

USENIX Association

**Proceedings of the
18th USENIX WOOT Conference
on Offensive Technologies (WOOT '24)**

**August 12–13, 2024
Philadelphia, PA, USA**

© 2024 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 978-1-939133-43-4

Conference Organizers

Program Co-Chairs

Adam Doupé, *Arizona State University*

Alyssa Milburn, *Intel*

Program Committee

Brandon Azad, *Apple Inc.*

Antonio Bianchi, *Purdue University*

Fraser Brown, *Carnegie Mellon University*

Juan Caballero, *IMDEA Software Institute*

Lorenzo Cavallaro, *University College London*

Sofia Celi, *Brave*

Jiska Classen, *Hasso Plattner Institute*

Jake Corina, *Independent*

Adrian Dabrowski, *CISPA Helmholtz Center for Information Security*

Audrey Dutcher, *Arizona State University*

Manuel Egele, *Boston University*

Aurélien Francillon, *EURECOM*

Fabian Freyer, *Apple Inc.*

Christophe Hauser, *Dartmouth College*

Xiali Hei, *University of Louisiana at Lafayette*

Yeongjin Jang, *Samsung Research America*

Alexandros Kapravelos, *North Carolina State University*

Vasileios Kemerlis, *Brown University*

Yongdae Kim, *Korea Advanced Institute of Science and Technology (KAIST)*

Daniel Klischies, *Ruhr University Bochum*

Pierre Laperdrix, *CNRS*

Aravind Machiry, *Purdue University*

Federico Maggi, *Amazon Web Services*

Dominik Maier, *Google*

Marius Muench, *University of Birmingham*

Colin O'Flynn, *Dalhousie University*

Fabio Pagani, *Binarly*

Mathias Payer, *EPFL*

Jam (Vie) Polintan, *Google*

Andreea-Ina Radu, *University of Birmingham*

Thomas Roth, *Hextree GmbH*

Jose Sanchez Vicarte, *Intel Corporation*

Martin Schwarzl, *Cloudflare*

Natalie Silvanovich, *Google*

Takeshi Sugawara, *The University of Electro-Communications*

Sam L. Thomas, *Binarly*

Dave (Jing) Tian, *Purdue University*

Stephen Tong, *Zellic*

Thomas Unterluggauer, *Intel Corporation*

Victor van der Veen, *Qualcomm*

Tom Van Goethem, *KU Leuven and Google*

Yingchen Wang, *The University of Texas at Austin*

Ned Williamson, *Google*

Xinyu Xing, *Northwestern University*

Yuval Yarom, *Ruhr University Bochum*

Stefano Zanero, *Politecnico di Milano*

Kyle Zeng, *Arizona State University*

Andrew D. Zonenberg, *IOActive*

Artifact Evaluation Committee Chair

Marius Muench, *University of Birmingham*

Artifact Evaluation Committee

Asmita, *University of California, Davis*

Tolga Atalay, *Virginia Tech*

Aurélien Hernandez, *EURECOM*

Adrian Herrera, *Interrupt Labs*

Jingmei Hu, *Amazon*

Doreen Joseph, *University of California, Davis*

Endong Liu, *University of Birmingham*

Jing Liu, *University of California, Irvine*

Zeyan Liu, *University of Kansas*

Sabrina Manickam, *Vellore Institute of Technology and Max Planck Institute for Security and Privacy*

Andrea Monzani, *University of Milan*

Paul Olivier, *LAAS-CNRS*

Samuel Pélissier, *INSA Lyon, Inria*

Davide Rusconi, *University of Milan*

Nathan Rutherford, *Royal Holloway, University of London*

Mahsa Saeidi, *Oregon State University*

Amit Samanta, *University of Utah*

Ryan Tsang, *University of California, Davis*

Billy Tsouvalas, *Stony Brook University*

Jayakrishna Menon Vadayath, *Arizona State University*

Nils Wiersma, *Netherlands Forensic Institute*

Lennert Wouters, *KU Leuven*

Yingao (Elaine) Yao, *University of British Columbia*

Matteo Zoia, *University of Milan*

Steering Committee

Aurélien Francillon, *EURECOM*

Yanick Fratantonio, *Google*

Casey Henderson-Ross, *USENIX Association*

Martina Lindorfer, *TU Wien*

Clémentine Maurice, *CNRS*

Collin Mulliner, *Cruise*

Colin O'Flynn, *NewAE Technology and Dalhousie University*

Mathias Payer, *EPFL*

Sara Rampazzi, *University of Florida*

Yuval Yarom, *Ruhr University Bochum*

Sarah Zennou, *Airbus*

External Reviewer

Mario D'Onghia

Message from the WOOT '24 Program Co-Chairs

We are delighted to introduce the proceedings of the 18th edition of the USENIX WOOT Conference on Offensive Technologies (WOOT '24).

What a journey it has been for WOOT: this year we've not only returned to USENIX (back to being co-located with USENIX Security) but WOOT has also become a full conference, complete with formal proceedings. This would not have been possible without the support of the amazing WOOT community.

To further WOOT's goal of bringing together academics and practitioners in the field of offensive security research, we introduced a new practitioner track aimed at encouraging more participation from non-academic researchers, and in particular advocating submission of existing work which would benefit from publication in a more formal and complete form.

The 18 accepted papers out of 51 (35% acceptance rate) are split between 3 short papers in this new practitioner track, and the remaining 15 full-length papers in the academic track. This technical program covers a range of different areas of offensive security, and we look forward to seeing them at the conference, along with the demos, posters, and lightning talks which make up the rest of the WOOT program.

Many members of the USENIX staff worked hard to guide our way and have organized much of the conference, and we deeply appreciated their help and wisdom. We'd also like to extend our gratitude to the many people who volunteered so much of their time and energy to help WOOT happen this year—this includes the Steering Committee, led by Aurélien Francillon; the many members of the Program Committee and Artifact Evaluation Committees; our AEC chair, Marius Muench; and our social media chair, Tom Van Goethem.

Finally, we want to express our appreciation for the generous support of WOOT's sponsors, without whom WOOT would not be possible.

Adam Doupé, *Arizona State University*

Alyssa Milburn, *Intel*

WOOT '24 Program Co-Chairs

18th USENIX WOOT Conference on Offensive Technologies (WOOT '24)

August 12–13, 2024
Philadelphia, PA, USA

Monday, August 12

Practitioners at Work

Achilles Heel in Secure Boot: Breaking RSA Authentication and Bitstream Recovery from Zynq-7000 SoC 1
Prasanna Ravi and Arpan Jati, *Temasek Laboratories, Nanyang Technological University, Singapore*; Shivam Bhasin, *National Integrated Centre for Evaluation (NiCE), Nanyang Technological University, Singapore*

WhatsApp with privacy? Privacy issues with IM E2EE in the Multi-device setting 11
Tal A. Be'ery, *Zengo*

Introduction to Procedural Debugging through Binary Libification 17
Jonathan Brossard, *Conservatoire National des Arts et Métiers, Paris*

Security Can Be Tricky

The Power of Words: Generating PowerShell Attacks from Natural Language 27
Pietro Liguori, Christian Marescalco, Roberto Natella, Vittorio Orbinato, and Luciano Pianese, *DIETI, Università degli Studi di Napoli Federico II*

Attacking with Something That Does Not Exist: 'Proof of Non-Existence' Can Exhaust DNS Resolver CPU 45
Olivia Gruza, Elias Heftrig, Oliver Jacobsen, Haya Schulmann, and Niklas Vogel, *National Research Center for Applied Cybersecurity ATHENE, Goethe-Universität Frankfurt*; Michael Waidner, *National Research Center for Applied Cybersecurity ATHENE, Technische Universität Darmstadt, Fraunhofer Institute for Secure Information Technology SIT*

Amplifying Threats: The Role of Multi-Sender Coordination in SMS-Timing-Based Location Inference Attacks . . 59
Evangelos Bitsikas, *Northeastern University*; Theodor Schnitzler, *Research Center Trustworthy Data Science and Security and Maastricht University*; Christina Pöpper, *New York University Abu Dhabi*; Aanjhan Ranganathan, *Northeastern University*

Embedded Security

MakeShift: Security Analysis of Shimano Di2 Wireless Gear Shifting in Bicycles 75
Maryam Motallebigohmi, *Northeastern University*; Earlence Fernandes, *UC San Diego*; Aanjhan Ranganathan, *Northeastern University*

Engineering a backdoored bitcoin wallet 89
Adam Scott and Sean Andersen, *Block, Inc.*

Oh No, My RAN! Breaking Into an O-RAN 5G Indoor Base Station 101
Leon Janzen, Lucas Becker, Colin Wiesenäcker, and Matthias Hollick, *Technical University of Darmstadt (TUDa)*

Tuesday, August 13

Hardware Security

RIPencapsulation: Defeating IP Encapsulation on TI MSP Devices 117
Prakhar Sah and Matthew Hicks, *Virginia Tech*

Reverse Engineering the Eufy Ecosystem: A Deep Dive into Security Vulnerabilities and Proprietary Protocols . . 133
Victor Goeman, Dairo de Ruck, Tom Cordemans, Jorn Lapon, and Vincent Naessens, *DistriNet-KU Leuven*

SoK: Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems 149
Xi Tan and Zheyuan Ma, *CactiLab, University at Buffalo*; Sandro Pinto, *Universidade do Minho*; Le Guan, *University of Georgia*; Ning Zhang, *Washington University in St. Louis*; Jun Xu, *The University of Utah*; Zhiqiang Lin, *Ohio State University*; Hongxin Hu, *University at Buffalo*; Ziming Zhao, *CactiLab, University at Buffalo*

Memory Corruption Is a Solved Problem

Not Quite Write: On the Effectiveness of Store-Only Bounds Checking171

Adriaan Jacobs and Stijn Volckaert, *DistriNet, KU Leuven*

SoK: On the Effectiveness of Control-Flow Integrity in Practice 189

Lucas Becker and Matthias Hollick, *Technical University of Darmstadt*; Jiska Classen, *Hasso Plattner Institute, University of Potsdam*

Exploiting Android’s Hardened Memory Allocator 211

Philipp Mao, Elias Valentin Boschung, Marcel Busch, and Mathias Payer, *EPFL*

Physical Attacks

Breaking Espressif’s ESP32 V3: Program Counter Control with Computed Values using Fault Injection 229

Jeroen Delvaux, *Technology Innovation Institute*; Cristofaro Mune, *Raelize*; Mario Romero, *Technology Innovation Institute*; Niek Timmers, *Raelize*

Basilisk: Remote Code Execution by Laser Excitation of P–N Junctions Without Insider Assistance 245

Joe Loughry, *Netoir.com*; Kasper Rasmussen, *University of Oxford*

SOK: 3D Printer Firmware Attacks on Fused Filament Fabrication 263

Muhammad Haris Rais, *Virginia State University*; Muhammad Ahsan and Irfan Ahmed, *Virginia Commonwealth University*

Achilles Heel in Secure Boot: Breaking RSA Authentication and Bitstream Recovery from Zynq-7000 SoC

Prasanna Ravi^a Arpan Jati^a Shivam Bhasin^{a,b}

^a*Temasek Labs, Nanyang Technological University, Singapore*

^b*National Integrated Centre for Evaluation (NiCE), Nanyang Technological University, Singapore*
Email: {prasanna.ravi, arpan.jati, sbhasin}@ntu.edu.sg

Abstract

Secure boot forms the backbone of trusted computing by ensuring that only authenticated software is executed on the designated platform. However, implementation of secure boot can have flaws leading to critical exploits. In this paper, we highlight a critical vulnerability in open source First Stage Boot Loader (FSBL) of AMD-Xilinx’s flagship Zynq-7000 System on Chip (SoC) solution for embedded devices. The discovered vulnerability acts as a ‘single point of failure’ allowing complete bypass of the underlying bypass RSA authentication during secure boot. As a result, a malicious actor can take complete control of the device and run unauthenticated/malicious applications. We demonstrate an exploit using the discovered vulnerability in form of first practical ‘Starbleed’ attacks on Zynq-7000 devices to recover the decrypted bitstream from an encrypted (using AES-256) boot image. The identified flaw has existed in the secure-boot software for more than 10 years. The vulnerability was responsibly disclosed to the vendor under CVE 2022/23822. The vendor thereafter patched the FSBL software and issued a design advisory. Our work therefore motivates the need towards rigorous security evaluation tools to test for such trivial security vulnerabilities in software.

1 Introduction

Due to the demand of System on Chips in sensitive applications, they support various security features such as secure boot, device authentication, bitstream encryption, readback protection, etc. However, the robustness of these security features remains unclear due to a lack of proper documentation and third-party evaluation/scrutiny. In this work, we perform an in-depth analysis of the *RSA authentication feature* of the Zynq-7000 SoC from AMD-Xilinx. AMD-Xilinx Zynq-7000 SoCs have been a market leader in the integrated FPGA and processor market, with wide adoption across several industries such as automotive, aerospace, industrial,

and healthcare sectors. We identified a critical *double fetch* security flaw in the RSA authentication feature within the First Stage Boot Loader (FSBL) provided by Xilinx. Its exploitation makes it possible to execute an unauthenticated software application on the Zynq-7000 SoC. The identified flaw is only present in the FSBL software and thus can be easily fixed through appropriate modification of the FSBL software.

Thus, the first contribution of our work is the *identification of a critical security flaw in the FSBL software to bypass RSA authentication*.

Upon bypassing RSA authentication, we utilize the unauthenticated software application to demonstrate a novel attack to recover the encrypted bitstream in the boot image, thereby subverting the bitstream encryption feature. To the best of our knowledge, there does not exist any prior work that has reported a bitstream recovery attack on the Zynq-7000 SoC. In this context, Ender et al. [3] in 2020 proposed the *Starbleed* attack, capable of breaking bitstream decryption on standalone Virtex-6 and 7-series Xilinx FPGAs. The design advisory from Xilinx as a response to the Starbleed attack claims that the Zynq-7000 SoC is resistant “due to the use of asymmetric and/or symmetric authentication in the boot/configuration process” [4]. Due to the security flaw found in the FSBL, we managed to identify a novel approach to mount the Starbleed attack on the Zynq-7000 device for full bitstream recovery.

Thus, as a second contribution of our work, *we present the first practical demonstration of the Starbleed attack on the Zynq-7000 SoC with practical validation on PYNQ-Z1 platform*

We have thus performed an end-to-end recovery of the bitstream exploiting the RSA bypass vulnerability and the Starbleed attack. We communicated our findings to Xilinx in a vulnerability disclosure on March 8, 2022.

Xilinx quickly confirmed the vulnerability on March 24, 2022, and also published a patch for the FSBL software on March 25, 2022 [6]. Information about the vulnerability was also published as a design advisory by Xilinx on April 28, 2022 [5]. Furthermore, we also investigated if the flaw in the FSBL software is also present in the BootROM code of the Zynq-7000 SoC. Analyzing the BootROM behavior presents significant challenges, since the BootROM code is unavailable or cannot be modified, as it is hard-coded within the SoC.

Thus, as a third contribution of our work, *we present a novel black-box analysis of the communication interface between the Zynq-7000 SoC and the NVM during BootROM execution.*

However, our analysis was able to positively confirm that the BootROM software does not suffer from the RSA vulnerability present in the FSBL.

Availability of Software

All the software used for this work is available in the public domain at the following link: https://github.com/PRASANNA-RAVI/RSA_Bypass_Vulnerability_Zynq_7000_SoC.

1.1 Threat Model

The boot image of the victim Zynq-7000 SoC device boots from a boot image stored in a Non-Volatile Memory (NVM) accessible to an attacker. The SoC typically consists of two components: (1) Programmable System (PS) which refers to the dual-core ARM Cortex-A9 processor and (2) Programmable Logic (PL) which refers to the FPGA fabric. The victim boot image has three partitions - FBSL, PL partition (bitstream to execute on the FPGA), and PS partition (software application to execute on the processor). The target device mandates RSA authentication of the boot image (i.e.) the RSA eFUSE is enabled, and all partitions in the victim boot image are encrypted as well as authenticated. This means that every partition has its corresponding RSA signature stored along with it, and is referred to as the Authentication Certificate (AC). Refer to Figure 1 for the structure of the victim boot image we consider for our attack. The attacker’s goal is to execute an unauthenticated application on the Zynq-7000 SoC.

2 Background

We now provide a brief background on the secure boot feature of the Zynq-7000 SoC, to facilitate the understanding of our attack, described later in Sections 3-6.

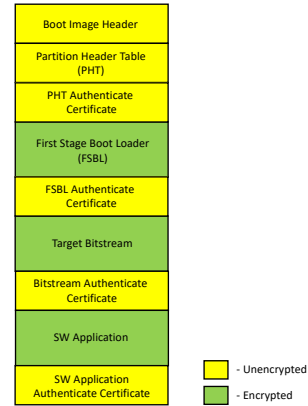


Figure 1: Authenticated victim boot image

2.1 Secure Boot of Zynq-7000 device

The central component of secure boot is the *secure boot image* which consists of various partitions that will be sequentially loaded securely into the appropriate locations within the Zynq-7000 SoC (either DDR, On-Chip Memory (OCM) or FPGA). The important components of a boot image are as follows:

- **Boot Image Header (BIH) and BootROM code:** The BootROM code is the first piece of software executed upon resetting the Zynq-7000 SoC. It is hard-coded onto the BootROM of the chip (and not part of the boot image), and cannot be modified. It initializes the device based on information in the BIH. Its main function is to retrieve the FSBL from the NVM, after which it authenticates the FSBL using its Authentication Certificate (AC that contains its RSA signature), and further decrypts the FSBL, before securely passing control to it.
- **Partition Header Table (PHT):** The PHT is a critical component of the boot image, which contains metadata information about each PL and PS partition in the boot image. Each partition has an associated entry of 64 bytes in the PHT and contains information such as encrypted partition size, decrypted partition size, total partition size including its AC, destination address in the device, location within the boot image, authentication status, etc. The PHT is used by the FSBL to get information about each partition in the boot image. *We remark that the PHT is present unencrypted within the boot image allowing an attacker to gain information about the metadata of each partition in the secure boot image.*
- **First Stage Boot Loader (FSBL):** The FSBL is responsible for loading each of the PS and PL partitions in the boot image into the appropriate

locations within the device (i.e.) PL partition is loaded into the FPGA, and PS partition is loaded into the DDR memory. The FSBL first retrieves the PHT from the NVM and authenticates it using its AC. Upon successful authentication, then FSBL securely loads the PL and PS partitions individually in the same manner, from the boot image, based on information in the PHT. However, if PHT authentication fails, then the FSBL simply aborts the secure boot procedure. After loading all the PS and PL partitions, the FSBL transfers control to the last software application that was loaded from the boot image. In this work, we use the official FSBL code for the Zynq-7000 SoC provided by Xilinx (FSBL version 2018.1).

- **Programmable Logic (PL) or Programmable System (PS) Partition:** After the FSBL, the remaining portion of the boot image is occupied either by a PL partition or a PS partition. For an authenticated partition, there is an Authentication Certificate (AC), that contains its RSA signature, which is appended to it in the boot image.

2.2 RSA Authentication in Zynq-7000 SoC

The Zynq-7000 SoC uses the well-known RSA-2048-based signature scheme for authentication. It is done with two keys: the Primary Key and the Secondary Key. While the primary key is stored in the eFuse of the device (fixed for a given device), the secondary key is associated with each partition. The primary key is used to authenticate the secondary key of a partition, and the secondary key is used to authenticate the partition data itself, thereby forming an authentication chain. The authentication operation (i.e.) signature verification is carried out by a cryptographic software library, part of the BootROM and FSBL of the Zynq-7000 SoC. Since an understanding of the intricate details of RSA authentication is not required for our attack, we refer the reader to [10] for more details.

RSA authentication is an integral component of the FSBL. FSBL is an open-source and modifiable piece of software. We analyze the FSBL source code to understand how it authenticates various components of the boot image.

3 Analyzing the RSA Authentication Procedure within FSBL

We noticed that the PHT authentication serves as a single point of failure in the secure boot procedure. If an attacker can bypass PHT authentication, he/she can mount

a tampered PHT that can be used to execute an unauthenticated application. We analyzed the PHT authentication procedure by FSBL (implemented within the `image_mover.c` source file in the `embeddedsdsw` project [9]). Refer to Figure 2 for a pictorial illustration of the authentication procedure of the PHT by the FSBL.

1. The FSBL first retrieves the PHT data from the NVM and stores it into a global variable denoted as `GVAR`. We denote the fetched PHT data from NVM as `PHT1`.
2. The FSBL then checks the status of the RSA eFUSE. If enabled, the FSBL again retrieves the PHT along with its AC. We denote the fetched PHT data as `PHT2` since it is retrieved at a different time than `PHT1`.
3. If verification of AC of `PHT2` is successful, then the FSBL uses the data in `GVAR` (`PHT1`) as the PHT to load the PS and PL partitions in the boot image.

In other words, the FSBL authenticates `PHT2` but uses the unauthenticated `PHT1` for secure boot. This is mainly because of the *double fetch* of the PHT data from the NVM which is external to the security boundary of the device. This is the critical vulnerability that we have identified that could be exploited to bypass PHT authentication.

We remark that our experiments were done on FSBL version 2018.1, they also applied to the latest FSBL version dated 23 Apr 2020¹, during the time of our research.

3.1 Related Works

Double fetch is a term referring to a bug that occurs when a process reads and uses the same value twice, expecting it to be identical while it is possible for an attacker to modify it between the two reads. This term was first coined by Serna [7], and there have been several works that have exploited double-fetch bugs in what is commonly referred to as Time-of-Check to Time-of-Use (TOCTTOU) attacks [1]. Well-known instances of such attacks include attacks on the Linux kernel [8], applications such as Firefox [11] and Intel BootGuard [2].

3.2 Exploiting the RSA Security Flaw in FSBL

We formulate an attack methodology to exploit the double fetch PHT, using an NVM emulator, which is configured to behave in the following manner during PHT authentication.

¹https://github.com/Xilinx/embeddedsdsw/blob/master/lib/sw_apps/zynq_fsbl/src/image_mover.c

1. When the FSBL fetches the PHT for the first time (PHT1), the NVM emulator provides a tampered PHT, configured according to the attacker's requirements. Thus, the tampered PHT is stored in GVAR variable, within the DDR.
2. The FSBL then checks the status of RSA eFUSE and if enabled, again fetches the PHT (PHT2) along with its AC. This time, the NVM emulator provides the valid PHT along with its AC.
3. The FSBL successfully authenticates PHT2, but now uses the tampered PHT1 for secure boot present in GVAR. Based on the tampered PHT1, the FSBL loads an unauthenticated application on the target device, thereby bypassing RSA authentication.

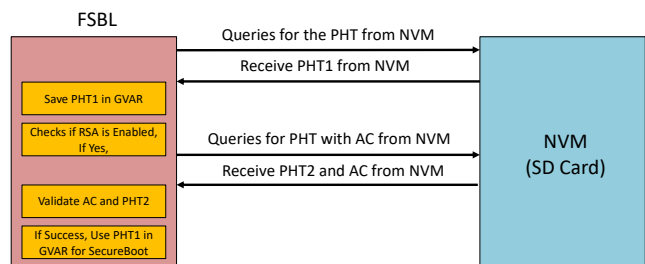


Figure 2: Authentication of the PHT by the FSBL

3.3 Proof of Concept (PoC) Attack Implementation

We started with a Proof of Concept (PoC) attack to demonstrate the presence of the double fetch vulnerability during PHT authentication. This was done not with an NVM emulator, but by performing manual modifications to the FSBL, to replicate the behavior of the emulator. We manually modified the PHT data in the GVAR variable after fetching PHT1, and the data is tampered with to load an unauthenticated PS partition (software application) from the boot image. This is the only modification done in the FSBL and does not aid the attack in any other manner. Within the boot image, the authenticated application is replaced with a malicious and unauthenticated application in the boot image. We ran repeated experiments using the tampered FSBL as well as the tampered boot image, and we were able to successfully load and execute the unauthenticated software application on the target device, which demonstrates the presence of the RSA bypass vulnerability.

However, this does not qualify as a real attack, since we made manual modifications to the FSBL that is encrypted within the boot image. Since the attacker does

not know the encryption key, it is not possible in a real-attack scenario. In the following, we thus attempt to perform a practical real-world attack by building a low-cost NVM emulator, that does not require making modifications to the FSBL in the boot image.

4 Practical Attack using SD Card Switcher Board

One approach to carry out a practical attack would be to implement the NVM emulator on an FPGA/ASIC. However, designing it requires significant engineering effort, and hence adopted a simpler approach. The basic requirement for our NVM emulator is to present a tampered PHT during the first fetch, and a valid PHT during the second fetch. To achieve this, we built an SD card switcher that can switch between two SD cards (SD Card 1 and SD Card 2) during the secure boot procedure.

The SD card switcher has two SD card slots, and we can choose the SD card to connect to the target, based on the logic level of a GPIO pin. The board also facilitates keeping the SD cards powered on from an external power source. This ensures that the SD card once initialized by the target device is powered on, even if the target device is powered off. In the following, we explain the proposed attack methodology using our SD card switcher board.

4.1 Attack Methodology

The two SD cards (SD cards 1, and 2) are loaded with two different attack boot images derived from the victim boot images. SD card 1 contains a boot image with the tampered PHT (i.e.) PHT1 (mapping to an unauthenticated attack application), while all the other contents match that of the victim SD card. SD card 2 contains a boot image with a valid PHT (PHT2) but with the authenticated software application replaced with the unauthenticated attack application. Refer to Figure 3 for a pictorial illustration of the boot images on both SD cards. We load both the SD cards onto the SD card switcher board and connect the SD card switcher to the Zynq-7000 SoC. The attack is carried out in the following manner:

1. The Zynq-7000 SoC first boots with SD card 2 mounted on the SD card switcher board. This is done to initialize SD card 2.
2. We now power off the Zynq-7000 SoC and switch to SD card 1. This is done while maintaining the power of both SD cards.
3. We now boot the Zynq-7000 SoC with SD card 1, which ensures that the tampered PHT (PHT1) during the first PHT fetch. After the first PHT fetch,

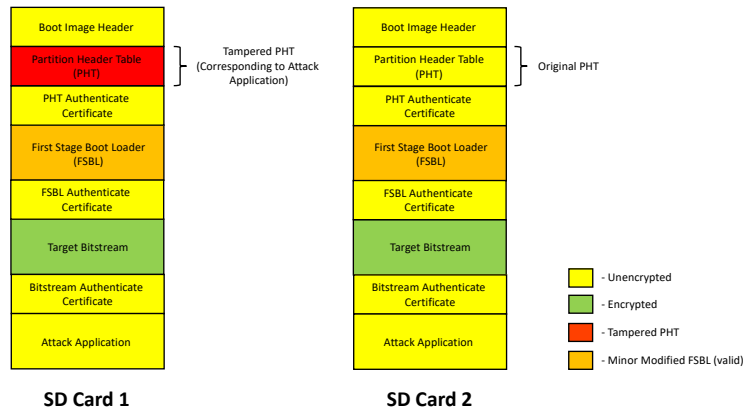


Figure 3: Boot Images of SD card 1 and SD card 2 within the SD switcher board

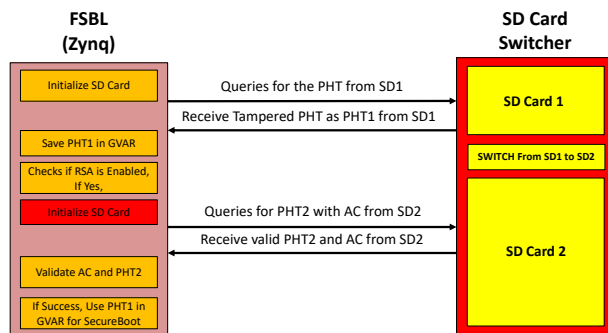


Figure 4: Improved Attack on the Zynq-7000 SoC using SD card switcher, with the modification done to the FSBL denoted in red.

we switch from SD card 1 to SD card 2. For our experiments, we added a manual delay between the first and second PHT fetches. However, this can be automated as the timing of the switch is constant for the Zynq device upon power up.

4. After the switch, we expect that the Zynq-7000 SoC will retrieve the valid PHT from SD card 2 (which was already initialized), which should be authenticated successfully by the FSBL. This should also ensure that PHT1 is used for secure boot, and will therefore execute the unauthenticated on the Zynq-7000 SoC.

4.1.1 Experimental Observations of Attack using SD Card Switcher Board

Our experiments reveal that the Zynq-7000 SoC halts operation after switching from SD card 1 to SD card 2, after the first PHT fetch. The FSBL is unable to connect to SD card 2, even though it is initialized. We hypothesize that the SD card peripheral on the target

device, which is oblivious to the switch tries to communicate with commands for SD card 1, while the switcher connects the device to SD card 2. To overcome this, we perform a minor modification to the FSBL, by adding to the InitSD function, to initialize SD card 2 after the switch. After this modification, we can successfully perform a bypass of the PHT authentication and load the unauthenticated application, demonstrating a successful RSA bypass. Refer to Figure 4 for a pictorial illustration of our improved attack using the SD card switcher.

Since our current setup still requires a modification to the FSBL, it does not qualify as a practical attack. We believe this limitation can be overcome using specialized hardware (FPGA/ASIC) to tamper the SD card interface at precise time instances. Nevertheless, our attack concretely exposes the flaw in the FSBL software to bypass RSA authentication.

4.2 Fixing the Flaw in PHT Authentication within FSBL

The vulnerability mainly arises from the retrieval of the same PHT data twice from the NVM and only using the data from the first fetch. This flaw can be patched by ensuring that PHT is only retrieved once from the NVM and authenticated immediately. This fix is implemented as part of the patched FSBL (dated March 25th, 2022) [6], and our manual analysis of the patched FSBL source code confirmed the removal of the double fetch vulnerability. AMD-Xilinx referred to our attack as a physical attack [6] and that the device "was not designed to be resistant to physical attacks". However, the identified vulnerability still exposes a critical flaw in the RSA authentication process, which enables a practical attack that enables it to completely bypass it. While we verified the BootROM of Zynq-7000 SoC for the same vulnerability, we have

not analysed other devices from AMD-Xilinx, and we leave this analysis for future work. This is not the first time that such double fetches have been detected in secure software [1]. In the following section, we show that an attacker can use the unauthenticated application to perform a novel bitstream recovery attack.

5 Starbleed for Bitstream Recovery

Ender et al. [3] in 2020 exposed a critical security flaw in the bitstream decryption protocol of standalone Virtex-6 and 7-series Xilinx FPGAs, which enables recovery of bitstream data, now well-known as the *Starbleed* attack. The only requirement is that the attacker requires access to the configuration interface of the FPGA (PL). In this work, we adapt the Starbleed attack to the Zynq-7000 device for bitstream recovery.

5.1 Attack Methodology

The attacker makes malicious changes to the encrypted bitstream, such that upon decryption, a targeted decrypted bitstream word is written into the Warm Boot Status Address (WBSTAR) register of the configuration interface. The WBSTAR register retains its value even upon FPGA reset and thus an adversary can access the decrypted word from the WBSTAR register. Similarly, full bitstream recovery can be performed one word at a time. Since the attacker now has control of the PS (through the attack application), we designed an attack application to carry out the attack by accessing the PL through the PCAP (Processor Configuration Access Port) interface. The application programs the PL with the tampered bitstream, but it results in failure of HMAC integrity check, triggering a HMAC error. The reference manual claims that readback of any register (including WBSTAR) is not possible unless the PL is configured with a valid bitstream. Thus, it was evident that the Starbleed attack could not be performed as on the Zynq-7000 SoC.

5.1.1 Attack Execution using Workaround

We identified a workaround to ensure that register readback is possible, even after programming the PL with a tampered bitstream.

1. Program PL with a valid encrypted bitstream.
2. Without initializing the PL again, we push the tampered attack bitstream through the PCAP interface. We observe that FPGA stays programmed (DONE signal is high) even though the tampered bitstream trigger an HMAC error.

3. We then issue read command to successfully read the WBSTAR register containing the decrypted bitstream word.

This technique of programming bitstreams without initializing is not recommended practice. We typically expect that PL is not configured properly without initialization. We observe that the tampered bitstreams were able to write the decrypted bitstream word to the WBSTAR register while ensuring that readback is also possible. However, after readback, the PCAP interface becomes unresponsive, and only a Power-on Reset (PoR) of the device could bring it back to normal working condition. Thus, we can only recover one bitstream word per secure boot, and the attacker needs to power cycle the device to recover every bitstream word. We can recover the bitstream at a speed of 32 bits per second, and an estimated recovery time of 46 days for our experimental bitstream of size 3.85 MB. We observe that the secure boot time after every POR reset serves as a bottleneck for our attack. While reducing the attack time is possible, we consider performance acceleration out of the scope of our work. This attack would not be possible without bypassing RSA authentication, and thus using the patched version of the FSBL (dated March 25, 2022) [6] can serve as a strong mitigation against the bitstream recovery attack.

6 Conclusion

In this work, we have identified a critical double fetch security flaw in the FSBL software of AMD-Xilinx's Zynq-7000 SoC, which enables bypassing the RSA authentication procedure, to execute an unauthenticated application on the target device. We experimentally validated a potential exploit using a custom-built SD card switcher board. We also analyzed the BootROM code for a similar vulnerability, and confirm that the same bug is not present (Refer to Appendix A). We then proceeded to demonstrate the first successful bitstream recovery attack on the Zynq-7000 SoC using the Starbleed attack technique. In essence, our work uncovers a simple double fetch vulnerability in the secure boot software of Zynq-7000 SoC, but such vulnerabilities are not new. Our work demonstrates a serious need for automated tools for identifying such trivial bugs. While there have been proposals for such techniques [8], the applicability of these tools to embedded devices is to be studied and forms an interesting direction for future research.

Acknowledgement

This work was supported in part by the "National Integrated Centre of Evaluation" (NICE), and in part by a facility of the Cyber Security Agency (CSA), Singapore

References

- [1] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic kernel {TOCTTOU} protection. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 107–124, 2022.
- [2] Peter Bosch and Trammell Hudson. Warpattack: bypassing cfi through compiler-introduced double-fetches. In *Hack in the Box Amsterdam*, 2019.
- [3] Maik Ender, Amir Moradi, and Christof Paar. The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series fpgas. In *29th USENIX Security Symposium*, 2020.
- [4] Xilinx Inc. Ar 73541-design advisory for 7 series/virtex-6 fpgas: Defeating bitstream encryption dated 04/27/2020.
- [5] Xilinx Inc. Design advisory for zynq-7000: Fsbl authentication attack dated 04/28/2022. https://support.xilinx.com/s/article/76974?language=en_US.
- [6] Xilinx Inc. Software patch of the fsbl software for zynq-7000 soc against fsbl authentication attack. <https://github.com/Xilinx/embeddedsw/commit/28111bd377ec77e8cbb5492e5a0a4f4d37b6c5e3>.
- [7] Fermin J Serna. Ms08-061: The case of the kernel mode double-fetch, 2008.
- [8] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How {Double-Fetch} situations turn into {Double-Fetch} vulnerabilities: A study of double fetches in the linux kernel. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1–16, 2017.
- [9] Xilinx. Fsbl software for zynq-7000 soc (image_mover.c). https://github.com/Xilinx/embeddedsw/blob/master/lib/sw_apps/zynq_fsbl/src/image_mover.c.
- [10] Xilinx. Zynq-7000 all programmable soc technical reference manual-ug585 (vl. 13).
- [11] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. Warpattack: bypassing cfi through compiler-introduced double-fetches. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1271–1288. IEEE, 2023.

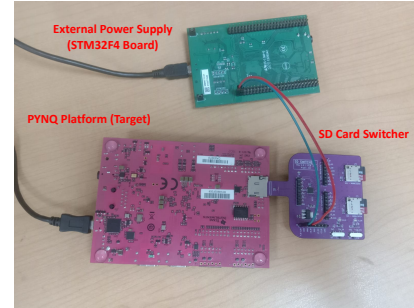


Figure 5: Attack Setup with SD Card Switcher

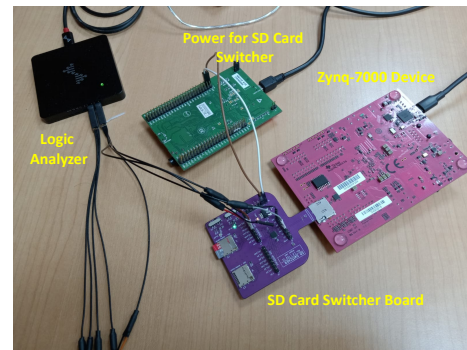


Figure 6: Experimental Setup: Logic Analyzer probing the SD Card interface to the Zynq-7000 device

A Assessing the RSA Authentication Procedure in BootROM

While we identified a flaw in the RSA authentication procedure in FSBL, we asked ourselves *whether the same flaw is also present in other operations during the secure boot*. We thus conducted a security analysis of the BootROM software, which also performs authentication of the FSBL itself, before FSBL starts execution. But, analysing the BootROM software is particularly challenging compared to the FSBL, since neither the BootROM source-code nor the binary is available. It is also hard-coded within the on-chip memory of the Zynq device, and hence cannot be modified. Thus, a code-analysis similar to that of FSBL is not possible. However, we observe that the BootROM loads data from the Non-Volatile Memory (NVM) (i.e.) SD card and thus monitoring the SD card interface during BootROM execution could provide critical information about its operation. In this respect, we utilize a logic analyzer to monitor the SD card communication during BootROM execution.

A.1 BootROM Analysis using SD Card Communication

In order to understand the data transfer between the SD Card (NVM) and the Zynq-7000 SoC during BootROM

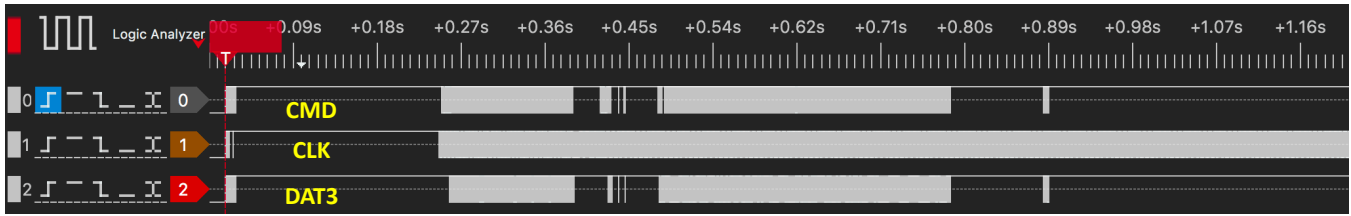


Figure 7: Data Transfer over SD interface covering the full boot-up of the Zynq device

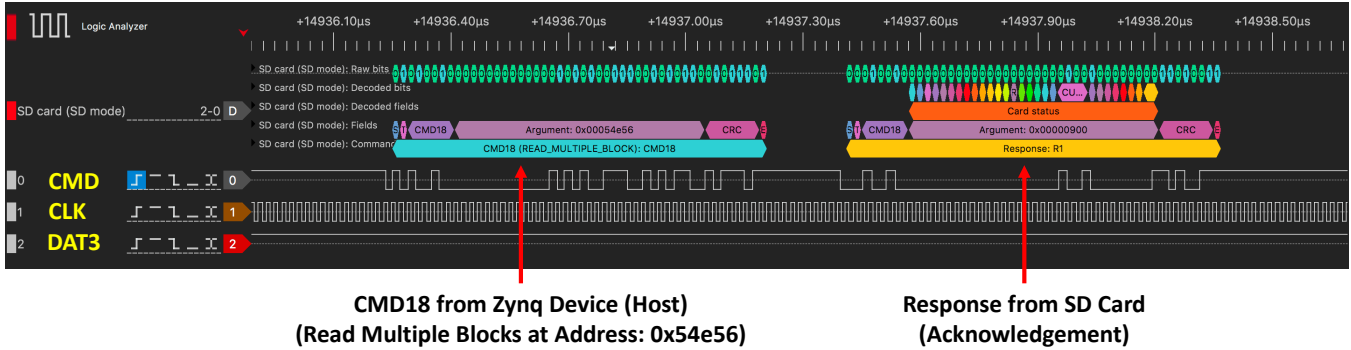


Figure 8: Visualization of packets transferred over the CMD line of the SD card interface

execution, we utilized a logic analyzer to analyze the communication between the SD card and the Zynq device, during FSBL execution. The reading/writing of data from/to the SD card occurs in blocks of 512 bytes, in a serial fashion, and in particular we monitored the commands CMD17 and CMD18, which can be used to read a single block and multiple blocks respectively.

We utilized the DSLogic Plus logic analyzer from DreamSourceLab to probe the SD card communication interface. Refer to Fig.6 for the picture of our experimental setup. We used to logic analyzer to probe the CMD, CLK and DAT3 lines (as a representative data line among DAT0-DAT3), and the captured signals can be viewed on the DSView software IDE. Please refer to Fig.7 for the data transfer over the SD interface during the entire boot-up phase of an authenticated and encrypted boot image. This captures the entire execution time of BootROM and FSBL. Channel 0 corresponds the CMD line, channel 1 corresponds to the CLK line and channel 3 corresponds to the DAT3 line. We only chose DAT3 as a reference for a data line, but any of the other data lines among DAT0-DAT2 can also be probed. Moreover, zooming into the data transfer allows us to visualize the packets within the DSView IDE, and refer to Fig.8 for the visualization of a command to read multiple blocks from the host (i.e.) CMD18 and the subsequent acknowledgement from the SD card.

A.2 Experimental Observations of BootROM Execution

We recall that the buggy software implementation of FSBL performs two PHT transfers (PHT1 and PHT2), when RSA authentication is enabled. While PHT1 only corresponds to retrieval of the PHT table, transfer of PHT2 corresponds to retrieval of the PHT along with the AC. If we identify such a redundant PHT transfer during BootROM execution, we can confirm that the vulnerability in the FSBL also exists in the BootROM. For our analysis, we considered three different types of boot images: (1) Non-secure (NSec), (2) Secure with only encryption (Sec_Encrypt) and (3) Secure with both encryption and authentication (Sec_Auth_Encrypt).

1. **Non-secure Image (NSec):** In this case, the BootROM is expected to only fetch the unencrypted FSBL. The size of the unencrypted FSBL in our boot image is ≈ 114.5 KB, which is equivalent to 225 blocks. Refer to Fig.9 for the data transfer corresponding to the retrieval of FSBL by the BootROM. We observe a total of 225 blocks being read from the SD card, using single block read commands (CMD17).
2. **Secure with only encryption (Sec_Encrypt):** When only encryption is enabled, the BootROM is expected to fetch the encrypted FSBL, whose

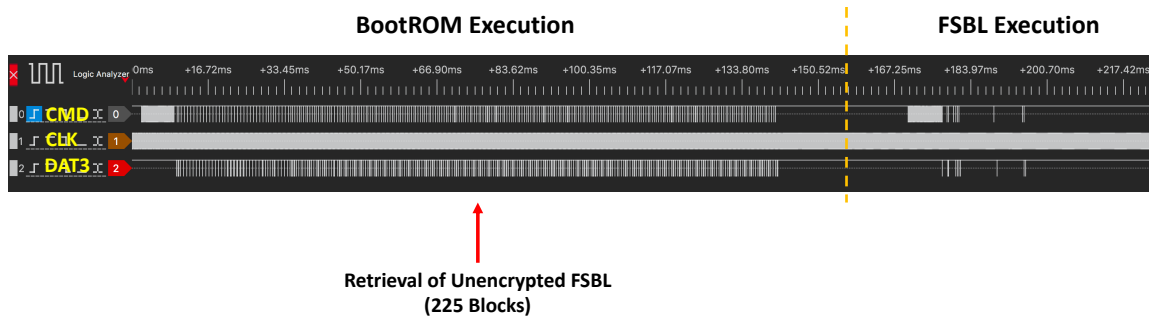


Figure 9: Retrieval of FSBL by BootROM in NSec image

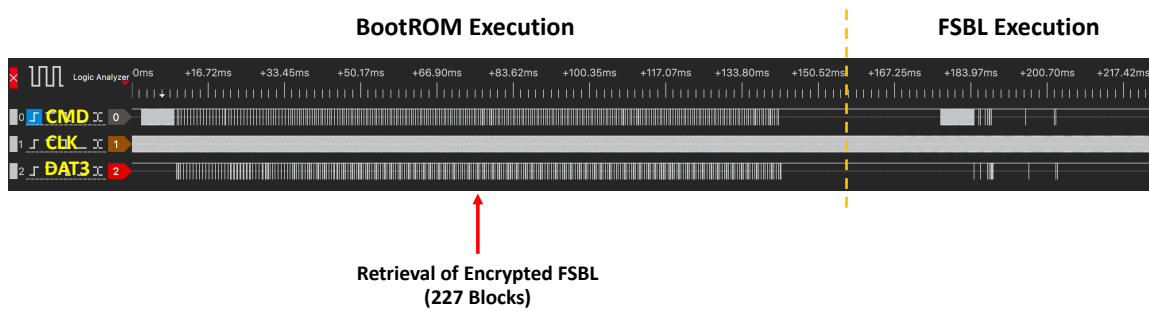


Figure 10: Retrieval of FSBL by BootROM in Sec_Encrypt image

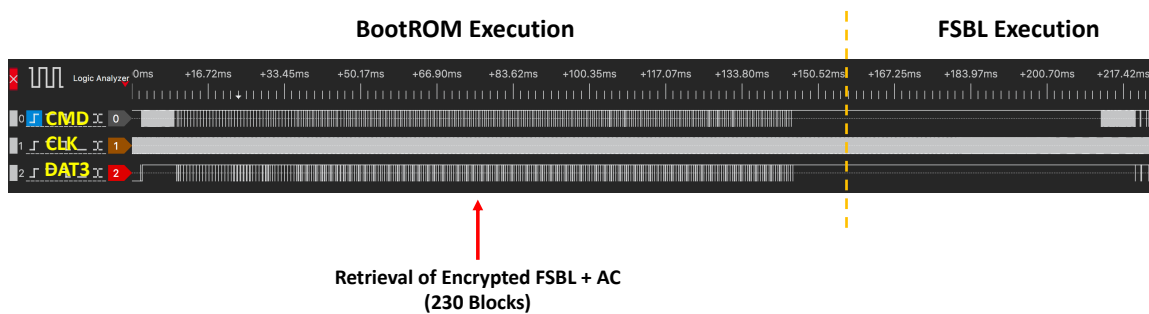


Figure 11: Retrieval of FSBL by BootROM in Sec_Auth_Encrypt image

size is roughly $115.5KB$ which is equivalent to 227 blocks. Refer to Fig.10 for the data transfer corresponding to the retrieval of the encrypted FSBL by the BootROM. We observe a total of 227 blocks being read from the SD card, using single block read commands (CMD17).

