002_installation.html`

[2] `https://learn.microsoft.com/en-us/windows/wsl/install`

```
7c980321158ebf7c7c03c53cee93248507584065
```

Our models can also be executed using the latest development version of Tamarin, however, that might affect the execution's runtime.

A.3.2 Basic Test

Make sure that the *tamarin-prover* executable is in the *$PATH*. To test if you correctly installed the Tamarin Prover, execute

```
$ tamarin-prover test
```

You should see a message containing
- a check for `maude`,
- a check for `Grapviz`, and
- a test for the unification structure (0 errors and 0 failures).

In the end you should see the following:

```
All tests successful.
The tamarin-prover should work as intended.
        :-) happy proving (-:
```

A.4 Evaluation workflow

A.4.1 Major Claims

Our artifact contains formal models of the SPDM specification and the means to execute them. The models include at the end of the file the main security properties of our analysis, and sanity checks expressed by the so-called *lemmas*, which are a formal representation using the Tamarin Porver's input language. The user can automatically reproduce the proofs for all lemmas by either using the python program or the instruction sets in our artifact. We state the security properties investigated in our analysis in the following:

(C1) Device attestation: For the part of SPDM that aims to provide device attestation, we prove **Responder Authentication** and **Measurement Authentication**. Definitions of those properties can be found in Section 4 of our paper. Our claims are proven with experiment **(E1)**.

(C2) Certificate KEX: For the key exchange based on certificates and public key cryptography, we prove **Responder Authentication**, **Mutual Authentication**, and **Handshake Secrecy**. Definitions of those properties can be found in Section 4 of our paper. Our claims are proven with experiment **(E2)**.

(C3) Pre-Shared KEX: For the key exchange based on pre-shared public keys, we prove **Mutual Authentication** and **Handshake Secrecy**. Further, for a restricted model (see section 4.5 in the paper), we also prove **Forward Secrecy**. Definitions of those properties can be found in Section 4 of our paper. Our claims are proven with experiment **(E3)**.

(C4) PSK Exchange: For the key exchange based on pre-shared symmetric keys, we prove **Mutual Authentication** and **Handshake Secrecy**. Definitions of those properties can be found in Section 4 of our paper. Our claims are proven with experiment **(E4)**.

A summary of our proof results and runtime can be found in Table 1 of our paper.

A.4.2 Experiments

In the following we list all the different models and explain how to execute them individually. Afterwards we introduce the python program to execute all models in experiments **(E1)-(E4)** at once.

We ran our models initially on an Intel(R) Xeon(R) CPU E5-4650L 2.60GHz machine with 756GB of RAM using 4 threads and the estimated runtime is based on this machine.

First navigate in the cloned repository's model folder

```
$ cd SPDM/TamarinModels
```

and then make the oracle file executable

```
$ chmod +x oracle
```

(E1): *[1 human-minute + $\sim$ 4 compute-minutes]:*
Device attestation is modelled in the file: *device_attestation.spthy*. The lemma *ResponderAuth* models Responder Authentication and the lemma *MeasurementAuth* models Measurement Authentication. The model file further contains several sanity lemma and helper lemmas to prove the above property. With the following instruction all of them will be executed.

Preparation: After cloning the repository and installing the software dependencies, navigate into the *TamarinModels* folder within the cloned repository.
Execution: Open the terminal in this folder and execute the following command:

```
$ python3 tamarin_wrapper.py
↪ device_attestation.spthy -p
↪ "auth,Sanity" -c 4 -t 1800
```

Results: The results are printed into the terminal, and a *.csv* file is also stored within the results folder.

(E2): *[1 human-minute + $\sim$ 52 compute-minutes]:*
The certificate based key exchange is modelled in the file: *key_exchange.spthy*. The lemma *resp_authentication_at_finish* models Responder Authentication and the lemma *mutual_authentication* models Mutual Authentication. Handshake Secrecy is modelled via 2 lemma: *secret_major_init_side* and *secret_major_resp_side*. The model file further contains several sanity lemma and certain helper lemma to prove

the above property. With the following instruction all of them will be executed.

Preparation: After cloning the repository and installing the software dependencies, navigate into the Tamarin-Models folder within the cloned repository.
Execution: Open the terminal in this folder and execute the following command:

```
$ python3 tamarin_wrapper.py
↪  key_exchange.spthy -p "Sanity"
↪ -c 4 -t 1800
```

Results: The results are printed into the terminal, and a *.csv* file is also stored within the results folder.

(E3): *[1 human-minute + ∼ 29 compute-minutes]:*
The preshared public key based key exchange is modelled in two file: *preshared_pk.spthy* and *preshared_pk_copy.spthy*. The lemma *mutual_authentication* models Mutual Authentication and is executed in *preshared_pk.spthy*. The other file executes Handshake Secrecy is modelled via 2 lemma: *secret_major_init_side* and *secret_major_resp_side*. The model file further contains several sanity lemma and certain helper lemma to prove the above property. With the following instruction all of them will be executed.

Preparation: After cloning the repository and installing the software dependencies, navigate into the Tamarin-Models folder within the cloned repository.
Execution: Open the terminal in this folder and execute the following commands:

```
$ python3 tamarin_wrapper.py
↪  preshared_pk.spthy -p
↪  "auth,Sanity" -c 4 -t 1800
```

```
$ python3 tamarin_wrapper.py
↪  preshared_pk_copy.spthy -p "fs"
↪ -c 4 -t 1800
```

Results: The results are printed into the terminal, and *.csv* files are also stored within the results folder.

(E4): *[1 human-minute + ∼ 3 compute-minutes]:*
The preshared public key based key exchange is modelled in two files: *preshared_psk.spthy* and *refl.spthy*. The lemma *mutual_authentication* models Mutual Authentication and is executed in *preshared_psk.spthy*. The other file executes Handshake Secrecy is modelled via 2 lemma: *secret_major_init_side* and *secret_major_resp_side*. The model file further contains several sanity lemma and certain helper lemma to prove the above property. With the following instruction all of them will be executed.

Preparation: After cloning the repository and installing the software dependencies, navigate into the Tamarin-Models folder within the cloned repository.
Execution: Open the terminal in this folder and execute the following commands:

```
$ python3 tamarin_wrapper.py
↪  attack_refl_preshared_psk.spthy
↪ -c 4 -t 1800
```

```
$ python3 tamarin_wrapper.py
↪  preshared_psk.spthy -p "Sanity"
↪ -c 4 -t 1800
```

Results: The results are printed into the terminal, and *.csv* files are also stored within the results folder.

Easier alternative
Instead of running all models independently, it is also possible to run all of them at once. On our computing device this took ∼ 1,5h.
Preparation: After cloning the repository and installing the software dependencies, navigate into the TamarinModels folder within the cloned repository.
Execution: open the terminal in this folder and execute the following command:

```
$ python3 tamarin_wrapper.py -f
↪  case_studies.tamjson
```

Results: While the results will be printed into the terminal, *.csv* files are also stored within the results folder.

Timeouts
Depending on the computing device it can happen that single lemmas do not terminate in the given timeout. You can either change the timeout at the `-t` parameter or if one is running all at once, change the `"timeout"` field within the `case_studies.tamjson` file.

A.5 Notes on Reusability

We conducted a first in-depth formal analysis of the three models of SPDM, and proved their main security properties. Ideally, we would verify all security properties on the complete model, however this seems beyond reach of the current state-of-the-art symbolic analysis tools. However, we modelled the protocol in a modular fashion s.t. models can be reused and adapted as the specification and standard evolve. We hope that our models can serve as a starting point for a unified model and encourage future work on the SPDM protocol.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: One Size Does Not Fit All: Uncovering and Exploiting Cross Platform Discrepant APIs in WeChat

Chao Wang
The Ohio State University

Yue Zhang
The Ohio State University

Zhiqiang Lin
The Ohio State University

A Artifact Appendix

A.1 Abstract

APIDiff is an automatic tool that generates test cases for each API and identifies execution discrepancies. APIDiff consists of three key components. The Test Case Generator is responsible for creating test cases for each API, initializing the parameters correctly, resolving dependencies between APIs, and mutating parameter values to achieve high coverage. The Code Executor executes the test cases on different platforms (Windows, Android, and iOS) to generate corresponding outputs and detect discrepancies that may pose security concerns. The Discrepancies Analyzer employs differential analysis and predefined policies to examine error codes and return values of tested APIs, enabling the identification of discrepant APIs.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

To ensure compliance with community practices, it is imperative to strictly adhere to the following requirements:

- **Thorough Analysis and Controlled Environment**: All analysis and execution of attacks must be conducted within a controlled environment, utilizing personal accounts and machines.
- **Confidentiality of Attack Code and Malware**: Any developed attack code and malware must be kept private and confidential to prevent any potential harm to users, miniapp developers, and platform providers.
- **Responsible Reporting**: Any discoveries made during the analysis should be promptly and responsibly reported to Tencent, following their specified guidelines and procedures.

A.2.2 How to access

Please find our project online: https://github.com/OSUSecLab/APIDiff/tree/f65137b3f8dc037021773134db40b1d384d542b7

A.2.3 Hardware dependencies

To run the tools, you require a specific environment with the following software and hardware requirements. Firstly, you need an operating system (OS) such as Linux, macOS, or Windows. Additionally, you need Node.js, which is a JavaScript runtime environment. You also need WeChat DevTools, which is a development tool specifically for creating and debugging MiniApps. In terms of hardware, you will need devices for conducting experiments. This includes Android devices, iOS devices, and Windows devices.

A.2.4 Software dependencies

Please ensure that you have the latest version of WeChat installed on your devices. In order to create and debug a MiniApp for API testing, it is necessary to have a WeChat account and install the IDE specified by Tencent.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To get started, please follow the steps below:

1. Clone the project: Use the Git command or a Git client to clone the project repository. For example, you can use the following command in your terminal:

```
git clone https://github.com/OSUSecLab/APIDiff/
```

2. Install additional dependencies:

```
# make sure node.js and npm is installed
node --version

# install for apitest-gen
cd apitest-gen && yarn && cd ..

# install for client
cd client && yarn && cd ..

# install for server
cd server && yarn && cd ..
```

A.3.2 Basic Test

You can test each component by simply executing the following commands:

- **apitest-gen**: `npx ts-node apitest-gen/src/apigen/main.ts`
- **client**: `npx ts-node client/src/index.ts`
- **server**: `npx ts-node server/agent/index.ts`

A.4 Evaluation workflow

Generate the test cases Make sure you have documents available for the test case generation. Note that the input document should be an array of API (Array in typescript) in JSON format. You can refer to Pre-processing for more information.

You can specify the input document file via `apitest-gen/src/apigen/config.ts` or via the environment variable `INPUT_DOC`.

You can specify the output directory via the same config file or via the environment variable `OUTPUT_DOC`.

Then, execute the `apitest-gen/src/apigen/main.ts` to generate test cases. (i.e., `ts-nodeapitest-gen/src/apigen/main.ts`)

Find the debug URL The debug URL is embedded in WeChat DevTools. You need a WeChat account to create and test a MiniApp, and via initiating a remote debug session you can obtain such a debug URL.

1. **Tweak the WeChat DevTools** Locate your WeChat DevTools installation directory, look for package.json inside package.nw (in macOS the location is `/Applications/wechatwebdevtools.app/Contents/Resources/package.nw/package.json`). Then, find the `-disable-devtools` flag, remove it and save the file.

2. **Initiate a remote debug session.** Choose a device target to start a remote debug session. Make sure you are using remote-debug 2.0 and enable LAN mode for best network latency. Make sure the remote debugger window is popped.

3. **Find the debug URL** Make sure the current focused window is the remote debugger, press F12 to open the chrome devtools. Switch to the Elements tab and find the webview tag (the selector is `body > div:nth-child(1) > div > div > div.debugger > webview`). Now you can find the debug string is inside the src property starting with `ws=`.

   An example debug string is `ws=127.0.0.1:40204`. You can now transform this string into the debug URL: `ws://127.0.0.1:40204`.

Run the server Make sure you have all dependencies installed. Then, you can start a server directly by the following command:

```
1  cd server
2  REMOTE_DEBUG_WS=<your debug URL> ts-node agent/index.ts
```

Make sure the *[evaluator init] global* message is prompted after running the server. If you did not see this message, the debug URL might be invalid due to timeout. You need to redo the previous steps. Note that the debug URL won't change as long as the WeChat DevTools is not closed or restarted.

You can also interact with the debugger via specifying *ENABLE_NODE_REPL* environment variable. The Node.JS REPL contains global objects for debugging. Please refer to *server/agent/index.ts* for more information.

The server will open a port for the incoming requests from the client. You can specify the port in server/agent/listener/config.ts

Run the client You can now run the client to start testing. You need to change the config file defined in `client/src/config.ts` based on your previous configuration. After that, you can run the client by the following command:

```
1  cd client && ts-node src/index.ts
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems

Zixuan Wang[⋆] Mohammadkazem Taram[♯⋆] Daniel Moghimi[†⋆]
Steven Swanson[⋆] Dean Tullsen[⋆] Jishen Zhao[⋆]

[⋆]UC San Diego [♯]Purdue University [†]UT Austin

A Artifact Appendix

A.1 Abstract

This artifact describes NVLeak, a collection of microarchitecture-level reverse-engineering tools and covert/side channel attack proof-of-concept, which exploit the microarchitecture design of Intel Optane DIMM. NVLeak also comes with a set of scripts to set up system environments, run experiments, collect results, and generate plots.

A.2 Description & Requirements

NVLeak artifacts are available online as a GitHub repo which contains reverse engineering, covert/side channel code, data parsing scripts, and documentation to use these tools. To reproduce the major claims in the main paper, we recommend using a server machine with Intel Optane DIMM, similar to *Server A* or *Server B* described in the main paper.

A.2.1 Security, privacy, and ethical concerns

This artifact does not exploit any security breaches on evaluators' machines. The reverse-engineering and covert/side channel code is run on the server machine with Optane DIMMs, and it is not destructive to evaluators' environments. The only code running on evaluators' machines is data parsing and plot generation scripts, which can run in the Docker images from NVLeak.

A.2.2 How to access

NVLeak code is hosted on GitHub[1].

A.2.3 Hardware dependencies

NVLeak exploits the microarchitecture designs of Intel Optane DIMMs and thus requires these DIMMs to reproduce the results presented in the main paper. An example server machine environment with Optane DIMM is shown in Table 1.

[1]https://github.com/TheNetAdmin/NVLeak/tree/588567e6ec30f2df9f260e60385031c94e94c75e

Table 1: NVRAM-equipped server system configuration.

Hardware	Configuration
CPU	Intel Xeon Gold 6230 20 Cores per socket, 2 sockets HyperThreading off
L1 Cache L2 Cache L3 Cache	32 KiB 8-way I-Cache, 32 KiB 8-way D-Cache, private 1 MiB, 16-way, private 27.5 MiB, 11-way, shared
DRAM	6 channels per socket DDR4, 16 GiB, 2666MHz
NVRAM	Intel Optane DIMM, 6 channels per socket 128 GiB, 2666 MHz Firmware: 01.02.00.5355

A.2.4 Software dependencies

NVLeak has a kernel module that compiles with Linux 5.4 or older versions (tested with 5.1 and 4.15). A newer kernel may have breaking changes to the filesystem APIs used by NVLeak and thus may fail the compilation. NVLeak requires `ndctl` (v67+) and `ipmctl` (v02.00.00.3885) to configure the Intel Optane DIMMs, which can be compiled and installed from their source code on GitHub. Additional NVLeak requires `e2fsprogs` (v1.46.4 or newer) to configure the Ext4 filesystem, and `sqlite3` (v3.31.1), `PMDK` library, `wolfSSL` (v4.2.0) for side channel attacks. NVLeak GitHub repo has more detailed documentation on installing and using these tools.

A.2.5 Benchmarks

NVLeak requires *NPPES NPI* dataset for SQLite side channels and provides a script to download this dataset.

A.3 Set-up

NVLeak provides a set of scripts to set up the server machine for experiments. Due to the space limit, we provide minimal instructions in this artifact appendix and describe the complete setup process in the NVLeak GitHub repo.

```
$ sudo -i su
# cd NVLeak/nvleak
# bash scripts/machine/machine.sh setup
# reboot
# bash scripts/machine/optane.sh reset
# bash scripts/machine/optane.sh setup \
      appdirect ni
# bash scripts/machine/optane.sh ndctl
```

Figure 1: Set up the Linux boot arguments and Optane DIMM operation modes.

```
$ ndctl list -u
[
    {
        "dev":"namespace1.0",
        "mode":"fsdax",
        "map":"dev",
        "size":"124.03 GiB (133.18 GB)",
        "uuid":"***",
        "sector_size":512,
        "align":2097152,
        "blockdev":"pmem1"
    },
    {
        "dev":"namespace0.0",
        "mode":"fsdax",
        "map":"mem",
        "size":"32.00 GiB (34.36 GB)",
        "sector_size":512,
        "blockdev":"pmem0"
    }
]
```

Figure 2: Optane DIMMs are successfully configured into the non-interleaved mode (the `pmem1` device is around 128 GiB, a single DIMM's size), and the kernel boot arugment `memmap` successfully creates an emulated PMEM device `pmem0` using DRAM.

A.3.1 Installation

NVLeak provides scripts to set up the system, including Linux boot commands and Optane DIMM operation modes, as listed in Figure 1.

Each NVLeak experiment requires a different set of tools and Optane DIMM configurations. In general, each setup involves three steps: (1) Install required tools, e.g., `sqlite3`; (2) Configure Optane DIMMs and mount them as Linux devices; (3) Compile the source code in NVLeak. Please refer to NVLeak GitHub's documentation for more details.

A.3.2 Basic Test

To check if the setup takes effect, run `ndctl` and check if a non-interleaved PMEM device and an emulated PMEM device are created, as shown in Figure 2.

A.4 Evaluation workflow

Table 2: Major claims and corresponding results.

Figure	Type	Claims
2	Reverse Engineering	L1/L2 NVCache sizes, their block sizes, and WPQ size
4		L1/L2 NVCache set structures
5		Wear-leveling policy
6		Wear-leveling's trigger condition
7		Robustness of wear-leveling data migration
17		Detailed pointer chasing results on Server A
18		Reverse engineering results on Server B
9b-c	Covert Channel	Cross virtual machine covert channel performance and signal
10		Filesystem inode-based covert channel
12	Side Channel	Access patterns of SQLite executing different SQL code
13		Access patterns of SQLite executing ranged queries
14		Access patterns of PMDK key-value store
15		Detected function calls from wolfSSL library
16	Mitigation	Effectiveness and performance of the PMDK-based mitigation

A.4.1 Major Claims

As shown in Table 2, we have made the following four major claims in our main paper:

(C1): NVLeak is able to reverse engineer the Optane DIMM's microarchitecture designs, including WPQ size, NVCache set structures, and wear-leveling mechanisms.

(C2): NVLeak can establish covert channels based on the recovered off-chip microarchitecture to break virtualization and file system isolation.

(C3): NVLeak can establish side channels to leak sensitive information from applications that use NVRAM as storage or memory.

(C4): NVLeak can mitigate the recovered vulnerability by patching the PMDK library's memory allocator.

A.4.2 Experiments

The major NVLeak experiments can be categorized into four types, as listed below. NVLeak GitHub repo provides complete documentation to set up hardware/software environments and reproduce results. The GitHub repo also contains scripts to collect and parse data, generate plots, and compile a LaTeX PDF with all plots organized as in the main paper.

(E1): Reverse Engineering *[1 human-hour + 6 compute-hours + 18 GiB disk]:*

Steps: Configure the Optane DIMM into non-interleaved mode, then compile and insert the NVLeak kernel module, and finally run the scripts to execute all experiments. NVLeak also provides Slack integration

to send experiment progress to the Slack channel configured by the user.
Results: Reproduce Figure 2-7 and Figure 17-18.
Docs: See *docs/reproduce/ReverseEngineering.md* in the NVLeak GitHub repo for more details.

(E2): Covert Channel *[1 human-hour + 16 compute-hours + 32 GiB disk]:*
Steps: Configure the Optane DIMM into non-interleaved mode, and create two separate Linux PMEM devices for the sender and receiver. Then compile the user space proof-of-concept, including QEMU and KVM-unit-tests. And finally, execute NVLeak's scripts to run experiments.
Results: Reproduce Figure 9-10.
Docs: See *docs/reproduce/CovertChannel.md* in the NVLeak GitHub repo for more details.

(E3): Side Channel *[2 human-hours + 1 compute-hours + 1 GiB disk]:*
Steps: Create two separate Linux PMEM devices for the attacker and victim. Then download the NPPES dataset and initialize the SQLite database. And finally, run NVLeak scripts to start experiments.
Results: Reproduce Figure 12-15.
Docs: See *docs/reproduce/SideChannel.md* in the NVLeak GitHub repo for more details.

(E4): Mitigation *[2 human-hour + 1 compute-hours + 1 GiB disk]:*
Steps: Download and compile the PMDK library, then execute NVLeak scripts to evaluate the effectiveness and performance of the mitigations described in the main paper.
Results: Reproduce Figure 16.
Docs: See *docs/reproduce/SideChannel.md* in the NVLeak GitHub repo for more details.

A.5 Notes on Reusability

NVLeak is able to exploit Optane DIMM's microarchitecture, and the user can establish attacks based on these hardware designs. But NVLeak is not limited to Optane DIMM as NVLeak is not bound to any Optane-specific hardware or software. One example is that NVLeak can run on DRAM, as shown in our main paper, Figure 10b.

We envision that NVLeak can be used to exploit future memory devices' microarchitecture designs, such as new memory products based on the Compute Express Link (CXL) technology. In fact, we have repurposed NVLeak to reveal PCIe performance characteristics (not shown in this paper) by attaching an FPGA to PCIe and using MMIO to map the FPGA memory for NVLeak to access. We hope NVLeak can facilitate future memory security research for not just NVRAM but even broader memory technologies.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software

Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth
University of Lübeck, Lübeck, Germany
{j.wichelmann, a.paetschke, l.wilke, thomas.eisenbarth}@uni-luebeck.de

A Artifact Appendix

A.1 Abstract

CIPHERFIX is a framework for finding and mitigating ciphertext side-channel leakages in software. It combines dynamic binary instrumentation and dynamic taint tracking to pinpoint vulnerable code parts. Then, it hardens the binaries against ciphertext side-channel leakage with the help of static binary instrumentation.

This artifact comprises our source code and usage instructions. We offer prebuilt Docker images which contain all necessary dependencies, library binaries and precompiled examples. The GitHub repository presents detailed instructions on how to build, run and extend CIPHERFIX.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our source code is available at `https://github.com/UzL-ITS/Cipherfix`. The commit used to reproduce the results in the paper is `0d05fcb`. The artifact contains our dynamic analysis (`static-variables`, `structure-analysis` and `taint-tracking`), the static mitigation (`static-instrumentation`) and our evaluation modules (`memwrite-tracer` and `evaluation`).

A.2.3 Hardware dependencies

For running CIPHERFIX, an AMD Zen1/Zen2/Zen3 CPU is highly recommended (we tested on an AMD EPYC 7763 and on an AMD EPYC 3151). Other x86 CPUs may also work, but CIPHERFIX does not have support for all instructions (e.g., AVX-512), so there might be unsupported instructions and therefore potential instabilities.

A.2.4 Software dependencies

We offer precompiled Docker images which contain all necessary dependencies.

For building the whole framework and the examples from scratch without Docker, please refer to the Prerequisites and Compiling sections in the README.

A.2.5 Benchmarks

The example targets for evaluating the performance and security of CIPHERFIX are located in the `examples` directory. The `cipherfix-examples-full` Docker image contains precompiled binaries for all examples.

A.3 Set-up

A.3.1 Installation

As we ship the artifact as a precompiled Docker image, only a Docker installation is required.

Our precompiled image (`ghcr.io/uzl-its/cipherfix-examples-full`) was built on Zen3, which helps reproducibility on other systems. For example, compilers may check for certain CPU features and then emit instructions which are not yet supported by our instrumentation framework.

If you want to rebuild the Docker images, follow these steps:

1. Clone the CIPHERFIX repository.
2. Run `./build-docker-images.sh` to build the Docker images. You may need `sudo`, if your local user is not member of the system's `docker` group.

A.3.2 Basic Test

Pull and run our precompiled Docker image:

```
docker run -it \
  ghcr.io/uzl-its/cipherfix-examples-full
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Dynamic Analysis

(Section 4.2.1) The static variable detection tool writes information about static variables in the program into a `static-vars.out` file.

(Sections 4.1, 4.2) The `static-vars.out` file is subsequently read by the taint tracking, which adds information about detected heap allocations and stack frames. The taint analysis then tracks which memory location contain secrets, and marks instructions that accessing them. All taint analysis results are written into a `taint.out` file.

(Section 4.3) To aid static instrumentation, a structure analysis tool collects basic blocks and register/flag liveness information. The results are written into a `structure.out` file.

The aforementioned files are acquired by experiment (E1).

(C2): Static Instrumentation

(Section 5) The static instrumentation tool reads the dynamic analysis results and generates hardened binaries, as shown by experiment (E2). The hardened binaries are functionally correct.

(C3): Functional Correctness and Overhead

(Section 6.2) The hardened binaries are functionally correct, but slower than the original ones. This is verified in experiment (E3).

A.4.2 Experiments

In the following, we describe how to verify the claims made in the previous section. For this, we have prepared a number of scripts that run CIPHERFIX for two representative targets.

For the specific steps, we refer to the Running the example targets section in the README file.

(E1): [Dynamic Analysis] [3 human-minutes + 0 compute-hours + 50 MB disk]:

How to: Follow steps 1 and 2 of the Running the example targets section in the README file.

The taint analysis may print a number of warnings and a few errors about unknown instructions. Usually, those can be safely ignored.

Results: The result files end up in `/cipherfix/cipherfix/examples/mbedtls/aes-multiround/` respectively `/cipherfix/cipherfix/examples/wolfssl/eddsa/`.

(E2): [Static Instrumentation] [10 human-minutes + 0 compute-hours + 15 MB disk]:

How to: Follow step 3 of the Running the example targets section in the README file.

Note that the `structure.out` files need to be manually extended with information about heap allocations functions, as described in the README.

Results: The static instrumentation was successful when there are several `.instr` files in the `/cipherfix/cipherfix/examples/mbedtls/aes-multiround/instr-fast-aesrng` and `/cipherfix/cipherfix/examples/wolfssl/eddsa/instr-fast-aesrng` directories.

(E3): [Functional Correctness and Overhead] [3 human minute + 0 compute-hours]:

How to: Follow step 4 of the Running the example targets section in the README file.

Results: The standard output shows the computed ciphertexts and signatures for both the original and the hardened binaries. If the outputs are identical, the functional correctness is given. The `Loop time` specifies the time needed for the cryptographic computations and allows computing the overhead.

A.5 Notes on Reusability

Our proof-of-concept implementation includes modules for the dynamic analysis and the static instrumentation module, as well as evaluation modules. Each of them has a fixed input and output format, which is documented in the `docs` folder in the repository. As long as these formats are followed, each module can be replaced without modification of the other components. For example, it is possible to use another dynamic analysis engine, or an alternative binary rewriting framework with better performance guarantees. See the Replacing Framework Modules section in the README for more information.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: Side-Channel Attacks on Optane Persistent Memory

Sihang Liu
University of Virginia

Suraaj Kanniwadi*
Cornell University

Martin Schwarzl
Graz University of Technology

Andreas Kogler
Graz University of Technology

Daniel Gruss
Graz University of Technology

Samira Khan
University of Virginia

A Artifact Appendix

A.1 Abstract

This paper presents reverse-engineering on Intel Optane persistent memory and demonstrates four attacks. This artifact has included all the source code and scripts for reverse-engineering Optane's internal caches and the following attacks: (1) a local covert channel based on Optane's internal caches, (2) a keystroke side-channel attack on a remote user via an Optane-backed key-value store, `pmemkv`, (3) a fully remote covert channel where the sender stealthily sends a message through texttpmemkv. and (4) Note Board attack where the sender leaves a message on Optane and the receiver can retrieve the message after a long time.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our demonstrated attacks are all based on unmodified systems. The evaluation system does not disable or modify any security-related features in the system. Therefore, there is no risk for evaluators.

A.2.2 How to access

The source code of our artifact is available at https://github.com/Systems-ShiftLab/optane_sec23_ae/

A.2.3 Hardware dependencies

The experiments require a hardware platform as listed below:

- CPU: Intel Cascade Lake.
- Persistent Memory: 1st generation Optane DCPMM. If the server contains multiple Optane modules, they must be running under non-interleaved mode.
- Network: one-hop Ethernet connection between the two Optane servers (for remote covert channels and side-channel attacks).

A.2.4 Software dependencies

The experiments require a software environment as listed below:

- Operating system: Ubuntu 18.04, kernel v5.4.
- Compiler: gcc and g++-7.5.
- Libraries and tools: PMDK v1.9, ndctl v68, ipmctl v02.00.00.3852, and websocket-client.
- Optane mode: The Optane memory must be running in *App Direct* mode using ipmctl.
- File system for Optane: The Optane device must be mounted in DAX mode.

A.2.5 Benchmarks

This artifact includes the following workloads:

- pmemkv server: an Optane-optimized key-value store server program. The covert channels and side-channel attacks are performed via this program.
- weServer: a WebSocket server library. This library enables the typer to send keystroke inputs to the pmemkv-based storage server.
- Keystroke inputs: a dataset from http://personal.ie.cuhk.edu.hk/~ccloy/files/datasets/keystrokes.zip. This input dataset is used to generate keystorkes with realistic inter-keystroke timings.

A.3 Set-up

A.3.1 Installation

To compile our microbenchmarks and run our experiments, we require the following prerequisite software:

*Suraaj Kanniwadi contributed to this work during his internship at the University of Virginia.