3. **Secure with both encryption and authentication (Sec_Auth_Encrypt):** When both authentication and encryption are enabled, the BootROM is expected to fetch the encrypted FSBL along with its Authentication Certificate (AC), whose size is roughly $116.8KB$ which is equivalent to 230 blocks.

Refer to Fig.11 for the data transfer corresponding to the retrieval of the encrypted FSBL along with its AC by the BootROM. We observe a total of 230 blocks being read from the SD card, using single block read commands (CMD17).

If there were a duplicate transfer of the FSBL, similar to that of the PHT, then we should have observed roughly 455 blocks being transferred. However, the number of blocks transferred from the SD card tallies with the expected number of blocks to be read. From this observation, we can positively confirm that the flaw identified in the FSBL is not present in the BootROM software.

However, this does not rule out the possibility of other vulnerabilities within the BootROM, that could be exploited for RSA authentication bypass.

WhatsApp with privacy? Privacy issues with IM E2EE in the Multi-device setting

Tal A. Be'ery, Zengo

Abstract

We recently discovered a privacy issue with Meta's WhatsApp, the world's most popular Instant Messaging (IM) application. Meta's WhatsApp suffers from a privacy issue that leaks the victims' device setup information (mobile device + up to 4 linked devices) to any user, even if blocked and not in contacts. Monitoring this information over time allows potential attackers to gather actionable intelligence about victims and their device changes (device replaced/ added /removed). Additionally, message recipients can associate the message with the specific sender device that sent it. The root cause for these issues stems from Signal's multi device protocol architecture, the Sesame protocol, and as a result these issues are not limited to Meta's WhatsApp only but probably relevant to most IM solutions, including the privacy-oriented Signal Messenger.

1. Introduction

End-to-End Encryption (E2EE) is a type of messaging that keeps messages private from everyone, including the messaging service. When E2EE is used, a message only appears in decrypted form for the person sending the message and the person receiving the message. The sender is one "end" of the conversation, and the recipient is the other "end"; hence the name "end-to-end".

Originally, most Instant Messaging (IM) apps did not support E2EE. However, as the importance and criticality of IM security had raised, E2EE became the security standard for modern communication and supported by modern IM apps.

Another aspect of IM communications that evolved over time is its multi-device support. Traditionally, Instant Messaging (IM) apps were bounded to a single device. However, as IM have gained popularity and became an important and even critical medium for communications, users wanted to have access to their IM conversations from every computing device they own. As a result, modern IM providers support the multi-device setting.

While each of these individual features (E2EE and multi-device) is critical for modern IM apps, supporting both simultaneously can lead to some security and privacy tradeoffs, as current E2EE solution expose some public cryptographic information about each of the devices, by thus compromising their users' privacy.

Contributions: Our main contributions are the following:

- We show the privacy and integrity implications of current popular multi-device solutions in IM apps.

- We demonstrate how attackers can easily subvert the WhatsApp client to obtain the victims' multi-device setup information.
- We suggest some practical measures to limit the exposure of such privacy leaks.

Overview: This paper is organized as follows: Section 2 provides a brick-and-mortar analogy to IM E2EE, Section 3 presents the Signal protocol and highlights the privacy issues in the multi-device setting, Section 4 shows how such privacy leaking attacks can be easily mounted by attackers against WhatsApp currently the world's most popular IM service, section 5 considers possible solutions. We conclude in Section 6.

2. Background

To better understand E2EE and its threat model we can use the postal service analogy:

Prior to E2EE, senders sent their letter in an envelope, but the envelope was not sealed. As part of its service, the post office opens the envelope and then puts it in another envelope and delivers it to the intended recipient.

This scheme has many advantages:

- Thanks to envelopes, eavesdroppers cannot see the contents of the letters.
- Thanks to the post office buffering, users do not need to meet to converse, but rather do so indirectly. This not only allows asynchronous conversations but also can protect user anonymity. Receivers can disclose only their nicknames to senders, and have the post service resolve from nicknames to true names and addresses. In fact, there is a privacy tradeoff between service and the conversation counterparty: If the conversing parties are directly connected, then the service is not exposed to the contents of the conversation, however the parties may uncover more metadata about each other and be able to break the "rules" of the protocol as the service is not there to enforce them. Generally speaking, it makes sense to assume that the service provider is more trustworthy than some counterparties that might be malicious.
- The post office can scan the contents of envelopes to make sure they do not contain bad content: Bombs, terror group messaging or pedophile photos.
- If letters are intended for multiple addresses (groups or a user with multiple houses) the post office can simply copy the message and send it to all addresses.

However, this scheme has a major drawback: Postal service employees are exposed to the contents of the letters and can leak them. The practical reasons for such leakage can vary: The service may act in negligence and mishandle user data, sell the data to advertisers for financial gain, be hacked by attackers, fail to restrict rogue employee access to private customer data, or even be served with a subpoena by the government.

To address this issue, E2EE was introduced. With E2EE, users send their message in locked boxes within the envelopes. Users provide their locks to the service when they join, but keep the keys themselves. When senders want to send a letter to a recipient they get the relevant padlock from the service and send their letter in a locked box within the envelope. As before, the post office opens the envelope and then puts it in another envelope and delivers it to the intended recipient. However, due to the locked box, the postal service personnel can no longer see the contents of the letter.

While E2EE indeed protects message content from the prying eyes of the service operator, it should be noted that:

- Even with E2EE, users must place some trust in the service provider, as the storing and forwarding messages, even encrypted, exposes metadata. Whether it's conversation related (counterparties, number of messages, length of messages, timing) or operational (online status, devices used, IP addresses which may have geo-location information).
- The newly added E2EE lock creates a new identifier for the user. When users lose their key, they must issue a new lock for the service. Aware attackers might leverage this information to deduce something changed on the user side.
- To make sure the E2EE lock is indeed of the intended user and not maliciously replaced by the service or a "Monster-in-the-Middle" (MITM) attacker, the sender must verify the lock's genuinity with the receiver using another independent channel. This requirement not only hinders the user experience but also jeopardizes the privacy of the users as they need to connect via additional service with additional identifiers.

But even with E2EE, users were still concerned: What happens if attackers break into their homes? Surely the system cannot prevent attackers from unlocking boxes and reading letters while they are still there and can use the keys, but we want to make sure that this privacy breach is limited to the exact period of the breach. Namely:

- Perfect Forward Secrecy (PFS): Attackers cannot open locked boxes that were locked before they broke into their victims' homes.

- Post Compromise Security (PCS): Attackers cannot open locked boxes that were locked after they left their victims' homes.

To achieve these properties, keys must be updated for every message, such that in case of compromise, the compromised keys are only useful for that message only. To do so, the two parties within the conversation are sending information to update the next locks and keys within their conversation.

It should be noted that while the first scenario of the pre-E2EE postal service privacy leak might be relevant at a large scale, for example to read the conversation of many users for serving ads or for mass surveillance, the case of post E2EE breaking into victims' homes does not scale well and mostly relevant to a small portion of the population consisting of highly targeted individuals. Since the contents of the messages themselves cannot be protected during the time of the attackers breaking in, the scenario for which PCS and PFS are relevant is only when attackers break into the victims' homes along with compromise of the service to get some of the victims' locked message boxes. Having such two successful independent attacks is a much less likely scenario than each of these attacks on their own.

3. The Signal protocol: From postal service analogy to real world crypto

3.1. The basic Signal protocol

WhatsApp is using the Signal protocol to implement E2EE's "postal service locked boxes" with public key cryptography. Users create their private and public key pair on their device when they join the IM service, and provide their public keys (possibly along with additional auxiliary data) to the IM service, which maintains the directory of the user's public keys. When parties wish to converse, the IM server provides them with their counterparty's public keys. It should be noted, as discussed above, that the newly added E2EE public key creates a new identifier for the user. When users lose their device, they must issue a new pair of keys for the service. Aware attackers might leverage this information to infer changes on the user side and leverage them to facilitate attacks.

Leveraging both parties' public keys, the parties can securely create a shared secret using the X3DH protocol, an extended version of the Diffie-Hellman protocol. This shared secret is then used to derive keys to encrypt the messages between the parties. While this in of itself might be sufficient to fulfill E2EE's promise, in order to fulfill the advanced properties of E2EE, namely the aforementioned Perfect Forward Secrecy (PFS) and Post Compromise Security (PCS), more is needed.

- Perfect Forward Secrecy (PFS): Attackers cannot read messages that were encrypted before they took over the victims’ device and app.
- Post Compromise Security (PCS): Attackers cannot read messages that were encrypted after they were removed from the victims’ device and app.

As discussed above, to limit breached key exposure and achieve PCS and PFS, a new key for each message needs to be created. To do so the Signal protocol introduced the “Double Ratchet” algorithm. As its name suggests, the solution consists of two “ratchets” preventing attackers compromising a key to “move it forward” to read future encrypted messages, or “backwards” to read past encrypted messages:

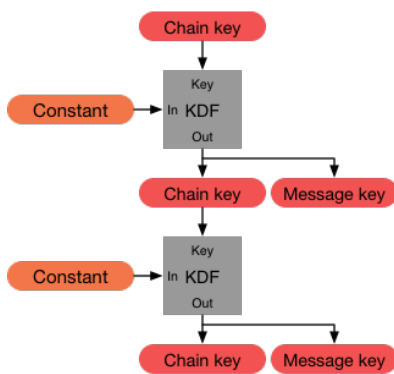


Figure 1 The Symmetric ratchet (source: signal.org)

The Symmetric ratchet (Fig 1): Ensures PFS, as it uses a one-way Key Derivation Function (KDF) to prevent attackers from calculating past keys from current keys.

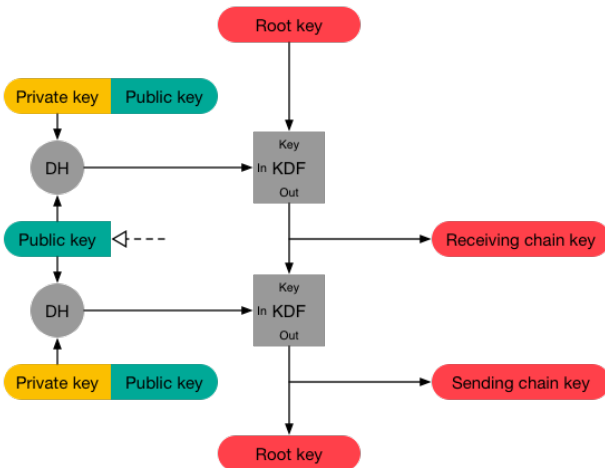


Figure 2 The Asymmetric ratchet (source: signal.org)

The Asymmetric ratchet (fig 2): This ratchet (sometimes called the “Diffie-Hellman/DH ratchet”) ensures PCS as it utilizes the entropy coming from the uncompromised other party to generate new keys.

Combining the symmetric and asymmetric ratchets together gives the Double Ratchet: When a message is sent or received, a symmetric-key ratchet step is applied to the sending or receiving chain to derive the message key.

When a new ratchet public key is updated via a received message, a DH ratchet step is performed prior to the symmetric-key ratchet to replace the chain keys.

3.2. Extending E2EE to the Multi Device setting: Existing solutions

As discussed above, in the pre E2EE era, the multi-device support requirements were trivial to solve. Since the IM server had access to the contents of the message, senders could just send their message once to the server, totally unaware of the receiver’s device setup and the IM server would handle its distribution to all of the receiver’s devices and sender’s other devices (so that their history would be up to date). However once E2EE is applied, the IM server cannot read the contents of the message and thus can no longer distribute them to all of the devices.

IM providers needed to address E2EE in the multi-device setting while still maintaining PCS and PFS requirements. Extending PFS and PCS definitions for the multi-device setting is quite natural:

- Perfect Forward Secrecy (PFS): Attackers cannot read messages that were encrypted before they took over the victim’s app on one device.
- Post Compromise Security (PCS): Attackers cannot read messages that were encrypted after they took over the victim’s app on one device and were removed from it.

There are two simple solutions to do so:

The “Leader” based solution: One of the user’s devices serves as the leader and the E2EE conversation happens between the parties leaders, in the same manner as if both users had a single device. The leaders then distribute the messages to their other devices, using E2EE between Leader and Devices. In the WhatsApp mobile based IM case, it would be natural to appoint the mobile device which was associated with the phone number that created the account as “leader” (or “primary device” in WhatsApp lingo).

This solution was applied by WhatsApp until mid-2021. However, the solution suffers from an obvious centralization drawback: When the leader device is offline, none of the other devices can communicate.

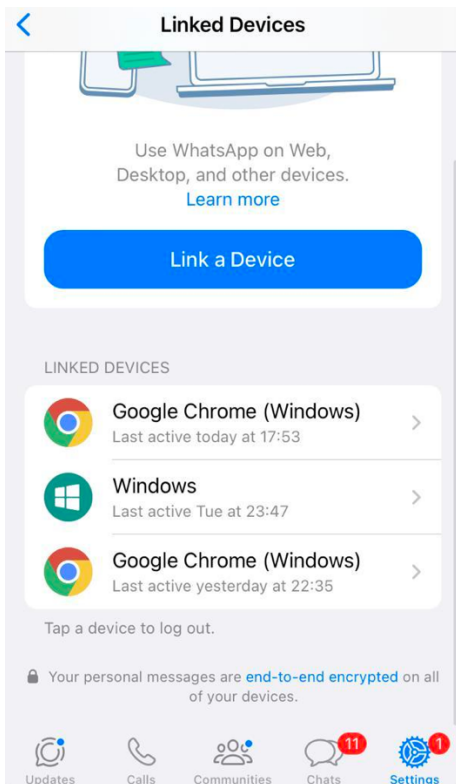


Figure 4 WhatsApp's linked devices screen of user X

User X's corresponding entries in the table matched this information as shown in Figure 5.

0	"g	4:0"	▶ {identifier:
1	"g	4:25.0"	▶ {identifier:
2	"g	4:26.0"	▶ {identifier:
3	"g	4:29.0"	▶ {identifier:

Figure 5 User X's corresponding entries in the identity store table

We had verified that such information is present even when the sender is not part of the receiver contact list and without actually sending messages to the receiver. Blocking the sender on the receiver side does not prevent it from getting device identity information.

We had responsibly disclosed our findings to Meta's bug-bounty program on January 9th 2024 but got politely rejected two days later, mainly because this is not an implementation bug but the way the protocol works by design.

Summing up, in order to obtain its victims' WhatsApp devices information, attackers need to:

- Know their victims' phone number.
- Add victims as contacts, no need to actually send a message to them.
- Use WhatsApp web client and monitor the identity-store table for information and changes.

5. Possible solutions

5.1. "Lockdown mode" to limit non-contacts access

This optional Lockdown mode will enable users to limit messages' reception to ones sent by their contacts only. Consequently, only the users' contacts will need and be able to view their device information.

While it does not fully prevent the privacy issue it presents a dramatic improvement compared to the current situation in which any user, including blocked users, can view that information.

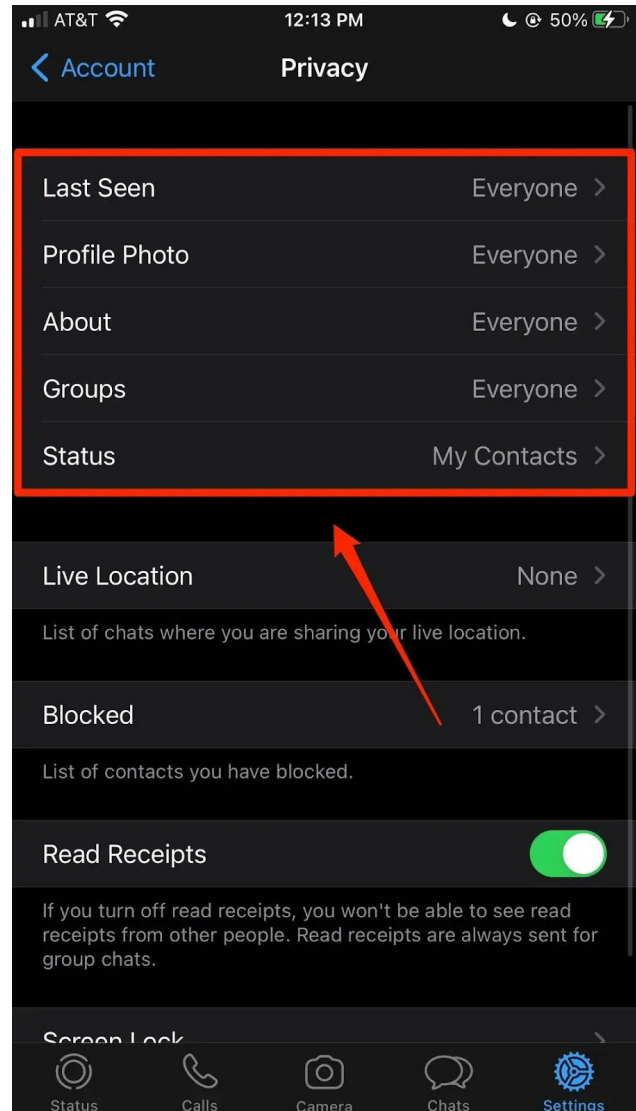


Figure 6 WhatsApp's privacy settings

This Lockdown mode can be beneficial to security and privacy aware users across the board and not just for this multi-device privacy issue, as it would protect them from receiving all kinds of malicious messages from non-contacts, which may include 0-days exploits, social engineering and phishing or even just spammy messages. The notion of limiting certain types of information to contacts only is

already present in WhatsApp as shown in Figure 6 and therefore already understood by its users.

5.2. Cryptographic solutions

To completely solve this issue a design change must be done, and the burden of distributing the messages needs to be removed from senders and placed on the receivers' instead.

As a result, the senders are only aware of a single recipient key, regardless of the number of the recipient's devices and are not aware of all recipients' devices and keys and cannot monitor changes to this setup.

A few researchers tried to suggest such solutions in the past, including a 2019 paper named "Multi-Device for Signal" that considers the multi-device scenario for the Signal protocol, which is used by WhatsApp (and others) and explicitly addresses and solves its privacy issues. It will be worthy to try and actually implement it or similar solution in popular IM E2EE solutions.

6. Conclusions

In this paper, we present the security and privacy tradeoffs of IM apps supporting both E2EE and multi-device. We demonstrate how attackers can easily subvert the WhatsApp client to obtain the victims' multi-device setup information and suggest some practical measures to limit the exposure of such privacy leaks.



Introduction to Procedural Debugging through Binary Libification

Jonathan Brossard

Conservatoire National des Arts et Métiers, Paris

Abstract

Assessing the existence, exact impact and exploitability of a known (or theoretical) memory corruption vulnerability in an arbitrary piece of compiled software has arguably not become simpler. The current methodology essentially boils down to writing an exploit - or at least a trigger - for each potential vulnerability. Writing an exploit for a weird machine involves several undecidable steps, starting with overcoming the reachability problem. In this article, we introduce the notions of “libification” and “procedural debugging” to facilitate partial debugging of binaries at the procedural level. These techniques allow the transformation of arbitrary dynamically linked ELF binaries into shared libraries, and the study of memory corruption bugs by directly calling the vulnerable functions, hence separating the memory corruption intraprocedural analysis from the reachability problem. Finally, we publish a framework [3] to implement such a libification under a permissive open-source license to facilitate its adoption within the security community.

1 Introduction

Triaging bugs has become an essential part of security. The Product Security function as a whole is becoming ever more critical for software manufacturers as legal frameworks around the globe mandate more clarity, speed, and transparency in dealing with existing and new vulnerabilities. The Cyber Resilience Act being implemented in Europe and the Executive Order on Improving the Nation’s Cybersecurity published in the US, for instance, both mandate the use of Software Bill of Materials (sBOMs) and their communication to clients and third parties, effectively rendering the superficial - software version based - vulnerability assessment of potential new CVEs affecting their software, seemingly more apparent.

However, assessing the actual existence, exact impact, and exploitability of a given memory corruption bug, as required by the above laws, has not become significantly more manageable over time. The current methodology to assess the

presence and impact of a given CVE in a piece of software essentially requires writing an exploit for each potential vulnerability. As such, this situation creates a seemingly unreasonable burden on Product Security teams, where triaging bugs requires performing operations like overcoming the reachability problem multiple times.

Writing exploits for a weird machine involves three steps: reaching, triggering, and exploiting. Much work has been done in automating the first step. Arguably, all of the fuzzing and dynamic testing performed hitherto follows this top-bottom approach, where execution starts from an application’s entry point, toward the leaves of the application, across the application’s call graph.

In this article, we aim to focus on the second step alone - without requiring solving the first one, which is undecidable in general.

Our methodology starts with modifying the ELF headers and dynamic section of an arbitrary dynamically linked ELF executable to turn it into a more workable shared library. The benefit of this technique is that any public function within the binary becomes callable without crafting an input to reach the attractive, potentially vulnerable function. Subsequently, we can render an arbitrary function within an ELF callable, even turn the entire ELF application into a callable API, and finally manually produce more limited, partial vulnerability triggers under the form of simple text files.

In the rest of this article, we will focus on memory corruption vulnerabilities unless stated otherwise and limit ourselves to C/C++ applications compiled as ELF binaries, as used under GNU/Linux and Unix-like operating systems, when implementing our framework. We will assume that the application’s source code is unavailable to the auditor.

Our first contribution is a methodology to transform an arbitrary ET_EXEC or ET_DYN dynamically linked ELF binary into a shared library. We provide a tool named the Witchcraft Linker to perform this operation on ELF32 and ELF64 executables alike, regardless of their architectures. Our second contribution is a methodology to invoke arbitrary C or C++ functions within ELF shared libraries without prior knowledge of their exact prototypes. We implement an original type of debugger, procedural-based, allowing the invocation of arbitrary C/C++ functions. This debugger, the Witchcraft Shell, noticeably does not use `ptrace()`, breakpoints, or single stepping. We name this new form of debugging procedural since analysis is performed at the granularity of function calls.

2 Previous Work

2.1 Exploitability of a Vulnerability

As noted by Wang et al. [49], in general, the only definitive way to prove the exploitability of a vulnerability is to write an exploit for that vulnerability. This constitutes proof by construction since the expert exhibits an exploit that demonstrates the exploitability of the vulnerability. On the other hand, proving that a vulnerability is not exploitable is a difficult problem, according to Suciu et al. [44]. Demonstrating the non-exploitability of a vulnerability via formal proof based on a crash analysis is sometimes possible despite the explosive nature of proofs based on symbolic execution [9] [49].

Green et al. [24] consider that when it comes to vulnerabilities such as memory corruptions, the fact that attacker controls the next instruction to be executed (the “Program Counter”) is a strong indicator of a function’s exploitability. However, the presence of countermeasures may not make this condition entirely sufficient [20].

2.2 Automatic Exploitation of Vulnerabilities Detected via Static Analysis

Several research projects focus on exploiting (or at least triggering) vulnerabilities detected using a preliminary static analysis to demonstrate that they are true positives. ExpRace [31], for example, focuses on a single class of vulnerabilities: race conditions in the Linux kernel. After having distinguished race conditions involving several variables (qualified as difficult) and race conditions involving a single variable in the kernel (qualified as easy), the authors propose a generic methodology for exploiting reusable single-variable race conditions on several cores, running under Intel processors, making it possible to trigger the previously identified vulnerability, taking advantage of the fact that an unprivileged process (or secondary thread) in user mode can significantly increase the race window using common system calls (`mmap()` and `mprotect()`) to trigger the synchronization of memory tables (“Lookaside Buffers translation”) between the cores of the

same Intel microprocessor. The tool is very specialized since it only addresses the problem of mono-variable race conditions in kernel mode under Linux.

The FUZE tool [51] aims at dynamically triggering, to prove their existence, “Use After Free” vulnerabilities in kernel mode under Linux. By combining open-source frameworks such as syzcaller (fuzzer), angr [46] (for binary analysis, function call graph generation, decompilation, and symbolic execution), and kernel mode debugging techniques (parsing the list of kernel modules, “LKM linked list”), it dramatically reduces the complexity of UAF vulnerability analysis by determining the few paths and system calls that can potentially modify a variable in kernel mode, then using combined fuzzing and symbolic execution techniques to generate user inputs capable of automatically triggering the vulnerability, and thus proving its existence.

The article “A Hybrid Interface Recovery Method for Android Kernel Fuzzing” [32] is also specialized. The problem raised by the authors is the addition of undocumented interfaces (system calls or ioctls) between user and kernel modes by mobile phone equipment manufacturers. These new interfaces are typically additions via proprietary kernel modules (the source code of which is unavailable, implying an analysis partially to be made in black box mode) to the Android kernel (which is based on Linux and is, therefore, open-source, auditable in white box mode). However, these interfaces are prime targets for privilege escalation attacks, where a program in unprivileged user mode will purposely call these extra interfaces to the privileged mode of the kernel to trigger vulnerabilities. Therefore, the analysis is gray, combining a white box analysis of the open-source Android part of the kernel and a black box analysis of the non-open-source, proprietary part added by the equipment manufacturer. The methodology followed is a taint analysis of proprietary modules, including type propagation, to find the prototypes of the interfaces introduced (whether they are new system calls in their own right or, more commonly, new valid ioctl calls on arbitrary device drivers). Once the prototypes of these interfaces have been determined, it becomes possible to use classic whitebox fuzzing tools, such as Syzcaller, by measuring the impact of calling these new system calls dynamically on the rest of the kernel (id est: by instrumenting only the open-source part of the kernel).

The PhD thesis “Finding race conditions in kernels: from fuzzing to symbolic execution” [52] proposes an original approach to the detection and exploitation of “time of check, time of use” (or TOCTOU) vulnerabilities, which are a subclass of race conditions, where a kernel resource is validated at time t , then read back and used at time $t+1$. The underlying fundamental issue is that this resource may have changed in the meantime, the Linux kernel being multi-tasking and concurrent, leading to false assumptions on the said resource core properties. It should be noted that several vulnerabilities of this type have been discovered on the Linux kernel

in recent years, hence the renewed interest in an automatic approach to the discovery and practical validation of this peculiar vulnerability subclass. The methodology followed is to modify the Linux kernel (using source patches) to instrument regions likely to contain TOCTOU vulnerabilities, selected by a preliminary static analysis, then use a fuzzer guided by symbolic execution toward these regions to be more purposefully scrutinized. This methodology is limited to TOCTOU vulnerabilities and does not apply to kernels whose code is unavailable.

Furthermore, the article “From source code to crash test cases through software testing automation” [16] offers a methodology for creating proof of exploits (id est: the automatic generation of user input triggering a vulnerability, previously identified in source code), by combining a preliminary static analysis (generation of the call graph of the application) of the application, a fuzzing engine to traverse this graph, and the use of a symbolic execution engine (named Triton) to guide the fuzzer toward the vulnerable function. Although the source code is essential to this methodology, it applies to several classes of vulnerabilities, giving it notable genericity.

2.3 Defense in Depth: Hardened Compilation Techniques

Countermeasures have been developed to prevent or limit the exploitability of vulnerabilities in compiled applications, particularly those developed in C or C++. Detecting and taking into account, where applicable, the presence of these countermeasures is critical when writing an exploit taking advantage of memory corruption.

Khalsan et al. [28] identify in particular the DEP (“Data Execution Prevention”) technique introduced in Microsoft Windows XP SP2, which makes the stack, dynamic memory, and variables in the data sections of an application non-executable. According to the authors, the non-execution of the stack is made possible thanks to hardware extensions (“NX” bit on AMD processors or “XD” equivalent on Intel processors). These countermeasures primarily aim to prevent the introduction and execution of shellcode [13] in all writable sections of the application. We also find the term W^X to name the segregation of variables (writable, non-executable) and code (executable, non-writable) in the literature [34] [10].

Khalsan et al. also describe the use of ASCII armoring, which ensures that all virtual addresses used by an application contain at least one 0x00 ASCII byte (in hexadecimal code). Given that the functions manipulating character strings end with a 0x00 (named ASCIIZ format), exploiting a stack buffer overflow vulnerability via the functions from libc making a copy of strings of characters is made impossible. Introducing ASCII armoring requires modifying the kernel and dynamic linker to provide armoring on the main binary and all its dynamic libraries.

Khalsan et al. detail the use of Address Space Layout Randomization (ASLR) [43], which consists of making the base address of a binary and each dynamic library in memory non-predictable. An attacker can no longer hardcode return addresses when writing an exploit. The introduction of ASLR typically requires modifying the kernel and the file format of executables to allow arbitrary relocation of protected binaries [34].

Khalsan et al. also describe binary protection techniques using canaries. These techniques have known several names, such as Propolice [21], StackGuard [15], and Stack Smashing Protection (SSP) [39]. This involves modifying the compiler in such a way as to introduce a canary (or “stack cookie”) before the return address in the stack, the integrity of which will be checked in the prologue of each instrumented function. If the canary has been modified, the stack is corrupted, and the program will be immediately terminated rather than risk arbitrary code execution by an attacker. These techniques have undergone several successive improvements until they no longer have any significant cost during the execution of the protected application [53].

Khalsan et al. finally detail FORTIFY (standardized in the ISO/IEC TR 247315 standard). This compilation option automatically replaces specific C library functions vulnerable to buffer overflows with functions including an additional argument, the maximum size of the destination buffer (which can often be inferred by the compiler). In the event of a stack buffer overflow during the program execution, the application is terminated rather than allowing the attacker to execute arbitrary code [30] [23].

These techniques have been extended to other architectures and operating systems, such as Linux [39], Android [33], OSX [39] or iOS [28].

Finally, there are protections against memory corruption at the hardware level of specific microprocessors, such as Intel Control Flow Integrity (CFI) [7] [29], which allows, by instrumenting the start of each block of code (an `endbr64` instruction is added at compile time under Intel x86-64) [29] to ensure that the control flow of the application has not been altered via memory corruption exploitation techniques such as `ret2libc` [10] or Return Oriented Programming (ROP) [40] [34] [1] [12] at any point in time. During a transfer of execution via branching or when returning to a calling function, the microprocessor can ensure whether the destination address is an `endbr64` instruction under x86-64 (respectively `endbr32` under x86) and terminate the application if this is not the case.

These countermeasures to exploiting memory corruption vulnerabilities are effective against their respective vulnerability subclasses but require activation (often at compile time) to operate correctly [50].

2.4 Binary Loaders and Binary Post-Compilation Instrumentation

The idea of statically or dynamically loading and instrumenting binaries is fundamental in analyzing compiled applications.

The most basic form of dynamic instrumentation is simply using the trap instructions to force an application to divert its execution flow, as seen in DTrace [14].

A more complex tool like Valgrind and its popular memcheck [42] memory sanitizer can perform a Just in Time (JIT) dynamic recompilation of executables. It is a complex framework that starts by transforming the original basic blocks of the application into an intermediate representation, then applies instrumentation and code optimization before translating the intermediate representation back to machine code [36]. Such an instrumentation is heavy and incurs an execution penalty of 10x or higher.

Some of the techniques available include dynamically rewriting a single basic block of code at a time, while the application is running, using a shadow memory mechanism. This is the foundation of tools like DynamoRIO [4] [6] [5], a framework reused in popular security tools such as WinAFL [55].

Dinesh et al., on the other hand, opt for a pure static rewriting of binaries to retrofit into binaries instrumentation that is usually introduced at compile time, such as AFL [54] and Address Sanitizer [41]. Their framework, named RETROWRITE [17], works by diverting the flow of execution through the insertion of trampolines. A preliminary static analysis involves building the control flow graph, which is a difficult problem in general [35] and undecidable [22].

This mechanism, where a preliminary disassembly and control flow recovery precedes a static rewriting of portions of the binary to introduce instrumentation code, is a popular design [2] [48] [37] [47], subject essentially to the same limitations: recovery of the control flow is undecidable in general [22].

To avoid this pitfall, Duck et al. [19] developed a suite of binary rewriting techniques, implemented under the E9Patch framework, that can insert jump trampolines without requiring an understanding of the binary's control flow. As such, their instrumentation is more robust and scales to large applications such as web browsers. They leverage techniques such as instruction punning [11], which may safely replace branching conditions and introduce trampoline code by overwriting exactly one assembly instruction.

Furthermore, it is worth mentioning the idea of recovering individual object files from a compiled binary [8] thanks to a control flow and data flow analysis. When individual compilation units can be unlinked, they may be subsequently relinked and instrumented.

Finally, custom loaders may allow the loading of Windows dynamic libraries under Linux [38] or rewriting Windows Executables so they may be loaded as DLLs [18].

In light of this state of the art, it seems relevant to introduce a more lightweight form of binary rewriting focused solely on modifying an application's metadata. As such, it shall not suffer from the limitations of the techniques based on control flow recovery or the runtime penalty of dynamic instrumentation.

3 Overview of the Libification Process

3.1 Libification: Methodology

In this section, we describe the production of a libifier, that is to say, a tool able to reliably and automatically transform an arbitrary ELF binary into a shared library. We detail this methodology, so it may be extended in the future, if necessary, to compensate for breaking changes in the GNU dynamic linker, or adapted to other toolchains.

The POSIX 2001 standard specifies the API of the dynamic linker, and in particular the `dlopen()` function, which allows loading an arbitrary shared library in memory:

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flags);
```

The `filename` parameter must point to the path to the library to be loaded on the file system.

The `flags` parameter controls the locality (local or global) of the symbols loaded in the address space, as well as the behavior of the dynamic linker. In particular, if the `RTLD_LAZY` bit is set, the dynamic linker performs lazy binding of symbols when necessary, as opposed to an immediate binding at load time if the `RTLD_NOW` bit is set, in which case the Global Offset Table may be safely remapped read-only.

In the remainder of this chapter, we will define a shared library as an ELF file that can be loaded in memory via `dlopen()`.

A minimal oracle to determine whether the dynamic linker can load an ELF file can be summarized with the following code:

```
int main (void) {
    void *handle = 0;

    handle = dlopen("./test.so", RTLD_NOW);

    if (handle <= 0) {
        printf(" !! ERROR: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    printf("Loading successful\n");
    return 0;
}
```

If successful, the return code from this oracle will be 0. It will be non-zero otherwise, and an error message stemming from the dynamic linker will indicate the cause of the memory loading error. Empirically, the work of the libifier will, therefore, be to modify the binary in a way that prevents the error returned by `dlerror()` from occurring. The developer of the libifier will then read the code of the dynamic linker, identify the cause of the error, and modify the libifier to patch the input binary to prevent this last error from occurring.

The goal - and hope - of the developer of the libifier is that through this iterative and empirical process, the shared library produced by the libifier will be able to pass all of the `dlopen()` parsing checks, and eventually be loaded in memory. There is no guarantee that such a libification will be or remain possible in the future or across an arbitrary corpus of executables since this libification is a reverse engineering technique and not a standardized feature of a dynamic linker guaranteed in any form or fashion.

3.2 Practical Libification

The operations performed by the Witchcraft Linker to libify an arbitrary ELF binary modify the ELF header, the dynamic section, and the GNU-specific symbols versioning section of an input executable.

First, within the ELF header, the libifier must ensure that the `e_type` field is set to `ET_DYN` since all shared libraries are of type `ET_DYN`.

Then, the dynamic section of the ELF must be parsed and possibly modified:

The `DT_BIND_NOW` shall be changed to `DT_NULL` if present in the `.dynamic` section.

The `DT_FLAGS_1` flags present in the `.dynamic` section may need to be modified: the `DF_1_PIE` and `DF_1_NOOPEN` bits must be removed if set. This last flag prevents an object from being loaded via `dlopen()`.

If the binary features constructors or destructors, those may not expect to be called from `dlopen()`. The Witchcraft Linker, therefore, features an optional command line argument to prevent constructors and destructors from being called. Within the `.dynamic` section, setting the values of `DT_INIT_ARRAYSZ` and `DT_INIT_ARRAY` to zero inhibits the instantiation of constructors, and setting the values of `DT_FINI_ARRAYSZ` and `DT_FINI_ARRAY` to zero inhibits the calls to destructors.

Finally, because the dynamic linker may refuse to load multiple versions of symbols if symbols versioning is in use within the libified binary, the Witchcraft Linker will simply zero out the entire section of type `SHT_GNU_versym`.

Currently, the Witchcraft Linker (`wld`) version `v0.0.6` can libify all of the binaries of a standard GNU/Linux distribution such as Ubuntu 22.04 LTS, so that they may be loaded via the `dlopen()` function of the GNU dynamic linker version 2.35.

3.3 Toward Procedural Debugging

Once the principle of libifying an ELF has been acquired, writing a debugger capable of loading a libified executable in its own address space is straightforward: simply load the libified binary via the `dlopen()` function of the dynamic linker. It appears appealing to integrate an interpreter into our debugger to allow a developer to interact with the functions exposed by the libified binary. Due to its tiny size, the choice of interpreter fell on the Lua language [25] since a Lua interpreter, including all its dependencies, occupies less than 500 kilobytes of memory footprint.

We wish to make the entire API available in the address space available to the Lua interpreter once the libified binary is loaded in memory. This API is made up, on the one hand, of the functions exported directly by the libified binary but also of the APIs exported by all the dynamic libraries loaded in memory by the dynamic linker when loading the libified binary in memory via `dlopen()`. The case of functions declared static and hence not exported at compile time is left aside for now¹. Obtaining these APIs can be done via the use of the `dlinfo()` function of the dynamic linker [27] [45].

By making the entire API available in memory exposed to the Lua interpreter, we simply make these APIs available to the developer. One of the advantages of this methodology is that a developer or security analyst may invoke any function loaded in the address space without worrying too much about the actual calling conventions or prototypes (number and type of arguments) of these functions. Additionally, they may do so without compilation from a Lua interpreter, which facilitates manual exploration of said APIs.

We name this technique, which allows invoking a single function at a time, “procedural debugging”.

3.4 An Empirical Assessment of the Side Effects of Libification

In this section, we address the question of the side effects introduced by the libification of a binary over its main security hardening properties.

We successively consider the following properties: the base address of the executable mapping (ASLR), the presence of stack cookies aimed at preventing buffer overflows, the stack’s executability, the presence of static relocations (RELRO), and the presence of Control Flow Integrity type protections (Intel FCF).

Libification of an `ET_DYN` binary does not modify its ASLR properties: the binary being initially mappable to an arbitrary address remains so. In the case of the libification of an `ET_EXEC` binary, which was initially only mappable to a fixed address, the ASLR is not modified either: the library

¹Static functions whose addresses relative to the base address of libraries or executables are known thanks to a preliminary control flow analysis may be named and called within the debugger.

thus generated is only mappable at the same address. Loading happens as if the binary had been transformed into a library by prelinking to the same base address [26].

The stack executability of a library loaded via `dlopen()` is determined by the stack executability of our debugger since the latter loads the library in its own address space. This debugger property can be arbitrarily changed via the `execstack2` application.

Libification does not modify the presence of static relocations (binary or library with the `BIND_NOW` flag in their dynamic sections).

The presence of stack cookies protecting the stack is intrinsic to each function since it is implemented by instrumenting the prologue and epilogue of each function. Libification, therefore, does not modify this property of the functions present in the libified binary.

The presence of Intel Integrity Protection (Intel FCF) type protections is characterized by the presence of `endbr64` instructions at the start of each basic block in each protected function. Libification does not modify this intrinsic property either.

Finally, this empirical study overall suggests that libifying an ELF binary into a shared library does not modify its fundamental security properties, particularly the countermeasures possibly introduced into the binary at compile time. In a nutshell, libification does not introduce notable security side effects from an exploitability standpoint.

3.5 Limits to Binary Libification

Libifying an `ET_EXEC` binary as a shared library generates a somewhat special shared library since it cannot be remapped to an arbitrary address. This induces a limit to our libifier. On the one hand, a libified library can generate a collision with the address space of a program trying to load it, as noted by beta testers³. On the other hand, it is not possible to load two libified `ET_EXEC` binaries initially provided with the same base address in our debugger.

3.6 Validation

The libification process and the WSH debugger were validated under GNU/Linux Ubuntu 22.04 equipped with an Intel 64-bit processor using the following binaries:

Software	Version	Status	Time
Google Chrome	114.0.5735.198	OK	< 0.01s
OpenSSH Server	8.9p1	OK	< 0.01s
Apache2	2.4.52	OK	< 0.01s
Nginx	1.18.0	OK	< 0.01s
GCC	11.4.0	OK	< 0.01s

²<https://linux.die.net/man/8/execstack>

³Thanks to Dan Kaminsky <https://github.com/endrazine/wcc/issues/26>

Furthermore, copying, libifying, and loading via `dlopen()` the 435 binaries in the default path of a default Ubuntu 22.04 AMD64 install took less than 3 seconds (in total) on a laptop featuring a core i-7 11850H CPU and 32Gb of RAM.

4 Conclusion and Future Work

In this article, we presented a methodology to transform a dynamically linked ELF binary into a shared library. We called this methodology “libification”.

We then introduced a very simple debugger able to load such a libified executable within its own address space, hence rendering nonstatic functions within the binary callable. We named this technique facilitating the invocation of arbitrary functions in isolation and out of context “procedural debugging”.

Thus, a security analyst seeking to experiment with a possible vulnerability within an executable manually may now directly invoke the function featuring the vulnerability via procedural debugging without needing to produce user inputs traversing the application’s call graph before reaching the vulnerable function. This is notable since the reachability problem is undecidable in general.

We verified the reproducibility of the libification process on some of the most complex user-mode binaries available under GNU/Linux, as well as across an entire widespread Linux distribution, which validates the generality of the approach.

In the future, we hope to be able to automatically generate scripts to trigger a vulnerability within a compiled binary, which would save significant time for Product Security teams.

Availability

The Witchcraft Compiler Collection [3], including the Witchcraft Linker described in this article, is published under a permissive dual MIT/BSD open-source license. The framework is available from <https://github.com/endrazine/wcc> and via the package managers of several GNU/Linux distributions, including at least Debian, Ubuntu, and Arch Linux.

References

- [1] Salman Ahmed, Long Cheng, Hans Liljestrand, N Asokan, and Danfeng Daphne Yao. Tutorial: Investigating advanced exploits for system security assurance. In *2021 IEEE Secure Development Conference (SecDev)*, pages 3–4. IEEE, 2021.
- [2] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

- [3] Jonathan Brossard. The witchcraft compiler collection. <https://zenodo.org/doi/10.5281/zenodo.11298208>, May 2024.
- [4] Derek Bruening and Saman Amarasinghe. Efficient, transparent, and comprehensive runtime code manipulation. 2004.
- [5] Derek Bruening and Timothy Garnett. Building dynamic instrumentation tools with dynamorio. In *Proc. Int. Conf. IEEE/ACM Code Generation and Optimization (CGO)*, Shen Zhen, China, 2013.
- [6] Derek Bruening and Qin Zhao. Building dynamic instrumentation tools with dynamorio.
- [7] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.
- [8] Mauro Capeletti. Unlinker: an approach to identify original compilation units in stripped binaries. 2016.
- [9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.
- [10] S Sibi Chakkaravarthy, Dhamodara Sangeetha, and V Vaidehi. A survey on malware analysis and mitigation techniques. *Computer Science Review*, 32:1–23, 2019.
- [11] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–332, 2017.
- [12] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.
- [13] Tsung-Huan Cheng, Ying-Dar Lin, Yuan-Cheng Lai, and Po-Ching Lin. Evasion techniques: Sneaking through your intrusion detection/prevention systems. *IEEE Communications Surveys & Tutorials*, 14(4):1011–1020, 2011.
- [14] Greg Cooper. Dtrace: dynamic tracing in oracle solaris, mac os x, and free bsd by brendan gregg and jim mauro. *ACM SIGSOFT Software Engineering Notes*, 37:34, 2012.
- [15] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *Linux Expo*, 1999.
- [16] Robin David, Jonathan Salwan, and Justin Bourroux. From source code to crash test-cases through software testing automation. *Proceedings of the 28th C&ESAR*, page 27, 2021.
- [17] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.
- [18] Aleksandra Doniec. Converts a exe into dll. https://github.com/hasherezade/exe_to_dll, 2020.
- [19] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, pages 151–163, 2020.
- [20] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2017.
- [21] Hiroaki Etoh and Kunikazu Yoda. Propolice: Protecting from stack-smashing attacks. *Technical Report, IBM Research Division, Tokyo Research Laboratory*, 2000.
- [22] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidirogrou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.
- [23] Jeff Gennari, Shaun Hedrick, Frederick W Long, Justin Pincar, and Robert C Seacord. Ranged integers for the c programming language. 2007.
- [24] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *Proceedings on Privacy Enhancing Technologies*, 2023.
- [25] Roberto Ierusalimsky. *Programming in lua*. Roberto Ierusalimsky, 2006.
- [26] Changhee Jung, Duk-Kyun Woo, Kanghee Kim, and Sung-Soo Lim. Performance characterization of prelinking and preloading for embedded systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 213–220, 2007.

- [27] David Keller, Timothy Roscoe, Reto Achermann, and Simon Gerber. Bachelor's thesis nr. 137b.
- [28] Mahmood Jasim Khalsan and Michael Opoku Agyeman. An overview of prevention/mitigation against memory corruption attack. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, pages 1–6, 2018.
- [29] Sandeep Kumar, Diksha Moolchandani, and Smruti R Sarangi. Hardware-assisted mechanisms to enforce control flow integrity: A comprehensive survey. *Journal of Systems Architecture*, 130:102644, 2022.
- [30] Marc-André Laverdière, Serguei A Mokhov, and Djamel Benredjem. On implementation of a safer c library, iso/iec tr 24731. *arXiv preprint arXiv:0906.2512*, 2009.
- [31] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. {ExpRace}: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380, 2021.
- [32] Shuaibing Lu, Xiaohui Kuang, Yuanping Nie, and Zhechao Lin. A hybrid interface recovery method for android kernels fuzzing. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 335–346. IEEE, 2020.
- [33] Héctor Marco-Gisbert and Ismael Ripoll-Ripoll. Sspfa: effective stack smashing protection for android os. *International Journal of Information Security*, 18(4):519–532, 2019.
- [34] Jonathan AP Marpaung, Mangal Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pages 744–749. IEEE, 2012.
- [35] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.
- [36] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory, 2004.
- [37] Trail of Bits. Mcsema. <https://github.com/lifting-bits/mcsema>, 2020.
- [38] Tavis Ormandy. Porting windows dynamic link libraries to linux. <https://github.com/tavis/loadlibrary>, 2017.
- [39] Conor Pirry, Hector Marco-Gisbert, and Carolyn Begg. A review of memory errors exploitation in x86-64. *Computers*, 9(2):48, 2020.
- [40] Yefeng Ruan, Sivapriya Kalyanasundaram, and Xukai Zou. Survey of return-oriented programming defense mechanisms. *Security and Communication Networks*, 9(10):1247–1265, 2016.
- [41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.
- [42] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.
- [43] Zhidong Shen and Weiyang Chen. A survey of research on runtime rerandomization under memory disclosure. *IEEE Access*, 7:105432–105440, 2019.
- [44] Octavian Suci, Connor Nelson, Zhuoer Lyu, Tiffany Bao, and Tudor Dumitras. Expected exploitability: Predicting the development of functional vulnerability exploits. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 377–394, 2022.
- [45] Justin Tracey. Building a better tor experimentation platform from the magic of dynamic elfs. Master's thesis, University of Waterloo, 2017.
- [46] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.
- [47] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.
- [48] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.
- [49] Yan Wang, Wei Wu, Chao Zhang, Xinyu Xing, Xiaorui Gong, and Wei Zou. From proof-of-concept to exploitable. *Cybersecurity*, 2(1):1–25, 2019.
- [50] Ye Wang, Qingbao Li, Zhifeng Chen, Ping Zhang, and Guimin Zhang. A survey of exploitation techniques and defenses for program data attacks. *Journal of Network and Computer Applications*, 154:102534, 2020.
- [51] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

- [52] Meng Xu. Finding race conditions in kernels: The symbolic way and the fuzzy way. 2020.
- [53] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 429–438. IEEE, 2006.
- [54] Michal Zalewski. Afl: American fuzzy lop. <https://lcamtuf.coredump.cx/afl/>, 2016.
- [55] Google Project Zero. Winaf. <https://github.com/googleprojectzero/winafl>, 2016.



The Power of Words: Generating PowerShell Attacks from Natural Language

Pietro Liguori*, Christian Marescalco**, Roberto Natella*, Vittorio Orbinato*, Luciano Pianese*
DIETI, Università degli Studi di Napoli Federico II, Naples, Italy
**{pietro.liguori, roberto.natella, vittorio.orbinato, luciano.pianese}@unina.it*
***c.marescalco@studenti.unina.it*

Abstract

As the Windows OS stands out as one of the most targeted systems, the *PowerShell* language has become a key tool for malicious actors and cybersecurity professionals (e.g., for penetration testing). This work explores an uncharted domain in AI code generation by automatically generating offensive PowerShell code from natural language descriptions using Neural Machine Translation (NMT). For training and evaluation purposes, we propose two novel datasets with PowerShell code samples, one with manually curated descriptions in natural language and another code-only dataset for reinforcing the training. We present an extensive evaluation of state-of-the-art NMT models and analyze the generated code both statically and dynamically. Results indicate that tuning NMT using our dataset is effective at generating offensive PowerShell code. Comparative analysis against the most widely used LLM service ChatGPT reveals the specialized strengths of our fine-tuned models.

1 Introduction

Offensive security practices, such as red teaming and adversary emulation, play a crucial role by helping us to understand how attackers take advantage of vulnerabilities and how to mitigate attacks [1, 2]. In these attacks, cybersecurity professionals emulate malicious post-exploitation actions, such as credential stealing, lateral movement across accounts and machines, data obfuscation and exfiltration, and more [3].

As Windows stands out as one of the most targeted OS [4], the *PowerShell* language has become a key tool for both malicious actors and cybersecurity professionals. This language is widely used to perform attacks since it can perform complex actions, such as establishing connections and accessing OS services and APIs without the need to deliver a malicious binary executable or payload on the target machine (e.g., “fileless” malware), making them harder to detect [5–8].

Unfortunately, writing offensive code demands a high degree of expertise and effort, restricting the adoption of offensive security practices. Therefore, the rise of automatic *AI*

code generators represents an appealing solution to unlock these practices to a broader spectrum of users [9].

AI code generators leverage ML models for Neural Machine Translation (NMT) to produce (offensive) code starting from inputs in Natural Language (NL), e.g., in the English language. The usage of NMT models is widespread across diverse software engineering tasks [10], yet their application in security-related scenarios is infrequent and not widely explored. This gap stems primarily from the lack of suitable corpora for training and evaluating code generators. The shortage of corpora for offensive code generation is an evident limitation: existing benchmarks [11–13] are derived from programming competitions and software interview questions (e.g., about algorithms and mathematics), or they focus on programs and languages that are not related to security (e.g., web applications in Python). Only a few security-oriented datasets are publicly available, targeting shellcodes in low-level programming languages [14]. As a result, there is a significant gap in the literature on offensive PowerShell code generation.

This work presents an assessment of AI code generators for PowerShell offensive code, a novel application of NMT. Given that generative models are predominantly trained on mainstream programming languages like Python and Java, we investigate strategies to repurpose these models for the PowerShell domain. To this aim, we adopt a combination of unlabeled and labeled datasets to train and evaluate models. Specifically, we first use a large collection of unlabeled (i.e., code only) samples of general-purpose PowerShell from various online repositories to pre-train ML models and refine their capabilities to comprehend and generate PowerShell code. Then, we build from scratch a manually annotated labeled dataset consisting of PowerShell code samples specifically crafted for security applications, which we pair with curated NL descriptions in English. We use this dataset to fine-tune three state-of-the-art NMT models (CodeT5+ [15], CodeGPT [16], and CodeGen [17]) to generate offensive PowerShell code. The dataset also serves as a ground truth for

the evaluation. We publicly share code, models ¹ and datasets as open data² to encourage further experimentation on this topic.

To perform our experiments, we formulate four key research questions (RQs) aimed at evaluating the models’ capabilities and the impact of the training strategies, performing static and execution analysis to assess the generated code, and comparing privately fine-tuned models with ChatGPT, the most widely used LLM service from OpenAI [18]. Table 1 summarizes the key findings of our analysis. To the best of our knowledge, this is the first work on the automatic generation of offensive PowerShell code from NL descriptions.

In the following, Section 2 discusses related work; Section 3 describes the research study; Section 4 shows the experimental results; Section 5 discusses the threats to validity; Section 6 discusses the ethical considerations; Section 7 concludes the paper.

2 Related Work

This work focuses on offensive code generation, involving machine translation techniques applied to the security domain for PowerShell code generation. Thus, we reviewed related literature in these areas.

ML for security-related PowerShell. Li *et al.* [19] designed a subtree-based de-obfuscation method and a semantic-aware PowerShell attack detection system. This work also demonstrates how the presented de-obfuscation method improves the performance of detection systems such as Windows Defender and Virus-Total. PowerDP [20] is a solution that aims to automatically identify malicious PowerShell commands through character distribution features and obfuscation multi-label classification also proposing a de-obfuscator method for recovering obfuscated commands. Even ML-based methodologies have arisen for detection purposes, as shown by Hendler *et al.* [21], who proposed several ML-based detectors demonstrating their effectiveness on malicious scripts. The authors also devised another solution [22] to achieve the same objective by retrieving information from Microsoft’s AMSI interface. Mimura and Tajiri [23] presented a lighter methodology, restricting detection only to word embeddings. Mezawa *et al.* [24] proposed an evaluation methodology for ML-based detectors based on a word-level machine learning model. Given the effectiveness of Abstract Syntax Trees (ASTs) in detecting obfuscated PowerShell scripts, Rusak *et al.* [25] proposed a hybrid approach that combines ASTs and deep learning to enhance detection methods for high-level obfuscation PowerShell malicious programs. We remark that research of ML for PowerShell focuses on *defensive* uses (i.e., detecting and de-obfuscating attacks), but none of these studies analyzed the *offensive* uses of ML (i.e., generating attacks), which are also

¹HuggingFace repo

²GitHub repo

Analysis	Main Findings
Capability Assessment	<ul style="list-style-type: none"> Models without fine-tuning (<i>zero-shot learning</i>) showed a limited ability to generate PowerShell code, often defaulting to Python syntax or incorrect PowerShell code. The fine-tuning phase significantly enhanced the models’ ability to generate syntactically correct and semantically relevant PowerShell code. Among the models, CodeT5+ and CodeGPT demonstrated notable improvements in generating offensive PowerShell code. Pre-training on a large PowerShell corpus had a varying impact on different models. While pre-training generally improved CodeT5+ and CodeGPT, especially with a limited number of epochs for fine-tuning, CodeGen did not consistently benefit from pre-training.
Static and Execution Analysis	<ul style="list-style-type: none"> All models achieved high syntax accuracy, indicating their strong capability to generate syntactically correct code. However, a significant number of warnings were identified, suggesting potential issues or suboptimal coding practices. The execution analysis showed that, despite textual differences between the ground truth and the generated code, the models are still able to generate offensive PowerShell code closely aligned with the intended malicious activities, in terms of events occurring in the system (e.g., on the filesystem, network, registry).
Comparison with public AI model	<ul style="list-style-type: none"> Our fine-tuned models outperform ChatGPT across all the metrics, showing that specializing the models on our fine-tuning dataset provides an advantage in the offensive PowerShell code generation task.

Table 1: Main findings.

relevant for red teaming and adversary emulation purposes, and which are in the scope of this paper.

Offensive Code Generation. Research on AI code generators for offensive security is still at an early stage. Gupta *et al.* [26] presented an outlook of the possibilities opened by ChatGPT for generating various types of cyber attacks, such as social engineering, phishing attacks, and malware creation. For each attack scenario, the paper shows qualitative examples of prompts submitted to ChatGPT, and the attack payloads generated as a result, including some snippets of PowerShell code. Similarly, Charan *et al.* [27] presented qualitative examples with ChatGPT and Google BARD to generate malicious

scripts (mainly in Python, Bash, and PowerShell) for the top 10 prevalent MITRE Techniques of 2022, showing the potential of these AI models for security applications. However, none of these studies systematically analyzed AI code generators, lacking in several aspects: (i) the evaluation was limited to a few examples, while systematic evaluation requires much larger datasets; (ii) the study lacked a ground truth for evaluating the correctness of generated code; (iii) they did not yet explore the potential of fine-tuning ML models for security-related code generation. The few studies in this direction focused on generating *exploits* in low-level languages (e.g., to attack memory management vulnerabilities). However, exploitation is only a limited part of the cyber kill chain, overlooking several more types of malicious code. Among these studies, Liguori *et al.* [28] proposed a dataset and approach for training and evaluating AI code generators for code security, by generating shellcodes in Assembly language. EVIL [29] automatically generates exploits for conducting code injection attacks via NMT by targeting both the generation of shellcodes in Assembly language and related Python code for encoding and obfuscating the shellcodes. DualSC [30] formalizes the automatic generation and summarization of shellcodes via a "Shallow" Transformer inspired by the T5 model and dual learning using the corpus provided by Liguori *et al.* [28]. ExploitGen [31] is an approach for generating exploit code in Python and Assembly based on the CodeBERT model. Differently from these studies, we presented a dedicated model for generating offensive PowerShell code, covering the entire cyber kill chain (e.g., including credential stealing, lateral movement, data exfiltration, and more tactics from the MITRE ATT&CK taxonomy). Moreover, we systematically analyzed the quality of generated PowerShell code by introducing a manually curated dataset to serve as a ground truth and evaluating the code statically and dynamically.

3 Research Study

The main objective of our research study is to understand whether NMT models can translate NL descriptions into code that accurately replicates the complexities of cyber attacks in PowerShell. This aspect is crucial as it explores the models' understanding of the unique syntax and semantics of this programming language.

Figure 1 provides an overview of this research study. We analyze various deep learning strategies to accurately generate code and introduce datasets to train and evaluate them. We study several state-of-the-art NMT models and introduce various approaches to evaluating the generated code, including the similarity of the generated code to ground truth and static and dynamic analysis of the code.

To help NMT models in the novel and ambitious task of generating PowerShell code from NL, we adopt a two-step process consisting of **pre-training** and **fine-tuning**. The pre-training phase aims to tailor NMT models (already pre-trained

on other programming languages) in the generation of PowerShell code. Armed with the pre-trained models, we proceed to the fine-tuning phase. This iterative process refines the models' capabilities, enabling them to generate offensive PowerShell code from NL descriptions.

The main problem in using NMT models is to have a sufficient set of data and to use them effectively to train the models themselves. Recognizing the lack of suitable datasets for offensive PowerShell code generation, in this study, we collect a large set of PowerShell programs used for penetration testing and adversary emulation. In addition to the code, we create descriptions of these programs in English to allow the model to translate English into PowerShell code. This dataset was created manually to verify that the programs were related to security and to ensure that the English language descriptions were complete and consistent with the code. The dataset is labeled since each sample includes both the text to translate into code and the code expected to be produced by the model (ground truth).

The creation of labeled datasets is inevitably limited by the availability of PowerShell security programs and the need to manually create English language descriptions for each program. To increase the amount of training data, in this study, we investigate an additional strategy, fully automated, to build an extended dataset of PowerShell programs, collecting PowerShell programs and the related text from the web (for example, comments in the code or description accompanying the code). As the collection is fully automated, this second dataset is non-labeled. The dataset includes programs not strictly related to security but includes, in general, PowerShell code used for various purposes. This dataset still contributes to the ability to generate security code since it allows the model to learn from further examples how to generate syntactically valid PowerShell code and to correlate the PowerShell code with the English language. We use this dataset to pre-train the NMT models, carrying out additional unsupervised training rounds.

Table 2 reports the statistics of both datasets, in terms of size, unique number of tokens, and average number of tokens for NL descriptions (only for fine-tuning data) and code.

Finally, we evaluate the models as follows:

- *Capability Assessment*: We compare the textual similarity of the code generated by the models with a ground-truth reference through automatic metrics. These metrics are an appealing solution to estimate the generated code since they are easy to tune and time-saving, hence overcoming the limit of human evaluation, which poses practical challenges for large-scale assessments.
- *Static analysis*: We assess the generated code to ensure that it adheres to PowerShell programming conventions and does not contain syntax errors.
- *Execution analysis*: We evaluate the capability of the

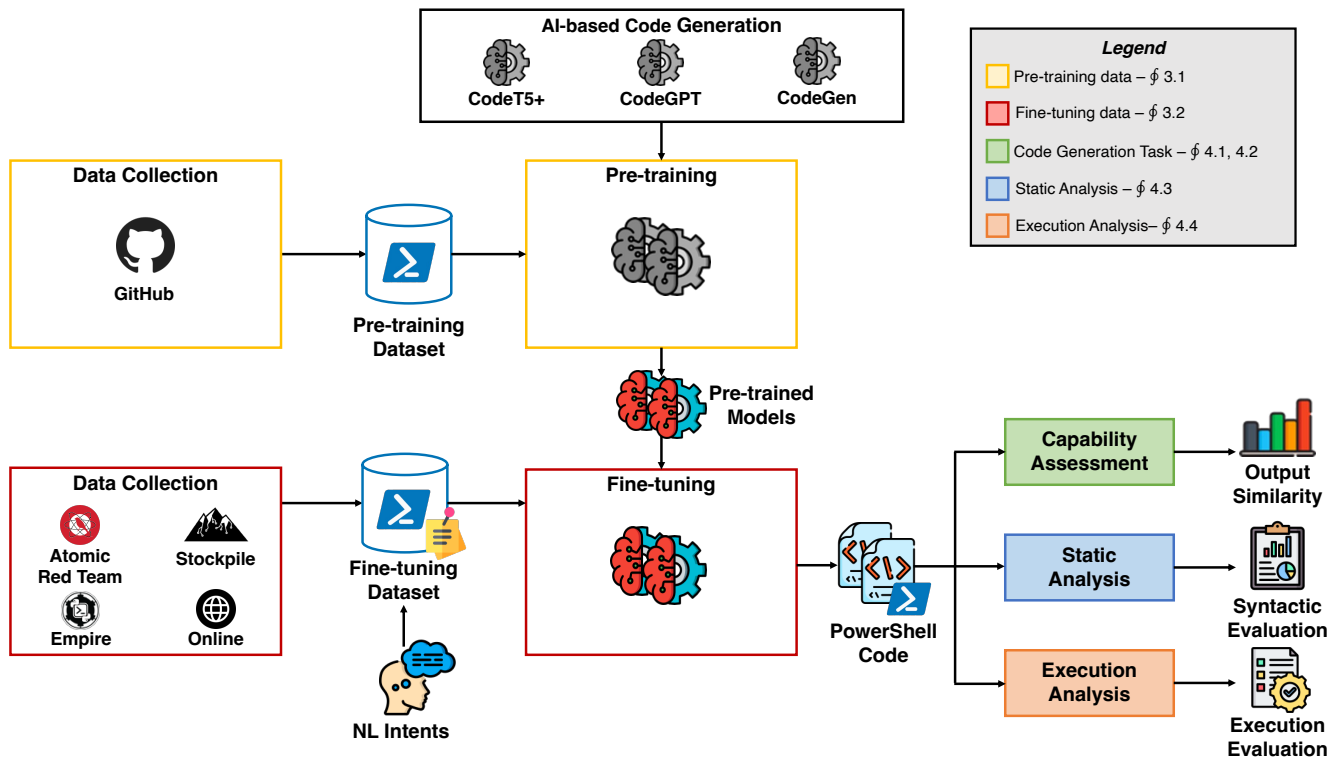


Figure 1: Overview of our research study.

generated offensive PowerShell code in executing malicious actions, replicating the behavior of the ground truth commands.

In the following of this section, we detail the pre-training (§ 3.1) and the fine-tuning data (§ 3.2), and the code generation task (§ 3.3).

3.1 Pre-training data (unlabeled)

Pre-training involves training the model on a large corpus of text data to learn general language representations before fine-tuning it for specific downstream tasks [32]. In other words, the parameters obtained from this step serve as a starting point for the later supervised training. Unsupervised or self-supervised pre-training is particularly attractive in the NMT context since large unlabeled data is available on the Internet. In this work, we leverage domain-adaptive pre-training (DAPT) [33]: given an NMT model pre-trained on massive, heterogeneous corpora, we perform additional rounds of unsupervised training with domain-specific data. Specifically, we leverage general-purpose PowerShell code for pre-training. The pre-training dataset aims to provide a valuable resource to enable the models' understanding of general-purpose PowerShell code. This dataset encompasses $\sim 90k$ samples extracted through the GitHub API. Specifically, we queried all

the repositories containing PowerShell code from the last decade (2013-2023) to encompass a broad spectrum of PowerShell code, then parsed the extracted data to remove unnecessary information, such as duplicates (inside the same repository), and logging and echo commands. In addition, we filtered out all the PowerShell commands with sizes greater than 1024, ensuring the dataset maintains a balanced representation of code complexities. This collection encompasses a diverse array of PowerShell scripts, spanning various application domains such as system administration, automation, and network management. Including a wide range of scripts reflects the versatility of PowerShell as a scripting language and provides models with exposure to the diverse ways PowerShell is used across different use cases.

The pre-training process depends on the model architecture. For decoder-only models, i.e., CodeGPT and CodeGen, we chose *Causal Language Modeling (CLM)*, also referred to as Language Modeling, as the pre-training objective. CLM has been extensively used as a pre-training task for transformer-based decoder-only models [34], such as in the GPT series [35–37]. CLM refers to language models that predict the next token or sequence of tokens in a sentence in a causal or autoregressive manner, where the prediction for each token depends only on the preceding tokens. By using masking, the model only attends to the left context in a unidirectional

Statistic	Pre-training Dataset	Fine-tuning Dataset
Dataset size	89,814	1,127
Unique Intents	-	1,077
Unique Commands	79,410	1,121
Unique tokens (Intents)	-	2,273
Unique tokens (Commands)	85,342	17,463
Avg. tokens per Intent	-	15.97
Avg. tokens per Command	12.71	15.49

Table 2: Statistics of the pre-training and fine-tuning datasets. The pre-training dataset does not contain NL descriptions (intents).

tional manner, ensuring that it cannot see "into the future". In the probabilistic framework, starting from the text sequence $x = (x_1, x_2, x_3, \dots, x_T)$, where x is the original sentence and x_t ($t = 1, 2, \dots, T$) is the t -th token, and T is the sequence length, an autoregressive model factorizes the likelihood of the input text sequence as $p(x) = \prod_{t=1}^T p(x_t | x_{<t})$, where p is the likelihood of the input text sequence [38]. Finally, models are evaluated by token-level accuracy. For CodeT5+, the pre-training objective is *Masked Language Modeling (MLM)*, as recent works show its effectiveness in code understanding tasks [39]. MLM refers to the prediction of missing tokens in a sentence based on the context provided by the surrounding tokens. Unlike the left-to-right language model pre-training, MLM considers both the left and right context. The approach is inspired by BERT [40], where 15% of the tokens in the encoder inputs are randomly replaced with sentinel token [MASK], and the decoder is tasked with recovering these tokens to reconstruct the complete snippet. The model is evaluated by token level accuracy only on the masked-out tokens.

3.2 Fine-tuning data (labeled)

The overarching purpose of this dataset is to serve as a comprehensive resource for training models in the translation of NL intents, i.e., descriptions of code snippets, into executable security-oriented PowerShell commands. Specifically, we focus on offensive PowerShell code, a key resource for cybersecurity exercises since Microsoft Windows represents the most targeted OS. By encompassing a wide array of sources, the dataset aims to expose models to the intricacies of real-world cybersecurity scenarios, enabling them to understand and generate PowerShell commands that align with those typical of cybersecurity operations. This holistic approach strives to ensure that models trained on this dataset are well-equipped to handle the complexities of real-world tasks and contribute meaningfully to offensive code generation, specifically PowerShell commands.

The dataset, consisting of 1,127 samples of PowerShell commands, is meticulously curated from the following

sources:

- *Atomic Red Team* [41]: renowned for its library of tests mapped to the MITRE ATT&CK framework³ [42], serves the purpose of replicating real-world adversarial tactics, techniques, and procedures (TTPs). This inclusion provides the dataset with a foundation rooted in a standardized and widely accepted framework, ensuring that the PowerShell commands align with recognized cybersecurity methodologies.
- *Stockpile* [43]: is a plugin for the CALDERA cybersecurity framework [1, 44] developed by MITRE and introduces a layer of sophistication by incorporating structured data integral for adversary emulation. Therefore, the dataset does not encompass raw PowerShell commands only but also captures the contextual information and relationships between commands within the broader context of adversarial scenarios.
- *Empire* [45]: a post-exploitation and adversary emulation framework integrated with MITRE ATT&CK, provides PowerShell commands representative of advanced malicious techniques, further enriching the dataset with nuanced and intricate scenarios.
- *Online sources*: we manually verified and selected additional offensive samples from several security-related online sources. We gathered samples from *HackTricks* [46], *Red Team Recipe* [47], and *Infosec Matter* [48], community-driven cybersecurity wikis about ethical hacking, penetration testing, and information security. By including diverse examples specific to the offensive PowerShell dataset, the model acquires a more profound understanding of the conventions and best practices unique to PowerShell security commands.

We manually curated the dataset to cover the highest number of tactics in the MITRE ATT&CK framework. In particular, the dataset covers 12 out of 14 tactics from the MITRE ATT&CK framework, the *de facto* standard for adversarial techniques representation, with varying numbers of techniques and sub-techniques per tactic. Figure 2 illustrates the number of entries for each ATT&CK tactic. Each entry in the dataset is annotated with an NL description extracted from the respective source. We manually annotated every sample that did not come with a predefined description. Moreover, we enriched all those descriptions that did not provide enough information about the specific PowerShell command. For instance, in the case of Atomic Red Team, the PowerShell commands represent implementations of the techniques in the ATT&CK framework. Consequently, these commands are

³The ATT&CK framework is a comprehensive knowledge base of the tactics, techniques, and procedures (TTPs) that adversaries leverage during cyberattacks, developed by MITRE.

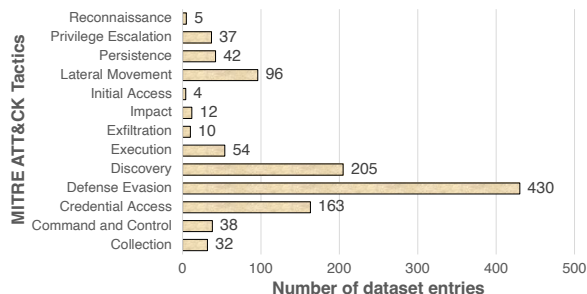


Figure 2: Mapping of fine-tuning dataset samples on the MITRE ATT&CK tactics.

often labeled with the technique name, which provides informative content about the technique itself rather than what the command does. To better understand how programmers and security experts describe PowerShell scripts and how to deal with ambiguities in natural language, we referred to popular books and manuals [49–51].

Finally, we notice that the size of our dataset is in line with other state-of-the-art corpora used to fine-tune ML models. In fact, in state-of-the-art code generation, the datasets for fine-tuning are relatively limited, in the order of one thousand samples [52].

3.3 Code Generation Task

To ensure the robustness of our study, we adopt the following state-of-the-art NMT models:

- **CodeT5+** [15] is a new family of Transformer models pre-trained with a diverse set of pretraining tasks to learn rich representations from both unimodal code data and bimodal code-text data. We utilize the variant with model size 220M, trained from scratch following T5’s architecture [53]. It has an encoder-decoder architecture with 12 decoder layers, each with 12 attention heads and hidden layer dimension of 768, and 512 for the size of position embeddings. We set the learning rate $\alpha = 0.00005$, batch size = 16, and beam size = 10.
- **CodeGPT** [16], a Transformer-based language model pre-trained on millions of Python functions and Java methods. The model architecture consists of 12 layers of Transformer decoders. We followed previous work for the implementation [54].
- **CodeGen** [17], an autoregressive language model for program synthesis with an architecture that follows a standard transformer decoder with left-to-right causal masking. The family of CodeGen models is trained in various sizes, including 350M, 2.7B, 6.1B, and 16.1B, and utilizes various datasets. Specifically, we leverage

CodeGen-Multi, initialized from CodeGen-NL and further pre-trained on BigQuery [17], a large-scale dataset of multiple programming languages from GitHub repositories, which consists of 119.2B tokens and includes C, C++, Go, Java, JavaScript, and Python.

In our experiments, we randomly split the fine-tuning dataset into training (the set of examples used to fit the parameters), validation (the set used to tune the hyperparameters of the models), and test (the set used for the evaluation of the models) sets using a typical 80%/10%/10% ratio.

To assess the performance of the models in generating offensive PowerShell code from NL descriptions, we used *output similarity metrics*, which compare the generated code with the code from the ground truth. This type of metrics is widely used to assess the performance of AI generators in many code generation tasks [55], including the generation of code for security contexts [28–31, 56]. The metrics are:

- **Bilingual Evaluation Understudy (BLEU) score** [57]. It measures the degree of n -gram overlapping between the string of each code snippet produced by the model and the reference, for values of n usually ranging between 1 and 4 [58, 59]. We implemented BLEU-4 score (i.e., with $n = 4$) computation employing the `bleu_score` module contained in the open-source Natural Language Toolkit (NLTK) Python suite [60].
- **Edit Distance (ED)**. It measures the *edit distance* between two strings, i.e., the minimum number of operations on single characters required to make each code snippet produced by the model equal to the reference. For the edit distance, we adopted the Python library `pylcs` [61].
- **METEOR** [62]. It measures the *alignment* between each code snippet produced by the model and the reference. The alignment is defined as a mapping between unigrams (i.e., 1-gram), such that every unigram in each string maps to zero or one unigram in the other string and no unigrams in the same string. To calculate the METEOR metric, we relied on the Python library `evaluate` by HuggingFace [63].
- **ROUGE-L**. It is a metric based on the longest common subsequence (LCS) between the model output and the reference, i.e., the longest sequence of words (not necessarily consecutive, but still in order) shared between both. We computed the ROUGE-L metric using the Python package `rouge` [64].

All metrics range between 0 and 1, with higher scores corresponding to a better quality of the generated code. To evaluate the generated PowerShell code, we also introduce additional evaluation metrics based on static and dynamic analysis that are specific to our context. These metrics will be introduced in the following sections.

3.4 Research Questions

We designed this research study to answer the following research questions (RQs):

▷ **RQ1:** *To what extent can NMT models effectively generate offensive PowerShell code for security applications from NL descriptions?*

RQ1 aims to establish a preliminary assessment of NMT models in generating PowerShell code for offensive security applications. This investigation seeks to shed light on the models' efficacy in translating NL descriptions into offensive code.

▷ **RQ2:** *What is the influence of the training strategies on NMT models' performance in offensive PowerShell code generation?*

RQ2 focuses on the impact of pre-training and fine-tuning on the quality of generated code. We analyze the influence of these training strategies by considering different configurations of the NMT models and their impact on their performance.

▷ **RQ3:** *How good is the generated code in terms of code quality and dynamic behavior?*

RQ3 aims to evaluate the generated PowerShell code in a deeper way than output similarity metrics, in terms of syntactic correctness and capability of executing malicious actions realistically, through behavioral comparison with the ground truth.

▷ **RQ4:** *How do fine-tuned NMT models, leveraging security-oriented training data, compared to a publicly available, closed-source model?*

RQ4 introduces a comparative analysis, evaluating the performance of the fine-tuned models against a publicly available general-purpose language model, specifically ChatGPT 3.5. This investigation strives to evaluate whether specialization on security-focused data provides an advantage in the offensive PowerShell code generation domain.

4 Experimental Results

This section presents an extensive evaluation of NMT models (CodeT5+, CodeGPT, and CodeGen) on the generation of offensive PowerShell code. First, we assess the models' capability of generating PowerShell code in their original configuration (§ 4.1) without further training. Then, we evaluate the impact of different training strategies, i.e., domain-adaptive pre-training and fine-tuning, on the performance of such models (§ 4.2). To provide further insight into the PowerShell code generation, we analyze the quality of the generated code in terms of syntactic correctness (§ 4.3) and dynamic behavior (§ 4.4), i.e., its ability to replicate the behavior of the ground truth code. Finally, we compare the fine-tuned models with a public AI model (ChatGPT) for all the previous analyses (§ 4.5) to benchmark their performance against a publicly available, closed-source model.

Model	Pre-training	BLEU-4 (%)	ED (%)	METEOR (%)	ROUGE-L (%)
CodeT5+	✗	0.04	8.87	4.69	1.08
	✓	0.01	6.96	1.86	2.68
CodeGPT	✗	0.23	12.31	4.08	1.19
	✓	0.28	15.67	2.55	3.41
CodeGen	✗	0.06	7.58	2.88	0.21
	✓	0.00	0.43	0.09	0.00

Table 3: Performance of models with and without pre-training on zero-shot.

4.1 Zero-shot Learning

To establish a baseline for the evaluation, we initially used the NMT models in their original configuration, asking them to generate PowerShell code. This is a *zero-shot learning task*, where an NMT model is applied for a different scenario than the one for which it was trained. In this way, we evaluate the current gap of existing models in generating PowerShell code. Table 3 shows the results of this analysis. In this task, the models are tested without any gradient updates, relying only on the intent provided by the test set for inference [36,37]. The non-pre-trained versions of the models tend to generate Python code, but their performance is generally low for the downstream task of generating offensive PowerShell code. Pre-training the models with general-purpose PowerShell code slightly improves the accuracy but is still not high. Among the pre-trained versions, CodeGPT is the only one that provides output close to valid PowerShell code, although it does not align well with the expected code indicated by the intent in natural language. In summary, regardless of pre-training, all models demonstrate the need for fine-tuning on a tailored dataset for optimal performance in generating offensive PowerShell code.

4.2 Impact of Training Strategies