- ndctl/daxctl v68
- pmdk 1.9.2
- pmemkv 1.3
- libuv 1.18
- websocket-client (pip package)

A.3.2 Basic Test

A small Reverse Engineering microbenchmark can be used as a self test. One can run `reverse/wear-level/script-ae.sh` as an example. If this fails, this can be due to two reasons:

- **Build failure**: The required libraries are missing.
- **Runtime failure**: It is not the case that the system has persistent memory mounted in dax mode with read/write permissions.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our *Reverse Engineering Framework* runs in user-space and helps unveil interesting internal components of Optane, such as:

- Size, associativity and replacement policies of Optane internal buffers.
- Wear-levelling behaviour exhibited by Optane when performing repeated writes to the same location
- Effect of concurrent writes on read performance

Experiments involved: E1, E2, E3, E4, E5

(C2): *Local Covert Channels* using 3 techniques (RMW, AIT and Read/Write contention) to transmit data covertly.
Experiments involved: E6

(C3): A *Keystroke Attack* which enables us to detect inter-keystroke interval between typists.
Experiments involved: E7

(C4): A *Remote Covert Channel* which allows us to achieve a bandwidth of 10 bps over the network.
Experiments involved: E8

(C5): The **Noteboard Attack**, which is an asynchronous, persistent covert channel.
Experiments involved: E9

A.4.2 Experiments

For ease of artefact evaluation, we provide a single script `runall-ae.sh` which builds our microbenchmarks, runs all experiments, collects data, and finally generates a concise report (`report/report.pdf`) with all the results. `runall-all.sh` is simply a wrapper script which does the following:

- **Run all experiments**: Each experiment directory has a `script-ae.sh` script. `runall-ae.sh` finds all these scripts and runs them all.
- **Copy all results**: Each experiment directory stores results in a `results-ae/` directory. The script copies them to a common `report/` directory.
- **Compiles the report**: The `report/` directory contains latex files which plot all the results.

Information about each experiment is listed below:

(E1): *[Reverse Engineering Heirarchy] [25 compute-minutes + 64GB pmem disk]*
How to: Run `reverse/user_lens/script-ae.sh`
Description: Runs the experiment with depicts the overall structure of Optane.

(E2): *[Reverse Engineering Bitmask Pointer Chasing] [3.5 compute-hours + 64 GB pmem disk]*
How to: Run `reverse/bit_pc/script-ae.sh`
Description: Runs experiments which depict Optane's internal buffer associtivity.

(E3): *[Reverse Engineering Replacement Policy] [10 compute-minutes + 1 GB pmem disk]*
How to: Run `reverse/replacement/script-ae.sh`
Description: Runs experiments which depict Optane buffers' replacement policy.

(E4): *[Reverse Engineering Wearlevelling Policy] [15 compute-seconds + 1 GB pmem disk]*
How to: Run `reverse/wear-level/script-ae.sh`
Description: Runs experiments which depict Optane's wear-levelling behaviour.

(E5): *[Reverse Engineering Read-Write Contention] [1 compute-minute + 1 GB pmem disk]*
How to: Run `reverse/read_write_cont/script-ae.sh`
Description: Runs experiments which depict read-write contention in Optane.

(E6): *[Local Covert Channel] [45 compute-minutes + 2 GB pmem disk]*
How to: Run `local_covert/script-ae.sh`
Description: Runs 2 processes (a sender and receiver) on the same machine, and facilitates covert communication, and measures the achievable bandwidth and accuracy.

(E7): *[Keystroke Side Channel] [1 compute-hour + 1 GB pmem disk]*
How to: Run `keystroke/script-ae.sh`
Description: Runs the keystroke attack as follows:

- Run a kv_server + prober on server A.
- Then, a client connects to server A from server B and sends keystrokes.
- These keystrokes can be detected via the prober on server A.

(E8): *[Remote Covert Channel] [35 compute-seconds + 2 GB pmem disk]*

How to: Run `remote_covert/script-ae.sh`
Description: Runs 2 processes (a sender and receiver) on machines connected across the network, and measures achievable accuracy and bandwidth.

(E9): *[Noteboard Attack] [1 compute-hour + 1 GB pmem disk]*
How to: Run `noteboard/script-ae.sh`
Description: Runs the noteboard attack as follows:

- Run a kv_server server A.
- Then, a client connects to server A from server B and encodes a message in wear-levelling metadata.
- Then another client connects to server A and retrieves this encoded message.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: ICSPatch: Automated Vulnerability Localization and Non-Intrusive Hotpatching in Industrial Control Systems using Data Dependence Graphs

Prashant Hari Narayan Rajput[1], Constantine Doumanidis[2], Michail Maniatakos[2]

[1]NYU Tandon School of Engineering, Brooklyn, NY, USA
[2]New York University Abu Dhabi, Abu Dhabi, UAE

A Artifact Appendix

A.1 Abstract

This artifact contains the source code for ICSPatch, a hot-patching tool for control application binaries on Codesys runtime-compatible Programmable Logic Controllers (PLCs). It can detect and patch out-of-bounds write/read, improper input sanitization, and os command injection vulnerabilities in control applications. It can patch these vulnerabilities via an LKM-based (Loadable Kernel Module) patcher or through JTAG.

Evaluating ICSPatch on a live setup requires a Codesys Codesys runtime-compatible PLC with either a Linux OS or JTAG connection. To facilitate a more straightforward evaluation, we also allow hotpatching `angr` *simulation instances loaded with vulnerable memory snapshots of control application binaries in case of missing physical devices. Furthermore, we package ICSPatch in a Docker container to minimize the initial setup steps, supporting multiple platforms. ICSPatch is tested on Wago PFC 100, PFC200 for Linux-5.10.21, and BeagleBone Black for Linux-4.19.82-ti-rt-r31.*

A.2 Description & Requirements

A.2.1 How to access

All the documents and source code for ICSPatch is available on GitHub at `https://github.com/momalab/ICSPatch/tree/v1.0`.
[Commit: 40803636849d24ab6a50e1c166d7522c7a1ceb6e]

A.2.2 Hardware dependencies

ICSPatch requires a 32 bit ARM architecture PLC supporting Codesys-runtime. In addition, a readily accessible JTAG port is also required for patching the control applications by using JTAG. However, ICSPatch also supports LKM-based patching, removing the need for an accessible JTAG port.

A.2.3 Software dependencies

ICSPatch is packaged in a Docker container. To run ICSPatch, manually install Docker as explained on this `https://docs.docker.com/engine/install/ubuntu/`.

In case of missing hardware requirements, utilize the captured memory snapshots of control application binaries (included in the repository) and evaluate ICSPatch by hotpatching the `angr` *simulation instance as explained* `https://github.com/momalab/ICSPatch/blob/v1.0/main/README.md`.

A.2.4 Benchmarks

We create a synthetic dataset of vulnerable control application binaries with their source code project files present at `https://github.com/momalab/ICSPatch/tree/v1.0/experiments/iec_projects` *and the corresponding memory snapshots for the WAGO PFC 200 included in the repository at the location:* `https://github.com/momalab/ICSPatch/tree/v1.0/main/src/bin/internal`. *ICSPatch can utilize the control application memory snapshots in the evaluation mode.*

A.3 Set-up

A.3.1 Installation

*For installing and running ICSPatch on the Docker container, build from the Dockerfile provided in the repository. The steps are explained in the **Installation** section at* `https://github.com/momalab/ICSPatch/blob/v1.0/main/README.md`.

Run the following commands to build and run the docker container.

```
cd ICSPatch/main
sudo docker build --pull --rm -f "Dockerfile" -
    ↪ t icspatch:latest "."
```

```
sudo docker images // List the images
sudo docker run -it icspatch:latest
```

To try ICSPatch on live PLCs, please build the LKM patcher for the target Linux Kernel.

A.3.2 Basic Test

To test the successful installation of ICSPatch, run the command `sudo docker run -it icspatch:latest`. *If ICSPatch executes successfully, the following prompt will be displayed on stdout:*

```
Select Vulnerability:
-------------------------
0. improper_input
1. oob_write
2. oob_read
3. os_command
4. exit
Choice:
```

A.4 Evaluation Workflow

Artifacts for ICSPatch have a detailed example for evaluating out-of-bound write in a control application binary for desalination plants, located at https://github.com/momalab/ICSPatch/blob/v1.0/main/README.md *in the section* ***ICSPatch for Evaluation****.*

The overall steps are as follows:

1. *Run the following command to execute ICSPatch in the Docker container and select the vulnerability for evaluation by entering the corresponding choice.*

```
sudo docker run -it icspatch:latest
```

2. *Next, select the appropriate mode of operation for ICSPatch. For evaluating ICSPatch without requiring a physical PLC, select* ***0****, as shown:*

```
Select Experiment:
-------------------------
0. Evaluate
1. Live
Choice: 0
```

3. *Next, select the test infrastructure when ICSPatch displays the following menu on stdout.*

```
Select Infrastructure:
-------------------------
0. aircraft_control
1. anaerobic_reactor
2. chemical_plant
3. desalination_plant
4. smart_grid
Choice:
```

4. *Some infrastructure might have multiple vulnerable control application binary examples. Please select the target control application binary for evaluation when a menu shows up similar to this:*

```
Select Test Sample:
-------------------------
0. bin/internal/chemical_plant/oob_write/
    ↪ code_1
1. bin/internal/chemical_plant/oob_write/
    ↪ code_2
Choice:
```

5. *After this, ICSPatch starts loading captured memory snapshots of the selected control application binary with a legitimate input (used to detect crashes that only impact the control application stack). After which the stdout displays the message.*

```
- Press Enter to continue to capture
    ↪ exploit input hexdump ...
```

6. *Press* ***Enter*** *to continue loading control application memory snapshot with exploit input, which results in displaying an output as shown below:*

```
***************************
RULE: OUT_OF_BOUNDS_WRITE_RULE
MESSAGE: OUT-OF-BOUNDS WRITE VULNERABILITY
    ↪ DETECTED
***************************
----- BLOCK DISASSEMBLY -----
Instruction # in block: 8
0xb6bbf8a0: stmhs r3!, {r1, ip}
0xb6bbf8a4: subshs r2, r2, #8
0xb6bbf8a8: stmhs r3!, {r1, ip}
0xb6bbf8ac: subshs r2, r2, #8
0xb6bbf8b0: stmhs r3!, {r1, ip}
0xb6bbf8b4: subshs r2, r2, #8
0xb6bbf8b8: stmhs r3!, {r1, ip}
0xb6bbf8bc: bhs #0xb6bbf89c
------ DEBUG INFO ------
* Instruction Address: 0xb6bbf8a0
* Exploit Memory Address: 0xb617ad5c
* Length: None
* Expression: 0x0
[*] Angr execution time of the control
    ↪ application: 5.889697313308716
* Found start node: 0x83f48d0 ...
```

```
- Localization start address list:
    ↪ [138365136] ...
----------0----------
[*] Starting exploit localization from
    ↪ address 0x83f48d0 ...
[*] Start address: 0xb6193fb4 End Address:
    ↪ 0xb6194018...
[*] Bounded by 0xb6193fb4 - 0xb6194018 ...
[*] Search successful for start node 0
    ↪ x83f48d0 ...
[*] Detected exploit location: 0xb6193ff0:
    ↪ str r6, [sp, #8]
[*] Detected exploit input: 0xb617aca0: ['0
    ↪ x2', '0x0', '0x200']
[*] Mermory value at exploit location: 0
    ↪ xb617aca0: 0x00000200
----------0----------

[*] Time for localizing vulnerability:
    ↪ 0.012766838073730469
* Selected vulnerability location is 0
    ↪ xb6193ff0 ...
* Exploit memory location is 0xb617aca0 ...

- Press Enter to continue to patching ...
```

*Here, **RULE** displays the name of the vulnerability identification rule triggered for the exploit input and the corresponding message in **MESSAGE**. It also detects the start node for DDG traversal for performing vulnerability localization. The start node in this example is detected as `0x83f48d0`. The traversal successfully detects the exploit instruction location at `0xb6193ff0` and the memory location for the input (to be validated by the patch) at `0xb617aca0`.*

7. *Press **Enter** to continue patching the vulnerability, which displays patch-related information such as the address table base address and the memory location for an empty location. Press **Y** when the prompt display:*

```
[*] Saved patch information detected. Use
    ↪ it? (Y/N):
```

This directs ICSPatch to use saved path information rather than connecting to an active local patch server.

8. *Finally, ICSPatch creates the patch, loads it in the `angr` simulation, and verifies it. Loading the patch in the `angr` simulation instance is similar to writing it into the live PLC with the LKM patcher. So, this can successfully test the patch created by ICSPatch, and the overall automated process.*

9. *Since the evaluation of ICSPatch does not require a connected PLC, once **Enter** is pressed on the prompt:*

```
- Press Enter to continue to patching live
    ↪ PLC ...
```

ICSPatch exits after 10 seconds when failing to connect to a local patch server deployed on a live PLC.

Instructions on GitHub also elaborate on how to use ICSPatch with a live PLC.

A.5 Evaluation and Expected Results

While running the experiments, as explained in Subsection A.4, ICSPatch displays the timings (in seconds) corresponding to every operation on the stdout. For instance,

```
[*] Time for localizing vulnerability:
    ↪ 0.012766838073730469
```

It should be noted that only the vulnerability localization time is representative of the live PLC scenario. All the other timings will change when tested with a live PLC. Furthermore, the LKM patcher captures the timing for the critical operation of redirecting execution flow by overwriting the `ldr` instruction, as explained in the paper.

USENIX'23 Artifact Appendix: ARGUS: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions

Siddharth Muralee[‡*], Igibek Koishybayev[†*], Aleksandr Nahapetyan[†], Greg Tystahl[†],
Brad Reaves[†], Antonio Bianchi[‡], William Enck[†], Alexandros Kapravelos[†], Aravind Machiry[‡]
[‡] Purdue University, {smuralee, antoniob, amachiry}@purdue.edu
[†] North Carolina State University, {ikoishy, anahape, gttystah, bgreaves, whenck, akaprav}@ncsu.edu

A Artifact Appendix

A.1 Abstract

ARGUS's artifact contains the source code and corresponding infrastructure to run our taint tracking tool. This is a modified version of the tool presented in the paper, which can generate all the taint summaries mentioned in the paper on the fly (rather than generating offline summaries). Also provides the datasets required to validate the tool and the claims made in the paper.

This document describes how to set-up our prototype, gives an overview of the requirements to replicate some of the experiments conducted in our evaluation, along with instructions to run them.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There exist no risks associated with executing ARGUS on any system. ARGUS is encapsulated as a Docker image, incorporating all the requisite dependencies necessary for conducting evaluations. It fetches files into Docker's isolated filesystem, thereby obviating any interaction with the host system's filesystem.

ARGUS doesn't directly interact with the repository apart from cloning it, so it is safe to run on any GitHub repository. However, a few of the vulnerabilities described in this document might still not be fixed, it is recommended to test these vulnerabilities using private forks of these repositories, so that an exploitable fork is not public.

A.2.2 How to access

Given that our paper is presently subject to an embargo, we will be provisionally providing all relevant code and datasets in the form of encrypted zip archives. These archives can be accessed at `https://github.com/purs3lab/Argus_artifacts`, under the commit hash `c8a2086`.

The decryption password for the `ARGUS.zip` archive is `d7e21ecf50fd0116a76957f285fda57f6426423af446b`. The `VWBench.zip` archive, however, is unencrypted.

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

The ARGUS is encapsulated as a Linux Docker image. Any system equipped with the capability to execute Linux Docker containers should suffice for the deployment of the tool for evaluation purposes.

The tool also can be executed outside docker, however, requires a Python version 3.8 and CodeQL installed. All the required Python packages can be installed via the Poetry Python package manager.

A.2.5 Benchmarks

The ARGUS was evaluated using two benchmarks:

- **VWbench:** This comprises a collection of vulnerable workflows, curated from security advisories previously reported and published. The VWbench encompasses 24 workflows, stored in the `vwbench.zip` archive, specifically within the `.github/workflows` directory.
- **Realworld Dataset:** This represents a collection of 2.8 million workflows, upon which our tool was assessed. A selection of representative workflows was chosen from this set to serve as sample PoCs, and added in the paper as listings.

A.3 Set-up

A.3.1 Installation

Given that the tool is packaged as a Docker container, the installation procedure merely entails the setup and construction of the container.

*Both authors made equal contributions to this work

1. Install Docker and Docker Compose via the command: `apt-get -y install docker.io docker-compose`

2. Extract the contents of `ARGUS.zip`, which should contain a directory named `Argus`

3. Navigate to the newly created folder and initiate the build process with the command: `docker-compose build`

A successful build, devoid of any complications, signifies that the tool is prepared for utilization.

A.3.2 Basic Test

To validate the proper functioning of the tool, we have retained the SARIF files corresponding to the actions/checkout action within the directory titled `saved_results`, nested inside the `Argus` folder.

These results can be regenerated by executing the following commands: `./run_check.sh`

The resultant SARIF file should be located in the `results` directory. The SARIF files should be consistent with SARIF file starting with `actions#checkout` inside the `saved_results` folder.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): ARGUS possesses the capability to identify all vulnerable workflows within the VWBench benchmark. This claim is substantiated by the results of Experiment E1.

(C2): ARGUS has been deployed to discover new bugs, a claim which is corroborated by Experiment E2.

A.4.2 Experiments

(E1): *[VWBench] [30 human-minutes + 2 compute-hours + 5GB disk]:* This experiment reproduces the VWBench benchmark for the vulnerabilities identified by ARGUS.
Procedure: Ensure that ARGUS generates alerts for each workflow in VWBench. The results will be located in the `results` directory.
Preparation: Extract the contents of `VWBench.zip` and upload it into a private GitHub repository. The workflows should be situated in the `.github/workflows/` directory of the repository. Generate a GitHub token to facilitate the tool's cloning of the GitHub repository. (We neither retain nor collect the GitHub tokens.)
Execution: Deploy the Docker container using the following command: `docker-compose run argus -mode repo -url <username>:<GHToken>@<url_to_git_repo>`
Results: The execution results should be found in the `results` directory. Each workflow should have an accompanying SARIF file containing the results. The SARIF format resembles JSON and can be viewed using online viewers such as as well as the SARIF viewer plugin on Visual Studio Code.

(E2): *[RWDataset] [2 human-hour + 2 compute-hours + 5GB disk]: This experiment reproduces several of the 0day vulnerabilities found by* ARGUS*, specifically the ones listed in the paper.*
How to: The list of vulnerable workflows and actions presented in the paper, is added to the file `rwvulns.md` in the `Argus` folder. The experiment requires running argus on these repositories and verifying that ARGUS can identify these vulnerabilities.
Preparation: None
Execution: To test the workflows run : `./run_test_docker.sh`
Results: The SARIF file present in the `results` directory can be used to identify the security vulnerabilities in these workflows and actions.

A.5 Notes on Reusability

For the large-scale evaluation delineated in our paper, we cached all reports corresponding to each version of each JavaScript and Composite action, as well as reusable workflows, within a MongoDB database. This procedure can be readily replicated by implementing minor modifications to the infrastructure responsible for report generation within our codebase, specifically within `argus_components/report`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: McFIL: Model Counting Functionality-Inherent Leakage

Maximilian Zinkus
Johns Hopkins University
zinkus@cs.jhu.edu

Yinzhi Cao
Johns Hopkins University
yzcao@cs.jhu.edu

Matthew D. Green
Johns Hopkins University
mgreen@cs.jhu.edu

A Artifact Appendix

A.1 Abstract

Our Artifact submission encapsulates the version of our tool, McFIL, which was used in the evaluation of the work. We provide the source code in a dedicated GitHub repository along with archived relevant dependencies. As configured, this artifact is prepared to reproduce our evaluation results in an offline setting rather than evaluate any online secure protocols. Interested parties are welcomed to independently install our software and evaluate it at their leisure, and to submit feedback, issues, and/or pull requests to the open source project.

A.2 Description & Requirements

Our software tool is intended to perform an iterative analysis of a target functionality. At each iteration, the tool gathers available information (constraints within a SAT solver), generates new constraint systems based on a randomized sampling algorithm, solves these SAT problems, and discovers an approximately greedy-optimal result. It then tests this result against an "Oracle," configured by default to be an offline instantiation of the target functionality as a test harness stand-in for an online secure protocol. Our test harness generates a secret at the beginning of the loop (withholding it from the main algorithm), and then iteratively discovers (partially or completely) this secret by generating and executing queries to the Oracle.

We evaluated McFIL on an Intel Xeon CPU E5-2695 v4 at 2.10GHz (72 threads) with 500 GB memory. Our evaluation targeted relatively smaller benchmarks to enable randomized repetition, and therefore did not stress this system to its limits. Our evaluation used the following software dependencies:

- CryptoMinisat 5.8.0 `https://github.com/msoos/cryptominisat/releases/tag/5.8.0`
- ApproxMC 4.0.1 `https://github.com/meelgroup/approxmc/releases/tag/4.0.1`
- Z3 4.8.15 `https://github.com/Z3Prover/z3/releases/tag/z3-4.8.15`
- louvain-community@8cc5382d `https://github.com/meelgroup/louvain-community`
- arjun@407ea7f5 `https://github.com/meelgroup/arjun`
- Python 3.8

A.2.1 Security, privacy, and ethical concerns

None directly, as our artifact is configured to evaluate functionalities in an offline setting by default, requiring the user to configure their target functionality. McFIL can be used in an "online" setting to directly evaluate real-world secure protocols and attempt to maximize leakage. We acknowledge that this could potentially be used to exploit target protocols, however, as a community we move forward and publish these tools to improve understanding and defense with the assumption that attackers will independently arrive at optimal attacks in secret.

A.2.2 How to access

The source code can be accessed at our GitHub release URL: `https://github.com/maxzinkus/McFIL-Release/releases/tag/release`

A.2.3 Hardware dependencies

Please refer to Description & Requirements above. We believe that the artifact can be run on a lower-specification machine sufficiently well to observe functionality and perform limited experiments (listed in Experiments) which indicate our broader results without requiring them to be fully re-run (which would take many compute-hours).

A.2.4 Software dependencies

Please refer to Description & Requirements above. We have bundled these dependencies within a Docker image for easier evaluation.

A.2.5 Benchmarks

This largely does not apply to our work. However, the included `target_funcs` folder contains example target functionalities which we evaluate against in our paper, and so these could be considered benchmarks in a sense. These are automatically discovered and used when their names are passed as command-line arguments to our tool such as `python3 main.py millionaires` for `target_funcs/millionaires.py`.

A.3 Set-up

Generally, in order to prepare a system for use with our tool, two groups of dependencies must be installed. First, the external package dependencies listed in Description & Requirements, and second the `python3 pip` dependencies in the software's `requirements.txt`.

A.3.1 Installation

[Mandatory] Instructions to download and install dependencies as well as the main artifact. After these steps the evaluator should be able to run a simple functionality test.

1. Install the dependencies
   - python 3.8: `sudo apt install python3`
   - `sudo apt install build-essential cmake zlib1g-dev libboost-program-options-dev libsqlite3-dev libgmp3-dev`
   - louvain-community: clone the repository and follow build instructions
     (a) `cd louvain-community ; mkdir build ; cd build ; cmake .. ; make ; sudo make install`
   - z3: clone the repository and follow build instructions
     (a) `cd z3 ; python scripts/mk_make.py --python ; cd build ; make ; sudo make install ; pip install z3-solver`
   - cryptominisat: clone the repository and follow build instructions
     (a) `cd cryptominisat ; mkdir build ; cd build ; cmake .. ; make ; sudo make install ; sudo ldconfig`
   - arjun: clone the repository and follow build instructions
     (a) `cd arjun ; mkdir build ; cd build ; cmake .. ; make ; sudo make install`
   - approxmc: clone the repository and follow build instructions
     (a) `cd approxmc ; mkdir build ; cd build ; cmake .. ; make ; sudo make install`
2. Fetch the software and install python dependencies
   (a) clone or otherwise fetch the source of McFIL
   (b) create a virtual environment `python3 -m venv venv`
   (c) install python dependencies `source venv/bin/activate ; pip install -r requirements.txt`

A.3.2 Basic Test

In order to determine if the dependencies are installed and the environment configured, the following commands should work without error with the virtual environment active:

- `cryptominisat </dev/null`
- `approxmc </dev/null`
- `python3 -c 'import z3 ; z3.SolverFor'`

Then, McFIL can be used to evaluate functionalities in the `target_funcs` directory such as:

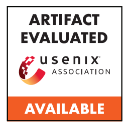
- `python3 main.py millionaires`
- `python3 main.py sugarbeets`

A.4 Notes on Reusability

McFIL is designed for use with functionalities of the user's choosing. A critical step in analyzing a novel functionality is accurately encoding it in the input format that McFIL expects. We recommend that users of our software use the existing `target_funcs` given target functionality examples as a basis (e.g. by copy-pasting them) to work from when defining new targets. McFIL requires that the target be implemented both "in the clear" (i.e. a correct python implementation of the function under test) and in a format the solver can understand. We provide `solver.py`, a support library which we hope makes encoding easier. All existing examples use this library and can be referred to for aid in its use.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: **Isolated and Exhausted: Attacking Operating Systems via Site Isolation in the Browser**

Matthias Gierlings, Marcus Brinkmann, Jörg Schwenk
Ruhr University Bochum

A Artifact Appendix

A.1 Abstract

Site Isolation, a security feature recently introduced to major browsers enables attacks on modern operating systems. To demonstrate the impact of Site Isolation attacks on web users we implemented a Site Isolation fork-bomb and a DNS Cache Poisoning Attack: *DNS Poisoning by Exhaustive Misappropriation of Network Sockets (DEMONS)*. Setup instructions, configurations, and the implementation of both attacks are part of our publicly available research artifacts. While DEMONS was assigned CVE-2020-6557 and patched by the Chromium Team,[1] the fork-bomb is still a threat to current browsers. We describe a way to mitigate the Site Isolation fork-bomb in Chromium-based browsers without measurable performance penalty and include both the patch and our performance measurement results in our artifacts.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All artifacts provided should be evaluated in a strictly isolated, physical or virtual lab environment. In case reviewers decide to evaluate attacks using public internet infrastructure they must employ measures to prevent *any* traffic from flowing between systems that are part of their evaluation setup and other systems.

A.2.2 How to access

The artifact repository is available at `https://git.noc.rub.de/gierlmds/isolated-and-exhausted/-/tree/usenix23_ae_summer`. Additionally we provide a mirror-copy[2] via Zenodo.

A.2.3 Hardware dependencies

Our experiments can be conducted either in a VM-based lab environment on a single x86 machine (VM setup) or alternatively on a set of physical hosts (hardware setup). In both cases, the setup consists of four distinct hosts (physical or virtual). For the VM setup an x86 desktop system with at least 4 cores/8 threads, virtualization support[3], at least 20 GiB RAM and more than 400 GiB of free disk space is required.

DEMONS VM Setup We recommend using Virtual Box 6[4] for virtualization and Kubuntu Linux 22.04 LTS as host OS.

DEMONS Hardware Setup For our hardware setup we used three desktop computers[5] to run the victim, the benign DNS server and the router. A ThinkPad T480s[6] was used to run the attacker infrastructure (webserver and spoofer).

Fork-Bomb Hardware Setup The fork bomb was evaluated on a Dell Latitude 5280, Intel Core i5 7200U, 8 GiB RAM, 240 GiB M.2 SATA SSD running Kubuntu Linux 18.04.5 LTS (Kernel 5.4.0-62).

A.2.4 Software dependencies

In addition to our artifacts additional software (binaries/installers) are required. This section lists the exact software versions used in testing our artifacts, their use is strongly recommended. While we did not verify that other versions than those listed below produce the same results, we believe that our artifacts will work with any version of Windows 10 and any release of Chrome with Site Isolation Support prior to version 86.0.4240.75, which introduces a fix for CVE-2020-6557. Using newer versions of Python 3 and the Python Websocket Client are unlikely to impact the experiments. Using different Virtual Box versions or other Hypervisors should also be possible but potentially affects performance and may require manual timing adjustments in the code.

- Windows 10 (1909 Build 18363.815)
- Ubuntu Linux 20.04.5 LTS Server
- Kubuntu Linux 18.04.5 LTS (Kernel 5.4.0-62)
- Kubuntu Linux 20.04.5 LTS
- Kubuntu Linux 22.04.1 LTS
- Chrome 83.0.4103.106 (Windows)

[1] https://chromereleases.googleblog.com/2020/10/stable-channel-update-for-desktop.html
[2] DOI: 10.5281/zenodo.7356538
[3] Intel VT-x or AMD-V
[4] 6.1.38-dfsg-3 ubuntu1.22.04.1 amd64
[5] Intel Core2Quad Q9400, 4 GiB RAM, Intel 82567LM-3 Gigabit NIC
[6] Intel i7-8550U, 40 GiB RAM, Intel Ethernet I219-V

- Firefox Nightly 86.01a (Windows, Linux)
- Chromium 83.0.4103.0 (Linux)
- Caddyserver 1.0.4[7] (Linux)
- Python 3.8.2 (Windows)
- Python Websocket Client for Python 3.8.2 (Windows)
- Virtual Box 6[4] (Linux)

It is recommended to download all binary dependencies except Caddyserver[7] before starting the setup procedure and transfer them via shared folders in case of a VM setup or via USB stick in case of a setup with physical hosts. This way the required software can be installed without internet connection. This also prevents unsolicited automatic browser/OS updates.

In addition to our artifacts the following software source code is required:

- Chromium 101.0.4951.64[7] (Linux)

A.2.5 Benchmarks

To benchmark the performance impact of our Site Isolation mitigation we used the performance profiler[8] integrated into the Chrome developer tools. The profiler can be accessed via the *Performance*-tab.

A.3 Set-up

A.3.1 Installation

The lab environment for the Site Isolation fork-bomb and DEMONS consists of four hosts, the victim, a router, a benign DNS server, and the attacker's server. This section contains detailed instructions on how to set up those hosts as virtual machines using Virtual Box. The instructions are also suitable for installation on native hardware. In this case, Virtual Box-specific steps should be omitted.