The evaluation of CodeT5+, CodeGPT, and CodeGen involved a meticulously designed test plan. More precisely, the models underwent three distinct fine-tuning scenarios: 3 Epochs, 10 Epochs, and 30 Epochs. This deliberate choice allowed us to assess the impact of prolonged fine-tuning on the models' ability to generate PowerShell code for offensive security tasks. In each scenario, we considered two training configurations: one with pre-training and the other without. This test plan allowed us to systematically explore the models' capabilities under varying conditions, providing a comprehensive understanding of their strengths and limitations. Table 4 shows the results.

In the 3 epochs setting, CodeT5+ exhibits low performance, regardless of pre-training, with a BLEU-4 score lower than 10%. In contrast, CodeGPT and CodeGen demonstrate notable performance even after a short fine-tuning period,

Model	Epochs	Pre-train. (%)	BLEU-4 (%)	ED (%)	METEOR (%)	ROUGE-L (%)
CodeT5+	3	X	4.22	35.11	28.83	22.26
		✓	4.57	35.96	30.57	23.99
	10	X	12.64	46.72	44.76	37.65
		✓	11.88	49.10	46.11	37.17
	30	X	17.40	50.92	47.61	39.05
		✓	18.50	50.23	47.87	38.86
CodeGPT	3	X	10.28	40.71	31.21	25.60
		✓	12.80	42.54	35.14	30.35
	10	X	16.22	46.39	40.50	33.52
		✓	17.93	49.88	45.12	37.12
	30	X	21.71	50.17	45.34	38.63
		✓	19.94	49.20	45.45	38.06
CodeGen	3	X	16.20	47.68	42.27	35.97
		✓	14.75	45.88	39.86	34.69
	10	X	19.15	50.52	46.76	37.63
		✓	19.04	48.45	43.25	35.25
	30	X	18.23	47.53	44.10	35.48
		✓	18.53	48.67	44.14	35.45

Table 4: Performance of models with and without pre-training and different number of epochs. Best results for each metric are **blue/bold**.

achieving a BLEU-4 score higher than 10% and an ED over 40%. Notably, after 3 epochs, CodeGen demonstrates superior performance compared to the other two models. In the 10 epochs experiment, CodeT5+ shows significant improvement, with BLEU-4 tripling to 12%. Moreover, ED, METEOR, and ROUGE-L experience a rise of 12-16%. CodeGPT also enhances its performance, surpassing CodeT5+ in terms of BLEU-4 score, although it faces challenges in achieving the same level of overall improvement. CodeGen remains ahead of the other models, even reaching an ED over 50%. For a more in-depth assessment of the models' adaptability, the training duration is extended to 30 epochs. CodeT5+ demonstrates superior performance over CodeGPT in ED, METEOR, and ROUGE-L metrics, while CodeGPT exhibits a higher BLEU-4 score surpassing 20%. Notably, both models achieve a high ED value of around 50%. CodeGen establishes its performance without further improvement compared to the 10 epochs versions.

To provide an estimate of the goodness of the results, we compared the results of the models with the performance of the state-of-the-art (SOTA). Since the task of generating PowerShell using NMT models is a task never addressed before, we compared the results with recent work investigating the effectiveness of existing models in the generation of different languages from NL, specifically, Python code [65] and in shell language [66]. We found that the best performance is 21% for BLEU-4 and 38% for METEOR in the case of the

Python language, and 25% for BLEU-4 and 44% for ED in the case of shell language. We notice that our results are in line with the ones of the SOTA. Even better, our best performance, represented by CodeT5+ without pre-training and 30 fine-tuning epochs, overcomes the SOTA over all the metrics.

We also assessed the impact of varying the number of epochs on fine-tuning time, with distinct differences observed between 3, 10, and 30 epochs for each model. For both CodeT5+ and CodeGPT, fine-tuning over 3 epochs takes approximately 20 minutes, whereas CodeGen requires double that time (40 minutes). Extending to 10 epochs, CodeT5+ and CodeGPT need around 35 and 39 minutes, respectively, while CodeGen's training time increases to 90 minutes. For the 30-epoch extension, CodeT5+ takes about 80 minutes, CodeGPT requires 110 minutes, and CodeGen extends its training time to 270 minutes. Finally, the comparison between the fine-tuning times of pre-trained and non-pre-trained models did not reveal evident differences, suggesting that the pre-training process does not introduce a significant computational overhead during the subsequent fine-tuning phase.

RQ1: To what extent can state-of-the-art NMT models effectively generate offensive PowerShell code for security applications from NL descriptions?

The evaluation of CodeT5+, CodeGPT, and CodeGen underscores their remarkable effectiveness in generating offensive PowerShell code for security applications from NL descriptions. CodeGen surpasses the other models in the 3 and 10 epochs settings according to all metrics. CodeT5+, designed with a specialized architecture for code generation tasks, consistently outperforms CodeGPT and CodeGen across various metrics in the 30 epochs setting. Particularly noteworthy is the comparison with SOTA performance in code generation tasks for different languages, such as Python and shell languages. Our best-performing model, CodeT5+ without pre-training and 30 fine-tuning epochs, surpasses the SOTA results, exhibiting superior performance across all metrics.

Considering the impact of pre-training further enriched our evaluation. Focusing on the 3-epoch experiments, CodeT5+ exhibits a slight improvement across all metrics, and CodeGPT extends the improvement to 2%-4% across all metrics. Conversely, CodeGen appears to have better performance without pre-training. Training the models for 10 epochs reveals a more pronounced distinction between the two versions. CodeT5+ pre-training results in a 2% increase in both Edit Distance (ED) and METEOR metrics. CodeGPT, on the other hand, shows a substantial displacement of 1.7%, 3.5%, 4.6%, and 3.6% for BLEU-4, ED, METEOR, and ROUGE-L, respectively. CodeGen maintains a negative displacement between the versions even with the extended training duration. When extending the fine-tuning duration to 30 epochs, pre-training

Natural Language Intent	Ground Truth	Model Output
Start a process with <code>DirLister</code> , wait for 4 seconds, and stop the <code>DirLister</code> process.	<code>Start-Process \${WebBrowserPassViewPath} ; Start-Sleep -Second 4 ; Stop-Process -Name "WebBrowserPassView"</code>	<code>Start-Process \${WebBrowserPassViewPath} ; Start-Sleep -Second 4 ; Stop-Process -Name "WebBrowserPassView"</code>
Invoke <code>Mimikatz</code> cmdlet with bypassed execution policy.	<code>powershell.exe -exec bypass -c " Invoke-Mimikatz "</code>	<code>powershell.exe -ExecutionPolicy Bypass -Command " Invoke-Mimikatz "</code>
Invoke compiled HTML Help (CHM) file with specific storage handler, HH file path, and CHM file path.	<code>Invoke-ATHCompiledHelp -InfoTechStorageHandler \$ { infotech_storage_handler } -HHFilePath \$ { hh_file_path } -CHMFilePath \$ { chm_file_path }</code>	<code>Invoke-ATHCompiledHTMLHelp -InfoTechStorageHandler \$ { infotech_storage_handler } -HHFilePath \$ { hh_file_path } -CHMFilePath \$ { chm_file_path }</code>
Get the <code>wininit</code> process and perform token manipulation to create a new process for <code>'cmd.exe'</code> .	<code>Get-Process wininit Invoke-TokenManipulation -CreateProcess 'cmd.exe'</code>	<code>\$wininit = Get-Process wininit Invoke-TokenManipulation -CreateProcess 'cmd.exe'</code>

Table 5: Illustrative examples of model output. The prediction errors are **red/bold**. ~~Slashed~~ text refers to omitted predictions.

did not consistently yield superior results. In this case, the performance of pre-trained models is comparable to non-pre-trained counterparts.

RQ2: What is the influence of the training strategies on NMT models' performance in offensive PowerShell code generation?

As the fine-tuning period extends, such as with 10 and 30 epochs, the benefits of pre-training diminish or even become counterproductive. In these cases, the performance of pre-trained models consistently falls below that of their non-pre-trained counterparts. This highlights the variable effectiveness of pre-training, dependent on the duration of fine-tuning. These findings underscore the interplay between the duration of training epochs and the usage of pre-training, emphasizing the importance of carefully considering these factors in model development.

Table 5 illustrates four cases of model predictions. They are examples from our test sets to highlight both successful and failed prediction cases. Row # 1 demonstrates the models' ability to generate a PowerShell snippet composed of multiple commands (separated by semicolons) without errors. The model correctly predicts the correct variables, e.g., `WebBrowserPassViewPath`, and command names, such as `Start-Process`, `Start-Sleep`. Row # 2 is indicative of the concept of implicit model knowledge. Indeed, the model can generate a correct command by leveraging alternative equivalent versions of PowerShell's option flags (e.g., `-ExecutionPolicy` instead of `-exec`). Row # 3 shows a relevant example of a failure case. It is possible to notice how the model correctly predicts the variable names and values except for one not referenced in the intent

(`-InfoTechStorageHandler`). In addition, the model fails to predict the correct command name, generating an additional word (HTML) based on the NL description. Finally, row # 4 illustrates another incorrect example in which the model is capable of generating the ground truth code, except for introducing an additional variable to save the output of the command (`$wininit =`).

Overall, we can conclude that these examples indicate the model's ability to generate complex PowerShell snippets, even though there is still some error margin, specifically related to omissions (e.g., variable names).

4.3 Static Analysis

We evaluated the generated code through *static analysis* to ensure that the code adheres to PowerShell conventions and does not contain syntax errors. The analysis was conducted on the top-performing models identified in the previous evaluation, namely the 30-epoch versions of CodeT5+ with pre-training, CodeGPT without pre-training, and CodeGen with pre-training. The static analysis leverages *PSScriptAnalyzer* [67], a static code checker for PowerShell modules and scripts. The primary purpose of *PSScriptAnalyzer* is to assess the quality of PowerShell code by analyzing its syntax, structure, and adherence to best practices. The rules are based on PowerShell best practices identified by the PowerShell Team and the community, organized into categories such as Cmdlet Design, Script Functions, Error Handling, Scripting Style, and Script Security. The severity levels (ParseError, Error, Warning, Information) associated with each rule indicate the importance and impact of adhering to the specific guideline. In this work, we focused on *parse errors*, which occur during the parsing phase of a program's execution, *errors*, occurring when code

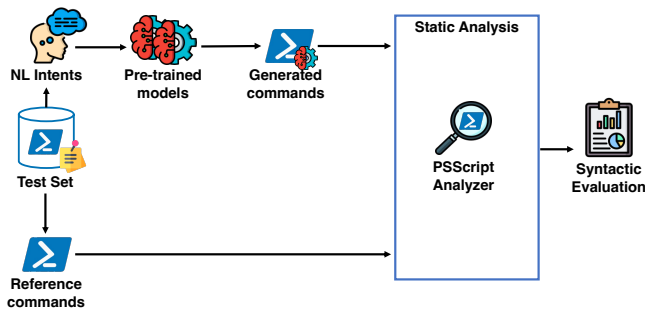


Figure 3: Static analysis workflow.

Model	Single Accuracy (%)	Comparative Accuracy (%)
CodeT5+	91.15	92.04
CodeGPT	98.23	98.23
CodeGen	98.23	98.23

Table 6: Syntactic evaluation for the best models.

does not meet specific high-severity rules (e.g., hardcoding computer names, using plain text passwords), and *warnings*, which typically highlight potential issues or coding practices that might lead to errors or security concerns.

We developed a syntactic analysis tool to streamline the process of detecting *parse errors*, *errors*, and *warnings* in PowerShell scripts. This tool automatically feeds PSScriptAnalyzer with PowerShell commands generated by the models during the testing phase. By doing so, our tool identifies errors and warnings in the generated code, assessing the overall syntactic quality of the models.

The syntactic analysis process begins with our test set, which consists of NL intents paired with reference PowerShell commands. These NL intents are fed into fine-tuned models to produce the PowerShell code. Both the generated commands and their corresponding references are then subjected to the syntax analyzer.

To assess the syntactic quality of the generated commands, we introduce two distinct metrics: *Single Syntax Accuracy* and *Comparative Syntax Accuracy*. The metrics are defined as follows:

- **Single Syntax Accuracy:** evaluates the percentage of commands without parse errors. This evaluation is independent of the reference commands from the ground truth.
- **Comparative Syntax Accuracy:** assesses the syntactic correctness of the generated commands by considering the results alongside the reference commands. When both commands present common parse errors, these are excluded from the counting process. Given that some reference commands include stub templates such as `<code>`

Test Set	ParseError (%)	Error (%)	Warning (%)
CodeT5+	8.85	1.94	35.92
CodeGPT	1.77	2.70	29.73
CodeGen	1.77	1.80	31.53
Ground Truth	2.65	0.00	39.09

Table 7: Summary of ParseError, Error, and Warning percentages for models and ground truth on the test set.



Figure 4: Counts for different warning types in each test set.

or `<command>`, the analysis filters out parse errors associated with these templates, specifically the *Redirection-NotSupported* and *MissingFileSpecification* errors.

The workflow for the syntactic analysis is depicted in Figure 3. Looking at the results in Table 6, it is possible to notice that all the models achieved a score greater than 90%, assessing their strong capability to generate syntactically correct code. CodeGPT and CodeGen, in general, demonstrate high performance across both syntax metrics. Table 7 summarizes the percentages for various severity types in the test set. Given that warning frequencies are consistently above 30% for all models, including the ground truth, Figure 4 enumerates the various warning types within each set.

4.4 Execution Analysis

The execution analysis aims to evaluate the generated offensive PowerShell code when running in an actual system. This involves assessing the ability of the code to behave as intended in terms of effects caused on the system. Therefore, we run both code from the ground truth and generated code, monitor their behavior at runtime, and compare the behavioral events

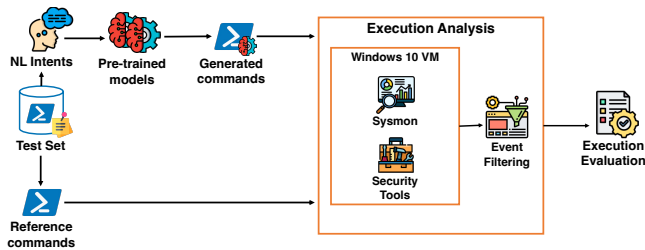


Figure 5: Execution analysis workflow.

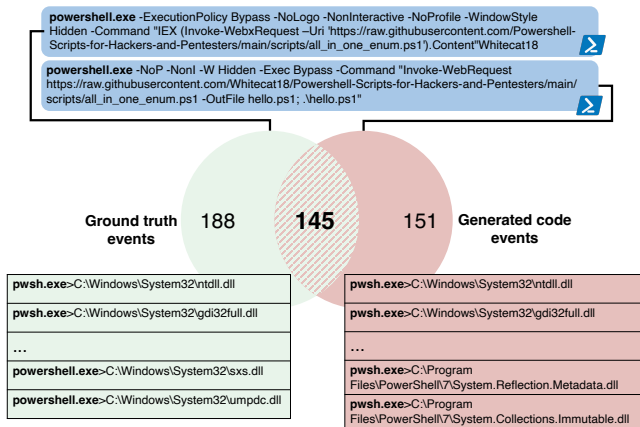


Figure 6: Comparison between events.

that occurred during their execution. The entire workflow for the execution analysis is shown in Figure 5.

We performed the experiments in a controlled and dedicated testing environment. The controlled environment consists of a virtualized Windows 10 system running in Virtual-Box 7. The system is equipped with a set of security-related tools, such as PowerSploit [68] and Mimikatz [69], that are invoked by many samples of offensive code in our dataset. We assume that these tools have been previously infiltrated by the attacker in a previous stage, as typical of advanced malicious campaigns. To monitor the execution of PowerShell code, we integrated Sysmon [70], a popular Windows service for gathering system events, including the filesystem, the network, and the Windows Registry. To be able to run the generated code on the system, we assume the scenario in which an attacker already bypassed part of the security mechanisms by deactivating the Microsoft Defender Firewall, Windows Defender, and Microsoft Defender SmartScreen.

The evaluation involved executing each command from both the generated ones and those from ground truth multiple times as a single-line PowerShell script. This generates a process through the standard Windows `System.Diagnostics.Process`. We filter the events recorded by Sysmon by filtering out records related to previous irrelevant

Model	Precision (%)	Recall (%)	F1-Score (%)
CodeT5+	97.26	80.94	88.35
CodeGPT	91.86	85.23	88.42
CodeGen	96.94	80.97	88.24

Table 8: Execution analysis results.

events and selecting records based on the Process ID (PID), focusing on both the parent process responsible for executing the PowerShell command and its child processes. The comparison has been performed comparing the events triggered by the generated command (called *retrieved records*) to those from the execution profile of the ground truth (called *relevant records*). The events that appear both when executing the generated code and the ground truth are *relevant records retrieved*. From these sets of events, we evaluate the *precision*, *recall* and *F1-score* of the generated code, defined as follows:

$$\text{precision} = \frac{1}{N} \sum_i \frac{\#(\text{relevant records retrieved})_i}{\#(\text{retrieved records})_i}$$

$$\text{recall} = \frac{1}{N} \sum_i \frac{\#(\text{relevant records retrieved})_i}{\#(\text{relevant records})_i}$$

$$\text{F1-Score} = 2 \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Figure 6 illustrates an example of event analysis: given the ground truth and the generated PowerShell command, we execute them and compare the set of events triggered by each command to measure their overlap. To avoid noise in the analysis due to events that only occur sporadically (e.g., because of non-determinism sources in the system), we identify such events by performing multiple repeated runs of the code and discard non-reproducible events from the analysis. After every command execution, the Windows environment is restored to a clean state, by reloading the virtual machine from a snapshot, to avoid interferences caused by the effect of previous commands.

The results shown in Table 8 outline how all models share an overall precision higher than 90% and an overall recall higher than 80%, likewise, the Execution F1-Score is very similar between the different models and higher than 88%. Thus, although there were differences found in the textual similarity analysis, the generated code closely matches the ground truth in terms of dynamic events.

RQ3: How good is the generated code in terms of code quality and dynamic behavior?

The syntactic analysis of the generated code showed that the models are indeed capable of generating high-quality

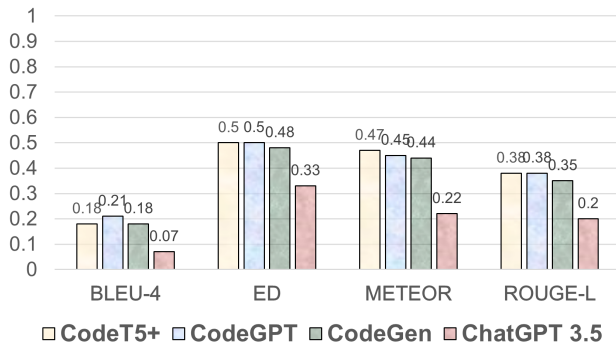


Figure 7: Comparison with ChatGPT on output similarity metrics.

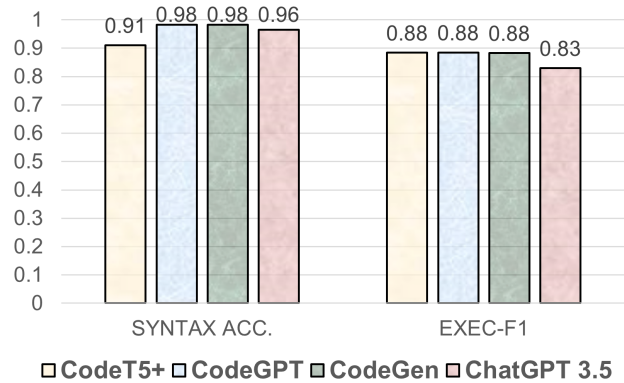


Figure 8: Comparison with ChatGPT on static and execution analysis evaluation metrics.

PowerShell code. CodeGPT and CodeGen achieve the best results in terms of Single and Comparative Accuracy, along with an amount of Warnings and ParseErrors comparable to the ground truth. The execution analysis revealed that the generated PowerShell code closely replicates the behavior of the ground truth code, generating the same events in the target system. This is indicative of the generated code’s capability of performing the malicious actions described in the NL intents.

4.5 Comparison with Public AI Model

In this study, we conducted a comprehensive evaluation by comparing the performance of our fine-tuned models, CodeT5+, CodeGPT, and CodeGen, with ChatGPT, the OpenAI LLM service widely used for a variety of tasks, including code generation [71]. The purpose was to assess the specialized capabilities of our models in generating PowerShell code for offensive security tasks and to benchmark their performance against a publicly available, closed-source model. We leveraged ChatGPT 3.5, which represents the most recent free version at the time of this work.

To assess the capabilities of the OpenAI model, we first provided a detailed description of the required task, i.e., the generation of PowerShell commands starting from NL descriptions, including an example of input and the desired output. Then, we provided a list of natural language code descriptions and asked ChatGPT to automatically generate the corresponding PowerShell code. Specifically, following works and guidelines on prompt engineering [71, 72], we leveraged the following prompt: I want you to act as a code generator. Given a natural language description of a PowerShell command, generate the corresponding PowerShell code.

Figure 7 shows the results of this analysis. The figure shows that our fine-tuned models consistently outperform ChatGPT

across multiple evaluation metrics. Specifically, ChatGPT exhibits a BLEU-4 score of 7.45%, an ED of 33.84%, a METEOR of 22.14%, and a ROUGE-L of 20.61%. In contrast, our fine-tuned models showcase superior overall performance across all output similarity metrics. The tailored training on the specialized fine-tuning dataset, designed specifically for offensive security code generation, results in more accurate code generation, enabling our models to surpass the capabilities of ChatGPT in this particular task. We also analyzed the syntactical quality of the PowerShell code generated by ChatGPT, obtaining a Syntax Single Accuracy of 95.58% and a Syntax Comparative Accuracy of 96.46%. These results underscore the commendable ability of ChatGPT to generate accurate and syntactically correct PowerShell code.

Finally, we extended the execution analysis to ChatGPT, following the same strategies described in Section 4.4, obtaining an overall Execution F1-Score of 82.92%. Despite the strong syntactic performance, ChatGPT remains one step below the fine-tuned models in the qualitative analysis of the generated PowerShell code. The results of this analysis are shown in Figure 8.

RQ4: How do fine-tuned NMT models, leveraging security-oriented training data, compare to a publicly available, closed-source model?

The comparative analysis with ChatGPT, a publicly available general-purpose language model, highlights the specialized strengths of privately fine-tuned models, CodeT5+, CodeGPT, and CodeGen, in offensive PowerShell code generation. The fine-tuned models consistently outperform ChatGPT across BLEU-4, Edit Distance, and METEOR scores. While showing notable performance on syntactic accuracy, ChatGPT achieves poorer results than the fine-tuned models for the execution analysis. This underscores the significance of

domain-specific fine-tuning and the benefits of training on security-oriented datasets, providing an advantage in generating offensive PowerShell code compared to a general-purpose language model. The results affirm the effectiveness of tailored training data for achieving superior performance in domain-specific tasks.

5 Threats To Validity

Model selection. The external validity of the study might be impacted by the choice of NMT models (CodeT5+, CodeGPT, CodeGen). To mitigate this, we carefully selected models with distinct architectures and capabilities, ensuring a representation of current advancements in the field [16,73,74]. This careful selection aims to ensure that our findings reflect broader trends in NMT model performance for code generation tasks.

Evaluation metrics. The reliance on output similarity metrics, although representing the most common solution in the field, poses a potential threat to construct validity, as these metrics may not fully encapsulate the correctness and functional adequacy of the generated PowerShell commands. To address this issue, our evaluation strategy encompasses a comprehensive suite of metrics, including similarity, syntactic, and execution metrics, each offering unique insights into the models' performance. By considering multiple variants of these metrics and aligning with common practices in code generation evaluation, we aim to provide a well-rounded assessment. No single metric is perfect, but analyzing them collectively allows for a more comprehensive evaluation of the code.

Fine-tuning data. The construction of our dataset, meticulously curated from several sources such as online repositories, Atomic Red Team, Stockpile, and Empire, introduces potential limitations regarding the generalizability of our models' performance across different offensive security contexts. To minimize the impact of these limitations, we sourced data from diverse origins and conducted manual verification of each sample in the labeled dataset, ensuring the completeness and coherence of descriptions with the intended programs. The diversity in data sources and the thorough verification process aim to diminish the influence of any singular source's peculiarities and errors in programs or descriptions, thereby enhancing the dataset's applicability and reliability for training and evaluating AI models in generating offensive PowerShell code. Furthermore, our approach to crafting NL descriptions, inspired by established styles found in PowerShell literature, mirrors real-world scenarios where such descriptions play a critical role in describing PowerShell commands. Finally, regarding the size of our dataset, we notice that it is in line with other state-of-the-art corpora used to fine-tune models, which are in the order of one thousand samples [52].

6 Ethical Considerations

Recognizing that attackers use attacks as a weapon, it is important to specify that the goal of the proof-of-concept (POC) is not to cause harm but to surface security weaknesses within the software. Identifying security issues allows companies to patch vulnerabilities and protect themselves against attacks.

Offensive security is a sub-field of security research that tests security measures from an adversary or competitor's perspective, employing ethical hackers to probe a system for vulnerabilities [75,76]. Our work aims to automate attack generation to explore critical vulnerabilities before they are exploited by attackers [77]. Indeed, our work simplifies the process of coding the attacks to surface security weaknesses within the software and can provide valuable information about the technical skills, degree of experience, and intent of the attackers. With this information, it is possible to implement measures to detect and prevent attacks [78].

7 Conclusion

In this paper, we assessed the feasibility of using NMT models to generate PowerShell code for security contexts. We aimed to demonstrate that AI-based code generators are indeed fit to generate PowerShell code, specifically, offensive PowerShell, which spans several applications in the cybersecurity domain. The evaluation of CodeT5+, CodeGPT, and CodeGen demonstrated that these models achieve significant performance on the code generation task, both with and without pre-training. Moreover, the study showed that domain-specific fine-tuning allows our models to outperform state-of-the-art privately fine-tuned models, i.e., ChatGPT. We also introduced two novel datasets for PowerShell code generation to use for pre-training and fine-tuning AI-code generators.

Future work includes further analysis of the generated code, such as sandbox execution of the offensive scripts, to understand whether the code can evade detection measures, along with more NMT models spanning several architectures and capabilities.

Acknowledgments

This work has been partially supported by MUR PRIN 2022, project FLEGREA, CUP E53D23007950001 (<https://flegrea.github.io>) and by an Industrial Ph.D. grant (PNRR - DM 117/2023) from MUR and DigitalPlatforms S.p.A, CUP E66E23000580003.

References

- [1] A. Applebaum, D. Miller, B. E. Strom, C. Korban, and R. Wolf, "Intelligent, automated red team emulation,"

Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016.

- [2] A. B. Ajmal, M. A. Shah, C. Maple, M. N. Asghar, and S. U. Islam, “Offensive security: Towards proactive threat hunting via adversary emulation,” *IEEE Access*, vol. 9, pp. 126 023–126 033, 2021.
- [3] E. M. Hutchins, M. J. Cloppert, R. M. Amin *et al.*, “Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains,” *Leading Issues in Information Warfare & Security Research*, vol. 1, no. 1, p. 80, 2011.
- [4] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas, “Mitre att&ck: Design and philosophy,” in *Technical report*. The MITRE Corporation, 2018.
- [5] Sudhakar and S. Kumar, “An emerging threat fileless malware: a survey and research challenges,” *Cybersecurity*, vol. 3, no. 1, p. 1, 2020.
- [6] I. Kara, “Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges,” *Expert Systems with Applications*, vol. 214, p. 119133, 2023.
- [7] Varonis, “What is Fileless Malware? PowerShell Exploited,” <https://www.varonis.com/blog/fileless-malware>.
- [8] Cybersecurity & Infrastructure Security Agency, “Identifying and Mitigating Living Off the Land Techniques,” https://www.cisa.gov/sites/default/files/2024-02/Join t-Guidance-Identifying-and-Mitigating-LOTL_V350 8c.pdf.
- [9] R. Natella, P. Liguori, C. Improta, B. Cukic, and D. Cotroneo, “Ai code generators for security: Friend or foe?” *IEEE Security & Privacy*, 2024.
- [10] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, “Large language models for software engineering: Survey and open problems,” *arXiv preprint arXiv:2310.03533*, 2023.
- [11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating Large Language Models Trained on Code,” *arXiv preprint arXiv: 2107.03374*, 2021.
- [12] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, “Codereval: A benchmark of pragmatic code generation with generative pre-trained models,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.
- [13] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, “ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation,” *arXiv preprint arXiv: 2308.01861*, 2023.
- [14] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, “Shellcode_IA32: A dataset for automatic shellcode generation,” in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, R. Lachmy, Z. Yao, G. Durrett, M. Gligoric, J. J. Li, R. Mooney, G. Neubig, Y. Su, H. Sun, and R. Tsarfaty, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 58–64. [Online]. Available: <https://aclanthology.org/2021.nlp4prog-1.7>
- [15] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, “Codet5+: Open code large language models for code understanding and generation,” *arXiv preprint arXiv:2305.07922*, 2023.
- [16] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html>
- [17] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “Codegen: An open large language model for code with multi-turn program synthesis,” 2023.
- [18] OpenAI, “ChatGPT,” <https://openai.com/chatgpt>.
- [19] Z. Li, Q. A. Chen, C. Xiong, Y. Chen, T. Zhu, and H. Yang, “Effective and light-weight deobfuscation and

- semantic-aware attack detection for powershell scripts,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1831–1847.
- [20] M.-H. Tsai, C.-C. Lin, Z.-G. He, W.-C. Yang, and C.-L. Lei, “Powerdp: De-obfuscating and profiling malicious powershell commands with multi-label classifiers,” *IEEE Access*, vol. 11, pp. 256–270, 2023.
- [21] D. Hendler, S. Kels, and A. Rubin, “Detecting malicious powershell commands using deep neural networks,” in *Proceedings of the 2018 on Asia conference on computer and communications security*, 2018, pp. 187–197.
- [22] A. Rubin, S. Kels, and D. Hendler, “Amsi-based detection of malicious powershell code using contextual embeddings,” *arXiv preprint arXiv:1905.09538*, 2019.
- [23] M. Mimura and Y. Tajiri, “Static detection of malicious powershell based on word embeddings,” *Internet of Things*, vol. 15, p. 100404, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2542660521000482>
- [24] Y. Mezawa and M. Mimura, “Evaluating the possibility of evasion attacks to machine learning-based models for malicious powershell detection,” in *International Conference on Information Security Practice and Experience*. Springer, 2022, pp. 252–267.
- [25] G. Rusak, A. Al-Dujaili, and U.-M. O’Reilly, “Ast-based deep learning for detecting malicious powershell,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2276–2278.
- [26] M. Gupta, C. Akiri, K. Aryal, E. Parker, and L. Praharaj, “From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy,” *IEEE Access*, vol. 11, pp. 80 218–80 245, 2023.
- [27] P. Charan, H. Chunduri, P. M. Anand, and S. K. Shukla, “From text to mitre techniques: Exploring the malicious use of large language models for generating cyber attack payloads,” *arXiv preprint arXiv:2305.15336*, 2023.
- [28] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, “Can we generate shellcodes via natural language? an empirical study,” *Automated Software Engineering*, vol. 29, no. 1, pp. 1–34, 2022.
- [29] P. Liguori, E. Al-Hossami, V. Orbinato, R. Natella, S. Shaikh, D. Cotroneo, and B. Cukic, “Evil: exploiting software via natural language,” in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 321–332.
- [30] G. Yang, X. Chen, Y. Zhou, and C. Yu, “Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning,” in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 361–372.
- [31] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, “Exploitgen: Template-augmented exploit code generation based on codebert,” *Journal of Systems and Software*, vol. 197, p. 111577, 2023.
- [32] A. M. Dai and Q. V. Le, “Semi-supervised sequence learning,” *Advances in neural information processing systems*, vol. 28, 2015.
- [33] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, “Don’t stop pretraining: Adapt language models to domains and tasks,” *arXiv preprint arXiv:2004.10964*, 2020.
- [34] T. Lin, Y. Wang, X. Liu, and X. Qiu, “A survey of transformers,” *AI Open*, 2022.
- [35] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, “Improving language understanding by generative pre-training,” 2018.
- [36] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [37] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [38] H. Wang, J. Li, H. Wu, E. Hovy, and Y. Sun, “Pre-trained language models and their applications,” *Engineering*, 2022.
- [39] R. Tufano, L. Pascarella, and G. Bavota, “Automating code-related tasks through transformers: The impact of pre-training,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2425–2437.
- [40] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [41] Red Canary, “Atomic Red Team,” <https://atomicredteam.io/>.
- [42] M. Corporation, “MITRE ATT&CK,” <https://attack.mitre.org/>.

- [43] MITRE, “CALDERA plugin: Stockpile,” <https://github.com/mitre/stockpile>.
- [44] MITRE, “CALDERA,” <https://github.com/mitre/caldera>.
- [45] Empire Project, “Empire,” <https://github.com/EmpireProject/Empire>.
- [46] Hacktricks, “Hacktricks,” <https://book.hacktricks.xyz/>.
- [47] R. T. Recipe, “PowerShell tips & tricks,” <https://redteamrecipe.com/powershell-tips-tricks/>.
- [48] I. Matter, “PowerShell commands for pentesters,” <https://www.infosecmatter.com/powershell-commands-for-pentesters/>.
- [49] Tutorial’s Point, “Learn PowerShell,” <https://www.tutorialspoint.com/powershell/index.htm>.
- [50] T. Lee, K. Mitschke, M. E. Schill, and T. Tanasovski, *Windows PowerShell 2.0 Bible*. John Wiley & Sons, 2011.
- [51] L. Holmes, *Windows PowerShell Cookbook: The Complete Guide to Scripting Microsoft’s Command Shell*. O’Reilly Media, 2012.
- [52] C. Zhou, P. Liu, P. Xu, S. Iyer, J. Sun, Y. Mao, X. Ma, A. Efrat, P. Yu, L. Yu, S. Zhang, G. Ghosh, M. Lewis, L. Zettlemoyer, and O. Levy, “LIMA: less is more for alignment,” *CoRR*, vol. abs/2305.11206, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2305.11206>
- [53] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020. [Online]. Available: <http://jmlr.org/papers/v21/20-074.html>
- [54] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html>
- [55] P. Liguori, C. Improta, R. Natella, B. Cukic, and D. Cotroneo, “Who evaluates the evaluators? on automatic metrics for assessing ai-based offensive code generators,” *Expert Systems with Applications*, vol. 225, p. 120073, 2023. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0957417423005754>
- [56] X. Ruan, Y. Yu, W. Ma, and B. Cai, “Prompt learning for developing software exploits,” in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, 2023, pp. 154–164.
- [57] K. Papineni, S. Roukos, T. Ward, and W. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318. [Online]. Available: <https://aclanthology.org/P02-1040/>
- [58] L. Han, “Machine translation evaluation resources and methods: A survey,” *arXiv preprint arXiv:1605.04515*, 2016.
- [59] D. Munkova, P. Hajek, M. Munk, and J. Skalka, “Evaluation of machine translation quality through the metrics of error rate and accuracy,” *Procedia Computer Science*, vol. 171, pp. 1327–1336, 2020.
- [60] NLTK, “Natural Language Toolkit (NLTK), bleu_score module,” 2023. [Online]. Available: https://www.nltk.org/api/nltk.translate.bleu_score.html
- [61] pylcs, “Python library pylcs,” 2023. [Online]. Available: <https://pypi.org/project/pylcs/>
- [62] A. Lavie and A. Agarwal, “Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments,” in *Proceedings of the Second Workshop on Statistical Machine Translation*, ser. StatMT ’07. USA: Association for Computational Linguistics, 2007, p. 228–231.
- [63] evaluate, “Python library evaluate,” 2022. [Online]. Available: <https://pypi.org/project/evaluate/>
- [64] rouge, “Python ROUGE Score Implementation,” 2021. [Online]. Available: <https://pypi.org/project/rouge/>
- [65] J. Shin, M. Wei, J. Wang, L. Shi, and S. Wang, “The good, the bad, and the missing: Neural code generation for machine learning tasks,” *arXiv preprint arXiv:2305.09082*, 2023.
- [66] J. Shi, S. Jiang, B. Xu, J. Liang, Y. Xiao, and W. Wang, “Shellgpt: Generative pre-trained transformer model for shell language understanding,” in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 671–682.
- [67] Microsoft, “PSScriptAnalyzer,” <https://github.com/PowerShell/PSScriptAnalyzer>.

- [68] Will Schroeder, “PowerSploit,” <https://github.com/PowerShellMafia/PowerSploit>.
- [69] Benjamin Delpy, “Mimikatz,” <https://github.com/gentikiwi/mimikatz>.
- [70] Mark Russinovich, Thomas Garnier, “System Monitor,” <https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon>.
- [71] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration code generation via chatgpt,” *arXiv preprint arXiv:2304.07590*, 2023.
- [72] Microsoft, “Prompt Engineering - Learn how to use AI models with prompt engineering,” <https://microsoft.github.io/prompt-engineering/>.
- [73] Y. Wei, C. S. Xia, and L. Zhang, “Copiloting the copilots: Fusing large language models with completion engines for automated program repair,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.
- [74] S. Tipirneni, M. Zhu, and C. K. Reddy, “Structcoder: Structure-aware transformer for code generation,” *arXiv preprint arXiv:2206.05239*, 2022.
- [75] S. Bratus, I. Arce, M. E. Locasto, and S. Zanero, “Why offensive security needs engineering textbooks,” *Yale Law & Policy Review*, p. 2, 2013.
- [76] J. G. Oakley, “The state of modern offensive security,” in *Professional Red Teaming*. Springer, 2019, pp. 29–41.
- [77] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, “Aeg: Automatic exploit generation,” in *NDSS*, 2011.
- [78] I. Arce, “The shellcode generation,” *IEEE security & privacy*, vol. 2, no. 5, pp. 72–76, 2004.



Attacking with Something That Does Not Exist: 'Proof of Non-Existence' Can Exhaust DNS Resolver CPU

Olivia Gruza^{*†}, Elias Heftrig^{*†}, Oliver Jacobsen^{*†}, Haya Schulmann^{*†}, Niklas Vogel^{*†},
Michael Waidner^{*‡§}

^{*}National Research Center for Applied Cybersecurity ATHENE

[†]Goethe-Universität Frankfurt

[‡]Technische Universität Darmstadt

[§]Fraunhofer Institute for Secure Information Technology SIT

Abstract

NSEC3 is a proof of non-existence in DNSSEC, which provides an authenticated assertion that a queried resource does not exist in the target domain. NSEC3 consists of alphabetically sorted hashed names before and after the queried hostname. To make dictionary attacks harder, the hash function can be applied in multiple iterations, which however also increases the load on the DNS resolver during the computation of the SHA-1 hashes in NSEC3 records. Concerns about the load created by the computation of NSEC3 records on the DNS resolvers were already considered in the NSEC3 specifications RFC5155 and RFC9276. In February 2024, the potential of NSEC3 to exhaust DNS resolvers' resources was assigned a CVE-2023-50868, confirming that extra iterations of NSEC3 created substantial load. However, there is no published evaluation of the attack and the impact of the attack on the resolvers was not clarified.

In this work we perform the first evaluation of the NSEC3-encloser attack against DNS resolver implementations and find that the NSEC3-encloser attack can still create a 72x increase in CPU instruction count, despite the victim resolver following RFC5155 recommendations in limiting hash iteration counts. The impact of the attack varies across the different DNS resolvers, but we show that with a sufficient volume of DNS packets the attack can increase CPU load and cause packet loss. We find that at a rate of 150 malicious NSEC3 records per second, depending on the DNS implementation, the loss rate of benign DNS requests varies between 2.7% and 30%. We provide a detailed description and implementation of the NSEC3-encloser attack. We also develop the first analysis how each NSEC3 parameter impacts the load inflicted on the victim resolver during NSEC3-encloser attack.

We make the code of our NSEC3-encloser attack implementation along with the zonefile and the evaluation data available for public use: <https://github.com/Goethe-Universitat-Cybersecurity/NSEC3-Encloser-Attack>.

1 Introduction

On 13 February 2024 a vulnerability,¹ termed *Preparing an NSEC3 closest encloser proof can exhaust CPU resources*, was registered as CVE-2023-50868 (short for Common Vulnerabilities and Exposures) in a list of publicly disclosed information security flaws. The description of the CVE says that the processing of responses sent by nameservers authoritative for DNSSEC signed zones can exploit maliciously crafted NSEC3 records to cause CPU exhaustion on a DNSSEC-validating resolver. By flooding the target resolver with queries, an adversary can trigger responses to the target resolver with specially crafted NSEC3 records exploiting this flaw. Computation of those NSEC3 records can significantly impair the resolvers' performance. In this work, we provide the first analysis of the vulnerability and an evaluation of the attack against popular DNS resolvers. We explain the impact on the resolvers' implementations using code analysis as well as monitoring of the CPU instruction count and measurements of the latency incurred on requests from benign clients.

Vulnerabilities in proof of non-existence. Domain Name System Security (DNSSEC) RFC4033 – RFC4035 was designed to protect the Domain Name System (DNS) against manipulation attacks by attaching digital signatures to DNS records. The DNS resolvers can use the public keys of the corresponding domains to authenticate the DNS records that they receive in responses. To provide an authenticated proof for resources that do not exist, RFC3845 defined NSEC records, which list the hostname before and the hostname after the requested hostname. The listing of hostnames in NSEC records exposed the domains to zone enumeration attacks, discussed in RFC4470. To mitigate zone enumeration attacks, the IETF standardized NSEC version 3 (NSEC3) in RFC5155. NSEC3 computes hashes over the hostnames and the resulting NSEC3 record lists the hashed names instead of plaintext names. Nevertheless, NSEC3 too was found vulnerable to zone enumeration attacks [3, 5, 10]. Although the privacy aspects of NSEC3 records were substantially explored, there was no evaluation

¹<https://kb.isc.org/docs/cve-2023-50868>

of the performance impact of NSEC3 records on DNS resolvers. In this work, we provide the first evaluation of the performance load induced on the resolvers by attacks with specially crafted NSEC3 records, which we dub the *NSEC3-encloser* attack. Although the potential degradation of performance by NSEC3 records was considered in RFC5155#§8.3, there was no evaluation of the impact on performance by attackers and the role of the NSEC3 parameters on the effectiveness of the attack. A recently registered CVE-2023-50868 does not explain the impact of the attack on the resolvers nor provides the evaluation of the attack.

NSEC3-encloser can exhaust CPU and lead to loss. We implement and evaluate an NSEC3-encloser attack that leads to increased CPU instruction counts on the affected resolvers, and also to loss of packets from legitimate clients. In our implementation of the attack, the NSEC3 records use the maximum number of iterations supported by the DNS resolver implementations, which follow the recommendation counts listed in RFC5155. We experimentally observe that using salt in the calculation of hashes in NSEC3 results in a more effective attack than attacks without the salt. The reason is that the salt value creates an additional input block which leads to an increased calculation time since the blocks are processed sequentially. At the same time, the salt value does not substantially increase the resilience to zone enumeration attacks since, in contrast to the traditional uses of the salt in hash computations like for passwords, the hashes are implicitly salted per zone by including the domain name in the computation process. This is also stated in RFC9276, and limits the benefit of using a salt in the first place.

Our contributions can be summarized as follows:

- We develop a tool for automated evaluation of the CVE-2023-50868 attack, expanding on the proof-of-concept in the CVE, and providing an automated setup to generate zones and queries. Our implementation creates multiple NSEC3 configurations setting different values for NSEC3 parameters, including a novel method for maximizing the number of NSEC3 records in DNS responses and varying salt length, all of which allow for testing different aspects of the resolvers' behavior. We make our tool open-source to facilitate reproduction of our work [6].

- We provide the first evaluation of an attack that exploits NSEC3 records for creating a load on DNS resolvers. In our evaluation, we also analyze the resolvers' behavior and limits introduced in RFC5155 and explain how the resolvers react to different values of NSEC3 parameters. We find that the salt increases the load on the resolvers by 30%, an aspect which was previously overlooked and not included in either CVE-2023-50868 or the PoC that the CVE made public. Our full fledged and automated attack evaluation allowed to identify the role of salt in increasing the CPU instruction counts on the resolvers. We also explore the limitations of the NSEC3-encloser attack, i.e., the high query rate required to load resolvers and the relatively low impact on traffic loss.

- We perform the first comparison of the NSEC3-encloser attack to other attacks on DNS, and explain the differences in performance and load, as well as in the vulnerabilities in resolvers' behavior that are exploited.

- We perform measurements of NSEC and NSEC3 configurations on DNSSEC-signed domains and find that 56% of the domains use NSEC which is vulnerable to zone enumeration, while 41% use NSEC3. 77% of those NSEC3 domains use a high number of hash iterations which exposes those domains for abuse to create load on victim resolvers.

Organization. This paper is organized as follows. In Section 2, we provide an overview of DNSSEC and the proof of non-existence with NSEC and NSEC3. We provide the details of the NSEC3 attack in Section 3. We evaluate the NSEC3 attack in Section 4, demonstrating the role of the parameters in the NSEC3 record on the impact of the attack. We measure real-world DNSSEC and NSEC/3 in Section 5. Finally, we review Related Work in Section 6 and conclude in Section 7.

2 Overview of DNSSEC and NSEC3

The IETF standardized DNSSEC RFC4033 – RFC4035 to enable DNS resolvers to detect if DNS records in responses are manipulated. The DNSSEC specification requires that the records in a zonefile are digitally signed. The zonefile contains DNS records as well as DNSSEC material, most notably DNSKEY, RRSIG, and DS records.

DNSSEC signatures are stored in RRSIG-type DNS records. The public keys used to validate the signatures are sent in DNSKEY-type records. DS records from a parent zone are used to authenticate individual Key Signing Key (KSK) type DNSKEY records in a child zone. This is done to delegate trust from a parent zone public key to a child zone public key. DS records use the same triple (owner name, algorithm, key tag) to identify a subset of candidate DNSKEYs as RRSIGs.

In addition to cryptographically attesting the validity of DNS records, DNSSEC also enables proofs for non-existing records, enabling authenticated denial of existence.

For this, RFC4035 defines Next Secure (NSEC) records for a precomputed denial of existence, that prove that a requested hostname does not exist. Each NSEC record contains a signed pair of consecutive hostnames, sorted canonically. Each query for a hostname not in the zonefile is answered by the name-server with a suitable NSEC record. For instance, a query for a non-existing hostname `b.x.org` is responded with a signed NSEC record for a pair of existing hostnames sorted canonically before and after the queried hostname: `a.x.org` and `c.x.org`. The resolver can then confirm the requested hostname does not exist as the NSEC record attests no domain name exists between `a.x.org` and `c.x.org`, proving non-existence of `b.x.org`. An example of a NSEC record is given below.

```

\\ Domain | TTL | RR type | Next hostname
   x.org   700   NSEC    a.x.org

\\ Resource record sets
   NS SOA RRSIG NSEC DNSKEY

```

Research showed that NSEC was vulnerable to zone enumeration attacks [3, 5, 10]. By enumerating a target zone, an adversary learns the IP addresses of all resources in the target zone. An enumerated list of resources can be exploited for other attacks, such as spam. To mitigate the threat introduced by NSEC records, RFC5155 designed NSEC3: a precomputed denial of existence. The idea of NSEC3 is replacing clear-text hostnames with hashes, which makes zone enumeration from the names significantly harder. The knowledge of the hashed hostname cannot be directly used for zone enumeration since cryptographic hash functions do not allow for the reconstruction of the plaintext hostname through preimage resistance. NSEC3 uses an additional record NSEC3PARAM which contains parameters for the NSEC3 validation, including the hash algorithm, the amount of iterations, and salt parameters. A single NSEC3PARAM record dictates the parameters for the entire set of NSEC3 records. This is needed to ensure that any query for a non-existent hostname maps to an NSEC3 record. The ‘salt’ contains hexadecimal digits and is appended to the domain name to make offline dictionary attacks harder. ‘Iterations’ indicates the number of times the hash function was computed.

The NSEC3 record contains a pair of ordered hashes. According to RFC5155, to create the NSEC3 records, the canonical hostname is hashed once and the resulting hash is re-hashed a number of times according to the iteration parameters in the NSEC3PARAM. Upon a query for a non-existent resource, the nameservers should return to the requesting resolvers a signed NSEC3 record that contains two hashes, one before the requested hostname and one after. The resolver can then hash the hostname to ensure the hashed hostname lies between the returned hashes, thereby proving the non-existence. An example of an NSEC3 record is given below.

```

\\ Hashed domain | TTL | RR type | Algorithm
   ej23jdn4jnd... 700   NSEC3    1 (SHA1)

\\ Flags | Iterations | Salt
   0      150      64ccb74...