Base System The base image serves as a common base installation for the router, the benign DNS and the attacker server. To prepare the base image, perform the following steps:

1. Install Oracle VirtualBox.[9]
2. Create a new 64-bit Ubuntu Linux VM with 2 cores and 2 GiB RAM and 15 GiB hard disk.
3. Enable the following acceleration settings: `VT-x/AMD-V`, `Nested Paging`, `PAE/NX`, `KVM Paravirtualization`
4. Install the Ubuntu 20.04 LTS server base image. This guide and the derivative guides assume that during the installation process `user` was chosen as username and `si-base` was chosen as the hostname.
5. Update the OS and install additional packages:

```
1 sudo apt update && sudo apt dist-upgrade && sudo
      ↪ apt autoremove --purge
2 sudo apt install build-essential curl git iptables
      ↪ -persistent
```

6. Insert the VirtualBox guest additions ".iso" into the virtual CD-ROM drive of your machine.
7. Install the VirtualBox guest additions.

```
1 sudo mount /dev/sr0 /media
2 cd /media
3 sudo ./VBoxLinuxAdditions.run
```

8. Add `user` to the `vboxsf` group

```
1 sudo usermod -a -G vboxsf user
```

9. Set an environment variable referring to the artifact repository base folder. It should be located at: `/home/user/isolated-and-exhausted`

```
1 export ARTIFACTS_REPO="/home/user/isolated-and-
      ↪ exhausted/"
2 sudo bash -c "echo ARTIFACTS_REPO=${ARTIFACTS_REPO
      ↪ } >> /etc/environment"
```

10. Clone the Isolated and Exhausted artifacts repository:

```
1 git clone https://git.noc.ruhr-uni-bochum.de/
      ↪ gierlmds/isolated-and-exhausted
      ↪ $ARTIFACTS_REPO
```

11. Shut down the system.

```
1 sudo systemctl poweroff
```

Attacker Server The attacker server runs the attacker's web server and the spoofer. Both the web server and the spoofer are containerized using Docker. To set up the attacker server perform the following steps on top of a *Base System* (cf. section A.3.1 Base System):

1. Clone the base system to a new virtual or physical host.
2. Name the cloned VM `Isolated and Exhausted Attacker`.
3. Boot the system and change the hostname to `si-attacker`:

```
1 sudo bash -c "echo si-attacker > /etc/hostname"
```

4. Install docker.

```
1 curl -fsSL https://download.docker.com/linux/
      ↪ ubuntu/gpg | sudo apt-key add -
2 sudo add-apt-repository "deb [arch=amd64] https://
      ↪ download.docker.com/linux/ubuntu $(
      ↪ lsb_release -cs) stable"
3 sudo apt update
4 sudo apt install docker-ce docker-ce-cli
      ↪ containerd.io docker-compose
```

[7] Will be obtained during setup (cf. subsection A.3)
[8] https://developer.chrome.com/docs/devtools/evaluate-performance/
[9] https://www.virtualbox.org/

5. Add `user` to the docker group.

```
1 sudo usermod -a -G docker $USER
```

6. Get the ubuntu base image from Docker Hub.

```
1 sudo docker pull ubuntu:bionic-20200311
```

7. Copy the attacker host configuration[10]

```
1 sudo rm -f /etc/netplan/*
2 sudo cp -R $ARTIFACTS_REPO/hosts/attacker/rootfs/*
      ↪ /
```

8. Download a copy of the caddy web server into the repository. For our evaluation caddy version 1.0.4 was used. Newer caddy versions should work but may require modifications to Caddyfiles.

```
1 cd $ARTIFACTS_REPO/hosts/attacker/webserver/
2 curl -JLO 'https://github.com/caddyserver/caddy/
      ↪ releases/download/v1.0.4/caddy_v1.0.4
      ↪ _linux_amd64.tar.gz'
```

9. Replace `/etc/resolve.conf` with a sym-link:

```
1 sudo ln -sf /var/run/systemd/resolve/resolv.conf /
      ↪ etc/resolv.conf
```

10. Add the base folder for docker volumes.

```
1 sudo mkdir -p /srv/docker/data/demons/attacker/
      ↪ spoofer/logs /srv/docker/data/demons/
      ↪ attacker/spoofer/results
2 sudo chown -R root:docker /srv/docker
3 sudo chmod -R 775 /srv/docker
```

11. Enable IPv6 forwarding and redirect all traffic on the attacker's subnet `2001:db8::/113` to the web server running on `2001:db8::1`.

```
1 sudo sysctl -w net.ipv6.conf.all.forwarding=1
2 sudo ip6tables -t nat -A PREROUTING -d 2001:db8
      ↪ ::/113 -j DNAT --to-destination 2001:db8
      ↪ ::1
```

12. Spoof the spoofers' source IP address.

```
1 sudo ip6tables -t nat -A POSTROUTING -s 2001:db8
      ↪ ::8001 -p udp -j SNAT --to-source 2001:db8
      ↪ ::8000:1
```

13. Make iptables rules persistent after reboot.

```
1 sudo netfilter-persistent save
```

14. Shut down the system.

```
1 sudo poweroff
```

15. Change the VM network settings
    - Right-click the Attacker VM in the VirtualBox Manager
    - Select the option `Settings` from the drop-down menu.
    - Make sure that `Adapter 1` and `Adapter 2` are enabled and that all other adapters are disabled
    - Attach `Adapter 1` to an `Internal Network` named `attacker_provider`. Set the adapter type to `Paravirtualized Network (virtio-net)`
    - For `Adapter 2` set the network type to `NAT`
16. Boot the system.
17. Create docker containers containing the attacker web server and spoofer.

```
1 cd $ARTIFACTS_REPO/hosts/attacker/
2 docker-compose up -d --build
```

18. Shut down the system.

```
1 sudo systemctl poweroff
```

19. Disable `Adapter 2` in the network settings.

[10] Using real hardware the network interface names may differ from the ones preconfigured in `/etc/netplan/00-installer-config.yaml` and must be manually adjusted.

Router

1. Clone the base VM image.
2. Name the cloned VM `Isolated and Exhausted Router`.
3. Change the VM properties to use 2 CPU cores, 1 GiB RAM.
4. Add three `Internal Network` adapters to the VM and assign the following network names:
    - Adapter1: `victim_provider`
    - Adapter2: `attacker_provider`
    - Adapter3: `dns_provider`
5. Boot the system and enable IPv6 forwarding

```
1 sudo sysctl -w net.ipv6.conf.all.forwarding=1
```

6. Change the hostname to `si-router`.

```
1 sudo bash -c "echo si-router > /etc/hostname"
```

7. Copy the configuration to the router VM[10]

```
1 sudo rm -f /etc/netplan/*
2 sudo cp -r $ARTIFACTS_REPO/hosts/router/rootfs/* /
```

8. Reboot.

```
1 sudo systemctl reboot
```

9. Enable and start the traffic shaping service.

```
1 sudo systemctl enable traffic-shaping.service
2 sudo systemctl start traffic-shaping.service
```

10. Check the MAC addresses of the interfaces `enp0s3` (`victim_provider`), `enp0s8` (`attacker_provider`) and `enp0s9` (`dns_provider`) and ensure that they are assigned to the corresponding VirtualBox adapters.
11. Shut down the system.

```
1 sudo poweroff
```

DNS Server

1. Clone the base VM image.
2. Name the cloned VM `Isolated and Exhausted DNS`.
3. Change the VM properties to use 2 cores, and 1 GiB RAM.
4. Boot the system and change the hostname to `si-dns`:

```
1 sudo bash -c "echo si-dns > /etc/hostname"
```

5. Install `bind9`.

```
1 sudo apt install bind9
```

6. Copy the configuration to the router VM[10]

```
1 sudo rm -f /etc/netplan/*
2 sudo cp -r $ARTIFACTS_REPO/hosts/dns/rootfs/* /
```

7. Disable `systemd-resolved`.

```
1 sudo systemctl disable systemd-resolved
```

8. Power off the system.

```
1 sudo systemctl poweroff
```

9. Change the VM network settings
   - Right-click the DNS VM in the VirtualBox Manager.
   - Select the option `Settings` from the drop-down menu.
   - Make sure that `Adapter 1` is enabled and that all other adapters are disabled.
   - For `Adapter 1` set the network type to `Internal Network` named `dns_provider`.

Victim VM Create a Virtual Box VM with 4 CPU cores and 8 GiB RAM and a 50 GiB hard disk or use an equivalent physical host. Note: Windows, Chrome and Firefox may perform fully automatic updates without prompting the user when an internet connection is available. Performing the victim installation offline solves this problem. The dependencies listed in subsubsection A.2.4 can be transferred to the victim VM via a shared folder, or via USB stick in case dedicated physical machines are used for the experiment.

1. Change the VM network settings.
   - Right-click the Victim VM in the VirtualBox Manager
   - Select the option `Settings` from the drop-down menu.
   - Make sure that Adapter 1 is enabled and that all other adapters are disabled
   - Attach Adapter 1 to an Internal Network named `victim_provider`.
   - Set the adapter type to `Intel PRO/1000 MT Desktop (82540EM)`.
2. Install Windows 10.
3. Install Chrome 83.0.4103.106 and Firefox Nightly 86.01a. Note: `chrome.exe` is expected to be located at `C:\ProgramFiles(x86)\Google\Chrome\Application\chrome.exe`. If chrome is installed elsewhere, the content of the variable `ATTACK_CMD` in `attack_simulator.py`, line 14 must be adjusted to point to `chrome.exe`.
4. Install the CA certificate from the artifact repository in Chrome:
   - Open Chrome and navigate to `chrome://settings/privacy`.
   - In the `Privacy and security` box click on `More`.
   - Click `Manage Certificates`.
   - Once the `Certificate Import Wizard` opened click `Next`.
   - Select the CA certificate from `isolated-and-exhausted/hosts/attacker/webserver/rootfs/srv/ca/ca.crt.pem` and proceed by clicking `Next`.
   - In the `Certificate Store` dialog click `Browse...` and change the Certificate Store to `Trusted Root Certification Authorities`
   - Proceed by clicking `Next` and the finish the import by clicking `Finish`.
5. Install the CA certificate from the artifact repository in FireFox Nightly:
   - Open Firefox and navigate to `about:preferences#privacy`.
   - Scroll down to the option group labeled `Security`.
   - Click on `View Certificates`
   - In the `Certificate Manager` Dialog select the `Autorities` tab and click `Import`.
   - Select the CA certificate from `isolated-and-exhausted/hosts/attacker/webserver/rootfs/srv/ca/ca.crt.pem` and proceed by clicking `Open`.
   - In the `Downloading Certificate` Dialog check `Trust this CA to identify websites.` and click `OK`.
6. Install Python 3.8.2
7. Install the Python Websocket Client (cf. subsubsection A.2.4)
8. Copy `websocket_client-1.3.2-py3-none-any.whl` to `C:\Users\user\Downloads`.
9. Open `cmd.exe` and execute the following commands to install the Python Websocket Client:

```
1 c:
2 cd \Users\user\Downloads
3 pip install websocket_client-1.3.2-py3-none-any.
    ↪ whl
```

10. Open the Windows `Network Connections` dialog via `Startmenu → Settings → Network & Internet → Ethernet → Change adapter options`.
11. Right-click the network connection (e.g. `Ethernet`).
12. Select `Properties` from the drop-down menu.
13. In the `Ethernet Properties` dialog select `Internet Protocol Version 6 (TCP/IPv6)`
14. Click `Properties`.
15. Adjust the Windows network adapter settings:
    - IPv6 Address: `2001:db8::4000:1`
    - Subnet prefix length: `98`
    - Default gateway: `2001:db8::7fff:ffff`
    - DNS server: `2001:db8::8000:1`
16. Confirm the changes by clicking `OK`.
17. Copy the artifact repository folder `isolated-and-exhausted` to `C:\Users\user\Documents`.

Chrome Build Environment This paragraph describes how to set up a build environment for Chromium and is mostly based on the official Chromium Build Instructions.[11]

1. Install Kubuntu 20.04 LTS on a host with at least 16 GiB of RAM and at least 300 GiB of free hard disk space.
2. Insert the VirtualBox guest additions ".iso" into the virtual CD-ROM drive of your machine.
3. Install the VirtualBox guest additions.

```
sudo mount /dev/sr0 /media
cd /media
sudo ./VBoxLinuxAdditions.run
```

4. Add `user` to the `vboxsf` group

```
sudo usermod -a -G vboxsf user
```

5. Install additional required packages.

```
sudo apt install git build-essential
```

6. Reboot the system.

```
sudo reboot
```

7. Clone the Chromium depot tools and add their path to the `PATH` environment variable:

```
git clone https://chromium.googlesource.com/
    ↪ chromium/tools/depot_tools.git
export PATH="$PATH:/home/user/depot_tools"
echo export PATH=\"\$PATH:/home/user/depot_tools\"
    ↪ >> ~/.bash_profile
```

8. Get the Chromium source code (this may take a while).

```
mkdir ~/chromium && cd ~/chromium
fetch --nohooks chromium
cd src
./build/install-build-deps.sh
git fetch --tags
git checkout tags/101.0.4951.64
```

[11] https://chromium.googlesource.com/chromium/src/+/main/docs/linux/build_instructions.md

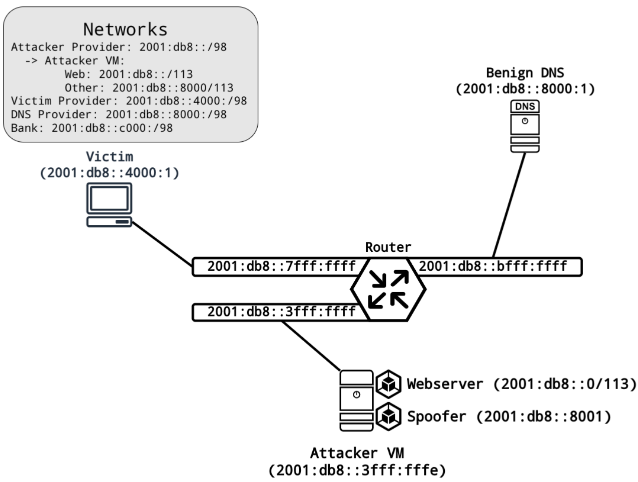


Figure 1: Network configuration of the Isolated and Exhausted evaluation lab.

9. Prepare the build. The command `gn args ...` automatically opens a file in the default text editor. Replace the contents of this file with the contents of the file `~/isolated-and-exhausted/site_isolation_patch/gn.args` from the artifacts repository.

```
gclient sync -D --with_branch_heads
gclient runhooks
gn args out/release
```

10. Finalize build preparations.

```
gn gen out/release
```

11. Build an unmodified Chrome (this may take a while).

```
nice -n 19 autoninja -C out/release
```

12. Build a version of Chrome patched with our proof-of-concept mitigation against the Site Isolation fork-bomb (this may take a while). Pass the contents of `~/isolated-and-exhausted/site_isolation_patch/gn.args` as arguments when invoking `gn args`.

```
git apply ~/isolated-and-exhausted/
    ↪ site_isolation_patch/si_patch.diff
gn args out/patched
gn gen out/patched
nice -n 19 autoninja -C out/patched
```

A.3.2 Basic Test

Testing Connectivity After successfully performing the setup (cf. subsection A.3) the lab network should be configured as shown in Figure 1. To verify the setup make sure that all hosts can communicate with each other by issuing `ping` commands. Note that Windows 10 will not respond to incoming ICMP echo requests but should be able to receive ICMP

echo responses from all other hosts in reaction to requests sent from the victim system. If mutual communication between all hosts works, ensure that DNS resolution works on the victim system. Open the windows command prompt (`cmd.exe`) and issue the command `nslookup evil.com`. You should receive a response resolving the domain to the IP address `2001:db8::1`.

Testing the Attacker Website To test the attacker web server, open a browser on the victim host and navigate to http://evil.com. You should see the website shown in Figure 2. If you properly installed the root CA certificate from the Isolated and Exhausted repository you should also be able to access the same website via HTTPS on port 443 without receiving a self-signed certificate warning message.

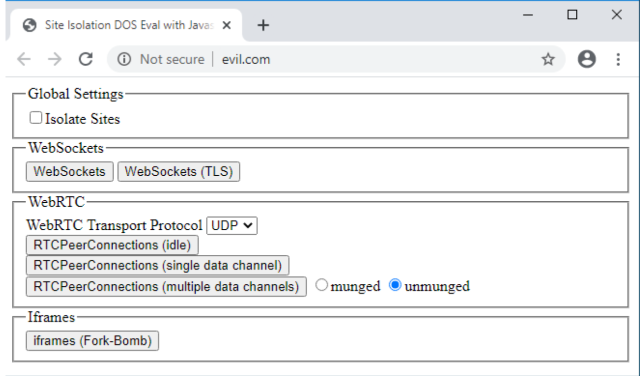


Figure 2: A website to test Site Isolation exploits. The website can be used to allocate UDP ports using WebRTC and to execute a Site Isolation fork-bomb.

Testing the Automated DEMONS Experiment Open an instance of the *Task Manager*, select the performance tab and focus the CPU graph. Next, open the windows command line (`cmd.exe`) and change your working directory to the victim host folder inside the Isolated and Exhausted artifact repository and execute the `attack_simulator.py` script.

```
1 c:
2 cd \Users\user\Documents\isolated-and-exhausted\hosts\
    ↪ victim
3 python attack_simulator.py
```

Once the `attack_simulator.py` script is running, an instance of the configured browser (or the malware attacker) should be started. In the Task Manager, you should be able to observe the CPU load spiking for a couple of seconds at the same time, the number of open handles will rise to a value around 130000 but little to no network traffic is observable during the DEMONS setup phase. Once the setup phase completes, CPU load will drop significantly. At the same time spoofed DNS responses sent by the poisoner consume a moderate amount of downstream bandwidth.

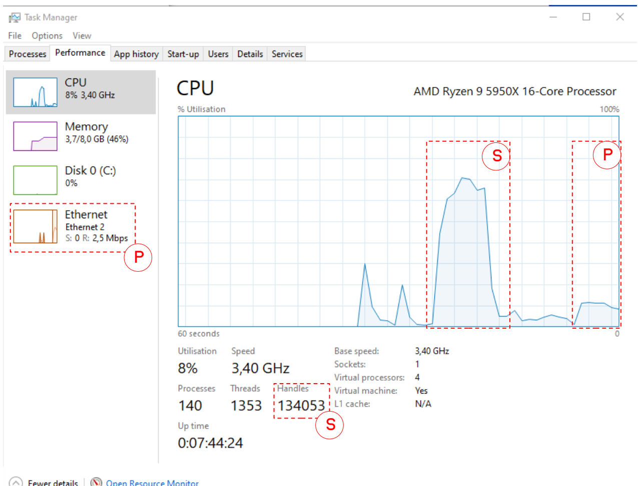


Figure 3: During the DEMONS Setup Phase (S) the CPU load and number of handles increase significantly, the Poisoning Phase (P) causes moderate CPU and network load.

Testing the Custom Chromium Build. Boot the Chromium Build Environment, open a console and change your working directory to the subfolder `src` inside the chromium repository. Then run both the unpatched and the patched version of Chromium. In both cases, you should be presented with a Chromium browser window. Make sure that you can access the Chromium developer tools.

```
1 cd ~/chromium/src
2 out/release/chrome
3 out/patched/chrome
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The Site Isolation fork-bomb can be used to implement DoS attacks against the operating system or the web browser.

(C2): Currently none of the tested browsers mitigates the Site Isolation fork-bomb. We propose an effective mitigation and implemented a proof-of-concept patch based on Chromium 101.0.4951.64. Our patch is efficient and does not measurably affect the browser's performance.

(C3): The impact of Site Isolation attacks goes beyond DoS. ***D****NS-Poisoning by* ***E****xhaustive* ***M****isappropriation* ***o****f* ***N****etwork* ***S****ockets (DEMONS)*, a DNS Cache Poisoning attack uses Site Isolation to poison the DNS cache of the Windows operating system, in the web attacker model.

A.4.2 Experiments

(E1): [Site Isolation Fork-Bomb (ad C1)] [1 human-hour]

```
1   START;END;RESULT;BURSTS;DURATION;CACHE_CONTENT
2   2022-09-19 11:52:20.508692;2022-09-19 12:02:23.901948;SUCCESS;5779;603;2001:db8::1|2001:db8::1
3   2022-09-19 12:02:33.976641;2022-09-19 12:36:50.096716;SUCCESS;20146;2056;2001:db8::1|2001:db8::1
4   2022-09-19 12:37:00.154377;2022-09-19 12:40:34.654990;SUCCESS;1998;214;2001:db8::1|2001:db8::1
5   2022-09-19 12:40:44.747582;2022-09-19 13:25:14.253180;ABORT;25000;2669;None
6   ...
```

Listing 1: Example excerpt of the DEMONS result log on the attacker server. Each line represents an iteration of the DEMONS experiment against the victim host and contains the following values in order from left to right: start date of the experiment end date of the experiment, outcome, duration of the experiment in seconds, the IP address found in the victim's cache for the target domain.

Preparation:

1. Boot the router, the benign DNS, the attacker server and the victim host.
2. Log into the victim system.
3. Start the Windows TaskManager and monitor the browser processes on the "Processes" tab.

Execution: Open Chrome 83.0.4103.106 and perform the following steps:

1. Visit the attacker web site in the lab `http://evil.com:80` (see Figure 2).
2. Note the number of browser processes running
3. Make sure the checkbox `Isolate Sites` is unchecked.
4. Click the button labeled `iframes (Fork-Bomb)`
5. Note that the number of browser processes has not changed significantly.
6. Make sure the checkbox `Isolate Sites` is checked.
7. Click the button labeled `iframes (Fork-Bomb)`
8. Observe the number of browser processes increase significantly until it stalls after some time.
9. If necessary kill the browser process or reset the victim system.

Results: Once the victim system is no longer able to create new browser processes the browser may crash and/or the victim OS becomes unusable. For a detailed description of effects we observed during our evaluation please refer to Table 5 in Appendix C of our work.
The number of processes created depends on the hardware, host OS, browser, browser version, and swap configuration. Even using identical hard- and software, the number of processes varies between runs. Table 2 of in our work lists the median of five measurements.

(E2): *[Site Isolation Fork-Bomb mitigation (ad C2)] [1 human-hour]*

Preparation: Start the Chrome Build Environment (cf. section A.3.1 Chrome Build Environment)

Execution: To verify that the changes introduced by our patch do not measurably impact the browser's performance, record the page load times of the Tranco-Top 5 websites using the profiling tools integrated into Chromium[12] (see Table 1).
Create at least one series of measurements for each Chromium and Chromium-Site Isolation-patched[13] by performing the following steps:

- Start the browser.
- Open the performance tab of the integrated developer tools.
- Open the website.
- Run the profiler and discard the initial result to avoid any caching effects.
- Run the profiler and calculate $t_{load} = t_{total} - t_{idle}$, where t_{total} is the total time recorded by the profile and t_{idle} is the time the browser was idle during loading.
- Repeat the previous step four more times.

Website	Page load time in ms Chromium 101.0.4951.64	
	unpatched	patched
google.com	752.1 ms	766.4 ms
youtube.com	5366.4 ms	5265.5 ms
facebook.com	643.3 ms	629.5 ms
netflix.com	1107.3 ms	1039.6 ms
microsoft.com	1305 ms	1243.3 ms

Table 1: The average page load time (in ms) of the Tranco-Top-5 websites shows no significant difference between an unpatched Chromium 101.0.4951.64 and the same browser version patched with our mitigation against the Site Isolation fork-bomb.

Results: Table 1 shows the average loading times for the Tranco-Top-5 websites using Chromium and Chromium Site Isolation-patched. There is no significant difference in performance between both browser versions (see `isolated-and-exhausted/site_isolation_patch/si_patch_performance_t_test.ods`). Since we only need two additional global constants, and one additional

[12] `https://developer.chrome.com/docs/devtools/evaluate-performance/?utm_source=devtools`

[13] To determine the page load time we recorded two series of measurements on two different days, to reduce the impact of server and network load.

local limit per tab/window, the effect of our patch on browser memory consumption is negligible.

(E3): [DNS Poisoning by Exhaustive Misappropriation of Network Sockets (DEMONS) (ad C3)] [5 human minutes + 12 compute hours]
A victim visits the attacker web site and becomes subject to the DEMONS (attack), if successful poisons the victim's DNS cache. A script automatically repeats the experiment until it is stopped manually.

Preparation:

1. Boot the router, benign DNS, attacker server and the victim host.
2. Log into a shell on the attacker server.
3. Start live monitoring of DEMONS experiment results (cf. Listing 1).

```
cd /srv/docker/data/demons/attacker/spoofer/
    ↪ results/
tail -f $(ls -1r | head -n1)
```

Execution:

1. Open `cmd.exe` on the victim system.
2. Change your working directory to the victim host sub folder in the artifacts repository and run the experiment:

```
c:
cd \Users\user\Documents\isolated-and-
    ↪ exhausted\hosts\victim
python attack_simulator.py
```

Results: DEMONS is probabilistic because the attacker must correctly guess a random 16-bit transaction ID. Given a large enough sample set, the attacker can poison the victim's DNS cache with a success probability of 36% or better. Each DEMONS experiment may have one of three results `SUCCESS` in case the attacker successfully poisoned the victim's DNS cache, `FAILURE` - a DNS response from the benign DNS server was cached by the victim or `ABORT` if neither the attacker nor the benign DNS served a valid record within a preset burst limit.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220912. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenix%20sec2023/.

Pool-Party: Exploiting Browser Resource Pools for Web Tracking

Peter Snyder
Brave Software

Soroush Karami
University of Illinois at Chicago

Arthur Edelstein
Brave Software

Benjamin Livshits
Imperial College London

Hamed Haddadi
Brave Software, Imperial College London

Abstract

We identify class of covert channels in browsers that are not mitigated by current defenses, which we call "pool-party" attacks. Pool-party attacks allow sites to create covert channels by manipulating limited-but-unpartitioned resource pools. This class of attacks have been known to exist; in this work we show that they are more prevalent, more practical for exploitation, and allow exploitation in more ways, than previously identified. These covert channels have sufficient bandwidth to pass cookies and identifiers across site boundaries under practical and real-world conditions. We identify *pool-party* attacks in all popular browsers, and show they are practical cross-site tracking techniques (i.e., attacks take 0.6s in Chrome and Edge, and 7s in Firefox and Tor Browser).

In this paper we make the following contributions: first, we describe *pool-party* covert channel attacks that exploit limits in application-layer resource pools in browsers. Second, we demonstrate that *pool-party* attacks are practical, and can be used to track users in all popular browsers; we also share open source implementations of the attack. Third, we show that in Gecko based-browsers (including the Tor Browser) *pool-party* attacks can also be used for *cross-profile* tracking (e.g., linking user behavior across normal and private browsing sessions). Finally, we discuss possible defenses.

1 Introduction

Browser vendors are increasingly developing and deploying new features to protect privacy on the Web. These new privacy features address the most common ways users are tracked on the Web: partitioning DOM storage to prevent tracking from third-party state, randomization or entropy reduction to combat browser fingerprinting, network state partitioning to prevent cache-based tracking, etc.

However, research has documented other ways Web users can be tracked, though in ways that may be difficult to conduct under realistic browsing conditions. Significantly among these are covert-channels that can be constructed through timing signals, or other side channels. These covert-channels allow sites to communicate with each other—or even other applications—in ways not intended by browsers. Such covert-channels can be used to reintroduce the kinds of cross-site tracking attacks the above-discussed browser protections were designed to prevent.

Browser vendors have responded to covert-channels in a variety of ways. Some covert-channels (e.g. timing signals from abusing HTTP cache state) have been addressed through platform wide improvements like network state partitioning. Other covert-channels have been addressed—or at least mitigated—through other protections, like isolating sites in their own OS processes. Others attacks have been left unaddressed, because browser vendors judge them to be impractical to execute in realistic browsing scenarios.

In this work we demonstrate that current browser protections are insufficient to prevent sites from using covert-channels to circumvent anti-tracking protections in browsers, including the protections deployed by the most privacy-focused browsers. We demonstrate this by defining a new category of techniques for constructing covert-channels, by exploiting the state of limited-but-unpartitioned resource pools in the browser. Because such covert-channels are exploited by two parties colluding in the same resource pool, we call this category of covert-channel "pool-party" attacks.

"Pool-party" attacks create covert-channels out of browser-imposed limits on pools of resources. When resource pools are limited (i.e. the browser only allows pages to access resources up to some hard limit, after which requests for more resources fail) and unpartitioned (i.e. different sites consume resources from a shared pool), sites can consume and release resources to leak information across security boundaries. Examples of such boundaries include site boundaries (e.g. the browser intends to prevent site A from communicating directly with site B) and profile boundaries (e.g. the browser intends sites visited in a "standard" browsing session to not be able to learn about sites visited in an "incognito" browsing session). More generally, attackers can use these covert-channels to conduct the kinds of cross-site tracking that the recent

browser features were intended to prevent.

We identify practical "pool-party" attacks in all popular browsers, both in browsers' default configurations, and non-standard, hardened configurations. We demonstrate "pool-party" attacks through three resource pools: WebSockets, Server-Sent Events, and Web Workers, and find that all browsers were vulnerable to at least one form of attack. We further identify other limited-but-unpartitioned resource pools in browsers that could be leveraged for "pool-party" attacks. Examples of such pools include certain kinds of resource handle (e.g. Web Speech API), or limits on how many network requests (distinct from network connections) can be in flight at once (e.g. DNS resolution), among others. Finally, we demonstrate that "pool-party" attacks are not just *theoretical* threats to user privacy, but *practical* threats that can be used to track users across sites. We show that in Gecko-based browsers (including the Tor Browser), "pool-party" attacks can create covert-channels *across profiles*, allowing sites to link behaviors in "private browsing" modes with standard, long term browser identities.

These findings are important for the development of browser partitioning. All browser engines support some forms of partitioning: WebKit partitions DOM storage and some kinds of network state, Gecko partitions DOM storage and network state, and Chromium partitions network state[1]. Brave has extended Chromium to also partition DOM storage.

1.1 Contributions

This work makes the following contributions:

- We **define a new category of technique for creating covert-channels in browsers**. We call this category of covert-channel "pool-party" attacks, and describe how the approach differs from the kinds of privacy attacks browsers currently aim to defend against;
- We **evaluate deployed browsers**, and find all popular browsers and browser engines are vulnerable "pool-party" attacks;
- We provide three **open-source, proof-of-concept** implementations of our attack that work in all browsers[2];
- We perform a **performance measurements** to evaluate the bandwidth and practicality of "pool-party" attacks, and find that "pool-party" attacks are a practical basis carrying out cross-site tracking attacks;
- We **discuss potential mitigation strategies** for how browsers could defend against "pool-party" attacks.

[1]At time of writing, network state partitioning is deployed for a portion of Chrome and Edge users, as part of the "NetworkIsolationKey" feature

[2]https://github.com/brave-experiments/pool-party-artifact/blob/master/static/inner.js

1.2 Responsible Disclosure

We have presented our findings to the following browser vendors (in alphabetical order): Apple, Brave, Google, Microsoft, Mozilla, Opera, Tor Project. All reports were made over 90 days in advance of this submission.

Microsoft and Opera responded that since the discussed vulnerabilities were in Chromium, they would wait for Google to address the problem. The Tor Project similarly said they would rely on Mozilla to address the vulnerabilities[3].

Some vendors have shipped fixes for the vulnerabilities identified in this work. Safari fixed the SSE event vulnerability in version 15.2[4], and Brave has released fixes for the WebSocket[5] and SSE[6] vulnerabilities.

Google[7] and Mozilla[8] also plan to address these vulnerabilities, though have not done so yet. These organizations are focusing on a mixture of browser-wide fixes (i.e. comprehensively partitioning all resource pools, not only the resource pools discussed in this work) and updates to Web standards (e.g. defining limits and the scope of connection pools).

2 "Pool-party": Definition and Background

This section provides context for how "pool-party" attacks relate to other Web tracking techniques, and how browser vendors' privacy models and goals have changed. This section also describes *why* existing browser protections fail to protect users against "pool-party" attacks.

This section first defines "Web tracking", followed by discussing how privacy models in Web browsers have improved, and why "pool-party" attacks allow trackers to violate intended privacy boundaries in all popular browsers. Next, we describe how "pool-party" attacks relate to both i) other covert-channels in browsers, and ii) conventional Web tracking techniques. We then explain why existing browser protections do not protect users against "pool-party" attacks, and conclude by describing how this work relates to a category of attack previously known to be possible, but not thought to be practical.

2.1 Web Tracking and Cross-Site Tracking

This sub-section gives a working definition of Web tracking. Our goal is not to provide a formal, unambiguous definition (the phrase "Web tracking" is used too broadly to likely allow for one), but instead to give a practical definition to build on through the rest of this work.

[3]https://gitlab.torproject.org/tpo/applications/tor-browser/-/issues/41381

[4]https://support.apple.com/en-us/HT212982

[5]https://github.com/brave/brave-core/pull/11609

[6]https://github.com/brave/brave-core/pull/16882

[7]https://bugs.chromium.org/p/chromium/issues/detail?id=1249658

[8]https://bugzilla.mozilla.org/show_bug.cgi?id=1730797

We use "Web tracking" to refer to a user being re-identified across conceptual contexts, without the user's expectation or consent. We use "context" to refer to a grouping of activities that the user expects to be separate from, and not accessible to, other similar contexts. This definition is similar to the W3C's proposed privacy principals[9].

Contexts might be divided by *time* (e.g. a site re-identifying a user revisiting the same site a week after first visiting, despite the user clearing browsing data), *application* (e.g. a site re-identifying a user visiting a site in Safari as the same user who previously visited the site in Chrome), *profile* (e.g. a site identifying that the visiting in an private/incognito browser session is the same user visiting the site in a standard browser session), or *site* (e.g. two sites colluding to learn that browser sessions occurring on each site belong to the person). The commonality is the users' reasonable expectation that things that happen in one context are not readily known and available to other contexts.

2.2 First-Party Site as Privacy Boundary

Browser vendors are converging on the first-party site as the Web's privacy boundary. All browsers include features intended to prevent sites from communicating across first-party site boundaries. Some browsers enforce this boundary by default; others only do so with opt-in "privacy" modes, but all browsers include such features.

Using the first-party site as a privacy boundary means that a third-party embedded under two different first-parties should not be able to confidently know it was the same person visiting each site, unless the user intentionally re-identifies themselves to the third-party.

The rest of this subsection documents that, and in what configurations, each browser uses the first-party site as their privacy boundary. In all the discussed configurations, browsers intended to communication across first-party site boundaries, and in all cases attackers can circumvent the intended privacy boundary through "pool-party" attacks.

Gecko Browsers. Both Firefox (as of version 103) and Tor Browser enforce the first-party site as the privacy boundary by default. In Tor Browser, the protection is sometimes called "first-party isolation" or "cross-origin identifier unlinkability"[10]. In Firefox, the feature is called "Total Cookie Protection"[11].

WebKit Browsers. Safari uses the first-party site as the privacy boundary by default. The WebKit documentation makes this privacy boundary explicit in their documentation, which mentions that they intended to protect against cross-site communication through covert-channels[12].

Chromium Browsers. Chromium *does not* enforce the first-party site as a privacy boundary by default. However, Chromium allows for configurations that do, by a combination of i) disabling third-party cookies (to prevent DOM storage communication across site boundaries) and ii) enabling Chromium's "`NetworkIsolationKey`" (NIK)[13] features (which partition caches and other network state by first-party).

Neither Chrome or Edge disable third-party storage by default, but both do enable NIK features for most users. We note though that even when Chrome and Edge are configured to use the first-party site as the privacy boundary, those browsers are vulnerable to "pool-party" attacks.

The Brave Browser uses a modified version of Chromium that, by default uses the first-party site as a privacy boundary. It does this by partitioning third-party DOM storage by first-party[14], and by enabling (and extending) Chromium's NIK system for all users[15].

2.3 Description of "Pool-party" Attack

"Pool-party" attacks manipulating pools of browser resources which are limited (i.e. the browser restricts how many of the resource can be used at one time) and unpartitioned (i.e. different contexts consume resources from the same pool). While the examples focused on in this work utilize either limited-but-unpartitioned pools of i) network connections or ii) thread handles, browsers include many other limited-but-unpartitioned resource pools that could be similarly exploited, such as pools of file handles, subprocesses, or other resource handles.

A "pool-party" attack occurs when parties operating in distinct contexts (contexts the user expects to be distinct and blinded from each other) intentionally consume and query the availability of the limited resources in a resource pool, to create a cross-context communication channel. Each context can then use the communication channel to pass an identifier, allowing each party to link the user's behavior across the two contexts. We note again that most commonly the two contexts considered here are two different websites running in the same browser profile, but could also be the same (or different) websites running in different browser profiles.

Algorithm 1 presents a simple-though-limited technique for conducting a "pool-party" attack, where sites can trivially transform this optimization choice into a cross-site tracking mechanism.

[9] https://w3ctag.github.io/privacy-principles/

[10] https://2019.www.torproject.org/projects/torbrowser/design/#identifier-linkability

[11] https://blog.mozilla.org/security/2021/02/23/total-cookie-protection/

[12] https://webkit.org/tracking-prevention-policy/

[13] https://github.com/shivanigithub/http-cache-partitioning#choosing-the-partitioning-key

[14] https://brave.com/privacy-updates/7-ephemeral-storage/

[15] https://brave.com/privacy-updates/14-partitioning-network-state/

Algorithm 1 Toy example of a "pool-party" attack.

Site A: $I_a \leftarrow$ *random N bits*
Site B: $I_b \leftarrow$ *empty string*
while $i \leftarrow I_a$ **do**
  Site A: Stop any playing videos
  if $I_a[i] = 1$ **then**
    Site A: Play a video
  end if
  Wait 5 seconds
  if Site B is able to play a video **then**
    Site B: $I_b[i] = 1$
  else
    Site B: $I_b[i] = 0$
  end if
end while

For this toy example, assume a browser vendor wants to improve performance by only allowing one video element to be loaded at a time, across all sites. If a video is currently playing on any page, the site will receive an error if it tries to play a new video. An attacker use this implementation choice to "send" a bite across site-boundaries by playing (or not) a video on one site, and checking on another site whether there is a video playing. An arbitrarily large message can be sent by repeating this process. A more realistic and efficient technique is presented in Section 3.

2.4 Relationship to Other Covert-Channels

"Pool-party" attacks differ from other covert-channel attacks by targeting intentional, application-imposed limits. This differs from many other covert-channel attacks in two ways, both of which increase the practicality of "pool-party" attacks.

First, "pool-party" attacks target application-level resources, while many other covert-channels target parts of the system below, or at least distinct from, the application (e.g. hardware restrictions like CPU caches, OS details like interrupt schedules or memory management, or language runtime features like garbage collection). This is significant because, the lower in the stack the attacker targets, the more likely the resource is (all other things being equal) to be shared with other actors on the system. This means that lower-level covert-channels are more likely to be noisy, and so more difficult to communicate over.

Second, related but distinct, "pool-party" attacks target browser-managed resources, resources that are, in most cases, intentionally shielded from other applications on the system. This again reduces the chance that colluding parties will have to contend with a noisy, unpredictable covert-channel.

2.5 Relationship to Other Tracking Methods

"Pool-party" attacks do not fall neatly into the categories usually used to describe browser tracking techniques. This subsection briefly describes the rough-taxonomy used in online-tracking research, and why "pool-party" does not cleanly fall into existing categories.

Browser	DOM Storage	Network State
Brave	$\oplus$	$\oplus$
Chrome	$\times$	$\ominus$
Edge	$\times$	$\ominus$
Firefox	$\oplus$	$\oplus$
Safari	$\oplus$	$\oplus$
Tor Browser	$\oplus$	$\oplus$

Table 1: State partitioning features in popular browsers (in alphabetical order). $\oplus$, $\ominus$ and $\times$ indicate the feature being available by default for all, some, or no users, respectively.

Stateful Tracking. "Stateful tracking" most commonly refers to websites using explicit storage APIs in the Web API (e.g. cookies, localStorage, indexedDB) to assign identifiers to browser users, and then read those identifiers back in a different context, to link the identity (or, browser behavior) across those contexts.

Stateful-tracking also describes other ways websites can set and read identifiers, by using APIs and browser capabilities not intended for such purposes. Examples of such techniques include exploiting the browser HTTP cache, DNS cache or other ways of setting long term state (e.g. HSTS instructions [37], favicon caches [35], or, ironically, storage intended to prevent tracking [13]).

Browsers increasingly protect users from stateful tracking by partitioning storage by context, mostly commonly be the effective-top level domain (i.e. `eTLD+1`) of the website. Giving each context a unique storage area prevents trackers from reading the same identifier across multiple contexts, and so prevents the tracker from linking browsing behaviors in different contexts. Partitioning explicit storage APIs is often referred to as **DOM Storage** partitioning. Partitioning caches and other "incidental" ways sites can store values is often called **network state partitioning**. Table 1 provides a summary of state partitioning in popular browsers.

Browser state partitioning strategies fail to defend against "pool-party" attacks because "pool-party" attacks do not rely on setting or retrieving browser state (at least not in the way state is generally discussed in this context, meaning the ways that sites can write state to the users profile). "Pool-party" attacks instead rely on implementation details of browser architecture, where device resources are limited-but-unpartitioned. While partitioning strategies could also be used to defend against "pool-party" attacks, as discussed in Section 5, certain aspects of these attacks make partitioning approaches difficult in practice.

Stateless Tracking. "Stateless tracking," (also often called "browser fingerprinting") refers to the category of Web track-

Browser	Coordination	Stability	Uniqueness
Brave	⊕	⊕	⊕
Chrome	×	×	×
Edge	×	×	×
Firefox	⊕	×	⊖
Safari	×	×	⊕
Tor Browser	×	×	⊕

Table 2: Stateless tracking protections in popular browsers (in alphabetical order). ⊕, ⊖ and × indicate the defense is available by default, off by default, or not available, respectively.

ing techniques whereby the attacker constructs a unique identifier for the user by combining a large number of semi-distinguishing browser and environmental attributes into a stable, unique identifier. Examples of such semi-distinguishing features include the operating system the browser is running on, the browser version, the names and the number of plugins or hardware devices available, and the details around the graphics and audio hardware present, among many others [19].

In contrast to "stateful" tracking techniques, "stateless" techniques do not require sites to be able to set and read identifiers across context boundaries, and so are robust to storage partitioning defenses. Stateless attacks instead rely on three conditions to be successful:

- **Coordination:** code running in different contexts must know to query the same (or at least sufficiently large intersection of) browser attributes.
- **Stability:** the browser must present the same values for the same semi-distinguishing attributes across contexts (otherwise the browser will yield different fingerprints in different contexts, preventing the attacker from matching the two fingerprints).
- **Uniqueness:** the browser must present enough semi-distinguishing attributes to allow the site to accurately differentiate between users (otherwise the attack will confuse two different users as the same person)

Browsers defend against "stateless" trackers by attacking any of these three requirements. A browser might prevent **coordination** by blocking fingerprinting code on sites, or prevent **stability** by making the browser present different attributes to different sites (such as in [18, 25]), or prevent **uniqueness** by reducing the entropy provided by each attribute. Table 2 provides a summary of deployed "stateless" defenses in popular browsers.

Browser defenses against "stateless" tracking techniques fail to defend against "pool-party" attacks because of differences in the nature of the attack. "Stateless" techniques target stable semi-distinguishing browser characteristics which are set by a page's execution environment. "Pool-party" attacks, in contrast, are enabled by sites consuming and reading the availability of limited resources in the browser across execution contexts. Therefore, unsurprisingly, browser defenses against "stateless" tracking provide no protection against "pool-party" attacks.

XS (Cross-Site) Leaks. "Pool-party" attacks are most similar to a category of attack loosely called "XSLeaks"[16], a broad collection of ways sites can send signals to each other in ways generally unintended by browser vendors. However, we note that in contrast to "stateful", "stateless," "pool-party" attacks, XSLeaks do not have a common cause or remedy; instead, XSLeaks can be largely thought of as a catchall for cross-site (or cross-context) techniques that do not not fit in another category. Examples of XSLeaks include timing channels stemming from a variety of causes, unintended side effects of experimental browser features[17], or misuse of other browser APIs[18].

The lack of a common cause of XSLeaks makes it impossible to generalize about defensive strategies or deployed browser defenses. Recent work in this area has identified ways sites can leak information across browser-imposed boundaries, including through unintended side effects in how browsers handle errors, implement cross-origin opener-policy (COOP), cross-origin resource policy (CORP), and cross-origin read blocking (CORB) policies, or limit the length of redirection chains, among many other signals [16].

We note though that "pool-party" attacks are most common to the "connection pool" attacks identified by the XSLeaks project[19]. This work makes the following contributions beyond the issues documented by the XSLeaks project, and the related work done by Kinttel et al. [16].

1. This work defines a larger category of attack than XSLeaks, where any limited-but-unpartitioned resource pool can be transformed into a covert-channel. The attack documented by the XSLeaks project is a subset of the larger category of attack discussed in this work. Network connection pools can be abused to conduct "pool-party" attacks, but other kinds of resource pool can too. For example, the Web Workers pool in Firefox thecan be exploited to conduct "pool-party" attacks. Section 5.3 identifies additionalm non-network-connection pools that can be exploited.

2. We demonstrate that attacks of this type are not just theoretically possible, but are practical, and are real-world threats to Web privacy.

3. The "connection pool" attack identified by the XSLeaks project relies on abusing the connection pool to create timing channels, while "pool-party" attacks utilize the

[16] https://xsleaks.dev/

[17] e.g. https://xsleaks.dev/docs/attacks/experiments/scroll-to-text-fragment/

[18] e.g. https://xsleaks.dev/docs/attacks/window-references/

[19] https://xsleaks.dev/docs/attacks/timing-attacks/connection-pool/

number of available resources in the pool to create the communication channel. This small difference is significant. Using the amount of available of resources in the pool as the communication channel makes the attack more robust to noise introduced from other sites, and mitigations against one form of the attack may not apply to the other.

3 Generic "Pool-party" Attack Algorithm

In the previous section we presented the category of "pool-party" attack in the abstract, explained why "pool-party" attacks are different from other attacks discussed previously in the literature, and why current browser defenses fail to protect against "pool-party" attacks. In this section we present a generic algorithm for conducting "pool-party" attacks, which can then be applied to any resource pool in a browser where the following conditions are met.

1. The resource pool is **limited**, meaning that sites can request resources from the pool until a global limit is hit, after which sites are prevented from accessing more resources, in a manner the site can detect.
2. The resource pool is **unpartitioned**, meaning that different contexts (e.g. sites, profiles, etc.) all draw from the same global resource pool. Put differently, the attack will fail if each context gets a distinct resource pool.
3. Sites can **consume** resources from the pool without restrictions, as long as the pool is not already exhausted.
4. After consuming resources, sites can **release** any number of those resources back into the pool.

Any resource pool where the above four criteria are met can be transformed into a covert communication channel between any two parties sharing the resource pool.

We have identified resource pools matching the above criteria in current versions of all popular browsers, even browsers that particularly emphasize their privacy features (e.g. Brave Browser, Tor Browser), and even when browsers are "hardened" by the enabling of non-default, privacy-focused features (as discussed in Section 2.5).

3.1 "Pool-party" Algorithm

We present a generic protocol for conducting a "pool-party" attack over limited-but-unpartitioned resource pools in all browsers. This protocol is presented as Algorithm 2, and provides a generic way a site can use a limited-but-unpartitioned resource pool to track users.

Protocol Inputs. The algorithm takes several inputs. First, the algorithm takes which resource pool will be exploited to conduct the attack, which also determines the size of the pool.

Algorithm 2 General algorithm for a "pool-party" attack.

Inputs.
POOL_SIZE $\leftarrow$ size of resource pool
PKT_SIZE $\leftarrow \lfloor \log(\mathit{POOL_SIZE}) \rfloor$
MSG $\leftarrow$ binary string to transmit
NEGOTIATE_INTERVAL $\leftarrow$
time to choose sender and receiver roles
PULSE_INTERVAL $\leftarrow$ time to transmit one chunk of data

1. Setup.
CHUNKS $\leftarrow$ MSG split into packets of size PKT_SIZE
RECV_MSG $\leftarrow$ *empty string*
START_TIME $\leftarrow \lceil$ NEGOTIATE_INTERVAL $+$
len(CHUNKS) $*$ PULSE_INTERVAL $\rceil$

2. Determining Initial Sender and Receiver.
Both sites: sleep until START_TIME
Both sites: consume resources until pool is exhausted
if "Site A" is able consume over $> 50\%$ of pool **then**
  SENDER $\leftarrow$ "Site A"
  RECEIVER $\leftarrow$ "Site B"
else
  SENDER $\leftarrow$ "Site B"
  RECEIVER $\leftarrow$ "Site A"
end if
Sender: consumes 100% of pool resources
Receiver: releases all pool resources
Both sites: sleep until
START_TIME $+$ NEGOTIATE_INTERVAL

3. Sending Data.
for $i \leftarrow 0..\mathit{len}$(CHUNKS) **do**
  Sleep until START_TIME $+$ NEGOTIATE_INTERVAL $+ i *$ PULSE_INTERVAL
  if SELF == SENDER **then**
    SEND_INT $\leftarrow$ *binaryToDecimal*(CHUNK[i])
    Consume all unheld resources in pool
    Release SEND_INT resources in the pool
  else if SELF == RECEIVER **then**
    Sleep for $0.5 *$ PULSE_INTERVAL
    Consume all unheld resources in pool
    RECV_INT $\leftarrow$ number of consumed resources
    Release all held pool resources
    RECV_STR $\leftarrow$ *decimalToBinary*(RECV_INT)
    RECV_MSG $\|$= RECV_STR
  end if
end for
Release all held pool resources

Attackers can precompute the largest pool available for each browser. The size of the resource pool (i.e. the number of resources in the pool available) is stored as `POOL_SIZE`.

The second input is the message being sent over the channel, which is a binary string of arbitrary length. The binary string to be transmitted is stored as `MSG`.

Third, the algorithm takes two time intervals, stored as

NEGOTIATE_INTERVAL and PULSE_INTERVAL. These intervals could be fixed across all attack methods, and trade faster transmission time (smaller values) against higher reliability (lower values).

Step One: Setup. To begin, the sender splits MSG into PKT_SIZE sized chunks, yielding a vector of bit-strings each of size PKT_SIZE, and the receiver constructs an empty buffer, RECV_MSG, to accumulate the received message into one packet at a time. The sending and receiving sites must choose the same time to start communication: the shared START_TIME is set to the next integer ECMAScript epoch time (in seconds) that is greater than a multiple of the full negotiation and message transmission time.

Step Two: Determining Initial Sender and Receiver. Both parties synchronize by sleeping until the START_TIME, and then determine which site will be the initial *sender* and which the *receiver*. This negotiation is needed because, neither site initially knows what other colluding site(s) may be open and available to communicate with, and thus no way of assigning roles in the protocol.

Sites determine *sender* and *receiver* by racing to exhaust the resource pool. The site that is able to consume more than 50% of resource in the pool assigns itself the role of initial *sender*; the site that is prevented from requesting the $50\%+1$ resource assigns itself as the initial *receiver*.

The *receiver* then releases the resources it holds, and the *sender* keeps consuming resources until the pool is exhausted.

Step Three: Sending Data. The third step of the protocol is where passing data across context (i.e. site) boundaries occurs. The *sender* and the *receiver* participate in this step of the protocol as follows.

The *sender* manipulates the state of the resource pool as follows for each c in their CHUNKS vector (recall that CHUNKS is a vector of binary strings, each of length PKT_SIZE). For each c, the *sender* first interprets the binary as positive integer representation (e.g. 0010010 becomes 18, etc), which is stored as SEND_INT. The *sender* then releases SEND_INT $+1$ resources from the pool and waits for a fixed period, PULSE_INTERVAL, to ensure that the the *receiver* has had time to read from the channel. Once the *sender* has finished sending their message, the *sender* releases all resources in the pool and proceeds to the next step in the protocol. Otherwise, the *sender* consumes all resources in the pool and repeats the current stage in the protocol to send the next c value.

Simultaneously, the *receiver* begins this stage of the protocol by waiting for PULSE_INTERVAL/2. Once that time has elapsed, the *receiver* tries to consume as many resources as possible, which will match the SEND_INT number of resources released by the *sender*, and stores this value (minus 1) as RECV_INT[20] Next, the *receiver* encodes RECV_INT in the inverse manner the *sender* used (e.g. 18 becomes 0010010), and concatenates the result onto the *receiver's* RECV_MSG. The *receiver* then releases all resources it holds, waits for the end of the pulse, and repeats the above process.

Step Four: Exchanging Roles. Finally, if desired, the two parties can exchange roles to pass data in the *receiver* to *sender*. This is trivially accomplished by each party assuming the opposite role, and continuing again from step 3. Otherwise, if there is no more data to transmit, both parties can abort the protocol. Note that the protocol itself does not provide a mechanism for the parties to indicate whether they wish to continue or end the protocol, though parties could easily signal such through the contents of the messages being passed.

4 Evaluation in Popular Browsers

In the previous section we presented a generic algorithm for turning limited-but-unpartitioned resource pools into cross-context communication channels, which in turn can be used to cookie-sync and track users across the Web. In this section we demonstrate three examples of such limited-but-unpartitioned resource pools in popular browsers, and measure how exploitable and practical they are for cross-site tracking.

Specifically, we show that practical forms of "pool-party" attacks can be carried out in popular browsers. We implemented three examples of "pool-party" attacks, using the WebSockets, Server Sent Events (SSE), and Web Workers APIs. Before this work, all Chromium browsers were vulnerable to the WebSockets and SSE attacks, Firefox was vulnerable to the Web Sockets and Web Workers attacks, Tor Browser was vulnerable to the WebSockets attack, and Safari was vulnerable to the SSE attack.

We assess the practicality of each of implemented "pool-party" attack through four measurements: **Availability**, the size of the relevant resource pool, and the kind of context-linking possible, **bandwidth**, or how long it takes to send a 35-bit identifier through the channel, **consistency**, or how often the identifier is sent correctly, and **background noise**, or how often sites on the Web use resources in each resource pool.

Finally, for all measurements of Chromium browsers, we configured each browser to enable all site-as-privacy-boundary features enabled (i.e. we enabled all browser features designed to allow communication across sites or site-partitions). Specifically, we disabled third-party cookies, and enabled all "Network Isolation Key" features to enforce cache and network-state partitioning (see Section 2.5 for more information on these "hardened" Chromium configurations see). We note the exception here was Brave, which partitions DOM storage and network state by default.

[20]Recall that, by construction, the *sender* is not able to obtain more than 2^{PKT_SIZE} resources, and so the *receiver* can be certain that values greater than the limit, or equal to zero, are not data the *sender* is attempting to transmit. If so, the *receiver* will exit the protocol.

4.1 Attack Availability

Methodology. We first checked the availability (and so, exploitability) of each example “pool-party” attack by experimenting with browsers and examining the source code of each browser engine to identify limited-but-unpartitioned resource pools in those browsers.

Specifically, we considered Web APIs that might hypothetically represent a finite pool of resources (network connections, threads, etc.). We then used the developer console in each browser to manually test whether each Web API would leak and whether that resource pool could be predictably exhausted. We conducted these tests as follows:

1. We opened a new tab,. visited a blank page, and checked if instantiating the candidate Web API repeatedly in a loop in the developer console caused errors after a finite and predictable number of calls.

2. Once that pool was exhausted, we then opened a second tab to a different site. Did we find that no more resources were available under the second site?

3. If we released a resource under the first tab, could a single resource now be consumed without error under the second tab?

4. Were we able to find logic in the corresponding browser engine code that was imposing this resource pool limit?

If the answer to all four of these questions was yes, then we concluded this Web API was vulnerable to the pool party attack for the tested browser. We thus proceeded to implement attacks against each vulnerable browser based on the algorithm presented in Section 3, implemented in JavaScript[21]. We then examined whether we could use the relevant resource pool to create a covert-channel and communicate across site boundaries, across profile boundaries, or both. Importantly, we tested whether the attack technique can be used to communicate between a standard-browsing profile, and a “private browsing mode” profile[22].

Results. Our *availability* measurements yielded several significant findings, summarized in Table 3.

First, we were able to identify exploitable limited-but-unpartitioned resource pools in all major browsers, which we were successfully able to exploit through “pool-party” attacks (though Safari and Brave both fixed some vulnerabilities during the “responsible disclosure” process). As noted, the resource pools targeted in each browser engine differ. We were able to use the relatively large WebSockets connection pool in Chromium- and Gecko-based browsers to conduct “pool-party” attacks. Safari’s WebSockets implementation was not exploitable, since WebKit does not restrict how many WebSocket connections can be opened simultaneously. Safari’s implementation of the SSE API, though, was previously exploitable before they fixed it. (Gecko’s implementation of the SSE API *was not* exploitable).

Firefox alone was vulnerable to the Web Workers form of the attack (a surprising finding given that Tor Browser uses the same Gecko engine).

Second, we found that Gecko-based browsers (i.e. Firefox and Tor Browser) were vulnerable to “pool-party” attacks in a way more concerning than other browser engines. While “pool-party” attacks can be used for cross-site tracking in all browsers, **in Gecko-based browsers “pool-party” attacks can be used to track users *across profiles***. Significantly, this means that, in Gecko-based browsers, sites can conduct “pool-party” attacks between private browsing sessions and standard browsing sessions. More concretely, a site running in a private browsing window can collude with a site running in a standard browsing window, and identify both sessions as belonging to the same person. This is particularly concerning since it violates the core promise of a private browsing session; that behaviors conducted using a private browsing are “ephemeral”, and cannot be linked other accounts or behaviors a user maintains. Additionally, this vulnerability undermines the work and research that has been done to strengthen private browsing modes in browsers (for example, [4, 8, 21]).

4.2 Attack Bandwidth

Methodology. We measured the bandwidth of each attack by measuring how long each implemented “pool-party” attack took to transmit a 35-bit string across the site (or in the case of Firefox’s WebSockets implementation, profile) boundaries.

We selected a 35-bit string for two reasons. First, because it is over 33-bits, or what is needed to uniquely identify the approximately 7.9 billion people on the planet, and two, because it aligns cleanly with the 5-bit packet size used in WebSocket and Web Worker experiments.

We conduct each measurement as follows. First, we manually open two tabs on a browser to two pages on two different sites we controlled. Each page includes an implementation of the relevant “pool-party” attack (Websockets and SSE in Chromium-based browsers, WebSockets and Web Workers in Gecko-based browsers, and SSE in Safari), implemented through JavaScript included in the page. We then experimentally varied the negotiation time and pulse time until we found the minimum times necessary to ensure that messages were passed accurately with a high success rate. We then conduct this measurement 100 times, using a clean browser profile for each measurement, and report the average.

Results. We report the results of our bandwidth measurements in Table 4. Times are reported in seconds, and the

[21] https://github.com/brave-experiments/pool-party-artifact/blob/master/static/inner.js

[22] This feature goes by different names in different browsers, but generically refers to the ability to run the browser in a way where stored values only last the lifetime of the browsing session.

Browser	Engine	Version	WebSockets	Web Workers	Server-Sent Events
Brave	Chromium	1.44.101	∗ 255	-	1,350
Chrome	Chromium	105.0.5195.125	255	-	1,350
Edge	Chromium	106.0.1370.42	255	-	1,350
Firefox	Gecko	105.0.1	† 200	512	-
Safari	WebKit	15.2	-	-	∗ 6
Tor Browser	Gecko	11.5.2	200	-	-