\\ Next hostname | Resource record sets
   kev723jd...   NS SOA RRSIG NSEC DNSKEY

```

RFC9276 defines the best current practice for setting and dealing with NSEC3 parameters, including considerations of Denial of Service (DoS) by Central Processing Unit (CPU) resource exhaustion through NSEC3 hashing. The only hash function standardized for use in NSEC3 records is SHA-1.²

²<https://www.iana.org/assignments/dnssec-nsec3-parameters/dnssec-nsec3-parameters.xhtml>

According to RFC5155#§7.2, the resolvers require a proof of the closest encloser, which proves that a subdomain of the requested hostname is the closest encloser of that name. The proof consists of up to two NSEC3 records: An NSEC3 record that matches the closest (provable) encloser and an NSEC3 record that covers the “next closer” name to the closest encloser. The first NSEC3 record proves that the encloser exists. The second NSEC3 record proves that the possible closest encloser is the closest, and proves that the queried hostname (and any subdomains between the queried hostname and the closest encloser) does not exist. These NSEC3 RRs are collectively referred to as the “closest encloser proof” RFC5155. An example in RFC5155 describes the closest encloser proof for the nonexistent hostname `alpha.beta.gamma.example.`: The owner might prove that `gamma.example.` is the closest encloser. The response contains the NSEC3 record that matches `gamma.example.`, and also contains the NSEC3 record that covers `beta.gamma.example.` (which is the “next closer” name).

According to the specification in RFC5155 to prove the nonexistence of a hostname in a query, a closest encloser proof and an NSEC3 record covering the (nonexistent) wildcard record at the closest encloser MUST be included in the response. This collection of (up to) three NSEC3 records proves both that the queried hostname does not exist and that a wildcard that could have matched the queried hostname also does not exist; if `gamma.example.` is the closest provable encloser to the queried hostname, then an NSEC3 record covering `*.gamma.example.` is included in the authority section of the response.

3 NSEC3-Encloser Attack

The NSEC3-encloser attack exploits computational complexity in hash calculation for closest encloser proofs. The idea behind the attack is to set up a malicious zonefile in a valid DNSSEC signed domain, then to cause the victim DNS resolvers to issue DNS queries for a non-existent resource in the domain of the adversary. We design our attack to be fully RFC compliant; both the client requesting resolution from the victim resolver as well as the nameserver containing the malicious zonefile fully conform to all RFC requirements. The goal is to create a zonefile that maximizes both the number of hash calculations and the computation effort per single hash calculation. We construct an attack on NSEC3 instead of NSEC as the former requires hash calculations for the closest encloser proof, which significantly increases computational load compared to NSEC. The core aspect of the NSEC3 attack lies in the construction of the proof of non-existence with NSEC3 records, which should lead to many hash calculations in the victim resolver. The adversary requests a resource that inflicts large complexity for the resolver to prove the closest encloser. In the following, we illustrate the attack concept with exemplary adversarial zonefiles.

3.1 Zonefile Construction

In the configuration of the zone, we follow DNSSEC and NSEC3 standard specifications. This ensures that the zonefiles are accepted by all standard compliant resolvers.

To maximize the attack impact, the attacker needs to trigger the maximum number of hash validations in a victim resolver. Since each NSEC3 record obtained from a DNS request results in a single hash calculation, this corresponds to maximizing the number of NSEC3 records for a given request. Following RFC5155, this number is limited to up to three NSEC3 records per DNS request, leading to a maximum of three hash calculations per request. Achieving this maximum number of NSEC3 records in each resolver request requires a specific zonefile configuration, which we illustrate in Figure 1. For a configured zone origin, the generated zonefile consists of the following non-NSEC3 (and non-RRSIG) records:

- ▷ The SOA, NS and DS records of the zone, present at the zone apex.
- ▷ Two DNSKEY records, one for the KSK and one for the ZSK.
- ▷ One NSEC3PARAM record at the zone apex, signaling NSEC3 usage to the authoritative nameserver.
- ▷ The A record for the nameserver domain.

The zone has two unique name entries, ATTACK.ER and NS1.ATTACK.ER. Following specification, both of these names require an NSEC3 record, proving the existence of the Resource Record sets (RRsets) listed for the names. However, to achieve three NSEC3 records in the response for an arbitrary resolver request, this is insufficient, as any domain existence or non-existence proof would require between one and two of these NSEC3 records. To validate an NSEC3 reply, resolvers need three different values from the nameserver: The closest encloser, proof that the “next closer” domain does not exist, and proof that no wildcard record exists covering the requested domain.

The closest encloser proves that a domain exists in the zone that is the nearest ancestor of the queried name. It establishes a context within which the non-existence of the queried domain can be asserted. In our example, the NSEC3 record with the hash of ATTACK.ER proves the existence of this hostname, and all subdomains will receive this record as their closest-encloser.

The next domain hash of an NSEC3 record provides evidence of the numerically subsequent domain name hash in the zone, confirming that no records exist between the domain name hash of an NSEC3 record and this next domain. For example, consider a nameserver has to prove the non-existence of a domain with a hash of 0x123. In the zone, the next smaller NSEC3 record has a hash of 0x111, with a next hash value of 0x222. Since the requested domain hash (0x123) is larger than 0x111 but not equal to 0x222, the requested domain probably does not exist in the zone. The nameserver must provide the

```
;; ZONE 'ATTACK.ER'
ATTACK.ER. 0 IN SOA NS1.ATTACK.ER. NS1.ATTACK.ER. 0 0 0
10 0
ATTACK.ER. 0 IN NS NS1.ATTACK.ER.
ATTACK.ER. 0 IN DS 35650 7 1 e8316...
ATTACK.ER. 0 IN DNSKEY 257 3 7 AwEA...
ATTACK.ER. 0 IN DNSKEY 256 3 7 AwEA...
ATTACK.ER. 0 IN NSEC3PARAM 1 0 150 -
HKHV...38AU.ATTACK.ER. 0 IN NSEC3 1 1 150 -
HKHV...38B0 ①
HKHV...38B0.ATTACK.ER. 0 IN NSEC3 1 1 150 -
QCQC...7U45 ②
NS1.ATTACK.ER 0 IN A 6.6.6.6
QCQC...7U45.ATTACK.ER. 0 IN NSEC3 1 1 150 - SN5U...89IT A
RRSIG ③
SN5U...89IT.ATTACK.ER. 0 IN NSEC3 1 1 150 - SN5U...89IU NS
SOA DS RRSIG DNSKEY NSEC3PARAM ④
SN5U...89IU.ATTACK.ER. 0 IN NSEC3 1 1 150 -
HKHV...38AU ⑤
[...] ;; RRSIG records
```

Figure 1: Generated attack zonefile example.

NSEC3 record proving that the “next closer” domain (the ancestor of the queried name just below the closest encloser) does not exist. The resolver can confirm that this domain name does not exist by validating that the next hash in the returned NSEC3 record is not the hash of the “next closer” domain. By inference, the queried name cannot exist, too, since the zone does provably not include one of its ancestors.

Finally, the resolver needs to ensure that no wildcard record covers the requested domain. The nameserver thus includes the NSEC3 record next-smaller of where the hash of the wildcard record corresponding to the level of the enclosed domain would be. These proofs may, however, overlap. For example, if the next domain corresponds with the NSEC3 entry for the closest encloser, the nameserver will only send the overlapping entry once, reducing the resulting computational effort in the resolver, thereby weakening the attack.

To force the authoritative nameserver to serve exactly three NSEC3 records to every request for a non-existing domain name and thereby maximize the impact of the attack, we develop a new scheme for NSEC3 records in the zone. The required records are described in the following. Note that H is the NSEC3 hash function used, generally SHA-1.

- (1) $H(\text{ATTACK.ER}).\text{ATTACK.ER}$ with next hash (3)
- (2) $H(\text{NS1.ATTACK.ER}).\text{ATTACK.ER}$
- (3) $(H(\text{ATTACK.ER}) + 1).\text{ATTACK.ER}$
- (4) $(H(*.\text{ATTACK.ER}) - 1).\text{ATTACK.ER}$ with next hash (5)
- (5) $(H(*.\text{ATTACK.ER}) + 1).\text{ATTACK.ER}$

NSEC3 records (1) and (2) are mandatory records and thus must be included in the domain. Further, in the attack setup, the adversary will trigger resolution of a non-existent sub-domain of the ATTACK.ER domain, resulting in (1) always contained in the reply as it is the closest encloser to all requests. Note that this closest encloser NSEC3 record also includes a next-hash value. If the resolver requests a domain which is, by chance, hashed to a value directly “after” the ATTACK.ER domain hash, the authoritative server would detect the overlap and only send a single NSEC3 record (1) to cover closest encloser and the next hash. To prevent this and force an additional NSEC3 record in the answer, we include an additional NSEC3 record (3) which covers the hash one larger than (1). Thus, (1) always has (3) as next hash and therefore never covers any other non-existent domain in the zone. It will therefore never overlap with the required “next closer” domain record.

Similarly, the attacker needs to ensure that none of the above mentioned records, by chance, covers the wildcard domain name, as the resolver would then, e.g., only need to send a single record for “next closer” and wildcard proof. To prevent this, a new record (4) is added, with a hash value just below the hash of the wildcard domain name, as this record will now always be included to proof non-existence of the wildcard domain. Conversely, this new record now also has a next hash value, which might by chance cover the “next closer” domain of the requested domain, again leading to overlap. Therefore, a new record (5) is added that ensures that the record (4) only covers two hashes. Thus, for every request to a non-existent domain, the nameserver must include three NSEC3 records: (1) for closest encloser, then (2), (3) or (5) for the “next closer” proof, and finally (4) for wildcard proof.

3.2 Maximizing the Impact

Using the above described zonefile as-is only results in three hash computations. However, the impact can be increased, both by adapting the DNS request from client to resolver, and by adapting the malicious zone.

3.2.1 Adapting the request

When a client requests a non-existent domain from the resolver, the resolver needs to conduct the above described checks to attest non-existence of the domain, including the check for the closest encloser. Crucially, the resolver cannot necessarily directly infer the closest encloser from the NSEC3 records. For instance, consider a nested sub-domain A.B.ATTACK.ER. The resolver receives a hash for the closest encloser, but does not directly know if the hash is for A.B.ATTACK.ER, B.ATTACK.ER, or ATTACK.ER. Instead, the resolver has to attempt for each candidate individually whether any of the NSEC3 records in the response proves the existence for the encloser. The algorithm for this is listed in

RFC5155. The resolver hashes the query name and matches the resulting hash against each NSEC3 record. If none of the records fit, it has to slice away the next label and try again, repeatedly hashing and matching. Therefore, the workload of the closest encloser proof depends on the number of labels below the closest encloser in the query name and, to a lesser degree, on the number of NSEC3 records in the nameserver response. Maximizing these numbers can incur a significant workload of calculating hashes on the resolver. Note that the maximum number of labels in the request is limited by the maximum request size of 255 bytes in RFC1035.

3.2.2 Adapting the zone

Using NSEC3 parameters in a malicious zonefile, the per-hash overhead can be greatly increased. In the following, we highlight the two NSEC3 parameters that can be manipulated to maximize impact.

3.2.3 Hash iteration count

NSEC3 supports hash iterations to increase computational effort for brute-forcing hash values. Hash iterations require that the hash of a domain name is re-iterated through the respective hash function for a set number of iterations. This mechanism, while improving security through hardening brute-force protection, can be exploited to increase computational load per calculation on the resolver, resulting in a stark increase in the number of hash calculations in the attack. For example, if the resolver needs to calculate three hashes for the three NSEC3 records in the zone, choosing an iteration count of 100 will result in a total of 300 hash calculations.

3.2.4 Adding a salt

Additionally to iterations, NSEC3 also supports protection against rainbow-table attacks [9] through the addition of a salt value to the hash. The salt is added to the plaintext domain name before hashing, which prevents pre-calculation of tables of potential domain names. The salt additionally increases the computational load for hash calculations, as SHA-1 (the only currently supported hash algorithm) exhibits an increase in computation time over longer plaintext inputs. The increase in computation time stems from the underlying blocks that are used as input to the hash functions; with more blocks of plaintext, the hash function takes linearly more time. Notably, when using iterations, the salt is not only added to the first iteration of the hash function but to all subsequent inputs to the function, increasing load for each of the iterations.

Our code-review yields that all investigated resolvers support both the hash iterations and the salting, following RFC specification. An exemplary implementation of the hash function in Unbound DNS resolver is given in code Listing 1.

Resolver	Iteration Limits		
	Patched version	50	150
Unbound	1.19.1	—	1.13.2
Bind9	9.18.24, 9.19.19	9.18.24, 9.19.19	9.17.13
	9.16.48		9.16.16
PowerDNS	5.0.0	5.0.0	4.5.2
Knot Resolver	5.7.1	5.7.1	5.3.1

Table 1: The limits introduced across resolvers over time.

```

1 nsec3_calc_hash(struct regional* region,
2     sldns_buffer* buf, struct nsec3_cached_hash* c) {
3     // [...] Init buffers and do sanity checks
4
5     // Write dname and salt to buffer
6     sldns_buffer_write(buf, c->dname, c->dname_len);
7     sldns_buffer_write(buf, salt, saltlen);
8
9     // Calculate first hash from buffer content
10    (void)secalgo_nsec3_hash(algo,
11        (unsigned char*)sldns_buffer_begin(buf),
12        sldns_buffer_limit(buf),
13        (unsigned char*)c->hash);
14
15    for(i=0; i<iter; i++) { // Iterate through number
16        // Insert previous hash and salt into buffer
17        sldns_buffer_clear(buf);
18        sldns_buffer_write(buf, c->hash, c->hash_len);
19        sldns_buffer_write(buf, salt, saltlen);
20
21        // Calculate hash from buffer content
22        (void)secalgo_nsec3_hash(algo,
23            (unsigned char*)sldns_buffer_begin(buf),
24            sldns_buffer_limit(buf),
25            (unsigned char*)c->hash);
26    }
27 }

```

Listing 1: Source code for NSEC3 iterations in Unbound.

The code snippet shows how the NSEC3 iterations are performed. The hash is calculated and written into the result. Then, a for-loop is entered which continuously writes the result of the previous hash calculation into a clear buffer, adds the salt and calculates the hash again, as long as the iteration count is below the limit. The code-example shows that Unbound, like all investigated resolvers, conforms to the specification in iterating the hash and adding the salt in each iteration.

Practical limits to iterations. The standard provides recommendations to the number of iterations a resolver may allow on a given NSEC3 record. We find from code review that these values are observed only in some resolvers; a subset of resolvers do not enforce these limits, while other resolvers set stricter limits in their standard configuration. This is not surprising, as RFC9276 encourages resolvers to choose their own limits to a value they seem adequate for current deployments. A detailed overview of enforced iteration limits in different resolver versions is presented in Table 1.

Practical limits to salt length. Generally, a longer salt value allows for longer calculation time of a given hash. However, the maximum length of the salt is limited by the available space of the salt field in the NSEC3PARAM record, only allowing up to 255 bytes of data for the salt. We find from code review that all resolvers allow this maximum salt length, with no resolver enforcing stricter length limits.

Resolver	Version	Iteration Limit
Bind9	9.16.1	RFC5155
Bind9	9.18.12	150
Unbound	1.17.1	150
PowerDNS	4.8.2	150
Knot	5.6.0	150

Table 2: Resolver versions and iterations limits in the test setup.

Thus the maximum attack impact can be achieved by querying the resolver with a deeply nested sub-domain, configure the nameserver to always deliver all three NSEC3 records, and using both the maximum number of iterations allowed by the resolver, and the longest possible salt length of 255 byte.

3.2.5 Generating the zonefile

To test different zone configurations with differing values for the NSEC3 parameters, we develop a script that automatically generates zonefiles from a singular JSON configuration file. We make the script publicly available to facilitate reproduction of our work [6]. This configuration file used in the script specifies the individual zones, the cryptographic parameters, such as key size and NSEC3 iterations, nameservers, TTL values, and relationship between the zones. The generation script written in Python parses a configuration, generates the defined records, creates all relevant DNSSEC signature and key records, and exports each zone to a file to be hosted by a nameserver implementation.

4 Evaluation of the Attack

To practically evaluate the impact of the attack, we deploy the resolvers and a nameserver with the attack zones in a local isolated setup. We send attack queries to the resolvers and measure the impact of the attack under different scenarios. Section 4.1 describes the test setup, Section 4.2 illustrates the influence of different parameters on the impact of the attack, and Section 4.3 delves into comparing the impact of the attack between different resolvers, highlighting differences in implementations that cause different reactions to attack requests. Finally, in Section 4.4, we show that the attack can sufficiently stall resolvers to cause a drop of benign client queries.

4.1 Setup

We deploy the five resolvers in Table 2 as Docker containers communicating via a network bridge with our nameserver for the attack requests, and the internet for benign requests. We additionally include the older Bind9 version 9.16.1 in our test environment to compare the impact of the (historic) iteration count limits defined in RFC5155 to the lower limits adopted by the current implementations. To serve the attacker zones, we set up an NSD 4.6.1 authoritative nameserver on our local

network which serves the generated zonefiles. This ensures that we accurately measure the attack impact on the resolvers, since the per-query overhead introduced by the authoritative nameserver is negligible. The nameserver is not reachable from the internet.

We generate and include zonefiles for different combinations of parameters in NSD for each test, each having a unique identifier as part of the domain name. The zones are generated as child zones `EXii.NSEC3.EXAMPLE.ORG` to a parent zone `NSEC3.EXAMPLE.ORG`, where `ii` is the two-digit zone identifier. It is unrealistic that an attacker can control zones at the domain tree root or some top-level domain, but since the impact of the attack depends on the length of the zone domain name, we select the reasonable-length domain name `NSEC3.EXAMPLE.ORG`. The parent zone contains signed DS records with the digest of the child zone KSK's, i.e., the zone has a complete and valid DNSSEC configuration and follows RFC specification.

Since the wire-format of the child zone domain is 24 bytes (including the root label), there remain 231 bytes for additional labels in an attacker query QNAME. We use a randomly chosen 4-byte label as the non-existent subdomain for the attack to prevent the resolver from answering queries from the caches. This effectively leaves 226 bytes for additional labels. Hence, the attack query names to the resolvers have the following format, resulting in 115 sub-labels:

(A.)¹¹³.abcd.EXii.NSEC3.EXAMPLE.ORG

Each resolver is configured to query the local NSD authoritative nameserver for any queries to `NSEC3.EXAMPLE.ORG` with the zone's keys added to the set of trusted keys of the resolvers. Furthermore, the resolvers have DNSSEC validation enabled and are run single threaded.

Our test setup is running Ubuntu 22.04 with a 12th Gen Intel® Core™ i7-1280P CPU at 4.8GHz.

4.2 Comparison of Attack Parameters

To compare the impact of the attack parameters, we execute the resolvers in a controlled environment and measure the attacker-induced CPU load for different rates of attacker queries per second and different parameter configurations. In our analysis, we identify how specific values for configurable parameters influence the CPU exhaustion impact on the resolvers, illustrating how to maximize attack impact as well as giving a numerical basis to choose appropriate limits for attack mitigations.

Our analysis includes key sizes, the number of NSEC3 iterations, and the length of the NSEC3 hash, influenced over the salt length. Each test case includes an incremental increase of the rate of attacker requests on the resolver to illustrate resolver behavior both under small scale and heavy attack.

We conduct multiple tests to find the ideal rate for increasing the attack rate and the maximum rate of attack in the

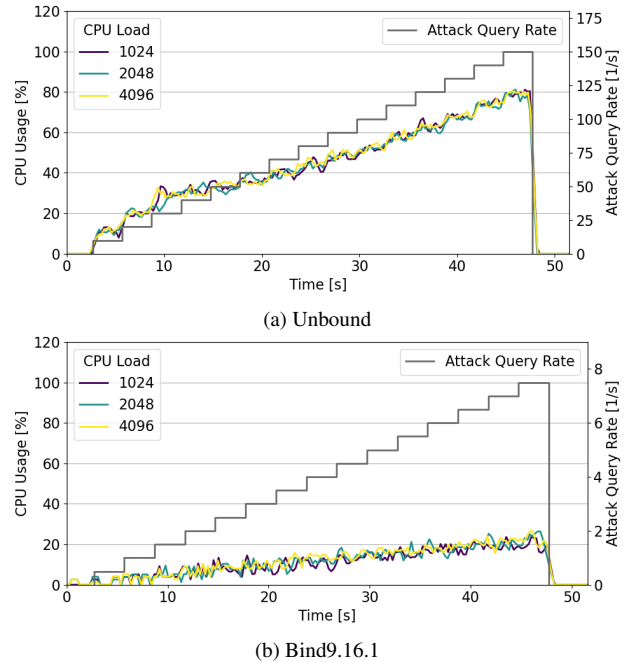
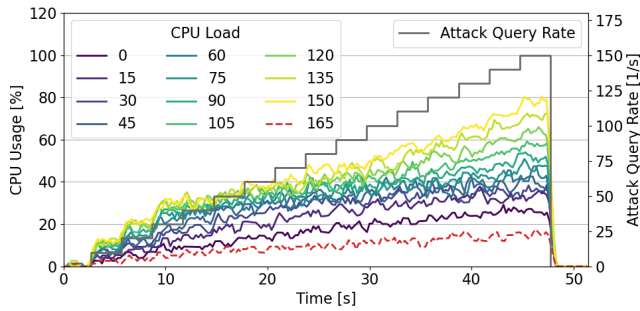


Figure 2: Comparison of CPU workload for different key sizes.

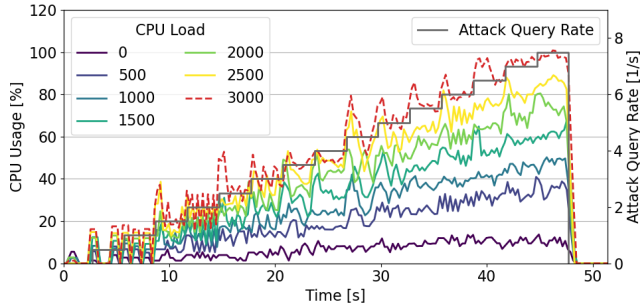
experiments. We find increasing the attack rate too quickly does not allow to distinguish the impact of a specific rate from natural fluctuations in CPU load resulting from CPU scheduling, while increasing it too slow wastes measurement time. Following our evaluation, we find increasing the attack rate every 3s as a suitable compromise. To identify a suitable maximum attack rate for the experiment, we continuously increase the rate of attack until we see artifacts caused from the experiment hardware struggling to keep up with sending enough requests to the nameserver. We find a value of 150 requests per second as a suitable maximum value where we did not observe any kernel- or hardware-induced artifacts in our measurement. A value of 150 requests per second is sufficient to cause 100% CPU load in all investigated resolvers. Finally, we choose to increase the attack rate with a delta of 10/3s to cause an observable difference between measurement steps, while also keeping the measurement fine-grained enough to see detailed effects at different steps.

For Bind9.16.1, which poses no strict NSEC3 iteration limit and therefore enables a much higher attack impact per request, we reduce the attack rate to enable similar fine-grained insights. We identify an attack rate delta of 0.5/s and an upper bound of 7.5/s suitable for our setup.

In our experiments, we find that the impact of different parameters is similar between the investigated resolvers. We will thus in the following section focus on the parameter impact on Unbound 1.17.1 and Bind9.16.1. The differences between resolvers will be discussed in Section 4.3.



(a) Unbound



(b) Bind9.16.1

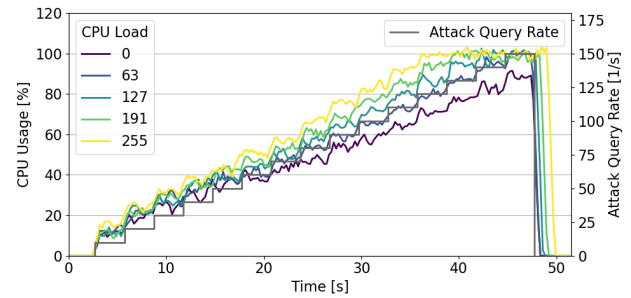
Figure 3: CPU workload for different NSEC3 iteration counts.

4.2.1 Key Size

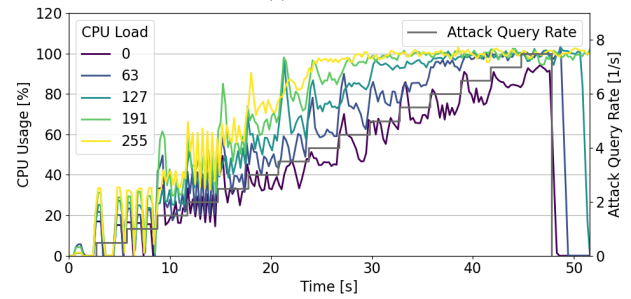
While no NSEC3 parameter per se, the key size influences the maximum allowed number of NSEC3 iterations as defined in RFC5155. We do not expect that the key size has a significant impact on the induced CPU workload since the load stems from the high number of hash calculations and not signature validation. Nevertheless, we evaluate whether this assumption holds for the tested resolvers. For this test, we fix the NSEC3 iterations at 150 and the salt length to 0 and compare the three different supported RSA key sizes of 1024, 2048, and 4096. The results are plotted in Figure 2. As expected, there is no significant deviation between the three curves in CPU load. Hence, for the subsequent tests, we use the key size 4096 as it allows for a much larger range of NSEC3 iteration values.

4.2.2 NSEC3 Iterations

For Unbound, we evaluate different NSEC3 iteration counts ranging between 0 and 165 in Figure 3a. We observe a clear correlation between higher iteration counts and larger induced workload which approaches a linear distribution as the attack query rate increases. The exception is 165 iterations which shows loads well below all other measurements. This is because the evaluated version of Unbound has a pre-configured limit of 150 NSEC3 iterations and disregards the zone with a higher iteration count as bogus without further validating the NSEC3 records it receives from the authoritative nameserver. Since processing the queries and validating the signatures has some constant overhead, an iteration count of 0 incurs more overhead than the rate 165.



(a) Unbound



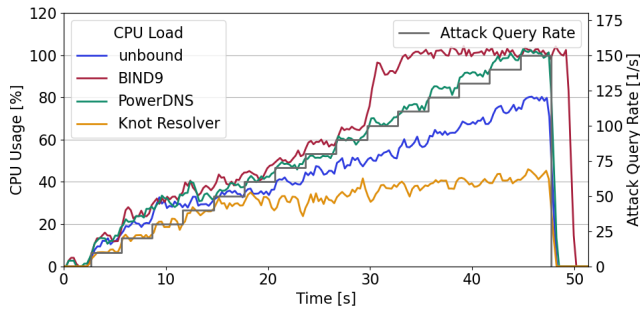
(b) Bind9.16.1

Figure 4: CPU workload for different NSEC3 salt lengths.

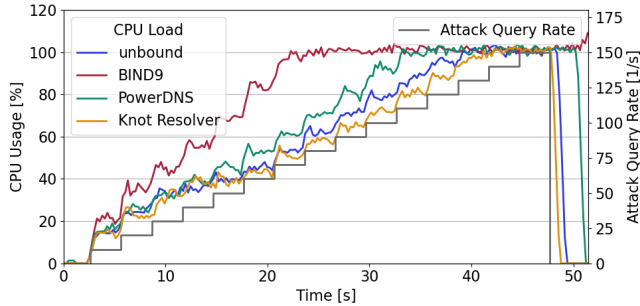
In the case of Bind9.16.1, no limits are enforced for the iteration values. As evident in Figure 3b, this allows us to query zones with iteration counts well above the 150 limit of all other tested resolvers. More significantly, we can use values above the 2500 iteration limit of RFC5155 which illustrates a significant vulnerability in this version of the resolver. At attack rates as low as 7.5 queries per second, we are able to max out the CPU load at 100% for this iteration limit. But, even for the standardized 2500 iterations, there is a significant load on the resolver, reaching up to 90% at an attack rate of 7.5/s. Thus, higher iteration counts can significantly increase the impact of the attack on resolvers.

4.2.3 NSEC3 Salt Length

Next, we compare different salt lengths in Figure 4. In this test, we use a key size of 4096 and set the NSEC3 iterations to the most impactful RFC5155-conform value of 150 for Unbound and 2500 for Bind9.16.1, respectively. For Unbound, we measure an increase of CPU load by approximately one third and for Bind9.16.1 by about one half when increasing the length of the salt from 0 to 255, with the load of the intermediate values distributed uniformly in-between. This is to be expected from the way the NSEC3 hashes are calculated. Since the salt is appended to the hashed domain/digest at each iteration, the additional workload of longer inputs to the hash function applies to every iteration of the hash function. SHA-1 is a Merkle-Damgård hash function, hence, the calculation overhead grows roughly linearly with the number of blocks the hash function is calculated on. With a block size of 512 bit, every 64 bytes added to the hash function input require



(a) 150 iterations, 0 byte salt



(b) 150 iterations, 255 byte salt

Figure 5: Comparison of CPU workload between resolvers

one more calculation of the SHA-1 hash function to compute the digest. Thus, a longer salt multiplies the total load on the resolver for each NSEC3 hash calculation by the number of blocks added through the concatenation of the salt to the digest per hash function execution. Overall, the increased load causes the CPU load to max out at 100% for Unbound at an attack rate of 110/s and Bind9.16.1 at 4.5/s for a salt length of 255 bytes.

4.3 Comparison of Resolvers

In this section, we compare the CPU load of the resolvers under the most effective parameter choices. Since the high iteration limit in Bind9.16.1 represents a special case, we limit the comparative analysis to the resolvers with an iteration limit of 150: Bind9.18.12, Unbound 1.17.1, PowerDNS 4.8.2, and Knot 5.6.0. As in the previous section, we execute an attack with an incrementally increasing attack rate of up to 150/s and measure the induced CPU load. We fix the zone NSEC3 iterations at 150 and repeat the test with salt lengths 0 and 255. The test results are illustrated in Figure 5.

4.3.1 Salt Length 0

Figure 5a plots the CPU rates of all resolvers with a salt length of 0. We can observe a clear differentiation of loads between the resolvers with only Bind9.18.12 and PowerDNS reaching the 100% CPU limit before the attack rate is maxed out, at 110 and 150 packets per second, respectively. Furthermore, we observe that Bind9.18.12 remains at 100% CPU activity for 2

Resolver	Attack Rate	Total Loss Rate	Adjusted Loss Rate*
Bind9.18.12	150/s	5.10%	7.01%
Bind9.18.12	110/s	16.42%	22.99%
Unbound	150/s	24.75%	34.66%
PowerDNS	150/s	1.97%	2.76%
PowerDNS	120/s	5.62%	7.87%
Knot	150/s	12.87%	18.01%

(*Total loss rate relative to the attack duration)

Table 3: Measured client request loss rate with an attack rate of 150/s over 40s, 150 iterations, and 255 byte salt.

more seconds after the attack has concluded, indicating that the resolver is falling behind processing the queries in real time. Notably, Knot is able to process the attack queries more effectively, only reaching a workload of up to 50% during the test.

4.3.2 Salt Length 255

For the test case with the 255 byte salt, we illustrate the measured CPU load in Figure 5b. In this scenario, all resolvers max out at 100% CPU load before the limit of 150 attack queries per second is reached. Bind9.18.12 reaches full load at 80/s, PowerDNS at 110/s, Unbound at 130/s, and Knot at 140/s. This confirms that the NSEC3 salt has a significant effect on the impact of the attack on all resolvers, roughly increasing the load by a third and — in the case of Knot — up to one half. Once more, we observe continuing CPU load after the attack has concluded, this time for all resolvers. The time of continued stalling correlates with how early in the attack the full CPU load is reached because, once rates continue to rise above the rate at which the CPU is at 100%, the resolver is unable to process the queries at the same rate as there are new incoming queries. Bind9.18.12 continues processing queries until after the measurement has concluded.

We can thus confirm that all examined resolvers are vulnerable to the attack. Knot generally performs best when stressed under the resource exhaustion attack for both attack configurations, while Bind9.18.12 shows the greatest vulnerability to the attack in terms of CPU load. In general, the effectiveness of the attack scales linearly with the attack query rate per second.

4.4 Effect on Benign Clients

Having established that high query rates are required for achieving high CPU load on the resolvers, the question remains whether the attack can be used to sufficiently stall the resolvers such that they fail to answer benign client queries. We evaluate this by continuously sending client queries at a rate of 10/s to the resolvers while simultaneously attacking the resolver with the NSEC3-encloser attack. The clients query unique uncached records from the resolvers and log whether they receive a reply. After 5s, we consider a client request timed out, i.e., too old to be of value to the client and

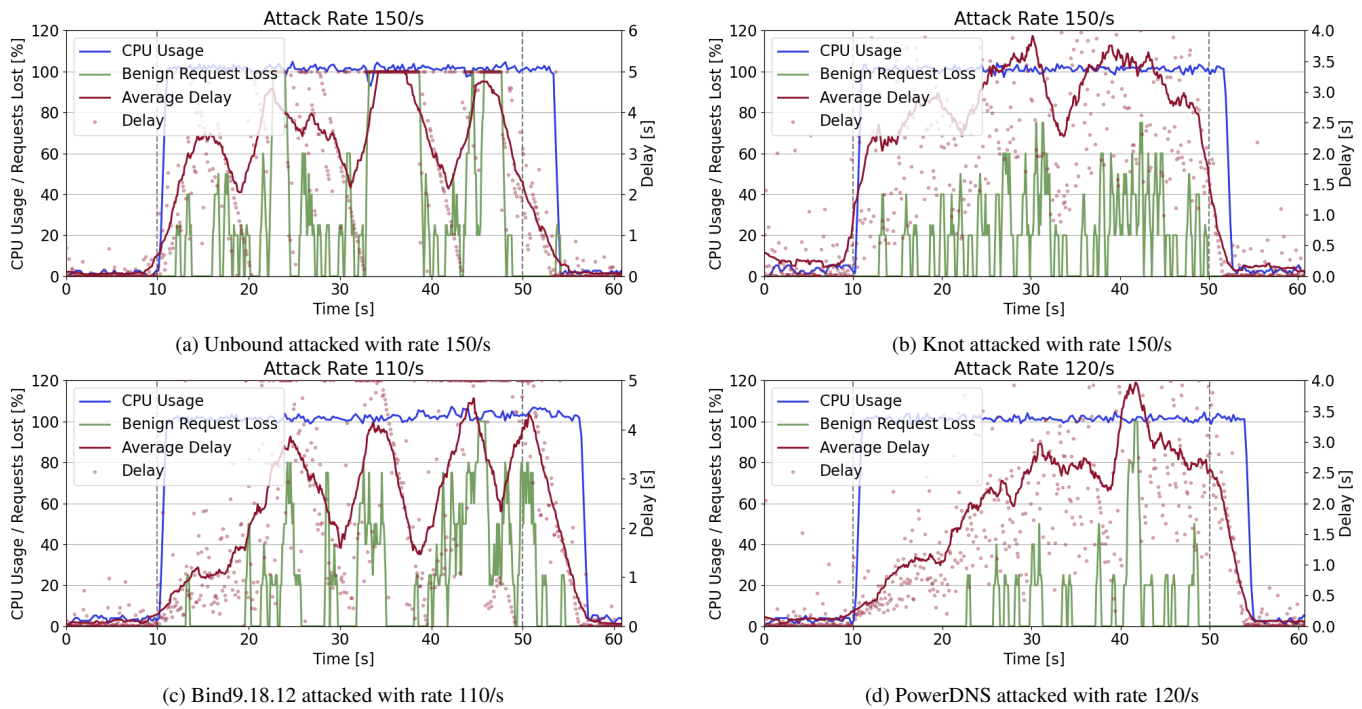


Figure 6: Comparison benign query delays and drops with 150 iterations and 255 byte salt

therefore lost. This is in line with the timeouts used by `dig`³ and `glibc`.⁴ Figure 6 shows the results for all tested resolvers, Table 3 lists the measured client loss rates.

We measure the resolvers at attack rates of up to 150/s, starting the attack 10s into the test and executing it for 40s. For both Unbound (Figure 6a) and Knot (Figure 6b), we achieve adjusted loss rates — the total loss rate during the entire test relative to the attack time — of 34.66% and 18.01%, respectively. For Bind9.18.12, 150/s is well above the attack rate at which CPU utilization reaches 100%, hence, the high number of stalled NSEC3 validations tend rapidly exhaust kernel and hardware resources and interfere with the measurement results yielding an adjusted loss rate of 7.01%. Bind9.18.12 reaches a peak adjusted loss rate of 22.99% at the rate of 110/s (Figure 6c). Similarly, PowerDNS, when attacked at 150/s, reaches a point where there are too many stalled attacker queries leading to lower loss rates in our setup. The evaluated peak rate for PowerDNS is 120/s where we measure a loss of up to 7.87% of queries at 100% CPU utilization (Figure 6d).

The results show that, even with full CPU exhaustion, the attack achieves no full client query loss, i.e., no comprehensive DoS. The key limitation of the attack is that every individual attacker query only causes a relatively minor load on the resolver, leaving ample opportunities to process and reply to client queries in-between the attacker-induced stalling periods.

We measure the number of instructions of all resolvers for 2000 queries over a span of 40s for uncached benign queries and attack queries. In their blog post,⁵ the developers from the Internet Systems Consortium (ISC) mention that the discovery of the NSEC3-encloser attack enables scaling the attack to 125 times as previously thought possible. In theory, the overhead of one attack query is made up of some constant portion (e.g., for querying the authoritative nameserver and verifying the signatures) and the hash calculation. The latter is dependent on the number of iterations, multiplied by the number of enclosed labels in the request (up to 125), roughly multiplied again by the hash operations incurred by hashing the digest plus salt (up to 4 additional hash blocks leading to a factor of approximately 3–5). This leads to an increase of instructions by a factor of up to $125 \cdot 5 = 625$ compared to a single query with a high NSEC3 iteration count and no additional labels/salt. In practice, hash operations are relatively cheap in terms of instructions, especially compared to asymmetric cryptography. Hence, compared to an uncached benign query, which incurs considerable overhead through recursive querying of nameservers, retrieving keys, and validating signatures, we measure an increase of instructions by a factor of 72 for Unbound, 41 for Bind9.18.12, 33 for PowerDNS, and 13 for Knot. The high factor for Unbound is mostly due to the low number of instructions for the benign queries, which is on average 65% lower compared to the other resolvers.

³<https://linux.die.net/man/1/dig>

⁴<https://linux.die.net/man/5/resolv.conf>

⁵<https://www.isc.org/blogs/2024-bind-security-release>

4.5 Comparison to PoC in CVE-2023-50868

Following our evaluation, we also look into CVE-2023-50868, which made the NSEC3 vulnerability public and contains a Proof of Concept (PoC) implementation of the attack.

To the best of our knowledge, neither the CVE-2023-50868, nor related blog posts contain any detailed evaluations of the impact of the attack on different resolvers. We contribute this evaluation, showing that resolvers differ in their vulnerability to the attack. For example, we find that Unbound is more vulnerable to the attack due to its internal scheduling of NSEC3 compared to e.g., PowerDNS.

We further identify the impact of different NSEC3 parameters on the severity of the attack. The PoC correctly identifies that maximizing the iteration count greatly improves impact on resolvers, which we confirm in our evaluations. However, the PoC lacks utilization of a salt value, which we show to also substantially increase the attack impact. Since salts extend the length of the hash-function input, they increase the required computation in every iteration of the hash, significantly increasing effort for the resolver.

We experimentally demonstrate that a query rate in the low hundreds is sufficient to exhaust a single CPU core on unmitigated, open-source resolver implementations at varying degrees. Using the attack, we were not able to achieve full DoS on any resolver. Our findings illustrate that the attack is not as powerful in stalling resolvers as other attacks, such as KeyTrap [7] and find that this is mainly due to the linear scaling of the workload induced relative to the attacker queries, compared to a quadratic increase in load for KeyTrap. However, an attacker can still use the NSEC3 to inflict harm on resolvers and achieve a degradation of service for benign clients using the victim resolver.

5 Measurements of Signed Domains

RFC9276 raises the best practice of omitting the use of both hash iterations and salts. We measured how NSEC3 is used in domains on the Internet and investigate their NSEC3 parameter configurations. To shed a light on how domains conform to RFC9276 and whether they use NSEC3 parameters which are suitable to be exploited in an attack, we next quantify how many domains on the Internet use NSEC3 and which parameter configurations they employ. During the week following 2024-03-10, we queried the nameservers of the Tranco Top-1M domains⁶ for the SOA, DNSKEY as well as DS records (located at the parent) and analyzed the DNSSEC configurations of the domains they serve. To collect information on the NSEC version and parameters used by the domains, as they are presented to the resolvers, we additionally issued queries for the records PTR-type RFC2317 at the according Tranco domain names. PTR-type records are used for reverse-mapping IP addresses to domain names and are

⁶<https://tranco-list.eu/list/Z333G/1000000>

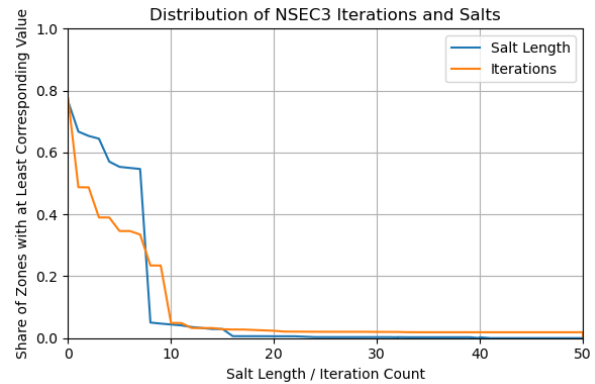


Figure 7: Share of zones which meet or exceed the configured Salt Length / Iteration Count in signed DNS zones.

most commonly located below the IN-ADDR.ARPA. domain. Therefore, we expect negative responses for these queries, indicating that no such resource exists. Our evaluations confirm that this methodology yields negative responses, i.e., containing either first-version NSEC or NSEC3 records, for 98.15% of the signed domains. We find 66339 (6.63%) of the Tranco Top1M domains to be signed. Out of these, 27761 (41.85%) use NSEC3 while 37354 (56.31%) use NSEC in its first version. 21522 (77.53%) of the domains using NSEC3 send records with an iterations count field value higher than 0, with a median of 5 iterations and a maximum of 500 iterations, while 21248 (76.54%) of the domains utilizing NSEC3 employ a salt. Where employed, the median salt length is 8 bytes and the maximum we find in our dataset is 64 bytes. We show the share of zones with salt lengths and iteration counts greater or equal to the respective value on the x-axis in Figure 7. The combination of both parameters, which imposes the highest NSEC3 hashing burden on resolvers is 500 iterations with a salt of 16 bytes length. According to the results of our evaluations, these domains can impose substantial load on the resolvers even with benign responses. Such domains could potentially be abused by adversaries to degrade the service of a vulnerable resolver by employing a moderate volume of malicious queries per second.

6 Related Work

DNS has a long history of Denial of Service (DoS) attacks which exploit different aspects of the DNS protocol to launch attacks against the DNS servers [1, 2, 4, 11]. Many of the attacks exploit a lack of limits on the functionalities performed by the DNS servers. For instance, [4] create a chain of CNAME records and force DNS resolvers to perform deep name resolutions, hence overloading the target victim authoritative nameserver with requests and achieving an amplification of 8.51. NXNSAttack [1] exploited a vulnerability that generated a flood of queries between the recursive resolver and the authoritative server creating a load on them both.

Subsequently [2] showed how to exploit delegations to an unresponsive authoritative server to cause computational load on DNS resolvers. Their attack differs from the NSEC3 attack in that they use plain DNS instead of DNSSEC, and create computational load through memory lookups and IO overhead instead of computational effort. Their attack achieves a higher instruction count amplification of 5600x over 70x with NSEC3. Still, the NRDelegation attack requires a high attack traffic volume of 500 requests per second to achieve substantial degradation of service, likely because the attack includes many IO operations, which allow the resolver to answer benign queries while waiting for IO responses. This explains why the NSEC3 attack, while resulting in a smaller instruction count amplification, can still achieve comparable degradation of service to NRDelegation.

The concept of complexity attacks on DNSSEC, specifically exploiting signature validations and hash computations was first introduced by [7]. Their work showed that DNSSEC was vulnerable to a new class of attacks that can exhaust CPU resources and thereby achieve Denial of Service on any DNSSEC validating resolver. Their most impactful attack, KeyTrap, achieves a full DoS of DNS resolvers for between 2min and 16h by exploiting colliding key-tags and a large number of signatures, leading to quadratic complexity in validating DNSSEC signatures. Further, their findings include an attack exploiting hash computations over the DS hash that connects a parent zone to a child zone. Specifically, in their attack, they include a large amount of DS hash records in the parent zone and point them to a single entry in the child-zone with a specific key-tag value. Exploiting colliding key-tags, they achieve quadratic complexity in hash computations, requiring the resolver to try each DS record in the parent zone against each DNSKEY in the child-zone. This computational effort allows for a DoS of the resolver. The NSEC3-encloser attack that we study in this work differs significantly in its single-request impact from the attacks described in [7]. Comparing to the KeyTrap attack, the NSEC3-encloser attack inflicts a modest 72x increase in CPU instruction count,⁷ while KeyTrap increases CPU instructions by a factor of 2000000x. Thus, with KeyTrap, a single attacker is able to DoS a resolver for an extended period of time, whereas with NSEC3, a large attack traffic volume is necessary, consisting of hundreds of DNS requests per second to exhaust the CPU of a victim resolver. This is expected, as KeyTrap exploits computationally heavy public key cryptography, while NSEC3 only uses hash calculations, which require less CPU resources. However, while requiring more traffic, the NSEC3 attack can still harm DNS resolvers, as it can create a heavy load on the attacked resolver and therefore lead to substantial degradation of service.

Our work is also related to downgrade attacks against DNSSEC [8]. The DNSSEC downgrade attacks however fo-

cus on disabling DNSSEC validation but do not have adverse effects on the availability of the victim resolvers.

7 Conclusions

We perform extensive evaluations of NSEC3-encloser attack and find that it can create a 72x increase in CPU instruction count on victim DNS resolvers. This is much less than the recently disclosed KeyTrap attack, which creates a factor of 2000000 increase in CPU instructions count. Our experimental evaluation shows that even the improved implementation of the NSEC3-encloser attack that we developed creates a relatively minor packet loss (between 2.7% and 30% depending on the resolver implementation), yet requires a high traffic volume from an adversary and can be easily detected. Therefore we do not expect to see such attacks in the wild. Nevertheless, our study shows that NSEC3-encloser attack points to a potential problem in the resolvers, that was also raised by the NSEC standard specification. In this work, we explore the practical aspects of NSEC across DNS resolver implementations.

We experimentally analyze the role of the different parameters in NSEC3 on the load created on the resolvers and show how to adjust the parameters to optimize the impact of the attack. Although the increase in CPU instruction set is lower than previous attacks on DNS, such as KeyTrap or NRDelegation, using about a hundred packets per second, the adversary can still create a sufficient load on the resolvers, eventually leading to packet loss. The load is created by the iterative application of the hash in NSEC3, and is further exacerbated by the application of salt to the computation of the hash. Multiple hash iterations with salt make zone enumeration attacks more difficult, requiring more resources from the attackers.

Such records can be exploited to exhaust resources on victim resolvers, as we experimentally demonstrate in this work. The effect of resource exhaustion may become even more severe with the new proposal NSEC5 which uses public key operations [12]. Our research essentially shows that there is a tradeoff between the privacy and the load on DNS resolvers, which can be exploited for attacks. This tradeoff is also aligned with the question raised by RFC9276: do the increased performance costs justify applying additional hash operations.

As RFC9276 points out, most of the names published in DNS are typically public and are rarely secret or unpredictable. RFC9276: “*They are published to be memorable, used and consumed by humans. They are often recorded in many other network logs such as email logs, certificate transparency logs, web page links, intrusion-detection systems, malware scanners, email archives, etc. Many times a simple dictionary of commonly used domain names prefixes (www, mail, imap, login, database, etc.) can be used to quickly reveal a large number of labels within a zone.*”

The fundamental question of the tradeoff between privacy of the resources in the DNS zones vs load on the DNS re-

⁷Measured on Unbound, average over 5 measurements

solvers poses an important decision that the research and operational community need to take.

Acknowledgements

This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) SFB 1119.

References

- [1] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. NXN-Attack: Recursive DNS inefficiencies and vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 631–648. USENIX Association, 2020.
- [2] Yehuda Afek, Anat Bremler-Barr, and Shani Stajner. NRDelegationAttack: Complexity DDoS attack on DNS recursive resolvers. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3187–3204. USENIX Association, 2023.
- [3] Jason Bau and John C Mitchell. A security evaluation of DNSSEC with NSEC3. *Cryptology ePrint Archive*, 2010.
- [4] Jonas Bushart and Christian Rossow. DNS unchained: Amplified application-layer DoS attacks against DNS authoritative. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 139–160. Springer, 2018.
- [5] Sharon Goldberg, Moni Naor, Dimitrios Papadopoulos, Leonid Reyzin, Sachin Vasant, and Asaf Ziv. Stretching NSEC3 to the limit: Efficient zone enumeration attacks on NSEC3 variants. Technical report, Boston University, 2015.
- [6] Olivia Gruza, Elias Heftrig, Oliver Jacobsen, Haya Schulmann, Niklas Vogel, and Michael Waidner. Goethe-Universitat-Cybersecurity/NSEC3-Encloser-Attack: WOOT’24 Artifact, May 2024.
- [7] Elias Heftrig, Haya Schulmann, Niklas Vogel, and Michael Waidner. The Harder You Try, The Harder You Fail: The KeyTrap Denial-of-Service Algorithmic Complexity Attacks on DNSSEC. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.
- [8] Elias Heftrig, Haya Shulman, and Michael Waidner. Downgrading DNSSEC: How to Exploit Crypto Agility for Hijacking Signed Zones. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7429–7444, 2023.
- [9] Himanshu Kumar, Sudhanshu Kumar, Remya Joseph, Dhananjay Kumar, Sunil Kumar Shrinarayan Singh, Ajay Kumar, and Praveen Kumar. Rainbow table to crack password using MD5 hashing algorithm. In *2013 IEEE Conference on Information & Communication Technologies*, pages 433–439. IEEE, 2013.
- [10] Harrison Mitchell. Taking the DNS for a walk; NSEC3 prevalence and recoverability.
- [11] Giovane C. M. Moura, Sebastian Castro, Wes Hardaker, Maarten Wullink, and Cristian Hesselman. Clouding up the internet: how centralized is DNS traffic becoming? In *Internet Measurement Conference*, pages 42–49. ACM, 2020.
- [12] Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Včelák, Leonid Reyzin, and Sharon Goldberg. Making NSEC5 practical for DNSSEC. *Cryptology ePrint Archive*, Paper 2017/099, 2017. <https://eprint.iacr.org/2017/099>.

Amplifying Threats: The Role of Multi-Sender Coordination in SMS-Timing-Based Location Inference Attacks

Evangelos Bitsikas^{*}, Theodor Schnitzler^{†‡}, Christina Pöpper[§], Aanjhan Ranganathan^{*}
^{*}*Northeastern University*, [†]*Research Center Trustworthy Data Science and Security*,
[‡]*Maastricht University*, [§]*New York University Abu Dhabi*
bitsikas.e@northeastern.edu, theodor.schnitzler@maastrichtuniversity.nl,
christina.poepper@nyu.edu, aanjhan@northeastern.edu

Abstract

SMS-timing-based location inference attacks leverage timing side channels to ascertain a target’s location. Prior work has primarily relied on a single-sender approach, employing only one SMS attacker from a specific location to infer the victim’s whereabouts. However, this method exhibits several drawbacks. In this research, we systematically enumerate the limitations of the single-sender approach, which prompted us to explore a multi-sender strategy. Our investigation delves into the feasibility of an attacker employing multiple SMS senders towards a victim to address these limitations and introduces novel features to bolster prediction accuracy. Through exhaustive experimentation, we demonstrate that strategically positioned multiple SMS senders significantly enhance the location-inference accuracy, achieving a 142% improvement for four distinct classes of potential victim locations. This work further highlights the need to develop mitigations against SMS-timing-based location inference attacks.

1 Introduction

SMS (Short Message Service) has emerged as a key vector in numerous cyber-attacks due to its widespread use for purposes such as two-factor authentication [21], identity verification [24, 25], and emergency alerts [24, 25]. Its prevalence, reliability, and global reach have made it a favored medium for malicious activities. Smishing attacks, for example, leverage SMS to distribute links that direct victims to phishing sites, aiming to steal sensitive information [14]. The Flubot virus utilized SMS links to spread trojan apps that compromised banking credentials, personal data, and disabled security features [9]. Beyond these, SMS has been exploited for spamming [8] and to propagate malware such as Simjacker and WIBAttack, which embed malicious commands within binary SMS messages [4, 28].

Most recently, a novel approach to ascertain the location of recipients was demonstrated in [6], utilizing the timing of silent SMS messages in conjunction with machine-learning techniques. This strategy exploits the delivery reports generated upon SMS reception as a timing attack vector for the sender. Rigorous experimentation across various countries, telecommunications operators, and a range of devices demonstrated that an attacker could deduce a recipient’s location by analyzing timing data from typical receiver locations. Although this method introduces an innovative side channel for localizing mobile users, it encounters notable limitations. Most importantly, there is a significant probability that the attack originating from a single source/mobile device can be detected and potentially be blocked by the victim’s service providers. This is more apparent when the attack requires a substantial amount of SMS transmissions to collect the necessary data. Additionally, as the number of possible victim locations increases, the method’s accuracy in predicting locations degrades due to the finite entropy available from single attacker-victim channel timing reports. As a result, there are classifications in which machine learning can perform poorly.

To tackle the above-mentioned limitations associated with single-sender-based SMS location inference attacks, this paper focuses on the following key research questions. The primary question we explore is whether using multiple coordinated SMS senders can improve the accuracy of localization predictions. We hypothesize that using senders from different locations could create unique timing side-channels which, when combined, could lead to more accurate classifications. This multi-sender approach can improve the prediction accuracy, especially as the number of potential victim locations increases. Additionally, using multiple SMS senders spread out geographically could also make the attack more resilient against being blocked, as the victim’s service provider now has to identify and block several senders. Optimizing the timing and pattern of SMS

sending could further reduce the likelihood of the attack being detected. Finally, we hypothesize that the attacker can collect a significantly smaller amount of data to conduct this attack efficiently, without compromising the model’s accuracy. Consequently, the adversary can save resources, as well as measurement collection and training time.

Motivated by the above hypothesis, in this paper, we make the following contributions:

- We identify limitations of single-sender SMS-timing-based location inference attacks and conceive *multi(ple)-sender SMS-timing-based location inference attack* in cellular networks. To establish a baseline for comparison with our multi-sender approach, we reproduced the single SMS sender-based localization attack described in prior work [6]. Interestingly, our data analysis highlights certain limitations inherent in the single-sender approach which serve as a crucial motivation for the development of our multi-sender approach.
- Through rigorous experimentation, we demonstrate the enhanced capability of multiple SMS senders, strategically placed across different locations, to coordinate and significantly improve the accuracy in determining a victim’s location. Our experiments reveal that the multi-sender MMS approach can reach up to 142% accuracy improvement for four classes. This further emphasizes that the effectiveness of the multi-sender attack strategy improves with an increasing number of potential victim locations, thereby overcoming a significant limitation of the single-sender approach.
- We highlight two substantial improvements and insights: (1) From the distinct timing side-channels generated by the multi-sender setup, we identify and introduce *new features* that are instrumental in boosting the prediction accuracy: the statistical mean, median, and standard deviation of the senders’ delivery time measurements, allowing us to effectively fuse the timings from multiple senders to improve the accuracy even further. (2) We investigate the required *sample sizes* for location inference attacks and demonstrate that already a few hundred SMS can yield strong results without the need for thousands of collected messages.

2 Background and Motivation

In this section, we provide the technical background for SMS delivery processes and then delve into the concept of *SMS-timing-based Location Inference Attacks*. We

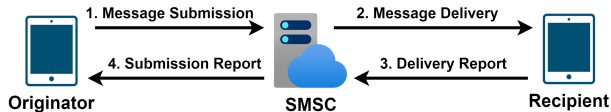


Figure 1: Brief representation of the SMS process, according to GSMA [11].

subsequently outline its limitations, which serves as the foundation for our research presented in this paper.

2.1 Overview on SMS Process

Short Message Service (SMS) is an inherent component of the cellular infrastructure and universally accessible across all network generations from 2G to 5G [1–3, 11]. Figure 1 briefly outlines the SMS delivery process involving the originator (sender), Short Message Service Center (SMSC), and the recipient (receiver).

The process begins with the message submission (Step 1) by the originator, who composes the message and sends it to the SMSC. Upon receiving the SMS, the SMSC performs the necessary network and validation checks and then forwards the SMS to the intended recipient. The SMSC ensures that the message reaches the recipient (Step 2), even if it means storing it temporarily, in case the recipient is unavailable immediately. Additionally, the originators have been informed by now that the submitted message was actually sent.

Next, once the recipient receives the message, the involved device sends the delivery report back to the SMSC. The report confirms that the message has been successfully delivered to the recipient’s device (Step 3). Finally, the report is sent to the originator via the SMSC, called the submission report (Step 4). This report ultimately confirms that the message was sent and delivered to the recipient successfully.

2.2 SMS-timing-based Location Inference

In an SMS-timing-based Location Inference attack, an attacker is interested in learning the current physical location of a specific victim by sending them (silent) SMSes. The attack builds upon the time elapsed between sending the SMS and the SMS being delivered to the victim and is conducted in two phases.

In the first phase (fingerprint generation), the attacker repeatedly sends SMSes to the victim while knowing their respective locations and measures the time it takes to deliver the SMS messages. By analyzing the resulting delivery timings and their distributions, the attacker is able to determine a unique fingerprint for each of the locations the victim has visited.

In the second phase (location inference), the attacker sends new SMS messages to the victim *without* knowing their current location, measures the time it takes to deliver them, and then *classifies* the collected timings by comparing them to the previously obtained fingerprints. Thus, the attacker can determine and re-identify the victim's location out of a set of known locations.

2.3 Limitations and Motivation

When the SMS-timing-based Location Inference Attack is carried out from a single sender at a fixed location, it has several drawbacks. In particular, the success and performance of the attack depend heavily on the specifics of the chosen location and its mobile network connection, such as the distance to the base station. The quality and reliability of the connection, along with the robustness of the collected data, may also vary depending on circumstances specific to the location, such as fluctuating numbers of people and concurrent mobile network connections throughout the day or week.

Another drawback is that during the initial phase of the attack (fingerprint generation), the attacker engages in non-standard behavior as a mobile network subscriber. Consequently, there is a risk that the adversary may be perceived as suspicious by the network operator and potentially be blocked, particularly if only a single static location is utilized.

From an organizational perspective, the attack outlined in [6] encompasses analyses at various levels of granularity, and a broad range of locations, from regional to worldwide attacks. However, the study lacks a thorough analysis of the sample size impact regarding the classification accuracy. This limitation implies that the attack requires additional evaluation.

Hence, we recognize the necessity for a more systematic evaluation of factors that could impact the SMS-timing-based Location Inference Attack's performance. This entails varying the adversary's location, systematically assessing the attack's performance with different receiving devices at the same locations, conducting repeated evaluations with varying sample sizes, and expanding the attack to encompass attackers operating from multiple vantage points simultaneously.

3 Multi-Sender Location Inference

3.1 Threat Model

We consider an attacker whose primary goal is to determine the presence of a victim's mobile device within a specific geographic area, without the intention to track the victim's exact movements.

The attacker is presumed to possess the victim's mobile number, enabling them to initiate various forms of SMS communications, including personal messages, undirected mass messages such as marketing advertisements, and notably, silent SMSes which the victim's device acknowledges without alerting the user. It is assumed that the attacker has access to an arbitrary number of smartphone devices, SIM cards, mobile numbers, and subscription plans. Furthermore, the attacker can deploy multiple sender devices in different geographical areas to collect data from the victim receivers simultaneously and combine them for location extraction. The adversary is assumed to possess the capability to utilize network services as a *conventional* user: leveraging several SIM cards, having the ability to send messages to any subscriber with a valid number, and maintaining a normal connection for the transmission of text messages and receipt of delivery notifications.

We emphasize that the attacker does *not* require physical access to the victim's mobile device, USIM cards, or any network infrastructure, nor do they seek to obtain or modify sensitive victim data such as cryptographic keys.

3.2 Attack Concept

The foundation of the multi-sender approach rests on the observation that fingerprints generated from the SMS exchanges between a single sender (attacker) location and a receiver can be limited in their effectiveness for accurate location classification. This limitation becomes particularly pronounced in complex environments, such as certain German locations in [6], where the variability and granularity of the urban landscape can dilute the distinctiveness of timing fingerprints.

To address these challenges, this work pioneers the integration of multiple attacker locations into the analysis framework. By orchestrating SMS exchanges from various (unique) attacker positions to the receiver, a richer and more nuanced dataset emerges. Each unique pairing of attacker and receiver locations contributes a distinct timing fingerprint to the dataset. These timing fingerprints, when aggregated, undergo further processing to distill additional dataset features, thereby forging more robust and comprehensive fingerprints. This enriched dataset plays a crucial role in enhancing the efficacy of machine-learning models during both the training and prediction phases.

For conducting a multi-sender location inference attack, we essentially replicate the attack methodology presented in [6] and simultaneously execute it from multiple locations. Consistent with previous work, the attack comprises two phases: fingerprint generation and location inference, but both are conducted from multiple sender locations. Basically, multiple instances of the

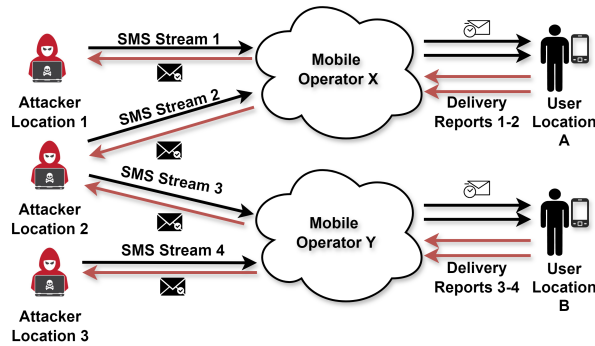


Figure 2: Multiple attackers in different locations establish SMS streams to send silent messages to the victim in various locations and receive delivery reports. This is possible even with distinct network providers.

single-sender location inference attack are executed in parallel.

Multi-Sender Setup. To gather data from multiple vantage locations and eventually enhance the accuracy of the location identification attack, the attacker deploys the setup at various geographical locations. Intuitively, by employing more attacking locations that are diverse, an adversary could generate more precise receiver location fingerprints. This distributed approach allows the attacker to collect measurements of the victim’s location from different “angles”, increasing the robustness and reliability of the subsequent analysis.

Attacking Process. The attacker, situated in multiple locations, initiates the process by sending a barrage of silent SMS messages to the victim. The victim, unknowingly participating in this scheme, moves across different locations at different times. The silent nature of these messages means that the receiver’s device does not notify the victim of the incoming SMS, thus keeping the process clandestine. Each time a message is received, the victim’s device automatically generates and sends back delivery reports as part of its standard operating procedure. These reports, unbeknownst to the victim, reveal valuable information for the attacker, notably the sent and delivered times. By analyzing the time discrepancies between when a message was sent and when the delivery report was received, the attacker can infer certain aspects of the victim’s location.

Since this procedure is repeated multiple times in the multi-sender attack, it accumulates a substantial dataset of measurements. The attacker categorizes the measurements based on the victim’s known locations during the attack, forming distinct datasets for each location. These datasets are then aggregated and analyzed to predict the victim’s location in the future. According to Figure 2, the attacker creates several SMS streams, which could be es-

tablished with different operators since the attacker can operate from different countries. The victim may also move to different countries and sends back the delivery reports to the corresponding SMS.

In the prediction stage, the attacker collects fresh measurements from the current location of the victim in the same fashion. These measurements serve as input for a machine-learning model that has been trained on the previously collected data, representing potential locations of the victim. Then, the model processes this input and outputs a prediction of the victim’s current location.

4 Experimental Validation

In this section, we detail our experimental validation of the SMS-timing-based location inference attack with multiple senders and report on our setup for data collection, processing, and evaluation.

4.1 Data Collection Setup

At the core of the attacker’s setup is the use of typical computer devices equipped with a smartphone running Android Debug Bridge (ADB). ADB allows for a wide range of communication with a connected device, in this case, to transmit silent SMS messages and record the sent and delivered timestamps. As in [6], the SMS transmission and recording of the timing metrics is conducted by an Android application, which also stores results for further processing. Controlling the application via ADB allows us to automate this process since it should be repeated multiple times to collect a sufficient number of timing metrics. This process also happens stealthily, without altering the victim, since the attacker utilizes silent SMSs which are accepted by the network operator. Moreover, the attacker’s equipment includes a SIM card, granting access to the cellular network.

Adhering to the aforementioned attacking concepts, over a period of 12 weeks, we repeatedly send SMS messages between smartphones in different locations in Germany and the Netherlands. We do not consider locations that are very far apart, as they are easier for an attacker to identify [6]. We use three smartphones, each placed in a fixed location that remains unchanged during the experiments, to send messages to four phones whose positions are periodically rotated. For sending SMS messages, we use two locations in Germany and one in the Netherlands. The receiving phones are placed in five different locations in Germany and three in the Netherlands (including the locations of the sending devices). Table 1 lists the devices we used for sending and receiving SMS messages, and Table 2 provides an overview of the locations used during our measurements and the amounts of data collected.

Table 1: Device Specifications

ID	Device	Chipset	OS	Model	Release
<i>Sending Devices</i>					
D	Samsung Galaxy A53	Samsung Exynos 1280	Android 12	SM-A536E/DS	2022
V	Nokia 5.3	Qualcomm Snapdragon 665	Android 11	TA-1234	2020
B	Huawei P8 Lite 2017	HiSilicon Kirin 655	Android 8	PRA-LX1	2017
<i>Receiving Devices</i>					
px6a	Google Pixel 6a	Google Tensor	Android 12	G1AZG	2022
a53	Samsung Galaxy A53	Samsung Exynos 1280	Android 12	SM-A536E/DS	2022
op7	OnePlus 7 Pro	Qualcomm Snapdragon 855	Android 11	GM1910	2019
p8l	Huawei P8 Lite 2017	HiSilicon Kirin 655	Android 8	PRA-LX1	2017

Table 2: Data Collection Summary

	Number of SMS per Receiving Device				Distances [km] to Sender		
	px6a	p8l	op7	a53	Sender B	Sender D	Sender V
<i>Receiver Locations in Germany</i>							
DE-1	3160	3280	420	–	11	0	140
DE-2	1540	1560	–	–	2	11	130
DE-3	4960	4540	8920	6900	0	11	129
DE-4	420	460	–	–	4	14	126
DE-5	1220	320	–	–	5	11	140
<i>Receiver Locations in the Netherlands</i>							
NL-1	7140	5500	0	1440	125	135	4
NL-2	5820	5280	10300	8700	129	140	0
NL-3	2020	960	1680	1120	125	136	7

Locations (Cities): *DE-1,5*: Dortmund, *DE-2,3,4*: Bochum, *NL-1*: Eindhoven, *NL-2*: Veldhoven, *NL-3*: Valkenswaard

Locations in the same country are chosen to be relatively close to each other. The distance from a receiving location to the *closest* sending device is 11 km at maximum, which also corresponds to the distance between the two sending devices in Germany.

4.2 Data Collection Procedure

We replicated the attack to use an Android app that sends one silent SMS at a time to a designated target phone number. Additionally, the app waits for the *Sent* and *Delivered* notifications and collects and stores the timestamps for the SMS transmission and both notifications. In line with previous work, we schedule 20 consecutive SMS transmissions on an hourly basis. We automate SMS transmissions by controlling the app remotely via a Python script issuing ADB commands to the smartphone. We simultaneously send SMS messages from all senders to the same receiver by scheduling the script to start once per hour at the same time for a specific receiver (i. e., :00 for the first receiver, :15 for the second receiver, ...) across all senders. While this does not

guarantee perfect sender synchronization due to potential offsets in their individual system clocks, we consider this a best-effort approach to approximate the behavior of an adversary simultaneously probing a specific target from multiple locations.

Our data collection tooling builds upon the code released by Bitsikas et al. [6] available on GitHub¹ and is extended to fit with the phones we use for sending messages. We also follow the guidelines provided along with the framework to implement the missing code handling the actual SMS transmission and timestamp collection procedures.

4.3 Feature Set Generation & Multi-Sender Fusion

To generate the timing features for each SMS transmission and combine the multi-sender datasets, we take the following steps:

¹<https://github.com/vaggelis-sudo/SMS-Location-Identification-Attack>

Step 1: Calculating the initial metrics. Following [6], we calculate the initial metrics for each SMS transmission in the collected dataset: the real sent duration T_{sent} , the real delivery duration T_{del} , the total delivery duration T_{tot} , and the delivery ratio P .

$$T_{sent} = t_{sent} - t_{tx} \quad (1)$$

$$T_{del} = t_{del} - t_{sent} \quad (2)$$

$$T_{tot} = T_{del} + T_{sent} \quad (3)$$

$$P = \frac{T_{del}}{T_{tot}} = \frac{t_{del} - t_{sent}}{t_{del} - t_{tx}} \quad (4)$$

Then, for every two consecutive SMS transmissions ($j-1$ and j), we calculate the differences in sent duration $T_{\Delta sent}$ and delivery duration $T_{\Delta del}$, respectively:

$$T_{\Delta sent} = (T_{sent}^j - T_{sent}^{j-1})/T_{sent}^{j-1} \quad (5)$$

$$T_{\Delta del} = (T_{del}^j - T_{del}^{j-1})/T_{del}^{j-1} \quad (6)$$

Moving beyond [6], the fingerprint does not conclude with this calculation, as we do not consider only one but multiple senders.

Step 2: Combining the sender datasets. Let D_i represent the dataset for sender i , where $i = 1, 2, \dots, m$, with n receiver locations. Additionally, let $t_{del,i,r,j}$ denote the delivery time of the j -th SMS transmission from sender i to receiver r . Finally, let $S_{i,r,j}$ represent the data associated with the j -th SMS transmission from sender i to receiver r , including $t_{del,i,r,j}$. Then, D_{concat} is the dataset resulting from the concatenation process, where each element is derived by matching $S_{i,r,j}$ from all senders based on the closest matching $t_{del,i,r,j}$.

For each $S_{i,r,j}$ in D_i , we seek to find $S_{k,r,l}$ in D_k ($k \neq i$) such that the difference in delivery times $|t_{del,i,r,j} - t_{del,k,r,l}|$ is minimal or zero, indicating the closest matching timestamps across different senders. This process occurs for every receiver separately and every available sender, until the new D_{concat} dataset contains per row the data of each sender to the same receiver, but synchronized. Algorithm 1 shows briefly the process.

Step 3: Fusing the sender datasets statistically. Given m senders, the number of unique combinations of two senders is given by the binomial coefficient:

$$\binom{m}{2} = \frac{m!}{2!(m-2)!} \quad (7)$$

For each pair of senders and for every z consecutive SMS transmissions (in this study, $z = 5$ ²), we calculate the Mean, Median, and Standard Deviation of

²We determined that the number should be less than 10 in our dataset to accommodate small sample sizes while not covering too many transmissions at a time. A middle value of 5 was chosen as a result.

Algorithm 1 Match and Concatenate SMS Transmissions based on Timestamps