Table 3: **Attack Availability**: Size of each resource pool used to conduct each instance of a "pool-party" attack. "-" denotes that the browser was not vulnerable. "∗" indicates that the vulnerability was fixed before this work was submitted. "†" denotes that the resource-pool can be exploited to conduct cross-profile attacks (in addition to cross-site).

Browser	Method	Setup	Send	Total	Success
Brave	SSE	3.0	5.0	8.0	100%
Chrome	SSE	2.0	5.0	7.0	100%
Edge	SSE	2.0	5.0	7.0	100%
Chrome	WS	0.1	0.5	0.6	100%
Edge	WS	0.1	0.5	0.6	100%
Firefox	WS	2.0	5.0	7.0	71%
Tor Browser	WS	2.0	5.0	7.0	73%
Firefox	WW	1.5	7.5	9.0	95%

Table 4: **Attack Bandwidth**: Number of seconds to transmit a 35-bit string). Times are reported in seconds; all values are reported over 100 runs.

transmission success rate (discussed in the next subsection) is reported as a percentage.

We find that our example "pool-party" attacks are practical. Even the slowest forms of the attack complete in under ten seconds (far below the average page dwell time of slightly under a minute [23]). Each attack could further be carried out between pages that are left open for a moderate amount of time, either because they get lost in a browser users ever-growing collection of tabs, or because the site is intended to stay open for a long time (e.g. sites that function and email clients, instant messaging applications, video streaming sites, etc). Our example "pool-party" attacks are fast enough that the could be conducted multiple times during an average page view (again, assuming an average page dwell time of slightly under one minute), as a simple error handling technique to account for noisy channels.

The "Setup" column in Table 4 corresponds to the "Determining Initial Sender and Receiver" section of Algorithm 2; the "Send" column measures the "Sending Data" steps.

4.3 Attack Consistency

Methodology. We also evaluated how consistently each "pool-party" attack example completed successfully, in the absence of other sites running on the browser. This measurement provides an upper bound on how practical the attack could be, as having other sites running in parallel in the browser will in some cases further reduce the success rate.

We measured the consistency of each attack using the same methodology described in Section 4.2. We again ran the attack 100 times, on two different pages in a single instance of the browser, each time in a clean profile. We then report the percentage of times the 35-bit string was received correctly.

Results. The results of our consistency measurement is also reported in Table 4. We find that most forms of the attack are either perfectly consistent (i.e. all 100 evaluations executed correctly), or consistent enough to be practical (i.e. the Web Worker attack on Firefox). The WebSocket attack was less consistently successful in the Gecko browsers. We investigated the root cause and found that the pool size was not consistently enforced; Firefox occasionally allowed additional sockets to be created, resulting in a corrupted message.

Web API	% page Loads	% of URLs Desktop	% of URLS Mobile
Web Worker	12.34%	12.29%	11.9%
WebSocket	9.55%	4.33%	3.72%
Server-Sent Events	0.79%	0.8%	0.06%

Table 5: **Attack background noise**: Web API metrics reported by the Chrome Platform Status service, as of August 9, 2022. Numbers reflect the % of page loads and % of URLs observed across all channels and platforms.

4.4 Attack Background Noise

Methodology. Finally, we evaluated how noisy the communication channels used in our demonstrative "pool-party" attacks are in practice. We build on the intuition that resource pools that are infrequently used by sites "in the wild" for being purposes) are easier to convert into practical side-channels. Put differently, if sites are already consuming and releasing resources in a resource pool for benign purposes, then other

sites intending to use it as a covert communication channel have to contend with more noise and uncertainty, and thus communication will be more difficult.

We estimate an upper bound on the presence of background noise per tab that could interfere with our "pool-party" attacks by using HTML & JavaScript usage metrics reported by the Chrome Platform Status website [23] to look at how often Web sites use WebSocket, Web Worker, and/or SSE capabilities.

We note that we initially measured background use on the Web through an automated, Web-scale crawl, using browsers instrumented to count how often the Web Workers, WebSockets and Server-Sent Events APIs were used. However, we abandoned this approach on realizing that an automated crawl would potentially *under report* background noise, since in some cases sites would only use these "advanced" browser capabilities on user interaction. This realization lead us to instead look for measurements of browsers under real-world use, and thus to Chrome telemetry.

Results. We report how often the relevant browser APIs are used during real-world browser use (as reported by "Chrome Platform Status") in Table 5. Reported numbers are of August 9, 2022. As noted, no browser feature is used on most websites; one reported feature, SSE, is used on less than 1% of websites, and less than 1% of page loads. Put differently, the vast majority of sites do not use any of these browser features, meaning that in the common case sites could use the resource pool without any interference from other pages.

Notes and qualifications. In practice, sites colluding in a "pool-party" attack would need to contend with the union of all open sites accessing resources from the relevant resource pool. Browser with large numbers of tabs open are therefore more likely to present interference to the two attacking tabs. The numbers resulting from our methodology are a lower bound on how noisy the given resource pool would be, and attackers might need to implement ways of communicating over noisy channels.

Additionally, we note that resources used by a site during the duration of a "pool-party" attack will not effect correctness, only bandwidth. Held resources merely reduce the total limit on a resource pool, but our algorithm can still proceed to send messages. Only resources that go from unconsumed to consumed by a site (or vise versa) during the attack will affect correctness.

Further, we note that the faster an attack completes, the less susceptible the attack is to errors introduced by background noise. This is for two reasons. First, attacks that finish quickly are less likely to be interrupted with benign background resource use (simply because there is less opportunities for background resource use). And second, attacks that finish quickly can engage in simple error correcting techniques to account for possible background noise (for example, conducting the attack multiple times and taking the majority result).

[23] https://chromestatus.com/metrics/feature/popularity

Nonetheless, some percentage of sites are likely to be calling the Web APIs in a more dynamic manner, so it's useful to understand how often these features are used in the wild.

5 Discussion

5.1 Implementation Challenges

We have shown that, a limited-but-unpartitioned resource pool whose resources can be consumed and released by scripts in web pages are sufficient to allow cross-site communication. A few additional anomalies arose, however, in the implementation of our algorithm that would be necessary to consider for the development of a robust message-passing script.

5.2 Drifting Resource Pool Limit

Because the global limit on a resource pool is not a central feature of web browsers, it is possible that developers may overlook certain behaviors of the resource pool. For example, we found that the Firefox global WebSocket pool limit was not strictly fixed. While our script continuously created and destroyed WebSockets to manipulate the number of unheld WebSocket vacancies in the pool, we observed that occasionally the total number of WebSockets that could be created increased. That is: while the initial limit on WebSockets was 512, after a few cycles of the algorithm, the total number of resources that could be held was observed to be 513. That number continued to increase over time. We attribute this behavior to a likely race condition that meant the limit on the total number of allow WebSockets was not consistently enforced, but occasionally a WebSocket slipped through and was not counted.

If the limit changed during a message cycle, then at least one value passed from sender to receiver in our implementation would be incorrect. That would result in an incorrect 35-bit message being recorded by the receiver. To minimize the effects of this anomaly, we ensured that the sender repeatedly attempted to consume more resources than the expected limit, so that even if the limit silently increased during one cycle, the the algorithm would correctly pass the message in subsequent cycles.

Delayed feedback. A second anomaly we observed in the browser-JavaScript implementation of our algorithm was an inconsistent delay in feedback from the resource pool when the pool limit was hit. If a site tries to consume an n-th resource when only $n-1$ resources were available, the attack script must receive an indication that the limit has been reached for the algorithm to succeed.

For example, to consume a resource in the WebSocket pool, it is necessary for the script to call `new WebSocket(...)`. In all cases, whether or not a limit on the WebSocket pool has already been reached, the call returns a WebSocket object. To determine whether the limit has been reached, it's necessary

to find out whether the WebSocket object is in a valid state or not. The state can be ascertained by using the `onerror` callback property on the WebSocket object; however this callback is not fired after a consistent time interval. Instead the time interval varied by as much as tens of milliseconds in some cases. Therefore, it is necessary to introduce a delay before checking for the presence of an error state.

In order to avoid introducing too much delay when attempting to consume n resources, we first create all n resource objects in a tight loop, and then wait for success or failure for all of the resources in parallel.

5.3 Additional Attack Vectors

In this work we demonstrated that "pool-party" attacks are possible and practical in all popular browsers, by exploiting the limited-but-unpartitioned implementations of WebSockets, Web Workers, and Server-Sent Events. However, there are many other "pool-party" attack opportunities in current browsers. This section details some additional APIs and browser capabilities that can be converted into covert-channels though "pool-party" attacks. We do not intend this to be a comprehensive list; we expect that there are many more "pool-party" attack vectors in browsers. We identified the below browser capabilities and APIs as likely exploitable by "pool-party" attacks based on their implementations in Gecko, WebKit and / or Chromium, though we did not build attacks to test the exploit-ability of all listed APIs.

Chromium. Chromium's DNS resolver has a global, unpartitioned limit of 64 simultaneous requests, which could be exploited by an attacker that could control the response time of DNS queries. Second, when Chromium browsers are configured to use an HTTP proxy, they impose a global limit of 32 simultaneous network requests. Third, several APIs in Chromium are thin-wrappers around OS-provided systems, and maintain a single global handle to the system process, effectively creating limited-but-unpartitioned pool of size one (e.g. the Web Speech API).

Gecko. Browsers built on Gecko have many of the same additional attack vectors as Chromium based browsers; Gecko has a global limit on the number of DNS requests that can be in the air at the same time, and a lower limit on the number of requests that can be open when using a HTTP(S) proxy.

WebKit. We identified far fewer additional limited-but-unpartitioned resource pools in WebKit than in other browser engines. We did not identify additional attack vectors in Safari (the most popular WebKit browser) beyond SSEs. However, other, less popular browsers also use WebKit, and some of these introduce additional limited-but-unpartitioned resource pools. For example, the GTK-based version of WebKit[24] uses a DNS resolver with a limit of 8 requests at a time, and a pre-fetch cache with a limit of 64 hosts.

[24] https://trac.webkit.org/wiki/WebKitGTK

5.4 Defenses and Constraints

Defending against "pool-party" attacks in browsers is difficult since, at root, systems will always have limited resources, and thus some underlying limited-but-unpartitioned pool that can be exploited by a sufficiently motivated party. Currently browser resource limitations are mostly explicit and intentional, but even if browsers removed such limits (e.g. limits on WebSocket connections), the underlying system would necessarily have a global limit, either explicitly (e.g. OS imposed limitations on open network sockets) or implicitly (e.g. systems have a finite amount of memory, and so unavoidably can only maintain a finite number of network sockets).

However, even if "pool-party" attacks will fundamentally always be possible, browsers can still take steps to limit how practical such attacks might be.

One approach is to lift browser-imposed limits on resource pools where applicable, and require attackers to contend with much larger system-maintained resource pools. This approach would make attacks much more obvious to the user, who would notice their system slowing down or other applications on the system impacted while the attack was being carried out. Such detectability might deter attackers, at least in the common case. Relying on the OS or system level limits would also make attacks more difficult to carry out. For example, the system network connection pool will be much "noisier" than the browser's WebSocket pool, making it much more difficult to use the resource pool as a reliable covert channel.

Second, browsers could take the opposite approach, and instead of dramatically widening the size of resource pools, browsers could maintain existing resource caps but partition resource pools the same way browsers increasingly partition DOM storage and network state (see Section 2.5). If resource pools were partitioned by site, a site would not learn anything by exhausting its resource pool; the resource pools for other sites would be unaffected. A determined attacker could regain the ability to conduct a "pool-party" attack by controlling a large number of sites, and using them to collectively drain the resource pool. Nevertheless, partitioning resource pools by site would make "pool-party" attacks significantly more difficult for an attacker to carry out.

Third, browsers could combine these two approaches, and simultaneously remove global limits on the size of resource pools, but limit the number of resources each site or context and use. Such a hybrid approach would achieve some of the benefits of both of the above approaches.

5.5 Applicability to Mobile

As part of this work, we checked whether mobile versions of each of browser were also vulnerable to "pool-party" attacks. To do so, we checked that the availability of the WebSockets and SSE attacks on mobile versions of each browser matched the availability of each attack on the desktop version. We

found that the availability of each attack was the same; attacks that worked on the desktop version of a browser also worked on the browser's mobile browser, and attacks that did not work on the desktop version also did not work on mobile.

We did not measure the bandwidth or stability of the identified "pool-party" attack on mobile, both because i. of limited resources, and ii. it being somewhat more difficult to conduct automated measurements on mobile browsers (e.g. most automation tools target desktop versions of browsers, or rely on imperfect simulations of mobile environments). Assessing the practicality of "pool-party" attacks on mobile browsers is an important area for future work.

Identifying Resource Pools. The vulnerabilities discussed in this work were identified through a combination of domain expertise, source code review, and manually interaction with each API in each browser.

We began by generating a list of candidate APIs, based on our experience in both browser and Wbb-app development. Our list of candidate APIs focused on features that i. needed to be handled in parallel (e.g. threads, I/O operations), ii. use limited system resources (e.g. file handles) or iii. are exclusive by nature (e.g. only one voice can speak at a time when using the Web Speech API).

Next, once we had our set of candidate APIs, we examined the source code for API's implementation in each browser engine. We looked for explicit limits on resources, both in the source code and surrounding comments. Often (though not always), limits were relatively easy to find, since they were encoded in constants or runtime flags.

Finally, we manually evaluated whether we could use these browser limits to conduct pool-party attacks by constructing two different test pages on different sites and seeing if we could detect on one site when the relevant resources were being exhausted by the other site.

As noted, this process is entirely manual; it is likely-to-certain that there are more vulnerable resource pools in browsers. Developing a system for systematically or automating the detection of vulnerable resource pools would be a valuable area for future work.

6 Related Work

Online tracking through feature misuse. Our work exists alongside a large body of work on ways browser features can be (mis)used by online trackers, to track their users in ways unintended by the browser vendor.

The largest volume of work in this area is on browser fingerprinting. We highlight significant work in the area, specifically those that identified new fingerprinting vulnerabilities in browsers. Mowery et al. [24] famously demonstrated that differences in how browsers executed drawing (i.e. canvas and WebGL) operations could be used to identify individuals, and Acar et al. [1] showed that differences in what fonts users have installed could be similarly misused. Englehardt et al. [6], as part of a project to measure privacy violations on the 1m most popular websites, show the WebAudio and WebRTC APIs could be used to track users. Olejnik et al. [27] showed the Battery Status API could be used for fingerprinting, and Olejnik and Janc [28] demonstrated the Ambient Light API could be similarly misused. Zhang et al. [42] found that in mobile browsers, websites can use unpermissioned access to motion sensors to identify users, and Starov and Nikiforakis [36] demonstrated that what browser extensions the user has installed can make users more identifiable. Eckersley [5] documented that a screen resolution and display size were practical fingerprinting vectors, and Nikiforakis et al. [26] showed that other display details contributed to identifiability. Laperdrix et al. [20] found that, ironically, the presence of a content blocker could help fingerprinters distinguish users. Iqbal et al. [12] identified dditional browser APIs that fingerprinting methods misuse (e.g. proximity sensor APIs, media capabilities, the Presentation API) by examining the JavaScript source code of known fingerprinting scripts and identifying the additional APIs those scripts abuse.

Distinct from browser fingerprinting, researchers have found other ways of misusing browser features to construct u user identifiers. Solomos et al. [35] transformed the browser's "favicon" cache into a persistent tracking mechanism, Janc et al. [13] showed that Safari's "Intelligent Tracking Prevention"[25] features could be abused to re-identify users, and Syverson and Traudt [37] showed how the browsers "HTTP Strict-Transport-Security" system could be re-purposed to construct and assign unique identifiers.

A parallel body of work attempts to prevent browser fingerprinting. Nikiforakis et al. [25] found browsers could resist fingerprinting by manipulating the values common Web APIs return. Laperdrix et al. [18] extended this approach by introducing small amounts of noise into the Web Audio and Canvas APIs, changes large enough to cause fingerprinters to misidentify users, but small enough that benign uses of features would not be impacted. Snyder et al. [34] suggested disabling Web APIs whose cost to the users (including additional identifiability) was higher than the benefit to them (in terms of desirable page behaviors), based on prior work finding that most browser APIs are rarely used at all [33]. Smith et al. [32] suggest a strategy for rewriting malicious code to prevent fingerprinting scripts from accessing the underlying, identifying values. Other works aim to prevent attackers from abusing features for tracking purposes by removing the entropy added by OS or hardware differences. Wu et al. [41] presents a method for removing hardware-induced differences in WebGL operations, and though not targeting fingerprinting attacks, Andrysco et al. [2] proposed a similar "improve privacy by making dissimilar systems execute similarly" approach for floating-point based channels in browsers.

[25] https://webkit.org/blog/7675/intelligent-tracking-prevention/

Attacks on browser partitioning and sandboxing. Our work also builds on, and exists along side, a large body of work documenting ways browser partitioning efforts can be circumvented, whether those partitions are enforced directly by the application, through OS-based process isolation, or otherwise. Again, the breath of work in this area makes a comprehensive discussion here impossible, so we discuss papers that are particularly significant, novel and/or recent.

Using timing methods to circumvent browser partitioning (in this case the same-origin-policy) dates back at least as far as 2000, when Felten et al. [7] presented a way sites could determine what other sites the user had visited by probing the HTTP cache and measuring timing differences. Bortz et al. [3] published similar foundational work on how sites could exploit timing differences in how, and how quickly, cross-site requests and resources were loaded to to learn about users' state on other sites. Since then, researchers have demonstrated many ways sites can circumvent browser imposed restrictions on what sites can learn about user behavior on other sites, and how sites can communicate with each other.

Schwartz et al. [30] document ways sites can create high resolution timers, with descriptions for how such timers can form covert channels across application boundaries. Smith et al. [31] show that sites can circumvent browsers attempts to partition a users browsing history by exploiting cache state and side effects in painting behaviors. Kohlbrenner and Shacham [17] presented a way of creating a covert channel across site boundaries by exploiting floating-point related timing channels in how browsers render SVGs. Gruss et al. [10] extended the Rowhammer [15] attack, previously used to leak information across OS process boundaries, to be exploitable be through site-included JavaScript code, to violate browser imposed process isolation. Jin et al. [14] show how security-focused protections like isolating sites in their own OS processes can be exploited to learn what sites the user is, or has recently, visited. Lipp et al. [22] demonstrated how sites could puncture site isolation protections and infer what the user was typing on a different site (or different application) by observing timing patterns in JavaScript execution, caused by the OS responding to key presses issued to other contexts. Vila et al. [40] presented a related attack, where a site could transform contention in the browser's main event loop (distinct from the event loop presented to an executing JavaScript context) into a cross-context side channel. van Goethem et al. [39] showed how other browser capabilities like service workers and the (now deprecated) application cache can be transformed into timer-based covert channels as well. As part of a larger project of creating an automated system for detecting cross-site information leaks, Knittel et al. [16] identified how many other browser features that had side effects that could be detected across site boundaries.

The cross-profile tracking attack against Gecko-based browsers described in this work build on other cross-profile attacks such as van Goethem and Joosen [38], which found that application efforts to isolate "incognito" browsing sessions from standard browsing sessions could be circumvented by using contention for disk and memory resources as a covert channel. Oren et al. [29] showed that contention in the CPU cache could be exploited by unprivileged, malicious JavaScript code to learn what sites a user was visiting in an "incognito" mode session. with JavaScript running on other sites, in other processes, or applications outside the browser. Gruss et al. [9] present a similar attack, though instead targeting timing channels stemming from memory deduplication.

Finally, Asankah [11] defines"ephemeral fingerprinting", where sites observe infrequent global events identify a user across contexts.

7 Conclusions

In this work we define a new category of practical privacy attack in popular Web browsers we call "pool-party" attacks. "pool-party" attacks allow sites to break out of the "contextual sandboxes" that browsers try to enforce, and so allow sites to circumvent privacy protections in even the most aggressively privacy-focused browsers. More alarming still, we find that "pool-party" techniques can be used to track users beyond cross-site tracking (specifically, that in Gecko-based browsers "pool-party" attacks can track users *across profiles*).

While some attacks in this category have been known to be *theoretically* possible, this work demonstrates that such attacks are *practical*, and must be dealt with as a real-world threat to the Web users' privacy. Further, we show that "pool-party" attacks can be carried out using a wider range of browser capabilities than previously documented, further emphasizing the severity of the risk to user privacy.

Web privacy has moved in two very different directions over the last two decades. *Privacy attacks* have moved in a dispiriting direction, with privacy violations becoming common place. This disappointing trend is reinforced by a combination of conflicting incentives from (some) browser vendors, backwards compatibility concerns, and user-harming financial incentives. *Privacy defenses* in browsers, though, have been recently moving in an encouraging direction. This is due to (in part) a combination of regulatory pressure, increasing user awareness, the tireless efforts of privacy-focused researchers and developers, and a virtuous competition between (some) browsers to own the "most private browser" title. We hope that this work helps the latter, to the determent of the former.

8 Acknowledgements

We'd like to thank Deian Stefan and Michael Smith from University of California, San Diego for contributing additional examples of limited-but-unpartitioned resource pools. We'd also like to thank Rainer Böhme from University of Innsbruck for refining and improving this work.

References

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1129–1140, 2013.

[2] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1369–1382, 2018.

[3] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *International Conference on the World Wide Web (WWW)*, pages 621–628, 2007.

[4] Gaurav Aggarwal Elie Burzstein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security Symposium*, 2010.

[5] Peter Eckersley. How unique is your web browser? In *International Symposium on Privacy Enhancing Technologies Symposium (PETS)*, pages 1–18. Springer, 2010.

[6] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1388–1401, 2016.

[7] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 25–32, 2000.

[8] Xianyi Gao, Yulong Yang, Huiqing Fu, Janne Lindqvist, and Yang Wang. Private browsing: An inquiry on usability and privacy protection. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 97–106, 2014.

[9] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *European Symposium on Research in Computer Security (ESORICS)*, pages 108–122. Springer, 2015.

[10] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[11] Asanka Herath. Ephemeral fingerprinting on the web. https://github.com/asankah/ephemeral-fingerprinting, 2020.

[12] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *IEEE Symposium on Security and Privacy (SP)*, pages 1143–1161. IEEE, 2021.

[13] Artur Janc, Krzysztof Kotowicz, Lukas Weichselbaum, and Roberto Clapis. Information leaks via safari's intelligent tracking prevention. 2020.

[14] Zihao Jin, Ziqiao Kong, Shuo Chen, and Haixin Duan. Site isolation enables timing-based cross-site browsing surveillance. In *IEEE Symposium on Security and Privacy (SP)*, 2022.

[15] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[16] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. Xsinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1771–1788, 2021.

[17] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security Symposium*, pages 69–81, 2017.

[18] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems*, pages 97–114. Springer, 2017.

[19] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, 14(2):1–33, 2020.

[20] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy (SP)*, pages 878–894. IEEE, 2016.

[21] Benjamin S Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security (ESORICS)*, pages 57–74. Springer, 2013.

[22] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *European Symposium on Research in Computer Security (ESORICS)*, pages 191–209. Springer, 2017.

[23] Chao Liu, Ryen W White, and Susan Dumais. Understanding web browsing behaviors through weibull analysis of dwell time. In *ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 379–386, 2010.

[24] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. *W2SP*, pages 1–12, 2012.

[25] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *International Conference on the World Wide Web (WWW)*, pages 820–830, 2015.

[26] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy (SP)*, pages 541–555. IEEE, 2013.

[27] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. The leaking battery. In *Data Privacy Management, and Security Assurance*, pages 254–263. Springer, 2015.

[28] Lukasz Olejnik and A Janc. Stealing sensitive browser data with the w3c ambient light sensor api, 2017.

[29] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1406–1418, 2015.

[30] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[31] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re: visited. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[32] Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. Sugarcoat: Programmatically generating privacy-preserving, web-compatible resource replacements for content blocking. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2844–2857, 2021.

[33] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser feature usage on the modern web. In *Internet Measurement Conference (IMC)*, pages 97–110, 2016.

[34] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 179–194, 2017.

[35] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. Persistent tracking in modern browsers. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[36] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *IEEE Symposium on Security and Privacy (SP)*, pages 941–956. IEEE, 2017.

[37] Paul Syverson and Matthew Traudt. HSTS supports targeted surveillance. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2018.

[38] Tom Van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[39] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1382–1393, 2015.

[40] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security Symposium*, pages 849–864, 2017.

[41] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. Rendered private: Making GLSL execution uniform to prevent webgl-based browser fingerprinting. In *USENIX Security Symposium*, pages 1645–1660, 2019.

[42] Jiexin Zhang, Alastair R Beresford, and Ian Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *IEEE Symposium on Security and Privacy (SP)*, pages 638–655. IEEE, 2019.

USENIX'23 Artifact Appendix: Ultimate SLH: Taking Speculative Load Hardening to the Next Level

Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, Yuval Yarom

The University of Adelaide, Adelaide, Australia
MPI-SP, Bochum, Germany
IMDEA Software Institute, Madrid, Spain
The University of Melbourne, Melbourne, Australia
Radboud University, Nijmegen, The Netherlands

A Artifact Appendix

A.1 Abstract

We provide the artifact of USLH in a GitHub repository. The artifact includes the PoC of leaking secrets from resolving branch conditions and variable-time instructions. The artifact includes a real word example of how LLVM-SLH fails to protect the OpenSSL library. Besides the demonstration of vulnerabilities, the artifact also includes a fix to SLH and a gadget searching tool implemented in LLVM.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

Running artifact does not need a root privilege. All data fed to the program are randomly generated. The provided code does not access files other than those described in the README. The artifact evaluation involves compiling the Clang and OpenSSL source code. Please follow the instructions and do not install these software; otherwise they may disturb the system wide configuration of Clang and OpenSSL.

A.2.2 How to Access

The artifact and documentation are available on GitHub: https://github.com/0xADE1A1DE/USLH/tree/e23d4292723b11fa56efb9c237b6db201be97bfa.

The source code for the USLH implementation of SLH is available at https://doi.org/10.5281/zenodo.7704637.

A.2.3 Hardware Dependencies

A machine with an Intel processor (8th Gen, 9th Gen, 10th Gen) running Ubuntu (not virtual machine) is necessary. The artifact has been tested on processor i7-10710U, running Ubuntu 20.04. To build the customized compiler, your machine has to have at least 8GB RAM.

A.2.4 Software Dependencies

The artifacts requires a GCC compiler to compile Clang.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

You will need to download and compile LLVM, but you do not need to install it. Instructions on building Clang is available at https://clang.llvm.org/get_started.html. Note that building Clang with Release version is sufficient for the artifact evaluation. You can find more instructions at README under the folder USLH/LLVM_FIX.

You will need to download and compile OpenSSL-1.1.1q, which is available at https://www.openssl.org/source/old/1.1.1/. Instructions on compiling OpenSSL with customized compiler and flags are available at the README in the folder USLH/PoC/openssl_leakage.

A.3.2 Basic Test

To evaluate the fix to LLVM-SLH, having a working customized Clang is necessary. After compiling the Clang, you should check if Clang is properly compiled by compiling a program with *clang* under $path_to_folder/build/bin/.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): Resolving branch conditions leaks secret if the secret resides in branch operands (E1, E4).

(C2): USLH prevents leakages from resolving branches by hardening branch conditions (E1, E4).

(C3): Variable-timing instructions leak secret by checking the cache status of a secret-independent memory. Specifically, when a sequence of floating point instructions is fed with a *slow value*, the secret-independent memory access may not be scheduled to execute (E2, E3).

(C4): USLH mitigates the vulnerability of variable-timing instructions by hardening the operands of variable-timing instructions. (E3)

(C5): We provide a LLVM backend pass to find potential gadgets (E5).

A.4.2 Experiments

(E1): [1/60 human-minutes, 1/3600 cpu-hour]: Leak secret from resolving the branch condition and fix it. A detailed instruction is available in README under the folder USLH/PoC/condition.
Preparation: You need to have the USLH compiled to mitigate the vulnerability. Please refer to README in the folder *USLH/LLVM_FIX* for more instructions.
Execution: You need to modify the *folder* in *compile.bash* to compile the program with and without fix to LLVM-SLH. You then run the executable file with a parameter, which is either 1 or 0.
Results: When executing *leak*, if the fed value is 1, the program should return a measurement with cache miss penalty. If the fed value is 0, the program should return a measurement with cache hit in most cases. When executing *fix*, no matter what value fed is, the program should always return a measurement with cache miss penalty.

(E2): [1 human-minutes 1/60 cpu-hour] Blocking reservation station under speculation. The code is available at USLH/PoC/test_rs_limit. README contains detailed instructions.
Preparation: None.
Execution: Run the command *python3 test.py $max $min* to test how many pairs of *sqrtsd, mulsd* can block the RS during the speculation. You need to change *val* in run.bash to test *fast value* ot *slow value*.
Results: For *slow value*, with fewer pairs of floating-point operations, the secret-independent memory will not be accessed during the speculation. The actual number of pairs is various from processors. On 11th and 12th Gen Intel processors, you may not see the effect as they have larger ROB and RS.

(E3): [1/60 human-minutes, 1/3600 cpu-hour] Leak secret from variable-timing instructions. The code is available at USLH/PoC/variable_time. README contains detailed instructions.
Preparation: You need to complete the last experiment and adjust the number of floating-point instructions manually. Note that you may want to reduce the number of pairs in this experiment as the vulnerable function is slightly different from the one in E2.
Execution: Execute the program with or without mitigation with *attack.bash* or *mitigate.bash*.
Results: The program processes a secret value bit-by-bit. It returns the eight measurements of accessing the secret-independent memory, the guessed secret and whether the guess is correct or not.

(E4): [1/60 human-minutes, 1/3600 cpu-hour] Leak secret from *BN_mul_word* in OpenSSL. The code is available at USLH/PoC/openssl_leakage. README contains detailed instructions.
Preparation: You need to install the OpenSSL with the customized Clang. Please refer to the README file for more instructions on how to compile OpenSSL with a customized compiler and flags.
Execution: Execute the program with *./crun $val* where *val* is either 1 or 0.
Results: When *val* is 0, the measurement should be cache hit; otherwise it should return a cache miss penalty. By fixing the OpenSSL with USLH, no matter what *val* is, it should always return cache miss penalty.

(E5): [Heavily dependent on processors and targets] Find gadgets. The code is available at USLH/LLVM_FIX. README contains detailed instructions.
Preparation: You need to have a compiled USLH.
Execution: Compile the program that you have interest with command *$path_to_binary/clang file -mllvm -x86-mir-analyze*
Results: If there is a gadget, the terminal prints *Found it –> function_name*. Then you need to refer the source code to review the code.

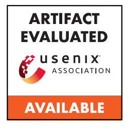
A.5 Notes on Reusability

USLH improves the LLVM-SLH by hardening more vulnerable operands or instructions. You can use the customized compiler to build safer programs. To play with different functionalities of USLH, you can enable features with command '-mllvm -x86-slh-xxx'. The LLVM backend pass performs static analysis on machine IR. You can use it to find potential leakages. To use it, you need to compile the program with command *-mllvm -x86-mir-analyze*.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at

https://secartifacts.github.io/usenixsec2023/.



USENIX'23 Artifact Appendix: Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions

A Artifact Appendix

A.1 Abstract

The goal of this artifact is to validate the microarchitectural leakage of CPU exceptions against the formal leakage models we proposed in the paper (named contracts). Concretely, this means reproducing a representative subset of the results described in Table 1 of the paper using our tool Revizor. The exceptions we tested form the rows of this table and the contracts are given as the columns. The contracts are ordered according to permissiveness, i.e., *CT-SEQ* does not allow any transient leakage, whereas *CT-VS-All* allows arbitrary speculative values.

Revizor is a random testing tool, i.e., all test cases are generated randomly. While we observed stable results during our experiments, we therefore cannot 100% guarantee that all results are reproduced within the indicated time frame.

The artifact of this paper includes the source code of Revizor, a set of scripts to run the experiments, and a description of how to run them.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Revizor includes a kernel module that disables the hardware prefetcher and initializes the performance counters. The tool also overwrites the OS-defined IDT to suppress the handling of exceptions on the running core. This may affect other jobs running on your system.

Revizor executes randomly generated programs in kernel space. These programs that are intended to throw exceptions. Even though the executor provides a stable and isolated environment, it may adversely affect the stability of your system.

A.2.2 How to access

The artifact is available on GitHub at https://github.com/vusec/SpeculationAtFault-AE/tree/cf2fa27ff5145a2dedfa8d4302a16d6e32aa5581

A.2.3 Hardware dependencies

Evaluating this artifact requires at least one physical machine with root access. Ideally, the reviewer has access to both one machine with Intel (KabyLake or CoffeeLake) and AMD (Zen+ or Zen3) CPU. If only one such machine is available, the experiments can still be reproduced for just that machine. For AMD Zen2, we expect to obtain the same results as for Zen3. Remote access to some of our machines may be granted upon request. To obtain stable results, the machine(s) should not be actively used by other software.

A.2.4 Software dependencies

- Linux v5.1+ and Kernel Headers
- python 3.9+, python3.9-venv, and pip

A.2.5 Benchmarks

None

A.3 Set-up

In this section, we provide a short version of the installation and configuration steps required to prepare the environment and run Revizor. Please refer to the README file of the repository for detailed installation steps.

A.3.1 Installation

1. Clone the repository.
2. Install software requirements:
```
# on Ubuntu
> sudo apt install linux-headers-$(uname -r)
> sudo apt install python3.9 python3.9-venv
```
3. Install Revizor python package: In the base directory:
```
# on Ubuntu
> cd revizor
> python3 -m venv ~/venv-revizor
> source ~/venv-revizor/bin/activate
```