```

1: Initialize  $D_{concat} = \emptyset$  as empty dataset
2: for each receiver location  $r$  from 1 to  $n$  do
3:   for each  $S_{i,r,j}$  in  $D_i$  for all  $i$  do
4:     Initialize a list  $L_{i,r}$  to hold data for concatenation
5:     for each  $D_k$  where  $k \neq i$  do
6:       Find  $S_{k,r,l}$  in  $D_k$  such that  $|t_{del,i,r,j} - t_{del,k,r,l}|$  is minimized
7:       Add  $S_{k,r,l}$  to  $L_{i,r}$ 
8:     end for
9:      $NewRecord_{i,r} \leftarrow Concatenate(L_{i,r})$ 
10:     $D_{concat} \leftarrow D_{concat} \cup \{NewRecord_{i,r}\}$ 
11:    Clear  $L_{i,r}$ 
12:  end for
13: end for

```

the delivery times. Let $t_{del,i}^{(s,r)}$ denote the delivery time of the i -th SMS in a sequence of z consecutive messages from sender s to receiver r . The statistics are calculated as follows:

$$\mu^{(s,r)} = \frac{1}{z} \sum_{i=1}^z t_{del,i}^{(s,r)} \quad (8)$$

$$\text{Median}^{(s,r)} = \text{Median}\{t_{del,1}^{(s,r)}, t_{del,2}^{(s,r)}, \dots, t_{del,z}^{(s,r)}\} \quad (9)$$

$$\sigma^{(s,r)} = \sqrt{\frac{1}{z-1} \sum_{i=1}^z (t_{del,i}^{(s,r)} - \mu^{(s,r)})^2} \quad (10)$$

Differences in these statistics for the delivery time between pairs of senders are calculated as their actual differences. For example, for means between sender pair (s_1, r) and (s_2, r) :

$$\Delta\mu^{(s_1,s_2,r)} = \mu^{(s_1,r)} - \mu^{(s_2,r)} \quad (11)$$

These differences, $\Delta\mu^{(s_1,s_2,r)}$, $\Delta\text{Median}^{(s_1,s_2,r)}$, and $\Delta\sigma^{(s_1,s_2,r)}$, are incorporated into the dataset for each sender pair accordingly, as additional features.

4.4 Multi-Sender Techniques

Simple Integration of Senders. In this method, the initial features are generated based on the timing data from individual sender-receiver pairings (Step 1). Subsequently, datasets corresponding to multiple senders are amalgamated (Step 2) *without* the application of sophisticated statistical fusion techniques (Step 3). Thus, we create datasets that are matched and concatenated based on the timestamps, but without incorporating unique feature types.

Specifically, we consider double- and triple-sender datasets as distinct (simple) approaches. For the double-sender cases, we create the BV, VD, and BD datasets,

while for the triple-sender cases, we create BDV, based on Table 2. The total number of features for double-senders is 12, and for triple-senders is 18, according to Algorithms 1-6 from Step 1. This exploratory step seeks to discern whether straightforward sender concatenation can bolster the machine-learning model's predictive accuracy compared to single senders and to statistically combined datasets.

Statistical Fusion of Senders. Advancing beyond the simple approach, the statistical combination of sender datasets represents a more refined approach to dataset enhancement. This technique encompasses a comprehensive process involving the generation of initial features (Step 1), the combination of sender measurements (Step 2) followed by the fusion of datasets from multiple senders through the statistical metrics (Step 3). Unlike the simple method, this approach enriches the combined dataset with additional features derived from the statistical analysis of delivery times: using the means, medians, and standard deviations between the sender measurements. For this approach, we use all three senders with their maximum sample size available for each receiver location.

In this work, we explore the following two strategies:

1. **Enhanced Mean Datasets.** Datasets statistically enhanced by the *mean* of the delivery time. A total number of 21 features is used, corresponding to the 18 combined features for the three senders and the 3 additional ones generated by the differences between the sender means.
2. **Enhanced MMS Datasets.** Datasets statistically enhanced by the *mean*, *median* and *standard deviation* of the delivery time. A total number of 27 features is utilized, correlated with the 18 combined features for the three senders and the 9 extra ones engendered by the differences between the sender means, medians, and standard deviations.

This dual-strategy approach aims to demonstrate the superiority of statistically enhanced datasets over both single-sender datasets and those trivially combined. The hypothesis posits that the inclusion of a broader array of statistical features not only increases the accuracy of location predictions beyond that achievable with simpler dataset combinations but also highlights the comparative advantage of the "Enhanced MMS" over the "Enhanced Mean" approach. This distinction underscores the principle that the depth and complexity of features within the dataset are pivotal to the refinement of model accuracy.

4.5 Attack Training & Prediction

In this study, we employ a Multilayer Perceptron (MLP) Classifier, a type of feedforward artificial neural net-

work, as the core predictive model to analyze the relationship between the features derived from SMS transmission data and the target outcomes. The MLP Classifier is instantiated with a specific configuration of hyperparameters to optimize its performance for the given dataset. The architecture of the neural network is defined by hidden layer sizes = (10, 40, 10), indicating a three-layered structure where the input data is first processed by a layer of 10 neurons, followed by a denser layer of 40 neurons, and finally, the information is aggregated through a layer of 10 neurons before reaching the output layer. This configuration is designed to capture the nonlinear relationships between the input features.

The model utilizes the stochastic gradient descent (SGD) algorithm for optimizing the network's weights. This choice is motivated by SGD's efficiency in handling large datasets and its capability to escape local minima during training. The regularization term, $\alpha=0.0001$, is set to a low value to prevent overfitting while allowing the model to learn complex patterns in the data. With learning rate='constant' and a max iteration of 5000, the learning rate is kept fixed across all epochs of training, and the model is allowed a substantial number of iterations to converge towards an optimal set of weights. Batch processing is employed with a size of 32 to leverage computational efficiency and stability in gradient descent updates. Model evaluation is conducted through a 10-fold cross-validation process providing a robust estimate of the model's predictive accuracy on various random data. Finally, the Accuracy metric is calculated to quantify the model's performance, offering a measure of how often the model predictions match the true labels. In our experimentation, we repeatedly run the model prediction with increasing numbers of samples per class, (i.e., 100, 200, 300, 500, 1000, 5000, and 10000), to analyze differences in the classification accuracy.

5 Experimental Evaluation and Results

We next describe the exact experimental setup we used in our experiments and then delve into our results with the multiple-sender approaches.

5.1 Single Senders: Baseline

We ran the classifications for single senders (D, B, and V) to establish the baseline for the subsequent improvement. Figure 3 illustrates the results of all classifications for all sample sizes. Generally, the lowest accuracy is observed for sender D on the device p8l with 5 classes (21%), while the highest accuracy is observed for sender B on the device op7 with 2 classes (82%). In fact, we make similar observations for the single sender classifications with [6], regarding the average accuracy scores

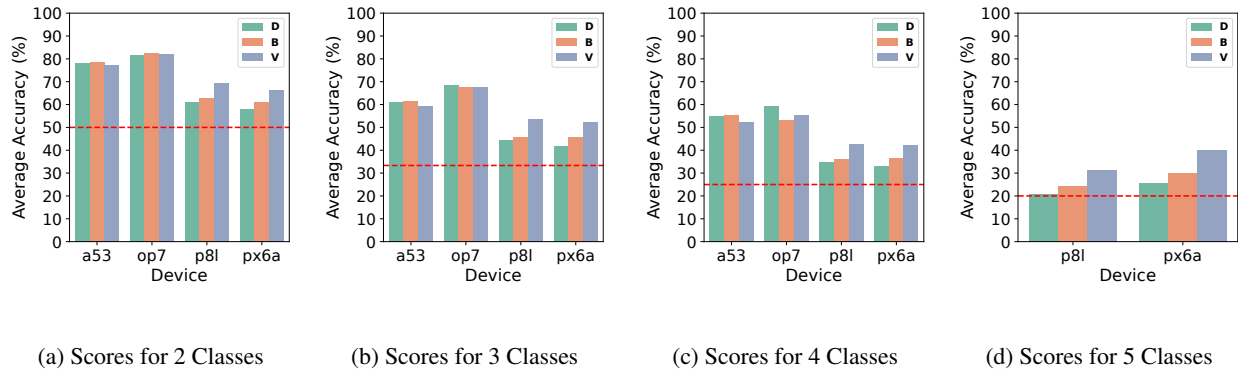


Figure 3: Average single-sender accuracy scores across devices and classes. These scores are considered the established baseline for which we provide improvement. The presented results take into account all possible sample sizes. The red dashed line indicates random guessing.

and the decline across the increasing number of classes. Specifically, for each device examined ranging from a53 to px6a, the data showcases a nuanced relationship between the number of classes involved in the classification task and the single sender accuracy scores. Notably, as the number of classes increases, a general trend of decreasing accuracy is observed, which is consistent across all devices. This trend is particularly evident when comparing results from 2-class configurations to those with 4 or 5 classes, where the average accuracy scores tend to diminish, highlighting the increased complexity and challenges associated with classifying a larger number of classes. Moreover, some devices and senders exhibit a more graceful degradation in accuracy as more classes are added. For example, V on px6a degrades from 66% with 2 classes to 40% with 5 classes, a relatively modest decline compared to D on p8l, which plummets from 61% with 2 classes to 21% with 5 classes.

In the comparative analysis of device performance, the op7 and a53 models significantly outperform the p8l and px6a devices across all metrics. In particular, the p8l and px6a devices achieve a maximum accuracy of 69% and 66%, respectively, when tested with sender V. Furthermore, sender V consistently surpasses senders B and D in performance on the p8l and px6a devices, highlighting a notable disparity in efficacy. Conversely, when evaluating the performance on the op7 and a53 devices, the results among senders B, D, and V demonstrate a remarkable uniformity, with only minimal variations in accuracy. The most significant discrepancy observed is a 6% difference between senders B and D when assessed with four classes on the op7 device. This suggests that while op7 and a53 provide more consistent and higher performance across different senders, p8l and px6a exhibit limitations, particularly in terms of accuracy and sender variability. Consequently, sender V not only shows higher accuracies across the board but

also appears to be more resistant to accuracy drops as the number of classes increases. This suggests that V's data might be inherently more separable or that V employs more consistent patterns in location-related behavior. Overall, the presence of differences in performance between the senders within the same device and class configuration underscores the variability in sender effectiveness.

5.2 Multiple Senders: Simple Combination

In this subsection, we start by comparing the double- and triple-sender accuracy scores with the single-sender scores. In Figure 4, we show all classification accuracy scores with the worst (minimum) and best (maximum) performances of the single- and double-sender data, across all devices and sample sizes. The aim here is to show the minimum and maximum improvement of the multi-senders with simple combinations, based on this collected dataset.

Reflecting on previous discussions, sender V consistently emerges as the top performer across all metrics, capturing both the lowest and highest scores. However, this trend does not uniformly extend to scenarios involving double- and triple-sender configurations. Initially, all multi-sender combinations yield superior accuracy rates compared to individual efforts by senders B and D, underscoring the premise that pooling sender data can enhance overall performance. Notably, in binary classification tasks, sender V is marginally eclipsed by combinations such as DV, BV, BD, and BDV, and similarly by DV, BV, and BDV in contexts involving three and four classes. On the contrary, the BD pairing underperforms for three and four classes, highlighting that sender D's contributions do not bolster the collective accuracy to the same extent as other senders in these specific instances.

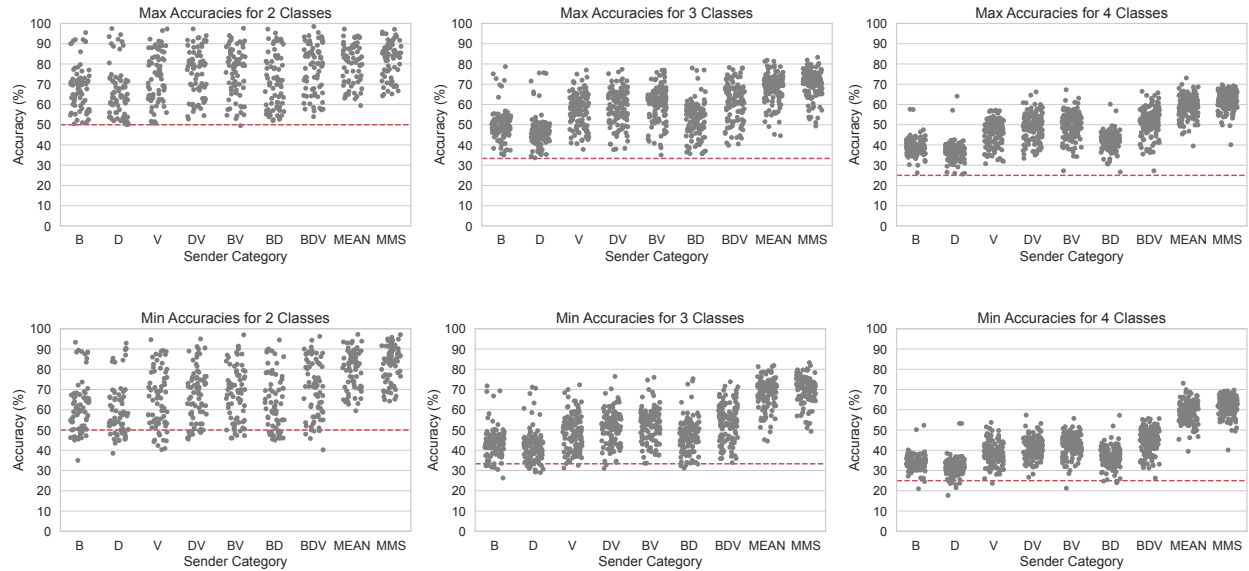


Figure 4: The scatter plots illustrate the accuracy points between different sender types and classes. All devices and sample sizes are considered. The plots with the **minimum** accuracy scores take into account the worst performance of the single- and double-sender data, while the **maximum** accuracy scores focus on the best possible (in this setup).

This phenomenon underscores a critical insight: a sender with generally lower performance can, in certain conditions, detrimentally impact the collective accuracy of multi-sender configurations.

To illustrate the enhancements in accuracy we achieved by integrating multi-sender data over single-sender benchmarks, we included Figure 5. This figure highlights the maximal accuracy improvements realized in our study for configurations involving two and three senders combined. It provides a detailed examination of the specific devices engaged in our experiments and quantifies the average accuracy enhancement across different class numbers. For each classification category, we pinpointed the lowest accuracy scores from single-sender scenarios and juxtaposed these with the highest-performing scores from multi-sender configurations across all sample sizes. This approach was designed to showcase the performance improvements achievable with multi-sender strategies within our dataset. The underlying principle is that the attacker can always adapt the classifications by choosing the best-performing multi-sender combination.

The analysis reveals that for devices a53 and op7, enhancements from multi-sender configurations are relatively modest for binary classifications. This is attributed to the already high performance of single-sender setups in these instances (as detailed in Figure 3). However, the narrative shifts significantly for classifications involving three and four classes, where we observe improvements

of approximately 20%. The scenario is even more pronounced for the p8l and px6a devices, which exhibit progressively larger gains in accuracy with an increase in the number of classes. Notably, the peak improvement recorded is an impressive 120% for the px6a device within four-class scenarios using three senders (namely, the BDV combination).

This data suggests a clear trend: *Classifications that initially present lower accuracy in single-sender formats tend to benefit substantially from the incorporation of multi-senders, particularly in multi-class classifications.*

5.3 Multiple Senders: Statistical Combination

In this subsection, we delve into a comparative analysis between the performance of individual senders and the aggregated results from multiple senders, specifically focusing on the statistically enhanced Mean and MMS datasets. These datasets incorporate data from all three senders at their largest sample sizes, representing the best dataset advancements explored in this study.

By observing Figure 4 once more, it becomes apparent that the Mean and MMS datasets exhibit superior performance for binary classifications compared to other methodologies. This is particularly noticeable in their minimum accuracy scores, which significantly exceed those achieved by alternative approaches. The gap between the Mean and MMS datasets is relatively narrow,

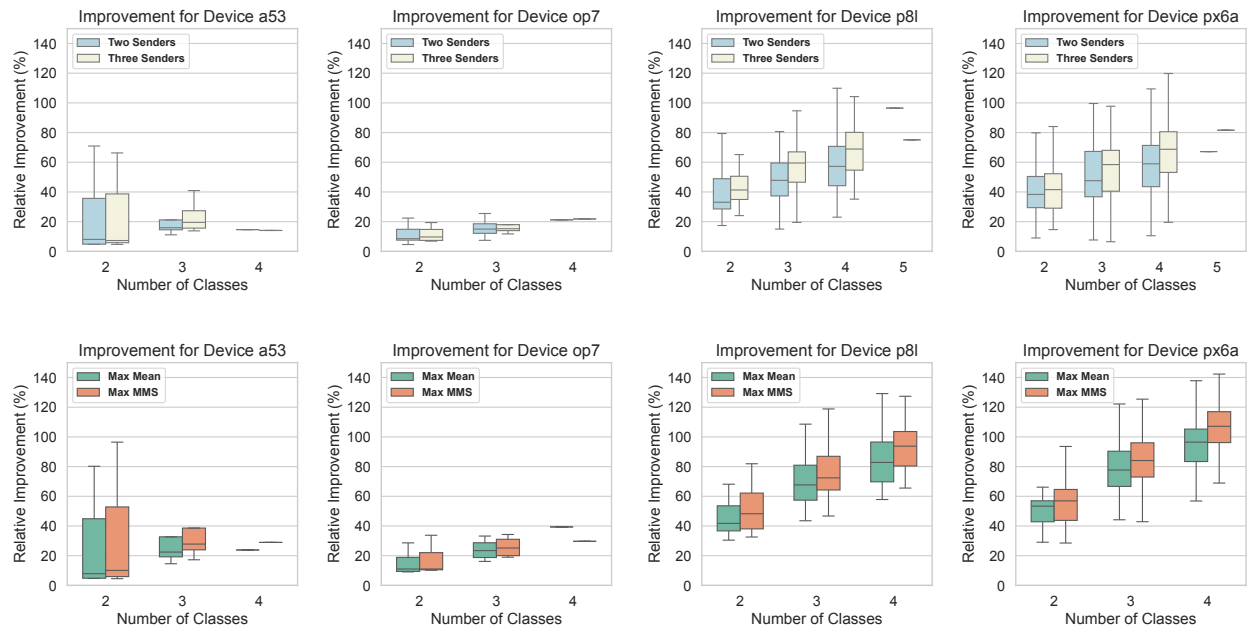


Figure 5: Best accuracy improvement of all multi-sender techniques from the single-sender baseline (not globally optimal), across all sample sizes. Lines in 4 and 5 classes indicate that there was only one classification, meaning one accuracy outcome.

with the MMS dataset showing a marginal enhancement in accuracy. However, the distinction in performance between these advanced datasets and other techniques becomes starkly apparent in the analyses for three and four classes. For these more complex classifications, the MMS dataset demonstrates a better performance than the Mean dataset, unlike the improvement observed in binary classifications. The results indicate that the MMS is currently the best-performing method for location identification, especially for multi-class classifications.

To further investigate the improvement of the Mean and MMS datasets per device, we study the corresponding boxplots of Figure 5 which illustrate the improvement percentages for the enhanced datasets for the four distinct devices. These plots reveal the percentage improvements of the advanced datasets across four distinct devices. For devices a53 and op7, the increments between the Mean and MMS methods are relatively modest. However, as we shift our focus to devices p8l and px6a, especially with an increasing number of classes, the distinction becomes more significant. The MMS dataset showcases the maximum improvement, reaching up to 142% for a four-class scenario on the px6a device. Furthermore, when juxtaposing the performance of the Mean and MMS datasets against results from two or three senders, the superiority of the MMS strategy becomes more evident. Particularly, the MMS dataset

demonstrates considerable superiority over the conventional multi-sender combinations, highlighting its effectiveness not just in enhancing accuracy, but also in providing a more consistent and reliable performance across varying class complexities and devices. This comparative analysis not only underscores the value of the MMS approach but also positions it as a notably advanced methodology within the scope of our investigation, significantly outpacing traditional techniques in terms of performance improvement. Still, Figure 5 displays our best improvements, but they are not considered as global optimal, since there might be ways to enhance these techniques even further. Finally, Figure 6 provides additional information comparing the Mean and MMS results to all single senders with all sample sizes.

5.4 Sample Size Comparisons

In machine learning, the sample size is a significant factor that influences the model's performance. A sufficient sample size ensures that the model can capture the diversity of the entire population within the data. Typically, larger sample sizes provide more data points for the model to learn from, which can lead to higher accuracy and reliability. In our work, we explore the connection between the model's performance and the sample size. Our goal is to determine whether the accuracy increases as the sample size increases. To this end, we

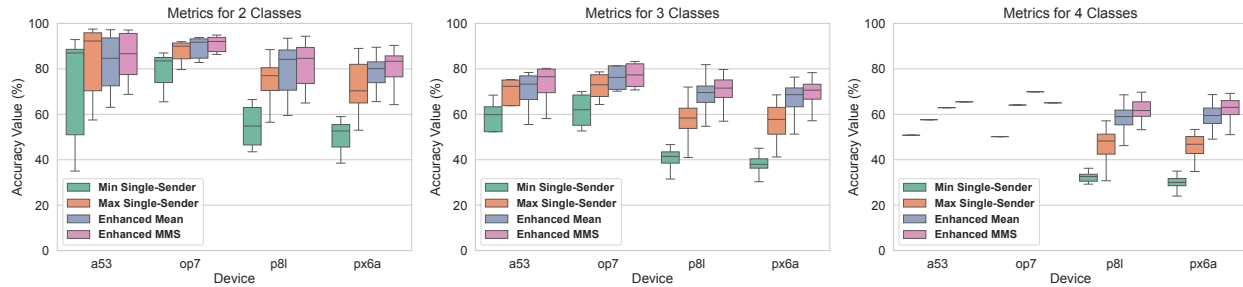


Figure 6: Accuracy boxplots between the single-senders and the enhanced multi-sender approaches for all classifications. The plots consider the **worst** and **best** performing accuracy scores for single senders. These distributions show that MMS achieves the best improvement (not global optimal).

meticulously analyze the performance metrics of single-, double-, and triple-sender results across a sample size range from 100 to 1000.

For single-senders B, D, and V, Figure 7 shows the average accuracy for all number of classes in each device, unveils a trend where accuracy generally stabilizes with an increase in sample size across various device contexts. For double-senders BD, DV, and BV, Figure 8 reveals a consistent pattern of steady or small improved accuracy with larger sample sizes, across all class numbers. This pattern persists into Figure 8, representing triple-sender configurations, where the trends once again affirm the model’s steady performance with increased data volume for each class number per device.

Regarding the classification, the trends give us the insight that the model might be well-tuned to the complexity of the task at hand, effectively capturing the patterns within the available data. In addition, this means that the key features and patterns necessary for making accurate predictions are already captured within the smaller dataset. Steadiness after a certain sample size also shows that the model’s structure is robust enough to perform reliably under varying dataset conditions.

Consequently, for the attacker, these are promising results as it is not necessary to collect large amounts of data, corresponding to the SMS transmissions, in order to conduct the location identification attack. This can be beneficial in reducing the measurement collection time, computational costs, and training time, making the model more efficient to develop and deploy, where acquiring large volumes of data is challenging or impractical. Additionally, this can also make the adversary less susceptible to detection, since the attacker can adapt to the least amount of SMS transmission and senders for the desired accuracy.

6 Discussion

In this section, we discuss the distribution of the sender locations in our study. Then, we provide our insights on the countermeasures against multi-sender SMS location inference attacks and explain their potential limitations.

6.1 Geographical Distribution of Senders

The strategic placement of sender locations, adhering to the principle of distancing them by several Kilometers, aims to capture diverse timing characteristics (e. g., via different routing), since the networks are black-box to the attacker based on our threat model. In our study, we utilize the most suitable locations from our options, for which we can collect a sufficient amount of data continuously and for a long time. We confine our options to two adjacent countries since it is more challenging to conduct the location inference attack in lower granularity levels. Expanding the number of senders and diversifying locations internationally as well can potentially improve the accuracy of attack even further.

6.2 Countermeasures

Ways to mitigate this attack can span from the elimination of silent SMSes and delivery reports to the implementation of more rigorous SMS filtering mechanisms for spam and flooding, which represents one of the most direct and practical countermeasures against location identification attacks [6]. Enhancing the core concept of resilient spamming/flooding filters, networks are encouraged to integrate advanced anomaly detection systems in order to accurately distinguish between normal and anomalous patterns of SMS traffic. However, it’s important to acknowledge that these systems primarily operate based on predefined rules and thresholds for anomaly detection, thereby limiting their efficacy to

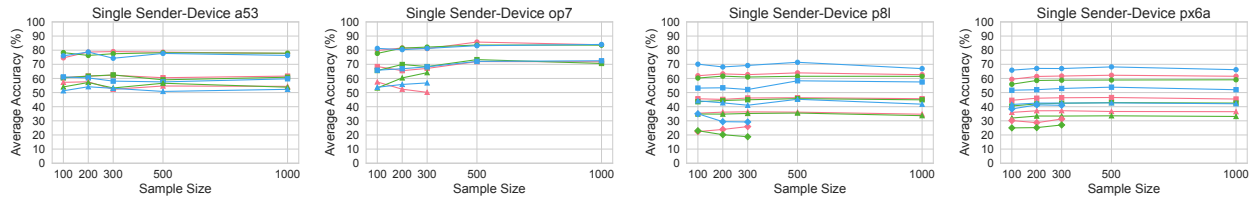


Figure 7: Single-Sender accuracy trend plots for each device, per number of classes. The trends behave steadily and continuously in most cases, as the sample sizes expand. We have included B, D, and V for 2 (○), 3 (□), 4 (△), and 5 (◇) classes.

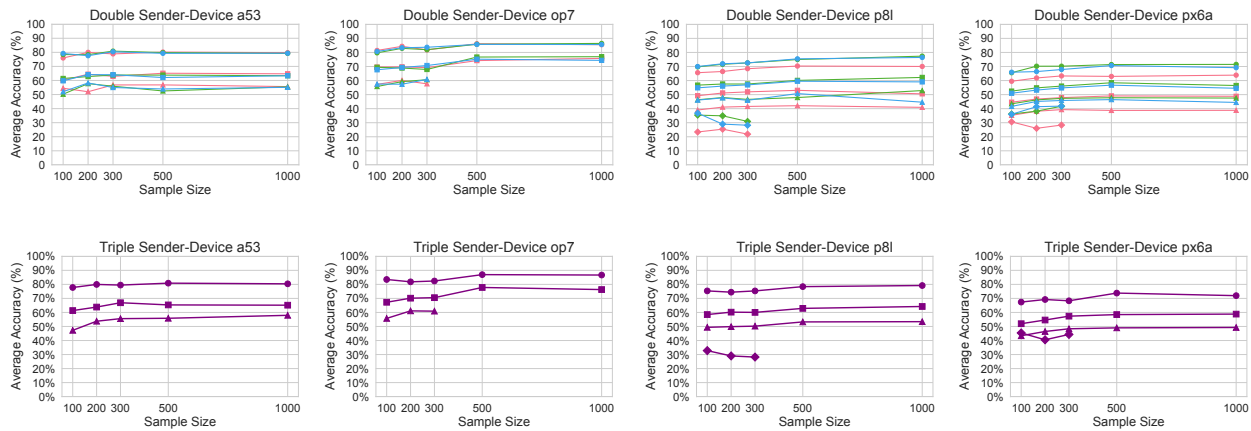


Figure 8: Accuracy trend plots per number of classes for two and three senders. The trend is rather steady and continuous as the sample sizes expand. We include BD, BV, DV, & BDV for 2 (○), 3 (□), 4 (△), & 5 (◇) classes.

merely delaying, rather than outright preventing, the execution of such attacks.

To further complicate the attacker’s efforts in utilizing timing information, the implementation of adaptive jitter mechanisms introduces a more nuanced counterstrategy. These mechanisms, capable of introducing variable delays in SMS processing, adjust dynamically in response to fluctuating network conditions and traffic patterns. This adaptability ensures that networks can impede side-channel analysis through effective timing obfuscation. Nevertheless, considering the sophisticated strategy of attackers deploying multiple senders across different geographical locations and leveraging various networks, the effectiveness of previously mentioned countermeasures could be compromised. To address this, networks could adopt a multi-layered defense strategy that also considers the following methods:

1. **Geographic Analysis of Source:** Implement anomaly detection systems that not only monitor the frequency and pattern of messages but also analyze the geographic origins of SMS traffic. By identifying unusual patterns of messages coming from mul-

iple locations (also through roaming) targeting a single number, the system can flag potential coordinated attacks.

2. **Adaptive Routing:** Dynamically alter the routing of messages based on real-time analysis to disrupt the timing measurements of attackers. This could involve randomizing the path messages take through the network or introducing variable delays for messages from identified suspicious sources and roaming.
3. **Joint Defense Initiatives:** Since the attacks can happen internationally from any location, it is imperative to establish shared intelligence on known attack patterns, including the use of multiple senders, across networks. Networks that work together can implement joint defense measures, such as coordinated blocking of attack sources and unified response strategies to emerging threats.

6.3 Limitations

In this work, we alleviated the problems of some limitations present in the location identification attack. First, the attacker is not constrained by one location only and can combine multiple sender measurements to significantly improve the model's accuracy. In addition, our sample size study showed that the attacker is not constrained by the data size in most cases, making the attack more efficient. The adversary has also the flexibility to choose the best-performing multi-sender technique per classification and is not restricted by one method only.

Despite the initial success of our experimentation, several challenges remain in multi-sender attacks. Firstly, while our study did not directly encounter coordination or resource challenges, expanding the attack to incorporate multiple senders may necessitate significant resources. This includes not only hardware but also logistical efforts to strategically position devices across various locations. Such expansion could substantially increase the complexity, cost, and effort required, potentially making the attack viable only for adversaries with substantial resources. Secondly, even though our experiments did not face any issues with anomaly detection systems, attacks conducted by multiple senders are more likely to be identified as anomalous, resembling patterns of spam or malicious activity more closely than those conducted by single senders. Lastly, our focus has largely been on closed-world scenarios, where the attacker has predefined knowledge of the victim's potential locations. The efficacy of multi-sender attacks in open-world scenarios, where the victim's location is unknown, remains less explored. We are planning to investigate these aspects of the attack in the future.

7 Related Work

Recent studies have increasingly focused on the exploitation of timing side-channel analysis for various security and privacy implications. Schnitzler et al. [27] explored the feasibility of distinguishing the location of message recipients in messenger applications using a technique based on timing differences, focusing on Internet infrastructure, similar to the concept examined by Bitsikas et al. [6] which was centered on cellular networks. This line of inquiry is part of a broader spectrum of research into timing side-channel analysis even across different web aspects, as evidenced by works such as Rasmussen et al. [23], Kohlbrenner et al. [15], Brumley et al. [7], and Goethem et al. [10], highlighting the versatility and risk of timing attacks in various online environments.

In the domain of cellular networks, a rich body of literature has methodically explored both active and pas-

sive techniques to localize cellular network users. Studies range from capturing specific identifiers to leveraging vulnerabilities within the network's paging messages and Radio Link Failure reports [12, 13, 17, 18, 29, 30]. The MAC layer and timing advance values have been investigated for their potential in enhancing localization accuracy [22, 26]. Notably, LTrack [16] demonstrated an improvement in localization accuracy to as precise as 20 meters, significantly enhancing tracking capabilities with minimal adversary involvement. Furthermore, Lakshmanan et al. [18] showed that by collecting data from the public scheduling channel and finding unique identifiers, one could trace a target's path with an accuracy of less than 1 kilometer.

Various SMS attacks have been demonstrated, exploiting vulnerabilities to extract sensitive user information or execute commands, as seen in the case of Simjacking [4] and studies on spamming, spoofing, DoS, and silent SMS in LTE networks [31]. Mulliner et al. [19] introduced a vulnerability analysis framework for monitoring unexpected smartphone behaviors leading to large-scale DoS attacks. Furthermore, audio call features have been explored for security applications, such as fingerprinting and anomaly detection to combat call redirection/hijacking. Techniques leveraging audio latency and network characteristics have been investigated, with notable examples including Sonar [20] and PinDrOp [5].

8 Conclusion

In this work, we explored various multi-sender techniques of the SMS location inference attack, which provide a substantial accuracy improvement compared to the single-sender approaches. Our results showed that the best-performing method for all devices, sample sizes, and number of classes was the multi-sender MMS method. Additionally, we performed an analysis on the effects of the sample size on the model's accuracy for single- and multi-sender attacks, which revealed that the attacker can leverage smaller sample sizes to conduct the attack saving measurement collection time, resources and reducing the possibility for detection. Finally, we re-examined the potential countermeasures with extra suggestions.

Acknowledgements

This work was supported by NSF grant 2144914, by UA Ruhr under the Research Alliance Ruhr program, and by the Center for Cyber Security at New York University Abu Dhabi (NYUAD). The authors would like to thank Michel Lang, Philipp Markert, and Lena Schnitzler for their help with data collection.

References

- [1] 3GPP. Digital cellular telecommunications system (Phase 2+)(GSM); Universal Mobile Telecommunications System (UMTS); LTE; 5G; Support of SMS over IP networks; Stage 3. Technical Specification (TS) 24.341, 3rd Generation Partnership Project (3GPP), 05 2022. Version 17.1.0.
- [2] 3GPP. Digital cellular telecommunications system (Phase 2+)(GSM); Universal Mobile Telecommunications System (UMTS); LTE; IP Multimedia Subsystem (IMS); Stage 2 . Technical Specification (TS) 23.228, 3rd Generation Partnership Project (3GPP), 05 2022. Version 17.3.0.
- [3] 3GPP. Digital cellular telecommunications system (Phase 2+)(GSM); Universal Mobile Telecommunications System (UMTS); LTE; Use of Data Terminal Equipment - Data Circuit terminating Equipment (DTE - DCE) interface for Short Message Service (SMS) and Cell Broadcast Service (CBS). Technical Specification (TS) 27.005, 3rd Generation Partnership Project (3GPP), 04 2022. Version 17.0.0.
- [4] Adaptive Mobile Security Limited. Simjacking. https://f.hubspotusercontent10.net/hubfs/8487362/Reports/AdaptiveMobile_Security_Simjacker_Technical_Paper_v1.01.pdf.
- [5] Vijay A. Balasubramanian, Aamir Poonawalla, Mustaque Ahamad, Michael T. Hunter, and Patrick Traynor. PindrOp: Using single-ended audio features to determine call provenance. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 109–120, New York, NY, USA, 2010. Association for Computing Machinery.
- [6] Evangelos Bitsikas, Theodor Schnitzler, Christina Pöpper, and Aanjhan Ranganathan. Freaky leaky SMS: Extracting user locations by analyzing SMS timings. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2151–2168, Anaheim, CA, August 2023. USENIX Association.
- [7] David Brumley and Dan Boneh. Remote timing attacks are practical. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.
- [8] Cloudmark. SMS spam overview — preserving the value of SMS texting. <https://www.cloudmark.com/en/resources/white-papers/sms-spam-overview-preserving-value-sms-texting>.
- [9] Europol. Takedown of sms-based flubot spyware infecting android phones. <https://www.europol.europa.eu/media-press/newsroom/news/takedown-of-sms-based-flubot-spyware-infecting-android-phones>.
- [10] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1985–2002. USENIX Association, August 2020.
- [11] GSM Association. Official Document NG.111 - SMS Evolution. Technical Specification (TS) 111-v2.0, GSM Association, 11 2020. Version 2.0.
- [12] Byeongdo Hong, Sangwook Bae, and Yongdae Kim. GUTI reallocation demystified: Cellular location tracking with changing temporary identifier. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [13] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy attacks to the 4G and 5G cellular paging protocols using side channel information. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.
- [14] Kaspersky. What is smishing and how to defend against it. <https://www.kaspersky.com/resource-center/threats/what-is-smishing-and-how-to-defend-against-it>.
- [15] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security Symposium*, 2017.
- [16] Martin Kotuliak, Simon Erni, Patrick Leu, Marc Röschlin, and Srdjan Capkun. LTrack: Stealthy tracking of mobile phones in LTE. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1291–1306, Boston, MA, August 2022. USENIX Association.
- [17] Denis Foo Kune, John Kölnsdorfer, Nicholas Hopper, and Yongdae Kim. Location leaks over the GSM air interface. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

- [18] Nitya Lakshmanan, Nishant Budhdev, Min Suk Kang, Mun Choon Chan, and Jun Han. A stealthy location identification attack exploiting carrier aggregation in cellular networks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3899–3916. USENIX Association, August 2021.
- [19] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. Sms of death: From analyzing to attacking mobile phones on a large scale. In *USENIX Security Symposium*, 2011.
- [20] Christian Peeters, Hadi Abdullah, Nolen Scaife, Jasmine Bowers, Patrick Traynor, Bradley Reaves, and Kevin Butler. Sonar: Detecting ss7 redirection attacks with audio-based distance bounding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 567–582. IEEE Computer Society, 05 2018.
- [21] Christian Peeters, Christopher Patton, Imani N. S. Munyaka, Daniel Olszewski, Thomas Shrimpton, and Patrick Traynor. SMS OTP security (SOS): hardening SMS-based two factor authentication. In *ASIA CCS'22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*, pages 2–16. ACM, 2022.
- [22] Benjamin A Pimentel. *Passive Geolocation in a 4G WIMAX Single Base Station Scenario*. Phd thesis, Naval Postgraduate School, Monterey California, 2013.
- [23] Kasper Bonne Rasmussen and Srdjan Capkun. Location privacy of distance bounding protocols. *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.
- [24] Bradley Reaves, Nolen Scaife, Dave Tian, Logan Blue, Patrick Traynor, and Kevin R. B. Butler. Sending out an SMS: characterizing the security of the SMS ecosystem with public gateways. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 339–356. IEEE Computer Society, 2016.
- [25] Bradley Reaves, Luis Vargas, Nolen Scaife, Dave Tian, Logan Blue, Patrick Traynor, and Kevin R. B. Butler. Characterizing the security of the SMS ecosystem with public gateways. *ACM Trans. Priv. Secur.*, 22(1):2:1–2:31, 2019.
- [26] John D. Roth, Murali Tummala, John C. Mceachen, and James W. Scrofani. On location privacy in LTE networks. *IEEE Transactions on Information Forensics and Security*, 12:1358–1368, 2017.
- [27] Theodor Schnitzler, Katharina Kohls, Evangelos Bitsikas, and Christina Pöpper. Hope of Delivery: Extracting User Locations From Mobile Instant Messengers. In *Network and Distributed System Security Symposium, NDSS '23, San Diego, CA, USA, February 2023*. The Internet Society.
- [28] Security Affairs. After simjacker, wibattack hacking technique disclosed. billions of users at risk. <https://securityaffairs.co/wordpress/91800/hacking/wibattack-sim-attack.html>.
- [29] Altaf Shaik, Jean-Pierre Seifert, Ravishankar Borgaonkar, N. Asokan, and Valtteri Niemi. Practical attacks against privacy and availability in 4G/LTE mobile communication systems. *ArXiv*, abs/1510.07563, 2016.
- [30] Ankush Singla, Syed Rafiul Hussain, Omar Chowdhury, Elisa Bertino, and Ninghui Li. Protecting the 4G and 5G cellular paging protocols against security and privacy attacks. *Proc. Priv. Enhancing Technol.*, 2020(1):126–142, 2020.
- [31] Guan-Hua Tu, Chi-Yu Li, Chunyi Peng, Yuanjie Li, and Songwu Lu. New security threats caused by IMS-based SMS service in 4G LTE networks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 1118–1130, New York, NY, USA, 2016. Association for Computing Machinery.

MakeShift: Security Analysis of Shimano Di2 Wireless Gear Shifting in Bicycles

Maryam Motallebighomi¹, Earlence Fernandes², and Aanjhan Ranganathan¹

¹Northeastern University, Boston, USA

²University of California, San Diego, USA

Abstract

The bicycle industry is increasingly adopting wireless gear-shifting technology for its advantages in performance and design. In this paper, we explore the security of these systems, focusing on Shimano’s Di2 technology, a market leader in the space. Through a blackbox analysis of Shimano’s proprietary wireless protocol, we uncovered the following critical vulnerabilities: (1) A lack of mechanisms to prevent replay attacks that allows an attacker to capture and retransmit gear shifting commands; (2) Susceptibility to targeted jamming, that allows an attacker to disable shifting on a specific target bike; and (3) Information leakage resulting from the use of ANT+ communication, that allows an attacker to inspect telemetry from a target bike. Exploiting these, we conduct successful record and replay attacks that lead to unintended gear shifting that can be completely controlled by an attacker without the need for any cryptographic keys. Our experimental results show that we can perform replay attacks from up to 10 meters using software-defined radios without any amplifiers. The recorded packets can be used at any future time as long as the bike components remain paired. We also demonstrate the feasibility of targeted jamming attacks that disable gear shifting for a specific bike, meaning they are finely tuned to not affect neighboring systems. Finally, we propose countermeasures and discuss their broader implications with the goal of improving wireless communication security in cycling equipment.

1 Introduction

Modern bicycles are cyber-physical systems that contain embedded computers and wireless links to enable new types of telemetry and control. The key motivating factors for moving away from traditional mechanical systems are the ability to gain insights about a rider’s physical performance, better responsiveness in gear shifting, customizability of how the gear shifters operate, and easier setup and maintenance.

Among all these technologies, we observe that the one with

the most impact on bike control and safety is wireless gear shifting.¹ It uses wireless links between the gear shifters and the derailleur — an electro-mechanical component that uses motors to move the chain between gears. Electronic control provides increased precision in shifts and is less prone to issues like cable stretch and contamination that plague mechanical gear shifting systems. Although wired electronic control of gear shifting exists, the current trend in the bicycle industry is to move towards wireless control. All major manufacturers now support wireless shifting (Shimano, SRAM, Campagnolo).

In this work, we analyze the security guarantees of wireless gear shifting. Any security vulnerability in this system can significantly impact the rider’s safety and performance, especially in professional bike races, where an attacker could target a victim rider to gain an unfair competitive advantage. In a professional race, a peloton of hundreds of riders are close to each other, often a few feet apart, and can reach speeds up to 40 mph. Any sudden changes to a bike’s performance can be catastrophic. For example, if an attacker were to target a subset of riders and shift the gears or jam the shifting operation, it could result in crashes and injuries. As another example, if the riders are climbing slowly (or descending at high speed), an attacker could shift a target rider’s bike into high gear or jam their shifting, leading them to lose their position in the race and even lose control of the bike itself.

The sport of professional cycling has a long and troubled history with the use of illegal performance-enhancing drugs — security vulnerabilities in one of the most critical components of the bike could be viewed as an attractive alternative method for people who want to compromise the integrity of the sport. Furthermore, our attacks do not leave any detectable trace, unlike the use of performance-enhancing drugs. As such, with the introduction of wireless gear shifting, one must adopt an adversary’s viewpoint — professional bike races are adversarial environments, and the technology must withstand motivated attackers. We focus on the Shimano 105

¹The other important component is the brakes, but these are mechanical systems.

Di2 [10] and Shimano DURA-ACE Di2 [16] wireless shifting systems. Shimano is a leader in the bicycle control system industry, commanding approximately 50% of the market share [15, 28]. We purchased a recent version of the control system and performed a black box security analysis, from capturing raw physical signals, examining their behavior on gear shifting, and finally performing packet structure/content analysis. This study seeks to address the following research questions: (1) What are the security guarantees provided by these wireless gear-shifting systems? (2) Do these wireless systems, when integrated into bicycles, maintain robust defenses against specific cyber attacks, such as replay attacks, similar to those observed in automotive key fob systems [20]? Have the lessons learned from analyzing similar systems contributed to the design of these wireless gear shifters? (3) What is the practical feasibility of executing the identified cyber attacks? In other words, what constraints and requirements would an attacker face in attempting to compromise these systems?

Our key contribution is the discovery of a record-and-replay attack that allows an unauthorized party to fully control gear shifting on a victim bike at ranges up to 10 meters without the use of amplifiers. This attack can be realized using commercial-off-the-shelf software-defined radios (SDR). The attacker only needs to record two signals — an upshift and a downshift.

We make the following contributions in this paper.

- **Analysis of the Shimano Wireless Gear Shifting Protocol.** We investigate the proprietary protocol used by Shimano for its wireless gear shifting. This process allows us to decode the communication framework of these systems, providing insights into their operational mechanics.
- **Identify Security Weaknesses.** Based on the analysis, we identify several security weaknesses within the protocol, notably the absence of replay protection mechanisms such as timestamps or sequence numbers. Despite the implementation of cryptographic primitives, these vulnerabilities present significant security risks.
- **Record-and-Replay and Targeted Jamming Attacks.** Leveraging the identified weaknesses, we successfully execute record-and-replay attacks. These attacks can cause unexpected gear shifts in arbitrary patterns by interacting at the physical layer, bypassing the need for extracting any cryptographic secrets and making the attack independent of the cryptographic layer. Furthermore, we explore the potential for targeted jamming attacks that specifically disable gear-shifting capabilities on targeted bicycles without impacting nearby cyclists.
- **Experimental Evaluation.** We conduct various experiments with two identical Shimano 105 Di2 wireless gear shifting systems. We also confirmed our findings on Shimano DURA-ACE Di2 system. Through these experiments, we executed

replay and jamming attacks utilizing SDRs and explored their effective range. Additionally, we examined the shifting system’s behavior in response to interference. Our experiments indicate that replay attacks using SDRs are effective up to a distance of 10 meters without amplification. The effectiveness of replayed packets persists as long as the pairing between shifters and derailleur remains unchanged.

- **Countermeasures.** We provide a discussion of potential countermeasures. Wireless gear shifting operates in a highly constrained environment — security mechanisms should not add significant time delay in shifting and must not degrade battery life. While implementing techniques such as timestamps has particular challenges, employing rolling codes or distance bounding within wireless gear shifting can effectively mitigate replay attacks.

Although our paper’s main focus is on Shimano’s gear-shifting systems, we also examine vulnerabilities in the communication protocol used for telemetry on bike displays, notably the ANT protocol. This protocol is widely used in Shimano and other low-power wireless data transmission systems, extending the relevance of our findings. We have shown that any nearby third party with Shimano’s private key and knowledge of the channel configuration can intercept all transmitted information. In our replay attack scenario, this information enables the attacker to determine the targeted bike’s current gear and replay the upshifting/downshifting commands to adjust the gear according to their preference. For example, the attacker waits until the rider is in gear 3 and then launches a downshift replay to move it to gear 2.

Our study aims to highlight vulnerabilities in wireless gear shifting systems, especially focusing on Shimano’s Di2, and offers a first look into the security challenges of bicycle wireless communication technologies. Through this work, we hope to contribute to the ongoing effort to secure wireless communications in cycling equipment.

Responsible Disclosure. We notified Shimano about the vulnerabilities, along with detailed information on replicating the attacks, part numbers of the devices we tested, and a description of countermeasures that might be helpful in this context. Shimano has acknowledged these vulnerabilities and is working on fixes at the time of this writing.

2 Wireless Gear Shifting: An Overview

In the cycling industry, all major manufacturers have ventured into developing wireless gear-shifting systems, aiming to enhance the cycling experience through technology. Brands like SRAM [17] and Campagnolo [3], alongside Shimano, have introduced their versions of wireless shifting, each bringing unique features and innovations to the market. These systems signify a leap forward in bicycle design, offering cyclists

Table 1: The equipment list on the test bikes for Shimano 105 Di2 groupset.

Item	Model	Firmware Version
Rear Derailleur	RD-R7150	ver 4.0.2
Front Derailleur	FD-R7150	ver 4.0.1
Right Shifter	ST-R7170-R	-
Left Shifter	ST-R7170-L	-
Battery	BT-DN300	ver 4.0.1

Table 2: The equipment list on the test bikes for Shimano Dura-Ace Di2 groupset.

Item	Model	Firmware Version
Rear Derailleur	RD-R9250	ver 4.0.7
Front Derailleur	FD-R9250	ver 4.0.3
Right Shifter	ST-R9270-R	-
Left Shifter	ST-R9270-L	-
Battery	BT-DN300	ver 4.0.1

improved performance, convenience, and integration. Due to Shimano’s significant market presence and role as a pioneering force in cycling technology, we’ve selected Shimano as our case study to examine the vulnerabilities inherent in wireless gear-shifting systems.

For our experiments, we chose the Shimano 105 Di2 and the Dura-Ace Di2 wireless gear-shifting systems as our case study. Tables 1 and 2 contain the equipment list, their respective model numbers, and firmware versions. We note that the two groupsets, Shimano 105 Di2 and Shimano Dura-Ace Di2, are compatible. Therefore, we tested pairing various shifters and derailleurs from these groupsets and confirmed that they all use the same protocol. Consequently, the vulnerabilities identified are consistent across both systems.

The Shimano gear-shifting system consists of four main components.

(1) Rear Derailleur: The rear derailleur is the core of the gear-shifting system and facilitates all wireless communications. This includes connections with the shifters, Bluetooth Low Energy (BLE), and ANT+ communications. It offers eleven gear levels (and, in some newer versions, 12 levels), ranging from the lowest to the highest.

(2) Front Derailleur: The front derailleur is wired to the rear derailleur through the battery and allows switching between two distinct gear levels, which are large gear changes.

(3) Right and Left Shifters: These components wirelessly transmit gear-shifting instructions to the rear derailleur using Shimano’s proprietary protocol, which we will explore in detail in the following section. One of the shifters controls the rear derailleur, and the other controls the front derailleur. This setting can be customized through the E-TUBE PROJECT [4] over BLE.

(4) Battery: The battery ports are connected to the rear and



Figure 1: Shimano’s RF communication.

front derailleur, ensuring they are powered for operation.

The Shimano system employs three key protocols to establish connections among its various components, each serving a distinct function. The communication methods within Shimano’s network are illustrated in Figure 1.

2.1 Bluetooth Low Energy

The Shimano E-TUBE PROJECT is a software tool that connects cyclists to their bike configuration. This platform can personalize the settings, such as customizing shifter button functions and conducting firmware updates. It employs BLE for efficient communication in many power-constrained devices, which fits the requirements of a system like E-TUBE PROJECT that aims to provide seamless and user-friendly interaction with bicycle components. While BLE is essential for configuring and updating the system, it *does not* control real-time biking actions such as shifting gears.

Also, the initial setup of shifters and the rear derailleur involves pairing them through the E-TUBE mobile app. Users need to register and connect the rear derailleur to the mobile app, then scan the QR code on the shifters to pair both shifters with the rear derailleur. Given that Bluetooth Low Energy (BLE) vulnerabilities have been extensively documented in existing literature [41], our paper did not focus on this aspect.

2.2 ANT+

ANT is a low-power wireless protocol designed to transmit information between devices efficiently and reliably. It is known for robustness and adaptability in different network setups, including mesh networks, making it ideal for gathering and sending sensor data.

Building on the ANT protocol, ANT+ is an enhancement that standardizes how specific data types are communicated. It establishes device profiles for consistent data transmission, like heart rate, bike speed, and cadence. This standardization allows devices from various manufacturers to work together seamlessly. In cycling, ANT+ plays a crucial role in the Di2 system. It wirelessly sends vital information such as gear position and battery life to compatible cycling computers,

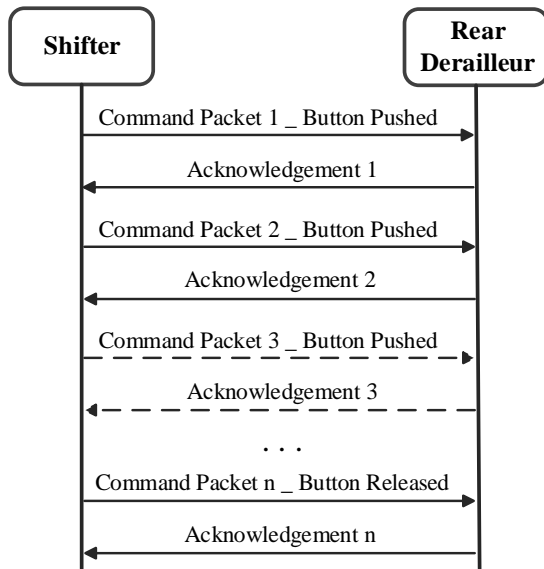


Figure 2: The sequence of command and acknowledgment packets between the shifter and rear derailleur after a button press. The user’s actions can influence the sequence and number of command packets, which are subsequently followed by an acknowledgment.

allowing riders to monitor these details in real-time during their rides. The frequency range for ANT devices spans from 2.400 GHz to 2.524 GHz, but 2.457 GHz is reserved specifically for ANT+ devices. These devices can operate using a public network key, a private one, or a managed network key owned privately, providing flexibility in network security and access [1]. In summary, ANT and ANT+ offer versatile and efficient solutions for wireless communication, especially in scenarios where reliable data transmission and interoperability are essential.

2.3 Shimano’s Proprietary Communication Protocol

In the Shimano Di2 system, gear shifting is controlled through a unique, Shimano-specific protocol. This protocol operates on the 2.478 GHz frequency band, facilitating communication between the rear derailleur and the shifters. However, Shimano’s official documentation does not disclose detailed information about this protocol, leaving specifics such as modulation, data rate, and packet structure unclear. Thus, we analyze Shimano’s proprietary communication protocol as a first step.

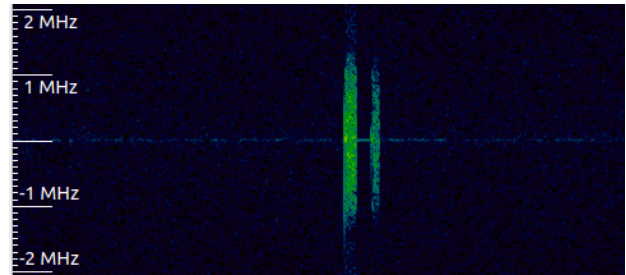


Figure 3: One command packet being transmitted from the shifter to the rear derailleur, along with the corresponding acknowledgment sent from the rear derailleur back to the shifter.

3 Analyzing Shimano’s Wireless Gear Control Protocol

We begin with an overview of Shimano’s Wireless Gear Control Protocol, providing a detailed examination of the command and acknowledgment packet sequences exchanged between the shifter and the rear derailleur. Next, we analyze the physical layer, focusing primarily on demodulating captured RF signals into binary data to understand the underlying communication mechanisms. We employ a black-box methodology to passively capture raw signals. Subsequently, we delve into the packet structures within the Shimano wireless communication protocol, exploring all components of the various packet types. Finally, we discuss the security weaknesses that could potentially threaten the protocol’s integrity.

3.1 High-Level Protocol Overview

The shifters send two types of commands to the rear derailleur — Gear Up and Gear Down. On each press of the shift button (either up or down), the shifter transmits at least three packets to the derailleur. Upon receiving each packet, the derailleur transmits an acknowledgment to the shifter. The quantity of packets transmitted is influenced by the speed at which the user presses and releases the shifter button. If the button is pressed and held, packets will be sent for the hold duration. Conversely, a single press of the button results in the transmission of at least three packets. Figure 2 illustrates the sequence of packets triggered by the user pressing the button, leading to one upshift on the rear derailleur. As noted, the sequence of command packets followed by an acknowledgment can vary based on the user’s actions. Figure 3 displays one command packet being transmitted from the shifter to the rear derailleur, along with the corresponding acknowledgment sent from the rear derailleur back to the shifter. Each command packet has an approximate duration of 112 μ s, while each acknowledgment packet is about 76 μ s.

If the shifter fails to receive an acknowledgment within a time frame, it initiates a burst transmission. Each burst

Table 3: Behavior of different packets during a replay attack: Pressing the button by the user results in the transmission of three packets from the shifter to the derailleur. To understand the packet’s functionality, we conducted experiments by replaying the packets both individually and in various combinations.

Setting	First Packet	Second Packet	Third Packet	Observations
A	1			This will cause the derailleur to shift up by one gear. The repeating signal will not function until you manually press the button once. After that, the signal can be replayed successfully once more.
B		1		Similar to A.
C			1	No reaction.
D	1		1	Works like a normal replay. Repeated many times.
E		1	1	Similar to D.
F	1	1		Every time the signal was replayed, it resulted in shifting twice instead of a single time.

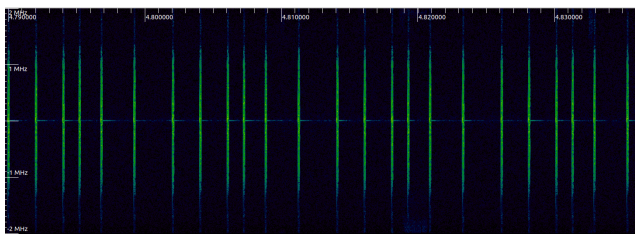


Figure 4: Segments from a burst sequence when acknowledgments are not received, showcasing a total of 748 packets transmitted over 1.5 seconds.

contains 748 packets and lasts 1.5 seconds. We captured the packet burst while the rear derailleur was disconnected from the battery. Figure 4 illustrates a segment of this burst. In Section 4.3, we discuss the relevance of the packet burst under conditions like interference.

In the next step, we conducted tests by individually transmitting the three captured packets associated with a single gear shift to analyze each packet’s effect. Furthermore, we experimented with various combinations of these three packets to examine the outcomes, acknowledging that redundancy among them might be designed to guarantee command reception by the derailleur to prevent potential interference.

Table 3 summarizes the functionality of the packets. We monitored how the packets behaved under different conditions (labeled A to F), continuously replaying the specific packet(s) relevant to each condition. This was then contrasted with a baseline scenario, wherein all three packets were replayed in their original sequence, mirroring the authentic command exactly. Our observations indicated that the behaviors of the first and second packets were strikingly similar.

On the other hand, replaying only the third packet triggers no derailleur action, leading us to speculate that this packet might serve as a "button released" command.

In scenarios D and E, eliminating the first or second packet does not affect the behavior of the packets compared to the

Table 4: Signals information derived from publicly available documents

Signal Feature	Value
Frequency	2.478 GHz
Bandwidth	2127 KHz
Modulation	GFSK
Emission Reference	<TX3064779>
Emission Designator	2M13F1D-

baseline scenario. This suggests that one of the packets may be sent as a form of redundancy. In both cases, D and E, repeatedly replaying the packets consistently triggers a single gear shift, akin to the baseline scenario. However, replaying the same packet in scenarios A and B does not lead to subsequent gear shifts after the successful initial replay.

Furthermore, in scenario F, sending both the first and second packets causes the gear to shift twice, which could mirror the situation where the user keeps the button pressed.

We clarify that our analysis in Table 3 focuses exclusively on individual shift events rather than MultiShift settings. MultiShift settings in Shimano’s wireless gear-shifting system allow multiple gear changes with a single button press, enabling quicker transitions across gears. This distinction is important as our experimental setup and data collection were designed to evaluate single-shifting actions.

In conclusion, our experiments revealed that the roles of the first and second packets might stem from redundancy and correspond to the user’s button press, while the third packet appears to be associated with the user releasing the button.

3.2 Physical Layer Analysis

The primary focus is demodulating the captured RF signals into binary data and subsequently examining their contents. We use a black box methodology that passively captures raw signals.

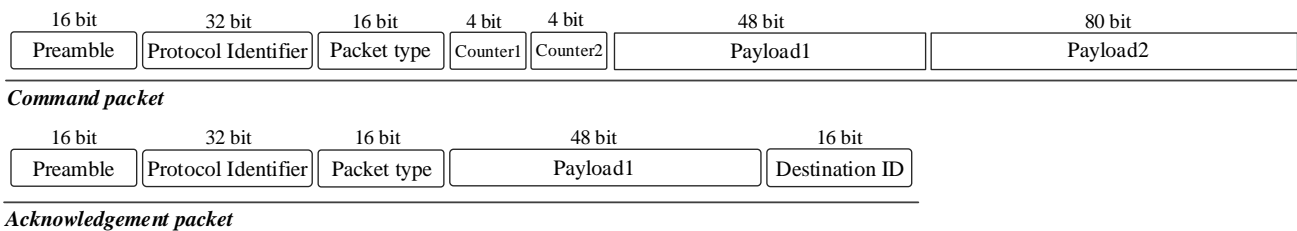


Figure 5: Overview of Command and Acknowledgment Packet Structures. Illustrating the differences between 200-bit command packets and 128-bit acknowledgment packets, including the content of their respective fields.

Table 5: Modulation/Demodulation parameters for Shimano’s proprietary protocol

Modulation Parameter	Value
Carrier Frequency	2.478 GHz
Data Rate	2 Mbps
Bit per Symbol	1
Frequency Deviation	-250 kHz/250 kHz
Gauss BT	0.5
Gauss Filter Width	1

The first step in analyzing a wireless signal is determining its precise frequency and modulation type. We found this data in documents from the Federal Communications Commission (FCC) and the Radio Equipment List (REL). Table 4 presents the information summarized from these documents. The communication between devices occurs at 2.478GHz and does not utilize frequency hopping. The signal’s bandwidth is 2127 KHz.

The term ‘Emission Reference <TX3064779>’ is identified as a distinct code or number linked to specific emission properties. The ‘Emission Designator 2M13F1D’ is recognized globally to categorize a signal’s bandwidth, modulation type, and content. ‘2M13’ details the required signal bandwidth. ‘F’ denotes the modulation type of the primary carrier as frequency modulation, ‘1’ represents a single channel carrying quantized or digital data without an additional modulating sub-carrier, and ‘D’ describes the nature of the transmitted information, highlighting the transmission of digital data.

Shimano gear shifting utilizes Gaussian Frequency Shift Keying (GFSK) for their proprietary communication protocol, a form of Frequency Shift Keying that applies Gaussian filtering to smooth out signal transitions or frequency shifts. GFSK is a prominent modulation technique employed across various wireless technologies, including Bluetooth, IEEE 802.15.4, and Z-wave. Dealing with GFSK presents more complexity in the analysis compared to systems using simpler modulation techniques like amplitude shift keying, where signal demodulation can be straightforwardly achieved using open-

source tools such as Inspectrum [8]. However, demodulating GFSK requires identifying the correct demodulation parameters, which increases the complexity. For Shimano devices, all specific modulation parameters were initially unclear. The FCC documents did not disclose any of these parameters.

We utilized Universal Radio Hacker (URH) [34], a tool specifically designed to analyze and manipulate wireless communication signals for our analysis. This tool facilitates the recording, analysis, and modification of signals across various wireless devices. However, the automatic parameter detection feature in URH failed to demodulate our captured signal effectively. We were unclear about the data rate, a critical piece of information for GFSK demodulation, which depends heavily on the correct sample/symbol ratio. Through a combination of trial and error and visual analysis of our signals, we identified the necessary parameters to successfully demodulate the captured data. Table 5 outlines the required modulation/demodulation parameters we identified. The Gaussian filter in GFSK modulation has a parameter called the time-bandwidth product (BT), which is the product of the filter bandwidth and the bit duration. The BT value affects the shape of the data pulses and the resulting GFSK signal.

3.3 Packet Structure and Content

Upon successful demodulation, we could distinguish two primary types of packets within the Shimano communication protocol: command and acknowledgment. Command packets, originating from the shifter, comprise 200 bits and are directed towards the derailleur. Conversely, acknowledgments follow these command signals and consist of 128 bits, transmitting from the derailleur back to the shifter. Figure 5 graphically illustrates these packet structures, annotated with their components.

In our analysis of numerous packet sequences, we identify and describe specific fields within the packets as follows:

Preamble: Each packet starts with a 16-bit preamble, represented as 0101010101010101. The preamble plays a critical role in various RF communication protocols by helping to synchronize the receiver’s timing with the sender’s signal. This synchronization aids in accurately detecting the beginning

Table 6: Analysis of different fields in Shimano command packets. It shows our observation of how different fields in the packets change under various conditions, helping to clarify how the fields are connected.

Action/Condition	Counter1 changes?	Counter2 changes?	Observations		
			Payload1 Consistency	Payload2 Consistency	Destination ID Consistency
Bike1, Upshifting	No	No	Yes	Yes	Yes
Bike1, Upshifting	Yes	No	No	No	Yes
Bike1, Upshifting	No	Yes	No	No	Yes
Bike1, Downshifting	No	No	Yes	No	Yes
Bike2, Upshifting	No	No	No	No	No
Bike1, Upshifting (Repairing)	No	No	Yes	No	Yes

of a new packet. Additionally, the preamble facilitates the adjustment of the receiver’s Automatic Gain Control (AGC) circuits to the strength of the incoming signal. We verified the correct demodulation of the preamble at the start of each packet during our adjustments of the modulation parameters.

Protocol Identifier: Following the preamble is a 32-bit protocol identifier, which remains unchanged across all packets captured under this specific Shimano’s proprietary protocol. This identifier functions similarly to the “access address” in BLE protocols and helps distinguish Shimano’s protocol from other traffic in the 2.4 GHz spectrum.

Packet Type: A 16-bit field follows, identifying the packet as either a command or acknowledgment packet. In our observations, every 200-bit command packet contains 0x8888 within this field, while acknowledgments are marked with 0x1010.

Counters: The first counter (Counter1) increments with each transmitted packet. The second counter (Counter2) increments only upon receiving an acknowledgment, indicating a successful transmission. If the shifter does not receive an acknowledgment and begins to emit a burst of packets, Counter2 remains constant, whereas Counter1 cycles through 15 possible values.

Payload: The next segment within command packets involves Payload1 and Payload2, together spanning 128 bits. We divided the payload into two parts because some parts of the payload in command packets are exactly replicated in the acknowledgments.

Table 6 offers a comprehensive look at our findings, illustrating how each field varies across different test scenarios and setups. For any given wireless shifting setup and command type (either upshifting or downshifting), Payload1 and Payload2 remain consistent as long as the counters are identical.

Specifically, Payload1 comprises a sequence that is present both in the command packets and in the acknowledgment packets. This section of the payload, a 48-bit sequence, is repeated in the acknowledgment packets to confirm which command the acknowledgment is intended for. On the other hand, Payload2 is exclusive to the command packets and does not appear in the acknowledgment packets. If two packets have similar values for counter1 and counter2, the values

of both Payload1 and Payload2 would be the same too. This consistency holds true while the same shifters are consistently paired with the same derailleur.

However, if the shifters are unpaired and re-paired, Payload2 will have a different value under the same counter conditions, while Payload1 remains unchanged even after unpairing. So, in other words, Payload2 is susceptible to changes upon reconfiguring the shifting system components.

Destination ID: The acknowledgment packets feature a 16-bit Destination ID at their conclusion. This ID corresponds to the identity of the shifter—the commanding device or the device that the acknowledgment is intended for. Through extensive testing involving various pairings of shifters and derailleurs, we consistently observed that the Destination ID is determined by the shifter that sends the command packet. In other words, the acknowledgment identifies and responds to the shifter initiating the command.

Additionally, our experiments revealed that the Destination ID remains constant, even after devices are unpaired and then repaired. This consistency indicates that the Destination ID is inherently linked to the shifter itself and does not change with different pairing configurations. It highlights that the identity encoded in the Destination ID is intrinsic to the shifter rather than being dependent on the pairing status or the particular session of interaction between the devices.

It is worth mentioning that the command packets lack a feature similar to the Destination ID, which is consistent and unencrypted, that would allow one to identify the receiver from captured messages.

For our analysis, we looked into the packets from shifters controlling both the rear and front derailleurs. We confirmed that the packet structure remains consistent for all command packets and constant across all acknowledgment packets.

3.4 Security Weakness

Our analysis revealed that Shimano’s wireless gear shifting protocol employs a form of encryption, which hinders attackers from creating and transmitting their own packets to the

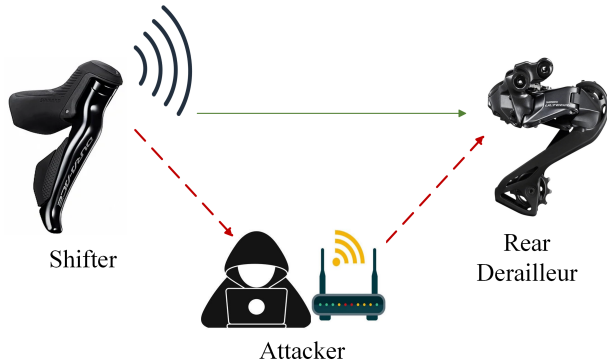


Figure 6: The attacker model for the replay attack. The attacker captures the command signal from the shifter, sends it to the derailleur, and later replays it.

derailleurs. However, the protocol does not offer protection against replay attacks, as the packets lack timestamps and sequence numbers, rendering the protocol vulnerable to such attacks. In this scenario, the attacker is not required to fabricate packets but can simply capture and replay them. We further describe this vulnerability with our experimental findings in the following section.

4 Replay and Jamming Attacks

In the following, we will explore the susceptibility of Shimano’s wireless gear shifting protocol to replay attacks in Section 4.2, and discuss targeted jamming against this protocol in Section 4.3.

4.1 Attacker Assumptions and Experimental Setup

The attacker is equipped with an SDR capable of transmitting and receiving signals in the 2.4 GHz band. All commercial off-the-shelf SDRs, such as the USRP B210 [5], HackRF [7], PlutoSDR [11], and LimeSDR [9], are potential options for this purpose. In our experiments, we used an USRP B210. While the attacker may opt for more advanced setup, e.g., amplifiers to extend the attack range, these are not essential components in our baseline attacker model.

A replay attack in RF communications is when an attacker captures the legitimate signals and retransmits them to execute authenticated actions on a system without authorization. This vulnerability poses significant risks as it can bypass various security measures, including data encryption. To perform a successful replay attack, the attacker does not need to know the packet’s format or contents. Replay attacks can even work against systems with encrypted protocols. In our targeted jamming attack, we capture and retransmit the signal similar to

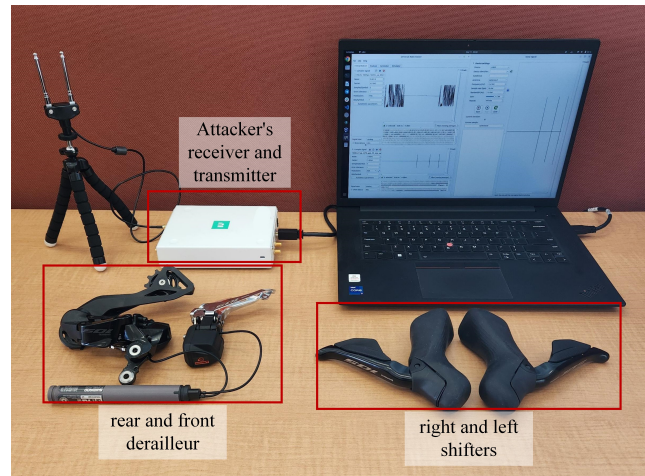


Figure 7: Our experimental setup, featuring the Shimano wireless gear shifting system (including shifters and derailleurs) and the USRP B210 as the attacker’s transceiver.

the replay attack. The methodology and details are explained in Section 4.3.

As previously mentioned, the attacker is equipped with an SDR; for our experiments, we used a USRP B210 to transmit and receive signals without external amplifiers. Figure 6 depicts the attack model. The attacker’s strategy involves capturing the signal emitted when the user engages the button to shift gears up or down. Once captured, replaying this signal enables gear shifting on the target’s bike. The attack works independently of the system’s current gear, effectively allowing for unauthorized control over gear adjustments. Figure 7 shows a photo of our evaluation setup.

A pre-requisite is that the attacker can capture a single upshift and downshift signal. There are several situations in which the attacker could collect these transmissions. An attacker does not need physical access to the bike; being in the vicinity is sufficient to capture the signal remotely in just a matter of seconds. For example, at a professional race, many individuals are within close proximity to a racer’s bike. The attacker can capture the signals on the fly as the victim rider is actually shifting their bike’s gears. Recall that our attack works irrespective of which gear the bike is currently in; thus, it is sufficient for the attacker to capture *any* upshift and *any* downshift signal. In a professional race, the attacker is easily within the signal range of the victim rider (e.g., riders are just a few feet apart).

4.2 Replay attack

We explored the mechanics and implications of replay attacks within the context of Shimano’s Wireless Gear Control Protocol. We detail our experimental setup and methodology using SDRs to transmit and receive signals, demonstrating how an attacker can exploit the system without needing to decrypt or

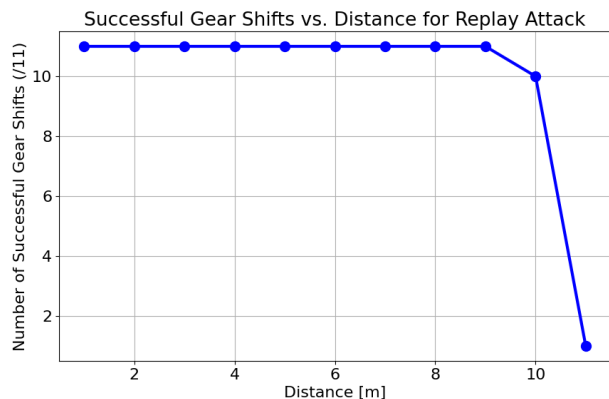


Figure 8: Assessing the Effective Distance for Replay Attacks without amplifier. Replay attack success rate vs distance from the target system.

even understand the signal’s content.

Figure 8 presents the outcome of our replay attack experiments. The distance is measured from the attacker’s transmitter to the bike’s rear derailleur. At various distances, we conducted tests to shift the gear from the lowest to the highest level, encompassing eleven levels in total. Our results indicate that the replay attack is effective up to a distance of 9 meters without encountering any failures. At a distance of 10 meters, we observed an average success rate of 10 out of 11 attempted gear shifts. Each test involved shifting through all 11 gears, from the lowest to the highest. Beyond 10 meters, the signal falls outside the effective range. Consequently, for the attack to be viable, the attacker must be within 10 meters of the target bike. All tests were conducted multiple times to ensure reliability. Specifically, each test was repeated at least five times across all shift levels. Additionally, for critical aspects of our study, such as measurements beyond 10 meters, we increased the number of repetitions to up to ten times to confirm the protocol’s effective range.

A critical point is that the attacker does not require direct physical access to the bike to capture and store the necessary signal. Once recorded, these signals can be reused at any future point without issue, owing to the lack of timestamps in the packet data.

The system completely lacks defenses against replay attacks, a finding reinforced by our ability to successfully replay the same signals two months after initially capturing the packets. Additionally, we conducted an experiment in which we recorded and replayed the signal, manually made at least 400 shifts, and subsequently performed the replay again, which proved to be effective. As long as the shifters remain paired with the same derailleur, the captured packets remain effective for replay.

Furthermore, by capturing just one instance of upshifting and one of downshifting from the targeted bike, an attacker

can create any sequence of gear shifts at varying intervals. This enables them to carry out attacks at any future point as long as the derailleur remains paired with the same shifters. We successfully created and executed our arbitrary sequences of upshifting and downshifting through replay attacks. In summary, this has the effect of creating an unauthorized shifter that completely controls the rear derailleur and the front derailleur of the victim. We successfully replicated the experiments by replaying the control commands (upshifting and downshifting) for the front derailleur and managed to take control of it.

4.3 RF Jamming Attack

A jammer operates as an RF transmitter, transmitting noise that interferes with wireless communications. Our study utilized two varieties of jammers: one generating random noise and another broadcasting sine and cosine waves. To assess the effectiveness of our jammers, we carried out tests under various conditions. Based on our observations, using sine and cosine waves for the jamming signal proved more effective than noise. Consequently, we conducted our experiments to assess the jamming range by generating sine and cosine waves.

We transmitted the generated signal precisely at 2.478 GHz to interfere with the communication, as this is the specific frequency used for all Shimano proprietary communications. Consequently, the jamming would affect all nearby bikes operating on this frequency. In our jamming tests, we positioned the shifter and the derailleur one meter apart, reflecting the typical distance between these components on a bike. We then experimented with the jammer at varying distances.

We use GNURadio [6] for generating our jamming signals and a USRP B210 to transmit them over the air.² The effectiveness of jamming depends on various factors, including the power of the jamming device, the type of signal being jammed, the environmental conditions, and the distance between the jammer and the receiver. The result of jamming can vary significantly based on these conditions. If the jammer is located anywhere within the one meter zone from the derailleur, the gear-shifting system becomes completely non-functional, losing all capability for successful communication.

Generally, jamming effectiveness increases as the jammer gets closer to its target. Outside the tested ideal jammer zone (1 meter from the derailleur), the jammer still disrupts communication to some extent, but it doesn’t completely disable the bike’s functionality. Our jamming range experiment was conducted using a baseline setup without the enhancement of amplifiers or directional antennas. There are multiple methods to make jamming more effective. Directional antennas, for instance, could intensify the jamming signal’s focus toward a particular area while lessening its effect elsewhere. The

²We note that although the devices operate in ISM band, care was taken to isolate the experimental setup in a separate area.

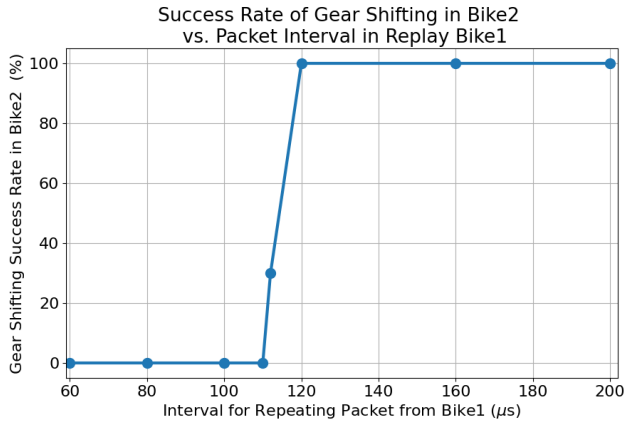


Figure 9: Analysis of Gear Shifting Success Rate in Bike2 Relative to Packet Interval Timing from Bike1: Demonstrating the Impact of Interference on Functionality

operational success and strategy of jamming can be greatly influenced by directionality, based on the jammer’s construction and strategic goals.

In a targeted jamming attack, the attacker would try to replay the signal on one bike so that it doesn’t cause any interference to other bikes. In our study, we labeled two Shimano wireless gear-shifting sets as Bike1 and Bike2. We captured an upshifting signal from Bike1 and replayed it at various intervals using a USRP B210. The targeted bike suddenly goes to the highest gear and stops there. Simultaneously, we attempted manual gear shifting on Bike2 in the vicinity. Our findings, illustrated in Figure 9, reveal that if the interval is less than 112 µs, which is equal to one packet length, Bike2 also stops working due to interference. The interference on Bike2 ceases when replay intervals exceed 112 µs, allowing normal communication due to sufficient time for transmitting command packets and receiving acknowledgments. In conclusion, when the attacker sends the replay packets with 112 µs interval, Bike1 ceases to function, whereas Bike2 or any other bike continues to operate normally.

5 Eavesdropping ANT Communication

Shimano utilizes the ANT+ protocol to transmit data, which devices like cycle computers can then pick up and display for cyclists. It’s important to note that the Shimano wireless gear shifting system does not send control commands via the ANT+ protocol. In ANT+ communication, multiple devices can connect to a single source. This means that with the Shimano network key, any nearby ANT+ receiver can pick up the data being transmitted. For instance, if two bikes are close together, the second bike can link to the first bike’s transmission and access its data simultaneously as the first bike is connected to its cycle computer. This allows an attacker to time their gear shifting replay based on precise knowledge of

Table 7: Configuration for Capturing Shimano’s ANT+ communication

Channel Frequency	57 (0x39), 2457 MHz
Network Key	A9-AD-32-68-3D-76-C7-4D
Channel Type	Master (0x10), Slave (0x00)
Device Number	1-65535, 0 searching
Device Type	1 (0x01)
Transmission Type	5 (0x05)
Channel Period	8198 counts, 4 Hz

what gear the victim rider is using.

We emphasize that eavesdropping on ANT+ communication does not form a core component of our attacker model. However, being able to target a specific gear through eavesdropping can indeed offer a strategic advantage to an attacker.

Table 7 outlines the configuration parameters necessary for capturing data using Shimano’s ANT+ protocol. The critical piece of information is the Shimano network key, a unique identifier that secures and enables communication on the Shimano ANT+ network. Cycle computers need this network key to capture ANT+ communications from Shimano devices. The Channel Frequency identifies the specific radio frequency used for communication, with channel 57 operating at 2457 MHz, which helps avoid signal interference. The Channel Type indicates whether a device acts as a ‘Master’ (initiating communication) or a ‘Slave’ (receiving data), with specific hexadecimal values for setup. The Device Number serves as a unique identifier within the ANT+ network. Numbers range from 1 to 65535, with 0 reserved for searching mode to connect with nearby devices. Device Type corresponds to the ANT+ standard for different device categories, with a type of 1 usually denoting a generic sensor. Transmission Type refers to specific patterns or information for device communication. The Channel Period details the frequency of data broadcasts, with ‘8198 counts’ equating to a 4 Hz rate, affecting both data timeliness and device battery life. Using the ANTware II [2] application and the correct configuration, we managed to intercept communications on the Shimano ANT+ network. ANTware II is a tool for managing ANT/ANT+ devices via an ANT+ USB stick. Figure 10 displays the captured packets during gear shifts from 8 to 1 on Shimano’s ANT+ network. The data highlighted in red represent the current gear values. Having a detailed knowledge of the network parameters and packet structure allows an attacker to easily replicate these packets.

6 Discussion

Shimano’s protocol incorporates basic encryption techniques to prevent attackers from creating counterfeit signals. Reverse engineering was notably demanding due to its use of GFSK modulation, which complicates demodulation when parame-