```
> pip install revizor_fuzzer-1.2.3-py3-none-any.whl
> cd -
```

4. Check installation:
```
> rvzr # Should print the following:
# usage: rvzr {fuzz,analyse,reproduce,minimize,
generate,download_spec} ...
# rvzr: error: the following arguments are
required: subparser_name
```

5. Install the executor:
```
> cd revizor/executor
> make uninstall
> make clean
> make
> make install
> cd -
```

6. Download the ISA spec:
```
> rvzr download_spec -a x86-64 -extensions BASE
SSE SSE2 CLFLUSHOPT CLFSH MPX -outfile base.json
```

A.3.2 Basic Test

From the base directory, on Intel CPU, `cd` into `intel/`. On AMD CPU, `cd` into `amd/`.

Run the basic test:

```
> rvzr fuzz -s ../base.json -c basic/seq-BP.yaml -i
10 -n 100
```

This command will start a small fuzzing campaign testing the Breakpoint exception with 100 test cases, each tested with 10 inputs. The command is expected to terminate without reporting a violation.

A.4 Evaluation workflow

The main results are summarized in Table 1 of the original paper. The evaluation workflow is designed to validate our leakage models. The list of experiments needed to do so depends on the CPU microarchitecture.

A.4.1 Major Claims

For each combination of exception and architecture, the following are the least permissive of our contracts that model the transient leakage induced by that exception.

C1 `#PF` complies with *CT-VS-All* on Intel Kaby Lake, with *CT-VS-NI* on Intel CoffeeLake, and with *CT-DH* on AMD.

C2 `#GP` complies with *CT-VS-CI* on AMD. On Intel, `#GP` does not satisfy any contract.

C3 (Intel only) `#BR` complies with *CT-DH*. (E5)

C4 ucode-assists comply with *CT-SEQ* on AMD, with *CT-VS-All* on Intel Kaby Lake, and with *CT-VS-NI* on CoffeeLake.

C5 `#DE` complies with *CT-VS-Ops* on Intel and AMD Zen3, and with *CT-VS-All* on AMD Zen+.

C6 `#UD`, `#DB`, and `#BP` comply with *CT-SEQ* on all machines.

A.4.2 Experiments

Our experiments serve two purposes: (1) validating our claims regarding which contract satisfies which exception on which machine, and (2) confirming Revizor's effectiveness in generating counterexamples. For each combination of CPU architecture and exception, we therefore propose one experiment that validates the correct contract and one experiment that finds a counterexample for the next more restrictive contract (if one exists).

In the interest of time, we run each experiment for 12h or until a violation is found. The timeout can be increased with the `timeout` option we included in the scripts. For our paper, each experiment ran for 24h. Remember though that Revizor is based on random testing, it is thus possible that a violation is not found within 12h. If this is the case, we suggest to repeat the experiment and increase the timeout.

How-to. We split our experiments according to the type of machine under test. Scripts for the experiments are grouped into a directory for Intel and one for AMD. For example, the scripts to reproduce **Intel E1** are stored inside `./intel/experiment_1/`. Inside each directory, there is one `run.sh` script to start the experiment. An optional timeout (given in seconds) can be set with the `--timeout` option (e.g., `./run.sh --timeout=86400` for a 24h timeout).

Running the script will create a subdirectory `results` inside the experiment directory, where logs are stored. When the script terminates, you can inspect the log to determine whether Revizor detected a violation. Violations (if any) are stored in subdirectories inside `results/violations/`. Each violation directory will contain the program, the inputs, and the configuration file.

Intel. On Intel, our claims can be confirmed with the following experiments.

E1: **C1 - page faults - violation** [1/2 machine hours]: Test each page fault class (invalid, read-only, SMAP) against *CT-DH*.
Result: violation (for all classes)

E2: **C1 - page faults - correct** [36 machine hours]: Test each page fault class (invalid, read-only, SMAP) against *CT-VS-NI* on CoffeeLake (and newer), resp. against *CT-VS-All* (on KabyLake and older).
Result: no violation

E3: **C2 - non-canonical accesses - violation** [12 machine hours]: Test non-canonical accesses against *CT-VS-All*.
Result: violation. Due to the complexity of the contract, finding a violation may take several hours (it was 11h

when we ran the experiment). Please increase the timeout if no violation is found within 12h.

E4: **C3 - Mpx - correct** [12 machine hours]: Test MPX against *CT-DH*.
Result: no violation

E5: **C4 - ucode-assists - violation** [1/6 machine hours]: Test both variants of ucode-assists (Access bit and Dirty bit) against *CT-DH*.
Result: violation (for both variants)

E6: **C4 - ucode-assists - correct** [24 machine hours] Test both variants against *CT-VS-NI* on CoffeeLake (and newer), resp. against *CT-VS-All* (on KabyLake and older).
Result: no violation

E7: **C5 - division - violation** [2 machine hours] Test both types of division errors (divide-by-zero and division overflow) against *CT-VS-NI*.
Result: violation (for both variants)

E8: **C5 - division - correct** [24 machine hours] Test both types of division errors (divide-by-zero and division overflow) against *CT-VS-Ops*.
Result: no violation

E9: **C6 - others - correct** [36 machine hours] Test `#UD`, `#DB` and `#BP` against *CT-SEQ*.
Result: no violation

AMD. On AMD, our claims can be confirmed with the following experiments.

E1: **C1 - page faults - violation** [1/6 machine hours]: Test each page fault class (invalid, read-only, SMAP) against *CT-SEQ*.
Result: violation (for all classes)

E2: **C1 - page faults - correct** [36 machine hours]: Test each page fault class (invalid, read-only, SMAP) against *CT-DH*.
Result: no violation

E3: **C2 - non-canonical accesses - violation** [1/12 machine hours]: Test non-canonical accesses against *CT-DH*.
Result: violation

E4: **C2 - non-canonical accesses - correct** [12 machine hours]: Test non-canonical accesses against *CT-VS-CI*.
Result: no violation

E5: **C4 - ucode-assists - correct** [24 machine hours] Fuzz both variants (Access bit and Dirty bit) against *CT-SEQ*.
Result: no violation

E6: **C5 - division - violation** [2 machine hours] Test both type of division errors (divide-by-zero and division overflow) against *CT-VS-NI*.
Result: violation (for both variants)

E7: **C5 - division by zero - correct** [12 machine hours] Test division-by-zero errors against *CT-VS-Ops* on Zen3 (or newer), resp. against *CT-VS-All* on Zen+ (or older). For Zen2 (which was not part of our setup), we expect *CT-VS-Ops* to hold as well.
Result: no violation

E8: **C5 - division overflow - correct** [12 machine hours] Test division overflows against *CT-VS-Ops*.
Result: no violation

E9: **C6 - others - correct** [36 machine hours] Test `#UD`, `#DB` and `#BP` against *CT-SEQ*.
Result: no violation

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

USENIX'23 Artifact Appendix: PROSPECT: Provably Secure Speculation for the Constant-Time Policy

Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, Frank Piessens

A Artifact Appendix

A.1 Abstract

The artifact contains the source code of the base Proteus processor extended with PROSPECT, alongside the benchmarks and security tests from our paper. All materials (except for the tool required for hardware cost measurements) are bundled into a Docker container and distributed on GitHub.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None, our artifact is contained in a Docker container, it does not perform any attacks against the host system and it does not use user data.

A.2.2 How to access

The artifact is available on GitHub at the following URL: https://github.com/proteus-core/prospect/tree/usenix_artifact.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Our artifact uses the following two tools, which are available for both Windows and Linux.

- Docker and 7 GB of disk space for the container (https://docs.docker.com/engine/install/).
- Xilinx Vivado 2022.2 Standard Edition, requiring approximately 55 GB of disk space (https://www.xilinx.com/products/design-tools/vivado/vivado-ml.html).

A.2.5 Benchmarks

Our evaluation uses modified benchmarks from the Spectre-Guard paper, which are included in our artifact.

A.3 Set-up

A.3.1 Installation

1. Install the two dependencies (Docker and Vivado). Our repository contains detailed instructions on setting up Vivado to minimize the required disk space.
2. Clone our GitHub repository or download the Dockerfile from the root directory (https://github.com/proteus-core/prospect/tree/usenix_artifact).
3. Build the Docker container by following the instructions in the `README.md` of the repository (building takes approximately 2 hours on a mid-range desktop).

A.3.2 Basic Test

The security evaluation can be run from the Docker container using the following commands:

```
// first, launch the container
$ docker run -i -t prospect

// inside the container, run the tests
# cd /prospect/tests/spectre-tests/
# ./eval.py /proteus-base/sim/build/base \
    /prospect/sim/build/prospect
TEST secret-before-branch
SECURE VARIANT:   Secret did not leak!
INSECURE VARIANT: Secret leaked!
[...]
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PROSPECT prevents the leakage of secrets from well-annotated programs via Spectre attacks. This is shown by experiment (E1) described in Section 6.2, which executes programs vulnerable to Spectre on the baseline and the extended secure implementation.

(C2): PROSPECT incurs no overhead on precisely annotated constant-time code. This is shown by experiment (E2), described in Section 6.2 (Runtime overhead) and Table 1.

(C3): PROSPECT only incurs a small overhead in terms of hardware cost. This is shown by experiment (E3), described in Section 6.2 (Hardware cost).

A.4.2 Experiments

(E1): [Security tests, 5 human-minutes]:

How to: The experiment is performed in the container by launching a script (identical to the basic test A.3.2).

Preparation: Launch the container with `docker run -i -t prospect` and navigate to the experiment with `cd /prospect/tests/spectre-tests`.

Execution: Run the following command:

```
./eval.py /proteus-base/sim/build/base \
/prospect/sim/build/prospect
```

This will run and evaluate the experiments with both the baseline implementation (first argument) and the PROSPECT-extended version (second argument).

Results: The results are displayed as text. The security evaluation should fail with the baseline implementation and succeed with the extension, validating claim (C1).

(E2): [Runtime overhead, 5 human-minutes + 9 compute-hours]:

How to: The experiment is performed in the container by launching a script.

Preparation: Launch the container with `docker run -i -t prospect` and navigate to the experiment with `cd /prospect/tests/synthetic-benchmark`.

Execution: Run the following command:

```
./eval.py \
/proteus-base/sim/build/base_nodump \
/prospect/sim/build/prospect_nodump
```

This will run and evaluate the experiments with both the baseline implementation (first argument) and the PROSPECT-extended version (second argument), using the variants compiled with no waveform dumping to save disk space.

Results: The results are displayed as text. The generated table should reflect Table 1 from the paper, validating claim (C2).

(E3): [Hardware cost, 1 human-hour + 2 compute-hours]:

How to: The experiment is performed in Vivado, using generated Verilog files from the Docker container.

Preparation: Follow the instructions under the heading *Hardware overhead* in `README.md` to obtain the Verilog files used for the synthesis and to set up the Vivado project (*Creating the Vivado project*).

Execution: Follow the instructions under the heading *Running the Vivado evaluation* in `README.md` to (iteratively) obtain the hardware costs of both the baseline and the PROSPECT-extended hardware design.

Results: The results of the synthesis should be interpreted according to the description under the heading *Interpreting the results* in the `README.md` and compared to the reported numbers in the paper under the heading *Hardware cost* (Section 6.2).

A.5 Notes on Reusability

Using the `newlib` board support package included in this repository and building on the scripts used for our benchmarks, it is possible to run other benchmarks on Proteus and PROSPECT, making additional benchmarking and security tests possible. The source code of PROSPECT can also be modified to investigate tradeoffs or to extend the offered security guarantees.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels

Ruiyi Zhang
CISPA Helmholtz Center for Information Security

Taehyun Kim
Independent

Daniel Weber
CISPA Helmholtz Center for Information Security

Michael Schwarz
CISPA Helmholtz Center for Information Security

A Abstract

As discussed in the paper, we reverse engineer undocumented properties of the `monitor-` and `mwait-` instruction family that help convert microarchitectural into architectural states. In three case studies, we show the versatility of our primitive. First, with Spectral, we present a way of enabling transient-execution attacks to leak bits architecturally with up to 200 kbit/s without requiring any timer. Second, we show traditional side-channel attacks without relying on a timer. Finally, we demonstrate that when augmented with a coarse-grained timer, we can also mount interrupt-timing attacks, allowing us to, e.g., detect which website a user opens. This artifact contains the description of several experiments and proof-of-concepts for the paper.

A.1 Description & Requirements

A.1.1 Security, privacy, and ethical concerns

In our experiments, we need to modify page table entries via the PTEditor library[1].

A.1.2 How to access

The source code for this paper is available on GitHub: https://github.com/cispa/mwait/tree/ae.

A.1.3 Hardware dependencies

We exploit the unprivileged idle-loop optimization instructions `umonitor` and `umwait` introduced with the new Intel microarchitectures (Tremont and Alder Lake). While the reverse engineering and analysis of all `mwait-` variants are generic both on Intel and AMD processors.

[1] https://github.com/misc0110/PTEditor

A.1.4 Software dependencies

We recommend Ubuntu 18.04 or 20.04 and all our experiments are tested on Ubuntu 20.04 LTS (Linux kernel 5.4).

A.1.5 Benchmarks

None.

A.2 Set-up

The individual proof-of-concept implementations are self-contained and come with a Makefile and an individual description that explains how to build, run and interpret the proof-of-concept. In order to run all the proof-of-concepts, the following prerequisites need to be fulfilled:

A.2.1 Installation

- Build tools (gcc, make)
- Intel latest CPUs (Tremont and Alder Lake)
- PTEditor
- Stress

A.2.2 Basic Test

The folder `Intel-umwait` contains the basic experiment to check whether `umonitor` and `umwait` work on the current tested CPUs.

A.3 Evaluation workflow

A.3.1 Major Claims

(C1): We exploit the unprivileged idle-loop optimization instructions `umonitor` and `umwait` introduced with the new Intel microarchitectures (Tremont and Alder Lake). Although not documented, these instructions provide architectural feedback about the transient usage of a specified memory region.

(C2): We experimentally confirmed that the Intel's undocumented timed `mwait` feature can be enabled by setting bit 31 in MSR (`0xe2`). We further reverse engineered the feature and found that bit 1 of the `ECX` register of the mwait instruction indicates that the timeout feature is used. The maximum waiting time is an implicit 64-bit timestamp-counter value stored in the `EDX:EBX` register pair.

(C3): With Spectral, we present a way of enabling transient-execution attacks to leak bits architecturally with up to 200/ without requiring any architectural timer.

(C4): We show traditional side-channel attacks without relying on an architectural timer.

(C5): We demonstrate that when augmented with a coarse-grained timer, we can also mount interrupt-timing attacks, allowing us to, detect which website a user opens.

A.3.2 Experiments

(E1): Intel-umwait
Preparation: Intel Tremont and Alder Lake CPUs
Results: Test if `umonitor/umwait` work on the current processors

(E2): trigger-tester
How to: We analyzed different wake-up triggers for all `mwait-` variants both on Intel and AMD machines, including cache coherence functions. Moreover, we analyzed the memory type of the monitored address range by modifying the page table via the library PTEditor.
Results: As shown in the Table 1-2 in the paper.

(E3): timed-mwait
How to: We reverse engineered the Intel's undocumented `timed-mwait` feature via a simple Linux kernel module.
Results: As claimed in the C2.

(E4): comparison
How to: We constructed a benchmark detecting fully asynchronous events with TWMand other conventional side-channel attacks for reference.
Results: As shown in the Figure 1-2, Table 3 in the paper.

(E5): covert-channel
How to: We created a timer-less covert channel with `umonitor` and `umwait`.
Results: As shown in the Figure 4 in the paper.

(E6): spectral
How to: We used the timer-less covert channel for spectre attacks.
Results: As shown in the Figure 5-6 in the paper.

(E7): aes-example
How to: We reproduced attacks on AES T-table implementation based on our Timer-less Timing Measurement.
Preparation: The deprecated OpenSSL 1.0.1e.
Results: As shown in the Figure 3,7 in the paper.

(E8): website-fingerprinting
How to: We detected network interrupts while opening a website.
Results: As shown in the Figure 8 in the paper.

A.4 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels

Andreas Kogler[1] Jonas Juffinger[1] Lukas Giner[1] Lukas Gerlach[2]
Martin Schwarzl[1] Michael Schwarz[2] Daniel Gruss[1] Stefan Mangard[1]

[1]Graz University of Technology [2]CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

We present Collide+Power, a technique that extends software-based power side channels to exploit the mere co-location of attacker-controlled data with victim data within CPU buffers, e.g., CPU caches. Collide+Power exploits that the *collision* of these values exposes the Hamming distance, *i.e.*, the bit difference between the values, in the power domain. Collide+Power can be mounted purely from software with any power-related signal, e.g., power consumption interfaces or throttling-induced timing variations.

The artifacts demonstrate the fundamental leakage enabling Collide+Power. First, we analyze the power leakage of the caches and evaluate our differential measurement method. Second, we compute the performance of the Correlation Power Analysis (CPA) for the raw channel and show that we leak precise victim data. Third, we analyze the effects of untargeted victim data within the cache lines. Finally, we demonstrate the attack PoCs for Collide+Power.

All the PoCs are tested on Intel, and some of the PoCs were also validated on AMD CPUs. Therefore, we only recommend Intel x86 CPUs to test the artifacts, for the best case, an *Intel Core i7-8700K*, *Intel Core i9-9980HK*, or *Intel Core i9-9900*.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifacts do not perform any destructive steps, and the worst risk for system security is a system freeze due to an unavailable Model Specific Register (MSR) read in the kernel module we provide. This *should* not happen if the system is set up as described above. We verified on our machines that the kernel module works as intended. If such a freeze occurs, data loss of unsaved files *could* happen. Therefore, we recommend saving all the work and doing a clean reboot before conducting the experiments. The PoCs only target the victim data of the provided programs. We do **NOT** target any other data on the system, nor do we read the personal files of the users. Furthermore, the power traces are saved locally and are not shared with the authors, nor does the provided framework send any information to us or any other server.

A.2.2 How to access

We provide the artifacts in a public GitHub repository. The most recent version of the artifacts is provided here: `https://github.com/iaik/collidepower`. The stable version of the artifacts with the included feedback from the artifact evaluation is provided here: `https://github.com/iaik/collidepower/tree/ae`.

A.2.3 Hardware dependencies

To reproduce the artifacts, we recommend a native *bare-metal* Intel CPU. We strongly recommend an *Intel Core i7-8700K*, *Intel Core i9-9980HK*, or *Intel Core i9-9900* CPU, as these CPUs showed the best leakage during our analysis (cf. Table 4 in the paper). For other CPUs not in the list, we designed the PoCs for an 8-way L1 cache and a 4-way L2 cache design with a pseudo-LRU replacement policy (cf. Section 4 in the paper) which can be checked with the `cpuid` command. If the cache uses a different number of ways, the PoCs need adaption, or the leakage *cannot* be guaranteed. Finally, we require an unfiltered Intel Running Average Power Limit (RAPL) energy measurement interface, meaning that for CPUs that support Intel Software Guard Extension (SGX), the Platypus patches might be active and obfuscate the energy measurements over the RAPL interface. Although we can exploit the throttling-induced timing variations with SGX enabled, we recommend disabling SGX in the bios to get *unmitigated* RAPL readings and significantly increase the practicality of the measurements.

A.2.4 Software dependencies

We require a Ubuntu 20.04 installation with Linux kernel version `5.4.` or `5.15`. The best case would be a fresh installa-

tion. The newer 5.19 kernel no longer supports the *nosmap* kernel argument, which is required for the initial leakage analysis. We detail this requirement in the provided readmes of the repository. Furthermore, we require access to the RAPL interface, which implies that the experiments must run on a bare metal machine and *should* not be a virtual machine as hypervisors block access to this interface. We require root privileges to insert a kernel module for the PoCs and to configure the Linux kernel boot command line. To build the PoC we require a built-essentials setup with gcc and make, which we list in the repositories readmes. Furthermore, we require the PTEditor to modify page tables. To post-process the recorded power traces, we use python3 with additional packages to provide installation steps in the readmes. Finally, the system should not be used during the measurements, *i.e.*, no other user must be logged in, and no program should be executed. We recommend using `ssh` to deploy and connect to the given machine.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The artifacts use two components: First, a C++ program with a kernel module performs the experiments and records the power traces. Second, a post-processing python framework to analyze the recorded traces. Installation of the required packages is performed using the `apt install` command. The python packages are installed with `pip3`. Finally, we adapt some kernel boot parameters to make the analysis more straightforward. For the detailed apt and python3 packages, please follow the provided readmes in the repository.

A.3.2 Basic Test

We provide the basic leakage analysis in the repository, which evaluates if the system exposes the exploited leakage. This is the smallest possible *basic test* we implemented since we cannot identify if the leakage exists on the system with other means. For detailed instructions, please follow the provided readmes in the repository.

A.4 Evaluation workflow

A.4.1 Major Claims

We provide artifacts verifying the following claims:

(C1): Using attacker-controlled data and victim data within the memory hierarchy exposes the combined Hamming distance leakage of both values in the power domain. We prove this claim with the initial leakage analysis experiment (E1) described in Section 4, whose results are reported in Table 2.

(C2): Using the differential measurement technique improves the correlation coefficients and the factors for the Hamming distance. We prove this claim with the same data as the initial leakage analysis (E1) using a different post-processing technique (E2). The differential measurement technique is described in Section 5, and the results are reported in Table 3.

(C3): We evaluate the raw channel leakage rates using our CPA and demonstrate that we can leak single nibbles as described in Section 7.2, where the results are shown in Figure 9. We prove this claim in the raw channel evaluation (E3).

(C4): We show that unmasked data does not influence the CPA success probability due to the differential measurement. This claim is described in Section 7.2, and the results are shown in Figure 10. We prove this claim in the victim data fill experiment (E4).

(C5): We show that Collide+Power with MDS-Power leaks data that is actively used on the hyperthread. This claim is described in Sections 6.1 and 7.3, and the results are shown in Figure 12a. We prove this claim in MDS-Power experiment (E5).

(C6): We show that Collide+Power observes a signal with Meltdown-Power for data that is only accessible within the Linux kernel. This claim is described in Sections 6.2 and 7.5; the results are shown in Figure 12a. We prove this claim in the MDS-Power experiment (E6).

A.4.2 Experiments

(E1): [30 human-minutes + 10 compute-hour + <5GB disk]:
How to: Follow the general setup guide. Build the provided PoC with a defined macro. Let the PoC record the power traces. Use the provided post-processing script to obtain the results.
Preparation: Reboot the machine and connect via SSH to the test machine. Build the program and the kernel module. Load the kernel module, stop all other programs, and follow the overall system preparation.
Execution: Execute the c++ program and pipe the output into a CSV file. Let the script run for at least the specified compute hours. Please note that the compute hours are estimates as the program only records samples for analysis. The more samples are recorded, the more accurate the analysis will get.
Results: Run the provided analysis script on the CSV.

(E2): [30 human-minutes + 0 compute-hours + <5GB disk]:
How to: Reuse the data from E1, the data for E2 is already included in the csv of E1.
Preparation: The same steps as E1.
Execution: None

Results: The same steps as E1.

(E3): [10 human-minutes + 5 compute-hour + <5GB disk]:

How to: Follow the steps of E1 but with another macro.

(E4): [10 human-minutes + 5 compute-hour + <5GB disk]:

How to: Follow the steps of E1 but with another macro.

(E5): [10 human-minutes + 20 compute-hour + <5GB disk]:

How to: Follow the steps of E1 but with another macro.

(E6): [10 human-minutes + 50 compute-hour + <5GB disk]:

How to: Follow the steps of E1 but with another macro.

Important Notes: The execution times for the experiments are estimates based on our CPUs that show a high correlation for the used interface. The longer the experiments is run, the more data is collected, resulting in a more accurate analysis. Furthermore, the experiments are designed to be terminated (`CTRL+C`) after the desired amount of data is collected. Finally, the created CSV files should always be valid during the experiments, which allows them to be copied to a different machine and be analyzed without the experiment to be stopped.

A.5 Notes on Reusability

The framework to analyze the traces is a general framework that can be reused for plotting and performing correlation analysis beyond Collide+Power.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: INCEPTION: Exposing New Attack Surfaces with Training in Transient Execution

Daniël Trujillo † — *ETH Zurich*
Johannes Wikner † — *ETH Zurich*
Kaveh Razavi — *ETH Zurich*

† Equal contribution first authors

A Artifact Appendix

A.1 Abstract

Our paper introduces the new TTE class of transient execution attacks and presents an end-to-end exploit INCEPTION. In particular, we make four major claims: 1) TTE allows manipulation of the BTB and RSB in transient execution, 2) a PHANTOMCALL allows manipulation of the RSB from an arbitrary instruction, 3) our end-to-end exploit INCEPTION leaks arbitrary kernel memory, and 4) `ibpb` overhead is between 93.1% and 239.2%. To back up these claims, this artifact reproduces experiments outlined in the paper, specifically those described in Section 8, Section 7.1, Section 7.3 and Section 9.

All experiments should be run on an AMD Zen microarchitecture. Our end-to-end exploit INCEPTION requires an Zen 1(+), Zen 2 or Zen 4 microarchitecture.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our exploit INCEPTION leaks kernel data from a user-provided address. Potentially private data located at the provided address will be printed to `stdout` and stored in a file (`data.bin`). An evaluator may choose to clear `stdout` and/or remove the output file after running this experiment.

Other than this, our experiments do not impose any security, privacy or ethical concerns.

A.2.2 How to access

The source code of INCEPTION is retrieved by cloning `https://github.com/comsec-group/inception.git`. The code for this artifact can be found under git tag `usenix-23-ae-final`.

A.2.3 Hardware dependencies

All provided code should be run on an AMD Zen microarchitectures. The end-to-end exploit works only on Zen 1(+), Zen 2 and Zen 4 microarchitectures.

A.2.4 Software dependencies

All experiments were ran on Ubuntu 22.04 LTS (Jammy Jellyfish), with a Linux kernel 5.19.0-28-generic. The following packages must be installed, available in the Ubuntu apt repository. `git build-essential clang linux-{image,headers,modules,modules-extra}-5.19.0-28-generic amd64-microcode=3.20191218.1ubuntu2, python3`.

A.2.5 Benchmarks

To evaluate `ibpb` as a mitigation, download UnixBench from `https://github.com/kdlucas/byte-unixbench` and place it under `./ibpb-eval`.

A.3 Set-up

The experiments are designed to run on bare-metal, *they will not work inside a virtualized environment*. You need an AMD processor similar to the ones we used in the paper.

A.3.1 Installation

1. Install Ubuntu 22.04
2. Install necessary dependencies (c.f. §A.2.4).
3. Boot the newly installed kernel.

A.3.2 Basic Test

Navigate to the path of the repository and run `./check.sh`. This script should show three times `PASS`. If the first line shows `PASS`, but the second or third line shows `FAIL`, all experiment but **E3** can be evaluated on your system.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): TTE allows manipulation of the BTB and RSB with code executed in transient execution.

(C2): We can manipulate the RSB from an arbitrary instruction using a PHANTOMCALL.

(C3): Our end-to-end exploit INCEPTION leaks arbitrary kernel memory.

(C4): `ibpb` can be used as mitigation against INCEPTION on Zen 1(+) and Zen 2, and has an overhead between 93.1% and 239.2%.

A.4.2 Experiments

(E1.1): [TTE of BTB] [1 minute]: this experiment executes a branch in transient execution and determines whether the state of the BTB has been changed. The experiments are described in Section 8 and depicted in Figure 9.
How to: These experiments should be carried out under `tte_btb/`.
Preparation: Install the kernel module under `kmod_ibpb`.
Execution: To build and run, run `./run_all.sh`.
Results: The script runs number of TTE tests. Lines starting with `sig_*` indicates a cache signal caused by TTE. The number is further represented within the array `rb`, where it should be significantly higher than the other numbers. The other numbers serves as indication for noise and should be low or 0.

(E1.2): [TTE of RSB] [10 minutes]: this experiment executes a branch in transient execution and determines whether the state of the RSB has been changed. The experiments are described in Section 5.2 and Section 8, and depicted in Figure 1 and Figure 9.
How to: Navigate to `./tte_rsb`.
Preparation: Before running this experiment, make sure the machine is quiescent. Find two sibling cores `CORE 1` and `CORE 2` on the target machine.
Execution: Follow the instructions in the provided `README.md`. Run `./tte_rsb.sh <CORE 1> <CORE 2> <OUTPUT DIR> <OPTIONAL CLANG ARGS>`.
Results: The output files in the `OUTPUT DIR` show the hits in the reload buffer for each return executed (one column for each return). If the RSB is uneffected by TTE, `stdout` should show a diagonal line. If this diagonal line is disturbed, entries are corrupted. If the last row of the output (`Hijacked`) shows hits, speculative return targets were hijacked using TTE.
Output files `*_16.txt` are the result of transiently executing 16 calls, and they should show a corrupted entries (disturbed diagonal line) on all AMD Zen microarchitectures. On Zen 3 and Zen 4, the output should show hijacked returns (hits in row `Hijacked`). Output files `*_32.txt` are the result of transiently executing 32 calls, and they should show hijacked returns for all AMD Zen microarchitectures. However, note that depending on the microarchitectural state, the desired number of calls do not always fit in the transient window. Therefore, output may not always show hijacked returns.

(E2): [TTE of RSB using PHANTOMCALL] [10 minutes]. This experiment shows that AMD's RSB can be manipulated with a recursive PHANTOMCALL. The experiments is described in Section 7.1 and depicted in Figure 5. The results of this artifact should resemble those shown in Figure 6. From the results produced by this experiment, it should be possible to conclude that we can hijack return instructions on all Zen microarchitecture, and that for Zen 1(+) and Zen 2 this only succeeds when a workload is running on the sibling hyperthread. However, the exact (number of) entries corrupted (and potentially returns hijacked) may differ slightly, since its dependent on various circumstances, as pointed out in the paper.
How to: Navigate to `./phantomcall/zen_1_2` when running on Zen 1(+)/Zen 2, or navigate to `./inception/zen_3_4` when running on Zen 3/Zen 4.
Preparation: Before running this experiment, make sure the machine is quiescent. Find two sibling cores `CORE 1` and `CORE 2` on the target machine.
Execution: Follow the instructions in the provided `README.md`. To start, run: `./recursive_pcall.sh {ZEN/ZEN2/ZEN3/ZEN4} <CORE 1> <CORE 2> <OUTPUT DIR> <OPTIONAL CLANG ARGS>`. As an example, if running on Zen 2, and if cores 1 and 9 are sibling hyperthreads, you may want to run: `./recursive_pcall.sh ZEN2 1 9 zen2_output`.
Results: The experiment produces up to two output files in the `OUTPUT DIR`:

1. `no_ht.txt`: this file contains the output of running the experiment on `CORE 1`, while `CORE 2` is disabled.
2. `ht.txt` (only for Zen 1, Zen + and Zen 2): this file contains the output of running the experiment on `CORE 1`, while running a workload on `CORE 2`.

The output files show the hits in the reload buffer for each return executed. The last row of the output (`Hijacked`) indicates hijacked returns (e.g. the executed return triggered the use of a transiently injected RSB entry). In case the experiment is successfull, we expect the following output:

- **no_ht**: For Zen 1(+) and Zen 2, this output should show a diagonal line which stops at a certain point, when it turns into a horizontal line. The last row (`Hijacked`) should not show hits. For Zen 3 and Zen 4, this experiment should show that we corrupt enough entries to hijack return instructions: some of the returns should show hits in the last row of the matrix (`Hijacked`).

- **ht** (only for Zen 1, Zen + and Zen 2): This output should show hits in the last row (`Hijacked`) for each column, indicating that the transient control flow was hijacked for each return executed.

(E3): [Leaking kernel memory with INCEPTION] [10 minutes]: this experiment shows that we can leak abitrary kernel memory using PHANTOMCALL, using the setup described in Section 7.3 and depicted in Figure 7.

How to: Navigate to `./inception/zen_1_2` when running on Zen 1(+) or Zen 2. Navigate to `./inception/zen_4` when running on Zen 4.

Preparation: Before running this experiment, make sure the machine is quiescent. Follow the instructions in the provided `README.md` on how to compile the required code for this experiment. Install the provided kernel module, which prints a kernel address containing a secret to `dmesg`, as described in `README.md`. When running on Zen 4, optionally enable AutoIBRS: `sudo wrmsr 0xC0000080 -a 0x200d01`.

Execution: Run INCEPTION: `./inception <KERNEL ADDRESS>`, where `KERNEL ADDRESS` can be found in the `dmesg` output.

Results: The leaked bytes are printed to `stdout`. The secret contains of 1024 As, 1024 Bs, 1024 Cs, and finally 1024 Ds.

(E4): [INCEPTION vs. `ibpb`] [10 minutes; 10 hours compute]: this experiment evaluates `ibpb` against INCEPTION.

How to: This experiment should be carried out under `./ibpb-eval`.

Preparation: Clone UnixBench, `git clone https://github.com/kdlucas/byte-unixbench`.

Execution: Run UnixBench (`./Run`) 5 times and save the results into a folder called baseline. Then reboot with the kernel parameter `retbleed=ibpb` (note: Zen3/4 requires the new kernel 6.2 parameter `retbleed=ibpb,force`) and run UnixBench 5 more times. Save the results in a folder called ibpb.

Results: To print the results in the format presented in Section 9, run `./parse.py <path>`. You may test the parser on pre-recorded results: `./parse ./raw/ee-tik-cn118`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: BunnyHop: Exploiting the Instruction Prefetcher

Zhiyuan Zhang†, Mingtian Tao†, Sioli O'Connell†,
Chitchanok Chuengsatiansup‡, Daniel Genkin§, Yuval Yarom†

† The University of Adelaide, ‡ The University of Melbourne, § Georgia Tech.

A Artifact Appendix

A.1 Abstract

We provide the artifact to demonstrate the power of BunnyHop in reverse-engineering the Instruction Prefetcher and Branch Target Buffer on Intel processors. The artifact further contains code to demonstrate BunnyHop-Reload, BunnyHop-Evict and BunnyHop-Probe in breaking KASLR and cache colored AES as well as monitoring a BTB entry cross hyper-threads.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

The evaluation of the BunnyHop-Evict involves installing a customized Linux kernel and a kernel module that does an AES encryption. To install the kernel, you may experience various *warnings* or *errors*. Please be careful when installing the kernel, and the users are on their own risk.

The provided code is only for the purpose of artifact evaluation. The authors are not responsible for any problems caused by using the provided code for other purposes.

A.2.2 How to Access

The artifact is available in GitHub repository: https://github.com/0xADE1A1DE/BunnyHop/tree/87abca5ef855593e4dc8e40e4b162d9f01026391.

The source code for cache colored kernel is available at https://doi.org/10.5281/zenodo.7704477.

A.2.3 Hardware Dependencies

To run the artifact, you need a machine with Intel processors (6th, 8th, 9th, 10th Gen), running Ubuntu OS natively (not on virtual machine). You need to enable hyper-threading.

To test the BunnyHop-Evict, a machine with Intel processor (6th $\sim$ 10th Gen) having four physical cores is necessary. Because the cache coloring we implement uses the last-level cache hash function for four core machines, we tested the BunnyHop-Evict on i7-6700 and i5-8265U.

A.2.4 Software Dependencies

You will need to install AssemblyLine and Mastik (Please refer to the README) to allocate code at any locations and use some side-channel technique APIs.

You will need essential packages to compile the customized Linux kernel. Please refer to https://phoenixnap.com/kb/build-linux-kernel for the full list of required packages.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

Users need to install AssemblyLine and Mastik before running any programs. You can find them under repository https://github.com/0xADE1A1DE/AssemblyLine and https://github.com/0xADE1A1DE/Mastik respectively.

Both AssemblyLine and Mastik are long term supported tools. In this artifact we use AssemblyLine available at https://github.com/0xADE1A1DE/AssemblyLine/tree/9fb095da7b5be01a121be9262e476f7a5cf71697 and Mastik available at https://github.com/0xADE1A1DE/Mastik/tree/8c4e550e9347e8b2f287f16f83015cd9d60414bb.

A.3.2 Basic Test

To test if two aforementioned tools are properly installed, you can run the experiment E1.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): The instruction prefetcher prefetches multiple memory lines. (E1) proves this.

(C2): The instruction prefetcher follows trained branch. (E2) proves this.

(C3): The instruction prefetcher is shared between hyper-threads. (E3) proves this.

(C4): The branch target buffer stores branches as long and short branches and different target bits are stored. (E4) proves this.

(C5): The BunnyHop-Reload technique can be used to break KASLR. (E5) proves this.

(C6): The BunnyHop-Evict technique can be used to bypass cache coloring and table preloading. (E6) proves this.

(C7): The BunnyHop-Probe technique can be used to monitor a branch status in BTB cross-threads. (E7) proves this.

A.4.2 Experiments

(E1): [1/12 human-minutes, 1/720 CPU-hour] Test prefetching depth. For more information, please refer to README under *BunnyHop/IP_RE/test_depth*

Preparation: Have AssemblyLine and Mastik installed.

Execution: Execute the *experiment.bash* to automatically run the test. The script tests for 20 memory blocks following the invoked function.

Results: Table 1 summarizes the result collected from different platforms. On 6th ∼ 10th Gen processors, you should observe that the prefetch depth is 14.

(E2): [1/12 human-minutes, 1/720 CPU-hour] Test the effect of trained branches on the instruction prefetcher. For more information, please refer to README under *BunnyHop/IP_RE/test_branch*

Preparation: Have AssemblyLine and Mastik installed.

Execution: Execute the *experiment.bash* to automatically run the test. The script tests for 60 memory blocks following the invoked function.

Results: You should observe that an instruction prefetcher follows the trained branches to prefetch memory blocks. Sample result is available under the folder.

(E3): [1/12 human-minutes, 1/720 CPU-hour] Test the behavior of the instruction prefetcher on hyper-threads. For more information, please refer to README under *BunnyHop/IP_RE/test_ip_operation*

Preparation: Have AssemblyLine and Mastik installed. Set the processor governor to performance. You will need to isolate two sibling cores at the boot time. (See README)

Execution: Execute *test_idle.bash* to run the test when the hyperthread is idle. Execute *test_busy.bash* to run the test when the hyperthread is busy with fetching infinite NOPS. You will need to change pinned cores (to two sibling cores) according to your machine configuration.

Results: You should be able to plot Figure 2 on machines with 6th ∼ 10th Intel processors.

(E4): [1/12 human-minutes, 1/720 CPU-hour] Test the target bits stored for long branch and short branch. For more information, please refer to README under *BunnyHop/BTB_RE/test_targetbits*

Preparation: Have AssemblyLine and Mastik installed. The core runs the test is isolated at the boot time.

Execution: You will need to compile the program with the command *gcc main.c -lassemblyline -o bh*. Then you execute the program with the command *taskset -c 1 ./bh > result.txt*. In the end, you plot the graph with the command *python3 plot.py*. The graph is saved as *result.py*.

Results: You should observe that the instruction prefetcher follows the trained branches to prefetch memory blocks. The sample result is available under the folder.

(E5): [1/3 human-minutes, 1/180 CPU-hour] Break KASLR with the BunnyHop-Reload. For more information, please refer to README under *BunnyHop/bunnyhop_fr*.

Preparation: You need to find the default physical address of the target branch and branch targets. Please follow the instructions on the README.

Execution: You need to compile the program with *make*. The code we provide guesses 256 BTB tag values. To run the code, execute *bash test.bash*.

Results: You will see the obtained BTB tag bits and a computed physical address after the randomization.

(E6): [5 human-minutes, 1/12 CPU-hour] Break AES and bypassing cache coloring and table preloading with the BunnyHop-Evict. For more information, please refer to README under *BunnyHop/bunnyhop_evict*.

Preparation: You need to compile and install a kernel that supports cache coloring. You also need to install an AES kernel module. Please follow the instructions on README.

Execution: You need to first obtain the address of an AES encryption and update the value (*base*) in *test.bash* under the folder *bunnyhop_evict/self-eviction/spy*. To compile and execute the code, run the command *bash test.bash*

Results: The randomly generated plaintext and timing result are saved in file *result_0xff.txt*. To plot Figure 5, you should run the command *python3 relation.py*. It reads fewer samples and plots a Pearson correlation graph. In a scenario that the measurement is noisy, you could run *python3 process.py* to find 16 peaks.

(E7): [30 human-minutes, 1/2 CPU-hour] Test the accuracy of the BunnyHop-Probe cross hyperthreads. For more information, please refer to README under *Bunny-*

Hop/bunnyhop_pp/hyperthread.
Preparation: You need to isolate two sibling cores at the boot time. You need to update the *experiment.bash* to pin the victim and spy on two sibling threads.
Execution: The experiment is similar to that of the Flush+Reload, and it requires the attacker to find a proper waiting cycles. The waiting cycles are determined by processors and CPU frequencies. More instructions on how to find proper waiting cycles are available at the README file.
Results: The result is written to *overall_result.txt* which indicates the bits that are correctly guessed. You can get an overall success rate with the command *python3 final_analyse.py*

A.5 Notes on Reusability

We provide the template code to generate aliased branches or NOPs in *BunnyHop/src*. They can be easily adapted for different purposes. We will later integrate the BunnyHop into Mastik.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Decompiling x86 Deep Neural Network Executables

Zhibo Liu, Yuanyuan Yuan, Shuai Wang*
The Hong Kong University of Science and Technology
{zliudc,yyuanaq,shuaiw}@cse.ust.hk

Xiaofei Xie
Singapore Management University
xfxie@smu.edu.sg

Lei Ma
University of Alberta
ma.lei@acm.org

A Artifact Appendix

A.1 Abstract

We provide source code of BTD and data used in our experiments. Our artifact is publicly available at `https://github.com/monkbai/DNN-decompiler/tree/b4f64783846b85cac4b0eb6c7a5595535cc858d3` with detailed documents. In the evaluation, user is able to use BTD to decompile 63 provided DNN executables into their original DNN model specifications, including ① DNN operators and their topological connectivity, ② dimensions of each DNN operator, and ③ parameters of each DNN operator, such as weights and biases, in `json` format.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact does not rise any ethical concerns. The experiments will not cause any risk for evaluators' machines security or data privacy.

A.2.2 How to access

The artifact is publicly available at `https://github.com/monkbai/DNN-decompiler/tree/b4f64783846b85cac4b0eb6c7a5595535cc858d3`.

A.2.3 Hardware dependencies

We ran our evaluation experiments on a server equipped with Intel Xeon CPU E5-2683, 256GB RAM, and an Nvidia GeForce RTX 2080 GPU. Logging and filtering all traces for all DNN executables in the evaluation takes more than a week and consumes nearly **1TB** disk storage. To ease the AE committee to review, we omit the trace logging process and provide the filtered traces in the docker image and evaluation data. The trace logger and filter are provided in MyPinTool and the trace_filter.py script. Without logging and filtering, the whole evaluation takes roughly **24 hours** and requires less than **120GB** of disk space. Besides, the symbolic execution may consume a lot of memory resources, so please make sure that the machine on which the experiment is run has sufficient memory.

*Corresponding author.

A.2.4 Software dependencies

BTD relies on IDA Pro (version 7.5) for disassembly, and because IDA is commercial software, we do not provide it in this repo; instead, in order to reduce the workload of AE reviewers, we provide the disassembly results directly as input for BTD. The scripts used to disassemble DNN executable into assembly functions with IDA are presented in our artifact. IDA Pro is not indispensable; any other full-fledged disassembly tool can be used to replace IDA.

A.2.5 Benchmarks

Table 1: Compilers evaluated in our study.

Tool Name	Publication	Developer	Version (git commit)
TVM	OSDI '18	Amazon	v0.7.0 v0.8.0 v0.9.dev
Glow	arXiv	Facebook	2020 (07a82bd9fe97dfd) 2021 (97835cec670bd2f) 2022 (793fec7fb0269db)
NNFusion	OSDI '20	Microsoft	v0.2 v0.3

Our evaluation covers above 7 models compiled with 9 different compiler options, including `Glow-2020`, `Glow-2021`, `Glow-2022`, `TVM-v0.7` (O0 and O3), `TVM-v0.8` (O0 and O3), `TVM-v0.9.dev` (O0 and O3), in total 63 DNN excutables. NNFusion-emitted executables are easier to decompile since they contain wrapper functions to invoke target operator implementations in kernel libraries (see our paper for more detailed discussion). Thus, in this evaluation we only focus on decompiling executables compiled by TVM and Glow.

Table 2: Statistics of DNN models and their compiled executables evaluated in our study.

Model	#Parameters	#Operators	TVM -O0		TVM -O3		Glow -O3	
			Avg. #Inst.	Avg. #Func.	Avg. #Inst.	Avg. #Func.	Avg. #Inst.	Avg. #Func.
Resnet18	11,703,912	69	49,762	281	61,002	204	11,108	39
VGG16	138,357,544	41	40,205	215	41,750	185	5,729	33
FastText	2,500,101	3	9,867	142	7,477	131	405	14
Inception	6,998,552	105	121,481	615	74,992	356	30,452	112
Shufflenet	2,294,784	152	56,147	407	34,637	228	33,537	59
Mobilenet	3,487,816	89	69,903	363	46,214	228	37,331	52
Efficientnet	12,966,032	216	89,772	546	49,285	244	13,749	67

A.3 Set-up

A.3.1 Installation

Download the packed docker image, then run the command below to unpack the .tar file into a docker image.

```
cat BTD-artifact.tar | docker import - btd
```

Create a container with the docker image.

```
docker run -dit --name BTD-AE btd /bin/bash
```

Open a bash in the container:

```
docker exec -it BTD-AE /bin/bash
cd /home
```

BTD can also be installed from source code, the detailed instructions are listed in our artifact.

A.3.2 Basic Test

To run the evaluation of operator inference:

```
cd DNN-decompiler
git pull
./op_infer_eval.sh
```

Inference results are written in the `output` directory. The output would be in format: `Compiler Option-Model-Operator Name/Type Pred: output`. For example, the output below indicates that a `libjit_fc_f` (Fully-Connected, FC) operator in the vgg16 model compiled with `Glow_2021` is *correctly* inferred as matmul (Matrix Multiplication).

```
GLOW_2021-vgg16-libjit_fc_f Pred: matmul
GLOW_2021-vgg16-libjit_fc_f Label: matmul
```

To run the evaluation of decompilation and rebuild:

```
cd DNN-decompiler
git pull
./decompile_eval.sh
```

This experiment will decompile and rebuild all 63 DNN executables. It takes 24 hours to finish all experiments. The output of rebuilt models and original DNN executables will be printed on screen (see example in Decompilation Correctness below). Corresponding decompilation outputs will be stored in the `evaluation` directory.

After executing `decompile_eval.sh`, for each directory in `evaluation`, a `topo_list.json` containing the network topology (①), a `new_meta_data.json` containing dimensions information (②), and a series of `func_id.weights/biases_id.json` containing all parameters of the decompiled DNN model (③) will be generated.

Each item in `topo_list.json` will be: [‘node id’, ‘func_id.txt’, ‘operator type’, [input addresses], ‘output address’, [input node ids], occurrence index].

Each item in `new_meta_data.json` will be: [‘<func_id>.txt’, [operator dimensions], ‘operator entry address (in executable)’, ‘operator type’, with_parameter, stride (if exists), padding (if exists)].

Examples can be found in README.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BTD is able to decompile all 63 DNN executables into model specifications that are (near) identical with input models. The decompiled model specifications can be used to rebuild new models that have identical output (with minor precision loss) as the output of original DNN executables.

A.4.2 Experiments

After decompilation experiments, all DNN model are rebuild with decompiled model structures and extracted parameters (stored in `.json` format). `decompile_eval.sh` will run each rebuilt model (implemented in pytorch) and the original DNN executable with the example image in binary format as input. The output would be like this:

```
- vgg16_tvm_v09_O3
- Rebuilt model output:
Result: 282
Confidence: 9.341153
```

```
- DNN Executable output:
Result: 282
Confidence: 9.341150
```

In the above example, both rebuilt model and DNN executable output result as `282` (see 1000 classes of ImageNet), and the confidence scores are `9.341153` and `9.341150`, respectively. While the confidence scores (or max values) are slightly inconsistent, we interpret that such inconsistency is caused by the floating-point precision loss between pytorch model and DNN executable, i.e., the decompilation is still *correct*.

(E1): [Decompilation Correctness] [10 human-minutes + 24 compute-hour + 120GB disk]: as described above.
How to: As described in A.3.2 **Basic Test**.
Results: The predicted label output by the original DNN executable and the rebuilt model should be identical.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Every Signature is Broken: On the Insecurity of Microsoft Office's OOXML Signatures

Simon Rohlmann
Ruhr University Bochum

Vladislav Mladenov
Ruhr University Bochum

Christian Mainka
Ruhr University Bochum

Daniel Hirschberger
Ruhr University Bochum

Jörg Schwenk
Ruhr University Bochum

A Artifact Appendix

A.1 Abstract

In our paper "Every Signature is Broken: On the Insecurity of Microsoft Office's OOXML Signatures", we present seven attacks, divided into five attack classes, on signed OOXML Word documents. The goal of each attack is to manipulate the displayed document content without invalidating the signature.

The attack classes Content Injection Attack (CIA), Content Masking Attack (CMA) and Legacy Wrapping Attack (LWA) are based on specification flaws that allow the attacker to correctly reference subsequently added files within the OOXML package. The root problem here consists of only partially signed relationship files. The Universal Signature Forgery (USF) and Malicious Repair Attack (MRA) attack classes exploit implementation flaws in the corresponding OOXML application. The reviewers can test the attacks under different Microsoft Office and OnlyOffice desktop versions during the evaluation. Since different versions of Microsoft Office cannot be installed simultaneously under one operating system, the applications must be installed individually. A bare-metal solution with macOS Monterey is required for the macOS versions of Microsoft Office and OnlyOffice Desktop.

We provide proof of concept (PoC) files for each attack to the reviewers. Here, the document numbered *01* is the source document, *02* is the first manipulation step (intermediate step), and *03* is the ready-to-run exploit. There is one exception for CIA, where the document *02* corresponds to the exploit.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The attacks are aimed at manipulating signed OOXML Word documents. The documents do not contain any macro code or similar that could influence the operating system. We have reported all vulnerabilities found during our investigations to Microsoft, OnlyOffice, as well as to the responsible standardization committee ISO/IEC JTC 1/SC 34. The vulnerabilities have been acknowledged by Microsoft. However, Microsoft has decided that the vulnerabilities do not require immediate attention. According to Microsoft, a potential fix in the future is not excluded. OnlyOffice has announced they are working on a fix.

A.2.2 How to access

- Stable URL: Link
- PoC files: Link
- To download Microsoft Office 2019, 2021 and 365, the Office Deployment Tool must be used with the appropriate configuration file:
  - Office Deployment Tool: Link
  - Configurations files: Link
- Microsoft Office 2016: Link
- Microsoft Office 2013: Link
- Microsoft Office for macOS 2019/2021/365: Link
- OnlyOffice Desktop: Link
- 7-Zip: Link
- Notepad++: Link

A.2.3 Hardware dependencies

A computer running Windows 10 and Ubuntu 22.04.1 or an equivalent Virtual Machine (VM) is required.

An Apple hardware running macOS Monterey is required to evaluate the attacks against Microsoft Office and OnlyOffice for macOS.

A.2.4 Software dependencies

Possibility to install the office packages.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

The packages for OnlyOffice and Microsoft Office 2013, 2016, as well as the macOS versions of Microsoft Office can be downloaded and installed directly. For Microsoft Office 2019, 2021 and 365, the Office Deployment Tool must be installed. After installation, *setup.exe* must be started from the console and the configuration file must be passed as an argument: *setup.exe /configure config-file.xml*

Since the certificate used to sign the PoC files was only valid for three months and expired on September 5th, 2022, the host system date must be set accordingly to a date between June 7th and September 5th, 2022. If the date is not corrected accordingly, a recoverable signature is displayed in Microsoft Word instead of a valid document signature.

A.3.2 Basic Test

From the PoCs folder, open the file */5-1_CIA/01_document_signed_by_trusted_entity.docx*. Microsoft Office will display a valid document signature on opening the document if everything is set up correctly. If a recoverable signature is displayed, the date was not reset correctly or there is no internet connection available to verify the certificate.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The attacker can manipulate a signed OOXML Word document. Microsoft Office for Windows displays the content selected by the attacker without invalidating the signature. This is proven by experiment **E1**. Sections 5, 6, and 7 of our paper contain the description of the attack technique, as well as our evaluation results. For this experiment, the attack classes CIA, CMA, LWA, USF, and MRA are relevant.

(C2): The attacker can manipulate a signed OOXML Word document. Microsoft Office for macOS displays the content selected by the attacker. The status of the signed document is displayed unchanged as a document protected by a digital signature. This is proven by experiment **E2**. Section 3 of our paper contains our evaluation results. Microsoft Office under macOS does not cryptographically check the signature, so direct document manipulation is possible. A dedicated attack technique is not required.

(C3): The attacker can manipulate a signed OOXML Word document. OnlyOffice Desktop for Windows, macOS, and Linux displays the content selected by the attacker without invalidating the signature. This is proven by experiment **E3**. Sections 5, 6, and 7 of our paper contain the description of the attack technique, as well as our evaluation results. For this experiment, the attack classes CIA, CMA, LWA, and MRA are relevant.

A.4.2 Experiments

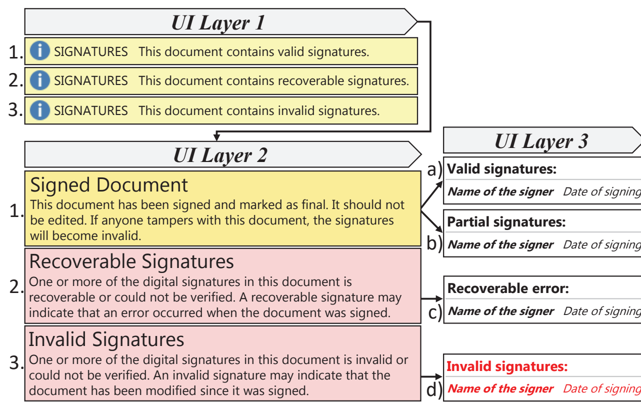


Figure 1: Representation of the different UI layers of Microsoft Office (Windows). A valid signature can combine the three UI layers *1. a)* or *1. b)*. *2. c)* corresponds to a signature with an unknown certificate state, e.g., if the certificate has expired, has been self-signed, or there is no Internet connection to validate the chain of trust. *3. d)* corresponds to an invalid signature, e.g., because an attacker modified the signed content. OnlyOffice Desktop provides only one UI layer to show the status of the document signature. This is essentially the same as the UI layer 3 shown and can reflect the combination *a)* or *d)*.

(E1): *[1 human-hour + 2 compute-hours + 10GB disk]: Manipulation of signed OOXML Word documents under Microsoft Office for Windows.*

Preparation: Download the provided PoC files to the system and launch a Windows version of Microsoft Office. All versions of Microsoft Office for Windows are equally vulnerable to the CIA, CMA, LWA, USF, and MRA attack classes. Microsoft Office 2013 has a bug that causes the UI layer 3 (Figure 1) to be left blank for any signed OOXML Word documents. The MRA attacks work on Microsoft Office 2019 and 2021 only if the office application has been activated/licensed. Set your host system to a date between June 7th and September 5th, 2022. The host system needs an Internet connection to validate the certificate trust chain.

Execution: One by one, open the files starting with *01* and *03* from the folders of the downloaded PoC files. The files *01* correspond to the unmanipulated original documents. Files *03* correspond to the documents signed

by the trusted entity and manipulated by the attacker. The CIA attack is an exception to this rule. Here, the file starting with *02* corresponds to the document manipulated by the attacker. Microsoft Office prompts to repair the documents when opening the files starting with *03* of attack class MRA. This repair prompt is part of the attack and must be confirmed. When closing, the document must not be saved, as this removes the attack vector.

Results: All documents starting with *01* show the content:

> *This document was created and validly signed by a trusted entity.*

All documents manipulated by the attacker contain the content:

> *This document has been manipulated by the attacker.*
> *The content is chosen by the attacker.*

The CMA attack in the *5-2_CMA_Font_Inj* subfolder is an exception. In the attacker document starting with *03*, all numbers were replaced with *6*, and the name was changed to *EVIL*.

For all attacks the UI layer 1 shows a valid signature (see Figure 2). Due to the use of test licenses of Microsoft Office applications, it may happen that the UI layer 1 is only displayed before or, in some versions, after the license query (to be canceled).

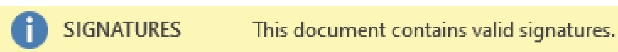


Figure 2: Valid signature under Microsoft Office's UI layer 1.

UI layer 2 is displayed after clicking on *File* (top left). The content of UI layer is the same as shown in Figure 3.

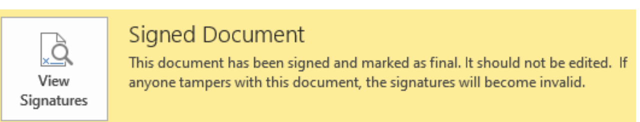


Figure 3: Valid signature under Microsoft Office's UI layer 2.

From UI layer 2, UI layer 3 can be opened by clicking on *View Signatures*. Here, the original signer *trusted.person.ooxml@gmail.com* is displayed as the signer for each manipulated document, although the content differs from the original signed document. For the CIA, CMA, LWA, MRA attack classes the signature type is specified as *Partial signature*, while for the USF attack class the signature type is specified as *Valid signature* (see *(a)* and *(b)* in Figure 1).

(E2): *[15 human-minutes + 10 compute-minutes + 5GB disk (for software)]: Manipulation of signed OOXML Word documents under Microsoft Office for macOS.*

Preparation: Download the provided PoC files to the system and launch a macOS version of Microsoft Office.

Execution: Open *01_document_signed_by_trusted_entity.docx* and *02_direct_manipulation_by_attacker.docx* files one by one.

Results: The document starting with *01* shows the content:

> *This document was created and validly signed by a trusted entity.*

The document manipulated by the attacker (starting with *02*) contains the content:

> *This document has been manipulated by the attacker.*
> *The content is chosen by the attacker.*

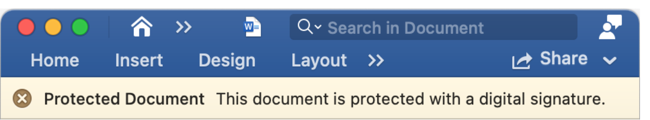


Figure 4: Message about a document protected by a signature in Microsoft Office for macOS.

Microsoft Office for macOS does not provide the UI layer described for the Windows variants. Microsoft Office for macOS only issues a message about a document protected by a signature (see Figure 4). This message also appears for the document manipulated by the attacker, although the included signature file (*sig1.xml*) does not contain any signature-relevant information. This shows that Microsoft Office does not perform any cryptographic verification of the signature.

(E3): *[30 human-minutes + 30 compute-minutes + 5GB disk)]: Manipulation of signed OOXML Word documents under OnlyOffice Desktop for Windows, macOS, and Linux.*

Preparation: Download the provided PoC files to the system and launch a version of OnlyOffice Desktop. All versions of OnlyOffice Desktop are equally vulnerable to the CIA, LWA, and MRA attack classes, as well as the style injection attack from the CMA attack class.

Execution: One by one, open the files starting with *01* and *03* from the folders of the downloaded PoC files. The files *01* correspond to the unmanipulated original documents. Files *03* correspond to the documents signed by the trusted entity and manipulated by the attacker. The

CIA attack is an exception to this. Here, the file starting with *02* corresponds to the document manipulated by the attacker.

Results: All documents starting with *01* show the content:

> *This document was created and validly signed by a trusted entity.*

All documents manipulated by the attacker contain the content:

> *This document has been manipulated by the attacker.*
> *The content is chosen by the attacker.*

The CMA attack in the *5-2_CMA_Style_Inj* subfolder is an exception. Since OnlyOffice has limited vulnerability to this attack, attackers can only hide existing content, but they cannot add new content. Thus, OnlyOffice displays the document manipulated by the attacker as an empty document.

Figure 5: Valid signature under OnlyOffice's UI layer.

OnlyOffice shows the signer and the signature status on the right side after opening the signed documents. On Windows, macOS, and Linux, a valid signature is displayed. Under Windows, the signer *trusted.person.ooxml@gmail.com* is displayed directly, as shown in Figure 5. On macOS and Linux, the issuer of the certificate is displayed first. With a right-click on the issuer, the certificate details can be shown. The same signer (*trusted.person.ooxml@gmail.com*) is displayed within the subject section (*CN*) of the certificate.

(E4): *[1 human-hour)]: Reproducibility of the manipulation of the displayed content of a signed OOXML document.*

The following steps describe the procedure to reproduce that the displayed content is not protected by the signature and thus remains arbitrarily manipulable. This applies to the attack classes CIA, CMA, LWA, and MRA. The USF attack requires that correct hash values are generated for the displayed content. The procedure required for this is described in our paper. The manipulations require 7-Zip and Notepad++.

CIA:

1. right click on *02_CIA_manipulated_by_attacker.docx*. In the subitem 7-Zip click on *Open archive*.
2. in the archive in the subfolder *word/* extract the *people.xml"*.
3. now open *people.xml* with Notepad++.
4. between the first element *<w:t></w:t>* you will find the displayed text. This text can be manipulated arbitrarily.
5. now insert the manipulated *people.xml* back into the archive in the subfolder *word/* and overwrite the old file.
6. When opening the document, the inserted content is now displayed, while the signature status remains valid.

CMA Font Inj.:

1. right-click on *03_CMA_Font_Inj_manipulated_by_attacker.docx*. In the subitem 7-Zip click on *Open archive*.
2. extract the *document.xml.rels* in the archive in the subfolder *word/_rels/*.
3. now open *document.xml.rels* with Notepad++.
4. delete the entry *<Relationship Id="rId4" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/fontTable" Target="fontTable.xml"/>*. This will remove the link to the malicious embedded fonts.
5. now add the manipulated *document.xml.rels* back into the archive in the *word/_rels/* subfolder and overwrite the old file.
6. when opening the document, the content without malicious fonts is now displayed, while the signature status remains valid.

CMA Style Inj.:

1. right-click on *03_CMA_Style_Inj_manipulated_by_attacker.docx*. In the subitem 7-Zip click on *Open archive*.
2. in the archive in the subfolder *word/_rels/* extract the *document.xml.rels*.
3. now open *document.xml.rels* with Notepad++.
4. delete the entry *<Relationship Id="rId1" Type="http://schemas.openxmlformats.org/office*

Document/2006/relationships/styles" Target="styles.xml"/>. This will remove the link to the malicious style elements.
5. Now add the manipulated *document.xml.rels* back into the archive in the *word/_rels/* subfolder and overwrite the old file.
6. when opening the document, the content without malicious style elements is now displayed, while the signature status remains valid.

LWA:

1. right-click on *03_LWA_manipulated_by_attacker.docx*. In the subitem 7-Zip click on *Open archive*.
2. in the archive in the subfolder *word/* extract the *document.xml*.
3. now open *document.xml* with Notepad++.
4. between the first element <*w:t*></*w:t*> is the displayed text. This text can be manipulated arbitrarily.
5. now insert the manipulated *document.xml* back into the archive in the subfolder *word/* and overwrite the old file.
6. when opening the document, the inserted content is now displayed, while the signature status remains valid.

MRA DDA:

1. right click on *03_DDA_manipulated_by_attacker.docx*. In the subitem 7-Zip click on *Open archive*.
2. in the archive in the subfolder *word/* extract the *document2.xml*.
3. now open *document2.xml* with Notepad++.
4. between the first element <*w:t*></*w:t*> is the displayed text. This text can be manipulated arbitrarily.
5. now insert the manipulated *document2.xml* back into the archive in the subfolder *word/* and overwrite the old file.
6. When opening the document, the inserted content is now displayed, while the signature status remains valid.

MRA ETA:

1. right click on *03_ETA_manipulated_by_attacker.docx*. In the subitem 7-Zip click on *Open archive*.
2. in the archive in the subfolder *word/* extract the *document.xml*.
3. now open *document.xml* with Notepad++.
4. between the first element <*w:t*></*w:t*> is the displayed text. This text can be manipulated arbitrarily.
5. now insert the manipulated *document.xml* back into the archive in the subfolder *word/* and overwrite the old file.
6. when opening the document, the inserted content is now displayed, while the signature status remains valid.

Direct manipulation under macOS:

1. create a signed Word document using Microsoft Office for Windows.
2. right click on the document. In the subitem 7-Zip click on *Open archive*.
3. in the archive in the subfolder *word/* extract the *document.xml* and in the subfolder *_xmlsignatures/* extract the *sig1.xml*.
4. now open *document.xml* with Notepad++. between the first element <*w:t*></*w:t*> is the displayed text. This text can be manipulated arbitrarily.
5. now insert the manipulated *document.xml* again into the archive in the subfolder *word/* and overwrite the old file.
6. now open *sig1.xml* with Notepad++.
7. delete the entire content of *sig1.xml*.
8. insert the manipulated *sig1.xml* back into the archive in the subfolder *_xmlsignatures/* and overwrite the old file.
9. when opening the document, the inserted content is now displayed, while the document is still displayed as a document protected by a signature.

A.5 Notes on Reusability

Since the certificate used to sign the PoC files was only valid for three months and expired on September 5th, 2022, the host system date must be set accordingly to a date between June 7th and September 5th, 2022.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix: Security Analysis of MongoDB Queryable Encryption

Zichen Gui, Kenneth G. Paterson, and Tianxin Tang
Department of Computer Science, ETH Zurich, Zurich, Switzerland

August 9, 2023

A Artifact Appendix

A.1 Abstract

This artifact appendix is provided alongside our paper. In this appendix, we describe the two phases of our attack on MongoDB QE. The first phase is *leakage extraction* (Section 4), followed by the second phase *inference attacks* (Section 7). Due to the complications with QE, leakage extraction is not feasible on a large-scale database, and we had to work with simulated leakage in our experiments (see Appendix B). Therefore, we provide two procedures for leakage extraction, one for real leakage (E1), and the other for simulated leakage (E2). The procedures for inference attacks exploiting leakage from `queryLog` and `opLog` can be found in E3 and E4, respectively.

A.2 Description & Requirements

Requirements. Leakage simulation and inference attacks require resources 10 GB RAM, 16 GB disk, and $\sim$ 3.3 GHz CPU on a single core (is recommend). For small-scale leakage extraction (e.g., 3K - 10K records), the same specification is adequate. However, full-scale leakage extraction involving building a database containing 3M records, requires 50 GB RAM and 600 GB disk. The instructions in this artifact only work for Linux/Unix systems. Ubuntu 22.04 is used for evaluation.

A.2.1 Security, privacy, and ethical concerns

In our experiments, we use the anonymized American Community Survey (ACS) micro data on the person level from 2012 and 2013 and the corresponding codebook, publicly available from https://www.census.gov/programs-surveys/acs/microdata.html. Our inference attacks do not in any way attempt to deanonymize this data.

A.2.2 How to access

URL. https://gitlab.com/mongodbqe/mongo/-/commit/4e9fc09377f26e1760fb510a0b998f777fd9e0f4

README.md and FAQ. We provide a `README.md` in our code repo for more comprehensive instructions. The `README.md` also contains a FAQ section to address common issues that you may encounter during evaluation.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Our artifact is evaluated on MongoDB 6.0.7.

General. The links provided above direct you to the packages of the latest version only. To reproduce our results with the specific versions of the packages we used in our artifact, please refer to the following example. The instruction works specifically for Ubuntu 22.04. We provide further instruction for other operating systems later.

- `mongod` from `Archive` and `crypt_shared` (6.0.7): https://www.mongodb.com/download-center/enterprise/releases
- `libmongocrypt` (1.7.4): https://www.mongodb.com/docs/manual/core/csfle/reference/libmongocrypt
- `mongoexport` (100.7.3): https://www.mongodb.com/docs/database-tools/installation/installation/
- `mongosh` (1.10.1): https://www.mongodb.com/try/download/shell
- Python 3.10 and the python package dependencies listed in Section A.3.1.

Example for Ubuntu 22.04.

- `mongod Archive` (6.0.7): https://downloads.mongodb.com/linux/mongodb-linux-x86_64-enterprise-ubuntu2204-6.0.7.tgz
- `crypt_shared` (6.0.7): https://downloads.mongodb.com/linux/mongo_crypt_shared_v1-linux-x86_64-enterprise-ubuntu2204-6.0.7.tgz
- `libmongocrypt` (1.7.4): Please refer to `README.md` in the code repository for the commands.

- mongoexport (100.7.3): https://fastdl.mongodb.org/tools/db/mongodb-database-tools-ubuntu2204-x86_64-100.7.3.tgz
- mongosh (1.10.1): https://downloads.mongodb.com/compass/mongodb-mongosh_1.10.1_amd64.deb
- Python 3.10 and the python package dependencies listed in Section A.3.1.

Similarly, for other operating systems/architectures, you can obtain the packages of specific versions by modifying the operating system/architecture of the links above. See the download links in "General" for examples.

A.2.5 Benchmarks

We use ACS 2012 as *auxiliary data* and ACS 2013 as *recovery target* in our experiments.

ACS 2012 https://www2.census.gov/programs-surveys/acs/data/pums/2012/1-Year/csv_pus.zip

ACS 2013 https://www2.census.gov/programs-surveys/acs/data/pums/2013/1-Year/csv_pus.zip

A.3 Set-up

A.3.1 Installation

1. Download or `git clone` the repo `mongo` from the provided URL in Section A.2.2.
2. Download `csv_pus.zip` for ACS 2012, ACS 2013, respectively listed in Section A.2.5. Unzip the files and get `ss12pusa.csv`, `ss12pusb.csv`, `ss13pusa.csv`, and `ss13pusb.csv`.
3. Place `ss12pusa.csv` and `ss12pusb.csv` in `mongo/acs_data/2012_person_records`; place `ss13pusa.csv` and `ss13pusb.csv` in `mongo/acs_data/2013_person_records`.
4. Work in `mongo/src/` and create a python virtual environment and install the required packages:
   - `python3 -m pip install --user virtualenv`
   - `python3 -m venv env`
   - `source env/bin/activate`
   - `pip3 install -r requirements.txt`
5. Install `mongod`, `crypt_shared`, `libmongocrypt`, `mongoexport`, `mongosh`; urls are listed in Section A.2.4.
6. Configure the library dependencies listed in 5 in `mongo/src/parameters.py`, from line 53 to line 56.
7. Configure the default MongoDB database path at line 57 in `mongo/src/parameters.py`. Please specify a directory that does not require root access.

A.3.2 Basic Test

Working in `mongo/src`, run `python3 check_config.py` to check whether the installation and configuration listed in Section A.3.1 are complete.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Real leakage about the data encrypted by QE can be extracted from `opLog` alone or from `queryLog` and encrypted document collection (Appendix B).

(C2): Leakage simulations for `opLog` and `queryLog` pass the correctness check, respectively, matching the real leakage (Appendix B).

(C3): The inference attack exploiting simulated query leakage (under uniform and Zipf distributions, with 100, 300, 500 queries per field) from `queryLog` achieve reasonable recovery rates (Section 7.2).

(C4): The inference attack exploiting simulated compaction leakage from `opLog` achieve reasonable recovery rates (Section 7.2).

A.4.2 Experiments

We have repeated our experiments for statistical reasons. The number of experiments can be adjusted based on time and resource constraints. Please refer to `README.md` for details. **Note that E3 and E4 can be run concurrently to reduce the waiting time.**

Sample output. Sample output for E1-E4 is provided in `mongo` repo. E.g., `E1_sample_output.txt`.

(E1): Leakage extraction (at a small scale). 5 human-minutes + 10 compute-minutes + 16 GB disk. The default number of records for this artifact is 3K. You can adjust the number of records using `--limit` argument. A full-scale leakage extraction with 3M records may take 2-4 days.
Preparation: Working in `mongo/src`, run `python3 export_acs_data_sample.py` to generate auxiliary information.
Execution: `python3 main.py` to collect `queryLog` and `opLog`, extract leakage, check its correctness for the sample dataset.
Results: Real leakage about the data can be extracted from logs successfully.

(E2): Leakage simulation. 5 human-minutes + 40 compute-minutes (depending on the number of experiments) + 16 GB disk:
Preparation: Make sure the setup stage in Section A.3 is complete. Work in `mongo/src`.
Execution: `python3 export_acs_data_simulated.py --start=0 --end=1` generates one instance of simulated leakage.
Results: Leakage is simulated for `opLog` and `queryLog` correctly.

(E3): Inference attack with simulated query leakage from `queryLog`. 5 human-minutes + 3 compute-hours (or 2 minutes if using `--fast`) + 16 GB disk:
Preparation: Work in `mongo/src_attack`. (Optional) Core attack parameters such as the number of it-

erations can be set from line 12 to line 29 of `mongo/src_attack/attack.py`.
Execution: If using simulated leakage from `queryLog` generated in E2, then run: `python3 attack.py --uniform n` for `n` queries from uniform distribution, `n in {100, 300, 500}`. Using `--zipf n` for Zipf distribution.
Results: The inference attack using the simulated query leakage achieves a reasonable recovery rate (see C3).

(E4): Inference attack with simulated compaction leakage from `opLog`. 5 human-minutes + 14 compute-hours (or 20 minutes if using `--fast`) + 16 GB disk:
Preparation: Same as in E3.
Execution: `python3 attack.py --oplog`
Results: The inference attack using simulated compaction leakage from `opLog` achieves a reasonable recovery rate (see C4).

A.5 Acknowledgement

We are grateful to the anonymous reviewers for their contributions in the artifact evaluation process. By incorporating their suggestions and refining the artifact, we have significantly enhanced its quality!

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.

USENIX'23 Artifact Appendix
AutoFR: Automated Filter Rule Generation for Adblocking

Hieu Le* Salma Elmalaki* Athina Markopoulou* Zubair Shafiq†

*University of California, Irvine †University of California, Davis

A Artifact Appendix

Adblocking relies on filter lists, which are manually curated and maintained by a community of filter list authors. We introduce AutoFR, a reinforcement learning (RL) framework to fully automate the process of filter rule creation and evaluation for sites of interest. Examples of filter rules are in Table 1. AutoFR is the first to balance the trade-off between blocking ads *vs.* avoiding visual breakage. The user gives AutoFR inputs (*e.g.,* the website to generate rules for, and breakage tolerance threshold w) to AutoFR. It will run our RL algorithm based on multi-arm bandits and generate filter rules that block ads while adhering to the given w threshold. This appendix details how to access our artifact (implementation of AutoFR and our dataset) and how to use and evaluate it.

A.1 Abstract

Our artifact includes the following. First, we open-source an implementation of the AutoFR framework on GitHub. Second, we provide our dataset of collected site snapshots on the Top–5K sites, which can be utilized to reproduce the filter rules we created or explore other algorithms to generate rules.

AutoFR's implementation follows Algorithm 1 and is illustrated in Fig. 4. Notably, it uses site snapshots (Fig. 5 and Sec 4.1), which are graph representations of how a site is loaded. We use them offline to run the reinforcement learning logic, which removes the bottleneck of waiting for a site to load during every visit. See Fig. 5 for more details.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

At its core, AutoFR visits websites automatically and creates filter rules that block ads with minimal visual breakage. Thus, there may be security issues if the user gives AutoFR a malicious site to visit. We advise testing AutoFR on sites that the user trusts. In terms of privacy, if AutoFR is used on a personal machine, websites may fingerprint or track the utilization of AutoFR.

A.2.2 How to access

GitHub: The repository is listed at `https://github.com/UCI-Networking-Group/AutoFR/tree/artifact-review`. It provides a detailed README.md on how to use AutoFR. The rest of this appendix will refer to `https://github.com/UCI-Networking-Group/AutoFR/tree/artifact-review`.

Dataset: The dataset and its detailed description are available at `https://athinagroup.eng.uci.edu/projects/ats-on-the-web/autofr-dataset/`. In summary, the dataset contains 1042 zip files, one per-site. Each zip file includes the raw collected data of outgoing HTTP requests, AdGraphs, annotated site snapshots, the action space, filter rules, and more. This matches Table 2. This includes a *"Top5k_rules.csv"* file that shows all the filter rules created within each zip file. Users must sign a consent form (at the bottom of the web page) before accessing the dataset. For *artifact reviewers*, we provide the direct Google Drive link to the dataset within a hotcrp comment.

A.2.3 Hardware dependencies

AutoFR was evaluated using Amazon EC2 instance m5.2xlarge, which has 8 cores, 32 GiB of memory, 35 GiB of storage, and up to 10 Gbps of network bandwidth. We recommend something similar, going as low as 16 GiB of memory with 20 GiB of storage. Our repository will provide a Dockerfile for easy setup.

Limitations. Currently, we do not support the running of AutoFR on M1 MacBooks (ongoing work to support it).

A.2.4 Software dependencies

AutoFR has been tested on a Debian 5.10 server (university) and Ubuntu 18.04.6 LTS (AWS EC2). Implementing the framework includes using several Python libraries, browser extensions, and prior work. The majority of the dependencies will be encapsulated in a Dockerfile. We list the major ones below and refer to the README.md of `https://github.com/UCI-Networking-Group/AutoFR/tree/artifact-review` for details.

Core Dependencies (Must Haves):

- Python 3.6+, git, pip3, virtualenv (or conda), docker
- If necessary, install with:
  1. sudo apt-get install git python3 python3-dev python3-pip
  2. pip3 install virtualenv
  3. We defer the docker installation to https://docs.docker.com/engine/install/debian/.

Dependencies (within Dockerfile):

- Python 3.6+: tldextract, networkx, adblockparser, pandas, numpy, selenium
- NodeJS: Ad Highlighter, Adblock Plus (browser extensions)
- C++: AdGraph (instrumented chromium)

A.2.5 Benchmarks

None

A.3 Set-up

For easy copy and paste of commands, we recommend using the https://github.com/UCI-Networking-Group/AutoFR/tree/artifact-review#setup.

A.3.1 Installation

1. Git clone our AutoFR repository (see Sec. A.2.2). The rest of the instructions assume you are in the project directory using a terminal window.
2. For artifact reviewers: "git checkout artifact-review"
3. git submodule update --init --recursive
4. Create a python virtual environment and activate it. We recommend using "virtualenv".
   (a) virtualenv --python=python3 [/save-path/autofrenv]
   (b) source [/save-path/autofrenv]/bin/activate
5. Install AutoFR:
   (a) pip3 install -e .
   (b) Make sure to have the period at the end of the command.
   (c) mkdir temp_graphs; mkdir -p data/output/
6. Build the docker container that AutoFR leverages:
   (a) docker build -t flg-ad-highlighter-adgraph --build-arg USER_ID=$(id -u) --build-arg GROUP_ID=$(id -g) -f framework-with-ad-highlighter/DockerAdgraphfile .
   (b) Make sure to have the period at the end of the command. This should run without any errors.
7. Done: You are now ready to run AutoFR.

A.3.2 Basic Test

Ensure you are in the project directory with a terminal window and your virtualenv activated as instructed in Sec. A.3.1.

1. Test whether your AutoFR environment has the necessary dependencies:
   (a) python scripts/autofr_controlled.py
   (b) The above command should print out a help message on how to use the script without errors.
2. View the docker image that you created:
   (a) docker image ls | grep flg-ad-highlighter-adgraph
   (b) The above command should print out the docker image called "flg-ad-highlighter-adgraph" with additional information such as its size.

A.4 Evaluation workflow

Disclaimer. As noted in our paper, the web changes naturally. AutoFR is only as good as its components. Thus, if a site does not serve ads that Ad Highlighter can detect or use obfuscation techniques, then AutoFR may not be able to generate rules for the given site. See Sec. 5.3.4 and 4.3. There may be other factors, such as w being too high to generate rules for, etc... Over time, AutoFR will improve as we maintain it, but we cannot guarantee that it will work on every website.

A.4.1 Major Claims

(C1): *Create Filter Rules:* Given inputs such as a website and hyper-parameters like the w threshold (breakage tolerance), AutoFR will generate filter rules that block ads with breakage that is within the w threshold. This is proven by the experiment (**E1**). Our results for the Top–5K sites are reported in Sec. 5.1, Table 2, Fig. 6(a-b), and Table 3 column 2. The w threshold ranges from 0–1; higher values mean the user wants to avoid breakage at the expense of not finding any filter rules that meet that criterion. In our paper, we use $w = 0.9$. See Sec. 3 and particularly 3.2.2 for details about our formulation.

(C2): *Reproducibililty:* Researchers can reproduce our results (*i.e.,* generate the same rules) by utilizing our collected site snapshots (provided in our dataset). This assumes the inputs to AutoFR are identical, and the same changeset/version of AutoFR is utilized (Sec. A.2.2). This is proven by the experiment (**E2**). Site snapshots are described in Sec. 4.1, Fig. 5, and Table 2. See further discussion in Sec. A.5.

A.4.2 Experiments

(E1): *Create Filter Rules: [10 human-minutes + compute-minutes vary on server + storage varies on site] × per-site*. See (**C1**) for more information.

How to: Run AutoFR to generate rules for a few given sites. Results will vary based on context, such as on the site, location, and given inputs. Repeat the below for a few sites. We recommend cricbuzz.com, yahoo.com, and sohu.com.

Preparation: Follow the instructions in Sec. A.3.1.

Execution:

1. Follow the below:
2. python scripts/autofr_controlled.py --site_url "https://cricbuzz.com" --chunk_threshold 6
3. The chunk size affects the number of docker instances spawned to visit the given website. Based on Sec. A.2.3, we recommend 6. If you have fewer cores, then decrease the chunk_threshold.

Results:

1. Follow the below. Directories given are relative to the project directory:
2. The terminal will display the rules that are outputted.
3. Go to directory "data/output/" to see the raw collected data, such as the outgoing HTTP requests, AdGraphs, and site snapshots.
4. Go to "temp_graphs" to see the outputted filter rules and other information.
5. Full explanation of the output is explained in our README: https://github.com/UCI-Networking-Group/AutoFR/tree/artifact-review#understanding-the-output.

Test the Rules In-the-Wild (optional):

1. Follow the below if you want to try the rules in your browser.
2. Install an adblocker, like Adblock Plus, into your browser (instructions depend on your browser).
3. Turn the rules given by AutoFR into per-site rules. For each rule, append the site it was created for. For instance, if the rule is ||*doubleclick.net*^ for the site cricbuzz.com, then change it to ||*doubleclick.net*^$domain=cricbuzz.com.
4. Configure the extension by going to its settings. Turn off all filter lists. Add in custom rules from the previous step. See https://help.adblockplus.org/hc/en-us/articles/360062859913-Add-a-custom-filter.
5. Refresh the site to see if ads are blocked. Note if there is any visual breakage.
6. Remember to undo the changes if you use the adblocker personally.

(E2): *Reproducibility: [5 human-minutes + 2 compute-minutes + no storage] × per-site*. See (**C2**) for more information. This is completely offline.

How to: Run AutoFR with existing site snapshots to reproduce the results. Repeat the below for a few sites. We recommend cricbuzz.com, yahoo.com, and sohu.com.

Preparation: Download the "Top5K_rules.csv" file. Open it and choose a zip file to download, described in Sec. A.2.2. Here we assume you chose AutoFRGEval_www.cricbuzz.com_ad3dce7b.zip. Unzip the file. Then, follow the instructions in Sec. A.3.1.

Execution:

1. Follow the below:
2. python scripts/autofr_use_snapshots.py --site_url "https://www.cricbuzz.com/" --snapshot_dir [zip name]/[Snapshots directory]
3. A full example is provided at step 7: https://github.com/UCI-Networking-Group/AutoFR/tree/artifact-review#reuse-site-snapshots
4. The script uses identical hyper-parameters utilized in our paper. Simply pass in the site URL (from the CSV) and the snapshot directory. Make sure not to change any of the directory structures or names.
5. We also provide a script that will automatically check the reproducibility. See the instructions in https://github.com/UCI-Networking-Group/AutoFR/tree/artifact-review#reuse-site-snapshots confirm_reproducibility script part.

Results:

1. Follow the below:
2. The terminal will display the rules that are outputted.
3. Open up our "Top5K_rules.csv" (Sec. A.2.2) and look for the corresponding row that matches the zip file name. Then compare the filter rules generated *vs.* the row information. They should match.

A.5 Notes on Reusability

By leveraging the site snapshots we collected in Sec. A.2.2, users and researchers can explore other ways to generate filter rules. This includes:

1. Be less conservative when dealing with site dynamics. For any given site, there are dynamics upon different visits to it. For instance, other images, text, and ads can be served to the same user. We capture these dynamics in our site snapshots by collecting multiple snapshots per-site. Our algorithm randomly selects one site snapshot to test a rule at a given time t step of our algorithm and puts the rule to "sleep" (*i.e.,* remove it from contention) if it does not block any requests. Instead, one can modify the algorithm so that it selects only site snapshots that will cause the rule to block at least one request. We discuss this in Sec. 5.2.2.

2. Explore other RL algorithms. In our paper, we formulate the problem of filter rule generation as a multi-arm bandits problem. Future work can freely explore different RL algorithms offline using site snapshots.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.