```

Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-08-3C-00-FF-FF
Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-07-3C-00-FF-FF
Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-06-3C-00-FF-FF
Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-05-3C-00-FF-FF
Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-04-3C-00-FF-FF
Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-03-3C-00-FF-FF
Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-02-3C-00-FF-FF
Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-01-3C-00-FF-FF
Received BROADCAST_DATA_0x4E
:: 4e, 00-00-FF-02-01-3C-00-FF-FF

```

Figure 10: Eavesdropping ANT communication. The data highlighted in red represent the current gear values in each packet.

ters are undisclosed. This is very different from the amplitude shift keying used by numerous security systems, which, due to its simplicity, leaves them more open to security breaches. In the current landscape, where many wireless devices communicate without encryption, exposing them to security threats, Shimano stands out by implementing encrypted communication, enhancing its defense against direct hacking. However, the system remains exposed to replay attacks. Below, we outline recommended strategies to mitigate the risk of replay attacks.

Effects of the Attacks. A modern bicycle typically has two derailleurs that control the chain position: rear and front. The rear derailleur typically has 11 or 12 levels, and the changes between levels are usually minor but still impact the rider’s performance. The front derailleur has two levels with large gear ratios. For example, consider a racer who is climbing a mountain. They will typically be in the smallest gear on the front. If the attack targets the front derailleur, causing it to move into a larger gear (i.e., harder for the rider), it can significantly impact rider performance, force them to stop, or even snap the chain. In professional races, any unintended changes to the gear position will have drastic consequences and affect the integrity of the sport. We believe that unauthorized gear changes through the attacks highlighted in our paper have a similar effect on the sport as performance-enhancing illegal drugs.

Size/Cost of Attack Device. In the current implementation of our signal capture and replay system, we utilize a setup comprising a SDR and a laptop. While effective, this configuration is not optimized for size or portability. However, with advancements in miniaturization and integrated circuit (IC) technology, it is feasible to reduce the size of the attack

device significantly. By custom designing specific circuits, we can integrate a receiver, a modest amount of memory for signal storage, and a transmitter into a compact, single System on a Chip (SoC) or small circuit board. This miniaturization process makes the attack system more discreet and enhances its portability and deployment ease. For example, researchers demonstrated relay attacks [20] on passive keyless entry systems with SDRs costing more than \$1500 in 2011. A few years later, the same attack was demonstrated using \$225 [12].

Countermeasures. Adding timestamps into wireless communications can mitigate replay attacks to some degree by allowing only messages sent within a designated timeframe, thereby rendering older, possibly replayed messages invalid. Nonetheless, integrating timestamps into wireless communication poses challenges. Effective use of timestamps requires precise synchronization between the devices. This can be challenging, particularly in settings where devices lack consistent access to a shared time source, such as the Internet or GPS signals.

Rolling or hopping codes stand as another prevalent strategy in wireless systems to prevent replay attacks. Within this framework, each transmitted signal is accompanied by a distinct code generated through a specific algorithm known to both the sender and receiver. These codes are one-time-use only, ensuring that once a code has been utilized for authentication, it is voided, prompting both devices to proceed to the subsequent sequence code. This method is especially prevalent in scenarios prone to signal interception and unauthorized reuse, such as passive keyless entry in cars and garage door systems. Although rolling codes significantly counter basic replay attacks, they are not foolproof against more sophisticated threats, such as code grabbing and delayed playback if an attacker intercepts the original code’s delivery to the receiver. However, this approach can significantly increase the difficulty of performing a replay attack in Shimano wireless gear shifting.

There are other types of countermeasures designed for specific applications that can be highly effective and useful. Particularly for Shimano bikes, implementing distance-based restrictions could be beneficial [35]. Since legitimate interactions occur only between shifters and derailleurs within limited distances, establishing range limitations on the receiver to only accept commands from close proximity can be helpful. This approach is based on the assumption that attackers are more likely to conduct replay attacks from a distance, so by restricting the range at which commands are accepted, we reduce the likelihood of successful remote attacks. However, securely measuring distance is a challenging problem [33, 36] in itself, and therefore, while it can reduce the risk of replay attacks, it should be used in conjunction with other security measures for comprehensive protection.

Our current observation indicates that it is likely that Shimano is not using any kind of rolling code or other mentioned

countermeasures. Our study reveals that the current security measures in Shimano's wireless gear shifting systems are insufficient to protect against replay attacks. The practical feasibility of executing these identified attacks demonstrates that attackers could exploit these vulnerabilities with relatively modest resources. Despite advancements in similar systems, the lessons learned have not been fully integrated into the design of Shimano's wireless gear shifters, leaving them vulnerable to specific cyber attacks such as those observed in automotive key fob systems. Moving forward, it is crucial to implement robust defenses, including rolling codes and other complementary security measures, to enhance the security guarantees of these wireless systems and safeguard against potential attacks.

E-TUBE PROJECT. The Shimano E-TUBE PROJECT is a platform that allows users to customize, update, and diagnose Shimano's electronic gear-shifting systems via BLE. When connected to the E-TUBE PROJECT over BLE, the rear derailleur would be out of operation. If the malicious attacker has any chance to physically access the derailleur, they can easily pair their phone with it and cause a DoS (Denial of service). The biker in this situation would not know what is wrong, and the only way to fix it is to completely disconnect the derailleur from the battery to cut off the power. Users are strongly advised to change the default passkey immediately after acquisition. Often, the initial pairing occurs at dealerships, which may result in the default code remaining unchanged if the end-user is not prompted to modify it. Furthermore, enhancing BLE security with unique, secure passkeys for each bike, rather than a standard default passkey, is recommended to prevent unauthorized access. If an attacker manages to connect the bike to his E-TUBE PROJECT, the implications can be severe. They could easily alter the bike's settings and even change the passkey, preventing any quick fix.

Future Work. In future research, we intend to expand our investigation into the security architecture of wireless gear-shifting systems beyond the scope of Shimano. We plan to analyze and compare various manufacturers' vulnerabilities and defense mechanisms, identifying common weaknesses and best practices within the industry. This comprehensive analysis will allow us to develop more robust security guidelines and recommendations for all wireless gear-shifting systems. Our goal is to ensure safer and more secure cycling experiences for users.

7 Related Work

In this section, we will review two primary areas of focus related to our work. First, we examine previous research on the reverse engineering of wireless systems. Second, we describe the security challenges posed by replay and relay attacks, exploring how these threats impact various technological domains.

7.1 Analysis of Proprietary Wireless Protocols

Many devices contain inherent security flaws. They often rely on security through obscurity by keeping their protocols and information secret, hoping this discourages efforts to reverse engineer and uncover potential vulnerabilities.

Various reverse-engineering studies have been conducted on different devices [18, 40], each with unique attributes and methodologies. For example, Garcia et al. [22] investigated the security of wireless smart cards used in payment systems, while Strobel et al. [39] examined a digital locking and access control system prevalent in corporations and educational institutions. Both studies required physically opening the devices to connect the wireless chips to a logic analyzer, which is invasive and could be easily detected compared to non-invasive techniques. Contrastingly, non-invasive reverse engineering, such as intercepting wireless communications using SDRs, offers a less detectable, scalable, and repeatable approach. This method avoids the complications of hardware tampering while still providing deep insights into wireless protocols. For example, [32] research on wireless mice and keyboards, which often use proprietary protocols in the 2.4 GHz ISM band.

Kim et al. [26] report instances in which authors could eavesdrop by recovering the 128-bit AES key. In [27], the process of demodulating RF signals into binary data for analysis was documented for a smart home alarm system known as SecuritasHome.

Researchers have recently adopted hybrid approaches for reverse engineering and launching attacks. Notable instances include Samy Kamkar's innovative methods for remote keyless entry systems [25] and Mike Ryan et al. [37] for electric skateboard control interfaces. Also, in [23], the authors focused on a case study with rolling codes.

Tools like URH [34] have aimed to streamline the reverse engineering process of wireless protocols, offering an open-source solution for signal capture and protocol analysis through SDRs. RFQuack [29] represents another advancement, a modular RF dongle system that allows for the customized development of dongles tailored to specific reverse engineering needs in wireless protocols. This tool underscores the evolving landscape of non-invasive techniques in security research.

In addition to academic studies, there have been non-academic reverse engineering efforts on the Shimano Di2 system [13, 14, 30]. However, these efforts primarily focus on reverse engineering the ANT communication protocol. To the best of our knowledge, none of these works have explored Shimano's proprietary protocol. Furthermore, none have investigated replay attacks or targeted jamming attacks on Shimano's command signals.

7.2 Replay and Relay Attack

Replay and relay attacks pose significant threats in wireless communications. It enables attackers to capture and rebroadcast packets for unauthorized access or service disruption, impacting various systems such as keyless vehicle entry, GPS, and remote garage door openers. For instance, previously, researchers have shown that through a relay attack, where a device is used to extend the communication between two legitimate devices, it's possible to unlock a vehicle and drive away even when the actual key is far from the car [20].

Similarly, GPS spoofing mirrors these concerns, with studies like [24, 31] demonstrating the potential for GPS signal manipulation, impacting navigation and timing. In RF communication, Roland et al. [21] explore relay attack risks in NFC transactions, commonly used in touchless payment and entry systems. The challenge in the abovementioned works would be relaying the signal in real-time. However, as shown in this paper, our attack on the wireless gear shifting system doesn't necessitate real-time relays and can be executed using any packet previously captured. RFID systems, crucial for secure access and transactions, face similar threats, with [38] addressing these system's susceptibilities to replay attacks.

Additionally, the increase in replay attacks on smart home systems underscores growing security gaps, as examined by Fernandes et al. [19], spotlighting exploitable weaknesses in smart home protocols.

8 Conclusion

The sport of cycling is an adversarial environment. Modern bicycles are cyber-physical systems that support wireless control of gear shifting. We conducted the first security analysis of the Shimano wireless shifting protocol and discovered its vulnerability to replay and jamming attacks. This allows attackers to target riders and take over control of the bike's gear shifting behavior. Allowing attackers such control can lead to negative outcomes on the performance of riders in professional races and can affect the integrity of the sport.

We discussed our analysis of Shimano's protocol with the hope that it would bring additional scrutiny to these technologies. We envision that future work will investigate the security of other wireless gear control manufacturers. Long term, we outlined countermeasures that manufacturers could use to reduce the impact of attacks. For example, a rolling codes system can reduce the attacker's ability to arbitrarily control gear changes.

Acknowledgements

The work was partially supported by NSF grant 2144914. We thank Keith Wakeham and Virgyl Fernandes for their technical expertise in cycling components, Andreas Noack for his

expert suggestions on URH, Yoshi Kohno for early discussions, the anonymous reviewers and our shepherd, Manuel Egele, for their insightful comments, and finally Stefan Savage, Geoff Voelker and the UCSD SysNet group for paper title ideas.

References

- [1] ANT / ANT+ Defined. <https://www.thisisant.com/developer/ant-plus/ant-antplus-defined>.
- [2] ANTware II. <https://www.thisisant.com/developer/resources/software-tools>.
- [3] Campagnolo. <https://www.campagnolo.com/>.
- [4] E-TUBE PROJECT. <https://bike.shimano.com/en-US/e-tube.html>.
- [5] Ettus Research. <https://www.ettus.com/products/>.
- [6] GNU Radio. <https://www.gnuradio.org/>.
- [7] HackRF. <https://greatscottgadgets.com/hackrf/one/>.
- [8] Inspectrum. <https://github.com/miek/inspectrum>.
- [9] LimeSDR. <https://limemicro.com/products/boards/limesdr/>.
- [10] New shimano 105 di2 delivers 12-speed, shifting and wireless performance. <https://bike.shimano.com/en-US/information/news/new-shimano-105-di2-delivers-12-speed-shifting-and-wireless-per.html>.
- [11] PlutoSDR. <https://wiki.analog.com/university/tools/pluto>.
- [12] Radio Attack Lets Hackers Steal 24 Different Car Models. <https://www.wired.com/2016/03/study-finds-24-car-models-open-unlocking-ignition-hack/>.
- [13] REVERSE ENGINEERING SHIMANO BIKE ELECTRONICS. <https://hackaday.com/2019/03/26/reverse-engineering-shimano-bike-electronics/>.
- [14] Reverse Engineering Shimano DI2. <https://titanlab.co/reverse-engineering-shimano-di2/>.
- [15] Shimano: Bike component monopoly at an attractive price. <https://seekingalpha.com/article/4572435-shimano-bike-component-monopoly-at-an-attractive-price>.
- [16] Shimano dura-ace di2. <https://bike.shimano.com/en-EU/product/component/dura-ace-r9200.html>.
- [17] SRAM. <https://www.sram.com/>.
- [18] CHOI, K., SON, Y., NOH, J., SHIN, H., CHOI, J., AND KIM, Y. Dissecting customized protocols: automatic analysis for customized protocols based on ieee 802.15. 4. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2016), pp. 183–193.
- [19] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security analysis of emerging smart home applications. In *2016 IEEE symposium on security and privacy (SP)* (2016), IEEE, pp. 636–654.
- [20] FRANCILLON, A., DANEV, B., AND CAPKUN, S. Relay attacks on passive keyless entry and start systems in modern cars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2011), Eidgenössische Technische Hochschule Zürich, Department of Computer Science.
- [21] FRANCIS, L., HANCKE, G., MAYES, K., AND MARKANTONAKIS, K. Practical relay attack on contactless transactions by using nfc mobile phones. *Cryptology ePrint Archive* (2011).
- [22] GARCIA, F. D., DE KONING GANS, G., MUIJERS, R., VAN ROSSUM, P., VERDULT, R., SCHREUR, R. W., AND JACOBS, B. Dismantling mifare classic. In *Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings 13* (2008), Springer, pp. 97–114.

- [23] GHANEM, A., AND ALTAWY, R. Garage door openers: A rolling code protocol case study. In *2022 19th Annual International Conference on Privacy, Security & Trust (PST)* (2022), IEEE, pp. 1–6.
- [24] HUMPHREYS, T. E., LEDVINA, B. M., PSIAKI, M. L., O’HANLON, B. W., KINTNER, P. M., ET AL. Assessing the spoofing threat: Development of a portable gps civilian spoofer. In *Proceedings of the 21st International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS 2008)* (2008), pp. 2314–2325.
- [25] KAMKAR, S. Keysweeper, 2015.
- [26] KIM, K., KIM, T. H., KIM, T., AND RYU, S. Aes wireless keyboard: Template attack for eavesdropping. *Black Hat Asia, Singapore* (2018).
- [27] LINDEBERG, A. Hacking into someone’s home using radio waves: Ethical hacking of securitas’ alarm system, 2021.
- [28] LLOYD, D. The fight for the crown: Shimano vs sram, 2018. <https://www.bicycling.co.za/bikes-gear/the-fight-for-the-crown-shimano-vs-sram/>.
- [29] MAGGI, F., AND GUGLIELMINI, A. Rfquack: A universal hardware-software toolkit for wireless protocol (security) analysis and research. *arXiv preprint arXiv:2104.02551* (2021).
- [30] MILLER, S. Shimano Di2 Security. https://www.youtube.com/watch?v=Kxocl08qB60&ab_channel=ShaneMiller-GPLama.
- [31] MOTALLEBIGHOMI, M., SATHAYE, H., SINGH, M., AND RANGANATHAN, A. Location-independent gnss relay attacks: A lazy attacker’s guide to bypassing navigation message authentication. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (2023), pp. 365–376.
- [32] NEWLIN, M. Mousejack: Injecting keystrokes into wireless mice. Retrieved January 10 (2016), 2019.
- [33] ÓLAFSDÓTTIR, H., RANGANATHAN, A., AND CAPKUN, S. On the security of carrier phase-based ranging. In *International Conference on Cryptographic Hardware and Embedded Systems* (2017), Springer, pp. 490–509.
- [34] POHL, J., AND NOACK, A. Universal radio hacker: A suite for analyzing and attacking stateful wireless protocols. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)* (2018).
- [35] RANGANATHAN, A., AND CAPKUN, S. Are we really close? verifying proximity in wireless systems. *IEEE Security & Privacy* 15, 3 (2017), 52–58.
- [36] RANGANATHAN, A., DANEV, B., FRANCILLON, A., AND CAPKUN, S. Physical-layer attacks on chirp-based ranging systems. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks* (2012), pp. 15–26.
- [37] RYAN, M., AND HEALEY, R. Hacking electric skateboards: Vehicle research for mortals, 2015. <https://www.defcon.org/html/defcon-23/dc-23-speakers.html>.
- [38] SINGH, A. K., AND PATRO, B. Security attacks on rfid and their countermeasures. In *Computer Communication, Networking and IoT: Proceedings of ICICC 2020* (2021), Springer, pp. 509–518.
- [39] STROBEL, D., DRIESSEN, B., KASPER, T., LEANDER, G., OSWALD, D., SCHELLENBERG, F., AND PAAR, C. Fuming acid and cryptanalysis: Handy tools for overcoming a digital locking and access control system. In *Advances in Cryptology-CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I* (2013), Springer, pp. 147–164.
- [40] STUTE, M., KREITSCHMANN, D., AND HOLLICK, M. Reverse engineering and evaluating the apple wireless direct link protocol. *GetMobile: Mobile Computing and Communications* 23, 1 (2019), 30–33.
- [41] ZHANG, Y., WENG, J., DEY, R., JIN, Y., LIN, Z., AND FU, X. Breaking secure pairing of bluetooth low energy using downgrade attacks. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 37–54.

Engineering a backdoored bitcoin wallet

Adam Scott
Block, Inc.

Sean Andersen
Block, Inc.

Abstract

Here we describe a backdoored bitcoin hardware wallet. This wallet is a fully-functional hardware wallet, yet it implements an extra, evil functionality: the wallet owner unknowingly leaks the private seed to the attacker through a few valid bitcoin transactions. The seed is leaked exclusively through the ECDSA signatures. To steal funds, the attacker just needs to tap into the public blockchain. The attacker does not need to know (or control) any aspect of the wallet deployment (such as where in the world the wallet is, or who is using it). The backdoored wallet behavior is indistinguishable from the input-output behavior of a non-backdoored hardware wallet (it is impossible to discern non-backdoored signatures from backdoored ones, and backdoored signatures are as valid and just “work” as well as regular, non-backdoored ones). The backdoor does not need to be present at wallet initialization time; it can be implanted before or after key generation (this means the backdoor can be distributed as a firmware update, and is compatible with existing bitcoin wallets). We showcase the feasibility of the backdoored wallet by providing an end-to-end implementation on the bitcoin testnet network. We leak an entire 256-bit seed in 10 signatures, and only need modest computational resources to recover the seed.

Version: 2024-05-30

1 Introduction

Bitcoin, introduced in 2008 [Nak08], transacts in 2024 over 5 billion USD per day. Bitcoin is a distributed ledger that accumulates signed transactions conveying movement of funds. Cryptocurrencies like bitcoin are designed so that access to cryptographic keys directly controls the ability to move funds. In other words, in a cryptocurrency system, a security or cryptographic mistake normally results in the loss of funds.

Cryptocurrency users typically handle cryptographic keys in one of two ways: either they engage with a *custodian* to handle keys on the customer’s behalf; or the users themselves

store keys in specific-purpose *hardware wallets*.¹ Hardware wallets are essentially small-scale HSMs that at a minimum store (or derive) keys and sign transactions using public-key cryptography. Examples of hardware wallets are Trezor [Sat22] or Ledger [SAS22]. Hardware wallets typically use a key derivation strategy such as BIP32 [Wui12] to simplify storage requirements and overall key management complexity. In a deterministic key derivation strategy like BIP32, all private keys are deterministically derived from a master secret seed using a suitable key derivation function (like HMAC-SHA512 in the case of BIP32).

This paper focuses on cryptographic subversion of bitcoin wallets. We assume an attacker designs a backdoor and can inject custom firmware on the hardware wallet unit. The objective of the backdoor designer is to steal the wallet funds after the wallet is deployed. The custom wallet firmware contains a backdoored implementation of the signature scheme. This signature scheme is sound: it produces valid signatures. In addition, signatures themselves contain an extra, subliminal message embedded onto the signature values. This subliminal message is recoverable only to the backdoor designer. In our scenario, the subliminal message is the secret seed that generates all wallet private keys. Note that it is very easy for the attacker to obtain the signatures: since signatures are part of transactions and hence public information, the backdoor designer can easily recover the seed by monitoring the blockchain.

Previous work. Simmons introduced the problem of subliminal messages in signatures schemes and gave precise constructions back in the 1980s [Sim83, Sim84, Sim85]. Later Young and Yung generalized and coined the term “kleptography” to refer to backdoored cryptographic algorithms designed to steal information through covert channels [YY96, YY97a, YY04]. They also gave a very efficient kleptographic version of DSA that just requires two leaked

¹Alternatively, the user can store the key directly on their personal device. This is considered to be riskier since general-purpose computers have a richer attack surface.

signatures [YY97b] assuming the signer is stateful. For formal definitions, see Ateniese et al. [AMV15, AMV20]. Cryptographic subversion is widespread: a notorious example of a (standardized) backdoor is Dual EC PRNG [SF07, BLN16].

Our contribution. This paper details the design and implementation of a flavor of Simmons backdoored DSA, heavily tailored towards an actual deployment onto a bitcoin hardware wallet. We focus on crafting a stealth backdoor that can be readily deployed in many different scenarios. We implement this backdoor end-to-end using the bitcoin testnet network and discuss optimizations and limitations. The backdoor designer can recover the whole seed with just 10 signatures and modest computational resources. We hope this example contributes to the development of more secure wallets and ecosystems.

Applicability to other cryptocurrencies. This paper focuses on bitcoin, but the ideas easily transfer to other cryptocurrencies that use ECDSA with minimal modifications. Similarly, while this paper focuses on “personal” hardware wallets like Trezor or Ledger, the same observations carry to special-purpose, cold storage solutions.

1.1 Attacker model

We assume the adversary has control over the wallet code. Gaining control over a hardware wallet firmware can be attained by a variety of paths, including supply-chain attacks on any wallet software components, a compromise of the build system, a malicious insider with write access to source code or a compromise of the firmware signing key.

On stealth backdoors. We note that when an attacker has achieved (essentially) remote-code execution in the hardware wallet firmware, there are many other exfiltration vectors available. For instance, a malicious wallet firmware could exfiltrate keys in unused fields in a bitcoin transaction, or resort to fancier physical exfiltration channels such as EM leakage or sound (if targeting a deep cold storage appliance). There is a plethora of work in this direction (sometimes called “covert channels”) [KA98, VP09, LU02, GMME15, MSST05, GSE20]. However, these exfiltration channels often introduce new assumptions that significantly increase the cost of the attack: either by requiring a compromise of other system components (as needed if the exfiltration technique relies on introducing new messages) or some kind of physical proximity (as needed in all EM / sound leakage techniques). Our exfiltration technique works in a significantly more constrained setting (our attacker model is weaker) since we rely on less assumptions. We do not require compromise of additional system components nor require physical proximity (not even knowledge about the specific deployment). Thus, our techniques are more broadly applicable and can be universally deployed on bitcoin wallets.

2 Design space

Informally, we design a ECDSA signing algorithm featuring extra functionality: the signature values (r, s) convey extra information that allow the backdoor designer to extract the BIP32 seed. The basic idea traces back to Simmons, 40 years ago.

2.1 Backdoored wallet requirements

We present in this section a brief reminder of what makes a good cryptographic backdoor plus some particular properties of good backdoored wallets. This concept was first introduced by Young and Yung [YY96]. The following properties are desirable:

- R1 Backdoor is efficient, both for signing and leaking. On the one hand, this means there is no noticeable performance loss in the signer (signing is fast), and that only a handful of signatures are required to leak the secret (the subliminal channel has appropriate bandwidth). We assume the attacker can afford some moderate computation when recovering the secret from the signatures.
- R2 Backdoor is stealth: backdoored signatures should be computationally indistinguishable from non-backdoored ones. This ensures the backdoor is undetectable from its input/output behavior. Relatedly, the backdoor should be sound: leaked secrets are unrecoverable to anyone without the backdoor recovery seed.
- R3 Backdoor is suitable for deployment in a typical cryptocurrency scenario. This means that the backdoor system should tolerate loss of signatures (for example, a transaction is signed but never broadcasted) or reordering (in general, we cannot assume the order the signatures are generated is the same as transactions in the blockchain). The backdoor should be able to leak arbitrary data, not just the long term ECDSA key. Also, in typical deployments of hardware wallets, each signature is generated under a different long-term key (this is the case in BIP32). This means the backdoor should work when the long-term key changes from signature to signature. In addition, the backdoor should be stateless: in some systems, non-volatile storage may not be available (either as a security feature) or storing extra data in non-volatile storage may not be desirable (to lower detection probability, and simplify backdoor deployment).

2.2 Backdoor syntax

We first revisit the syntax of non-backdoored signature schemes and later augment it to construct backdoored signature schemes.

Digital signature. We adopt the usual syntax for a digital signature scheme, consisting of the following three algorithms:

- $\text{Sig.KeyGen}() \rightarrow (\text{pk}, \text{sk})$ generates a public/private key pair.
- $\text{Sig.Sign}(\text{sk}, m) \rightarrow \sigma$ emits a signature σ on a message m using private key sk .
- $\text{Sig.Verify}(\text{pk}, m, \sigma) \rightarrow \{0, 1\}$ verifies signature σ on message m using public key pk .

For a definition of the ECDSA algorithm, see Appendix A.

Backdoored signatures. A backdoor signature scheme augments a signature scheme with two extra algorithms and modifies the signing algorithm:

- $\text{BSig.SysParam}() \rightarrow (\text{bp.sign}, \text{bp.extract})$. Generates backdoor system parameters required for signing bp.sign and for recovery bp.extract . The attacker typically runs this algorithm once to initialize the backdoor parameters.
- $\text{BSig.Sign}(\text{sk}, m, S, \text{bp.sign}) \rightarrow \sigma$. Generates a backdoored signature σ on message m . The signature σ conveys the subliminal message S (potentially only partially) to be exfiltrated.
- $\text{BSig.Recover}(\{\sigma_i\}, \text{bp.extract}) \rightarrow l$ recovers the subliminal message S from a set of signatures $\{\sigma_i\}$.

In practice, the signature scheme BSig.Sign may not fully leak the whole subliminal message S in a single signature, but only a part of it. To address this, the backdoor designer implements BSig.Sign to leak different parts of S on each invocation. Conversely, BSig.Extract takes multiple signatures $\{\sigma_i\}$ to reconstruct S . In other words, the backdoor designer may bake message fragmentation and reassembly into the backdoored algorithms.

3 Backdoor description / our construction

Basic idea overview. The backdoor designer wants to leak an arbitrary value S . For example, S could be the BIP32 seed that generates the whole wallet. The basic idea is to leak S in two steps. The first step is splitting S into several partial subliminal messages l_i . The second step is picking the ECDSA nonce to encode l_i into σ_i such that BECDSA.Extract can recover l_i from σ_i .

We write in Figure 1 the full description of the backdoored signature scheme BECDSA . In the next sections, we describe step by step the construction of BECDSA .

Backdoor description:

- $\text{BECDSA.SysParam}() \rightarrow (\text{bp.sign}, \text{bp.extract})$. Generate an ECIES public/private key pair esk, epk . Generate a backdoor recovery secret $b \in_R \mathbf{Z}_q^*$. Return $\text{bp.sign} \leftarrow (\text{epk}, b)$ and $\text{bp.extract} \leftarrow (\text{epk}, b)$.
- $\text{BECDSA.Sign}(\text{sk}, m, S, \text{bp.sign}) \rightarrow \sigma$. Signing consists of two parts. First we compute a subliminal message to leak $l = p(S, m, \text{bp.sign})$. Then we generate a signature that leaks l : compute a nonce r from the message m , subliminal message l and backdoor recovery secret b as $r \leftarrow l \cdot h_1(m, b) \in \mathbf{Z}_q$. Call $\text{ECDSA.Sign}(\text{sk}, m; r)$ and emit the signature.
- $\text{BECDSA.Recover}(\{\sigma_i\}, \text{bp.extract}) \rightarrow S$. Given a collection of signatures σ_i , extract each subliminal message $l_i \leftarrow \text{Extract}(\text{bp.recover}, \sigma_i)$ and then invert p using subliminal messages l_i to recover S .

Figure 1: BECDSA backdoor description.

Notation and parameters. We write h for a hash function, and add a subindex h_i when we want different hash functions to separate domains. We work with the prime order group \mathbf{G} of elliptic curve points. In the bitcoin case, this is the secp256k1 curve.

3.1 Step 1: construction rationale

Here we detail the construction rationale for step 1, performing fragmentation and assembly.

Mapping S to l_i . We need a way to map the 256-bit S secret to several “short”² L -bit l_i . Simply assigning l_i to each L -bit chunk of S would work, but has important drawbacks. This raw method requires the receiver to receive all the subliminal messages l_i in order, and does not tolerate loss of l_i . In the context of bitcoin transactions, these two points can be hard to guarantee in practice since the signer may not broadcast every transaction (loss of l_i) or transactions may be broadcasted in a different order in which they were generated.

A more robust approach is to use the message m itself to “select” which linear combination of few bits from S are actually leaked. Since the message is public, BECDSA.Recover can invert this operation. More precisely, we use bit vector-matrix multiplication to “compress” S :

$$l_i \leftarrow S\mathbf{M} \quad (1)$$

where S is seen as a 1×256 row vector of bits and \mathbf{M} is a

²Typically, L is in the range of few dozen bits. This is a constraint coming from §3.2.

$256 \times L$ public matrix. The output l_i is L bits long. Intuitively, each l_i “picks” a “random” combination of bits from S .

The matrix \mathbf{M} is spanned from the message m . Each message m generates a unique matrix \mathbf{M} . Since the message is public, the backdoor designer can easily reconstruct \mathbf{M} . For the concrete definition of \mathbf{M} see Appendix B.

Recovering S from $\{l_i\}$. Recovering S from multiple l_i is very easy once enough l_i are recovered from the signatures σ_i . Each l_i adds L linear equations to a linear system of equations over $\text{GF}(2)$. Solving for S recovers the seed.

Mapping properties. Using a map like Eq. (1) tolerates partial loss and reorder of some l_i . It also provides an entertaining feature: the backdoor designer gets an early progress report on how many bits from S are left to guess by computing the kernel dimension of the linear mapping. Note that it is not necessary to have a determined or over-determined system, but we only need to collect enough equations so that the last remaining bits can be bruteforced. We elaborate in §4.1.

3.2 Step 2: construction rationale

In what follows, r is the ECDSA secret nonce (also called short-term key) and the signature is the pair $\sigma := (c, s)$ where $c = f(g^r)$ (in our case, f just returns the x -coordinate of the curve point). For a complete description of ECDSA and notation, see §A.

Picking nonces. The basic working principle of backdoor-ing ECDSA signatures is that the signer picks the ECDSA secret nonce r to convey the subliminal message l_i , a way that the backdoor designer can recover r and thus l_i . At the same time, the nonce r should still be unpredictable for someone that does not know the backdoor secret b . This ensures that the security of the ECDSA signatures is preserved.³ An easy way to do this is by setting $r \leftarrow b \cdot l_i$.

Cross-stealing resistance. The basic method for cooking nonces $r \leftarrow b \cdot l_i$ makes a very fragile signature scheme. Since l_i is small, a collision between r_i can happen with large probability, leading to complete loss of security of the signing key (if the same key is used for different signatures⁴). To fix this, we diversify the backdoor recovery secret b on a per-message basis as $b_i \leftarrow h(b, m)$ where h is a suitable key derivation function and set $r \leftarrow l \cdot b_i$. This ensures the security of the signature scheme is preserved (no one but the backdoor designer can steal funds).

³Note that the “long-term” ECDSA key sk may change across invocations; thus, different signatures may correspond to different public keys. This happens in modern bitcoin wallets that are based on HD derivation [BIP32].

⁴While we cannot assume the same key sk is used for different messages, we cannot either discard this possibility, and the backdoor should yield a good signature scheme even if this is the case.

Recovering $\{l_i\}$ from $\{\sigma_i\}$. To recover l_i , the attacker computes the discrete logarithm of rG with respect of bG . This is relatively easy since l_i is small by construction. One straightforward procedure to solve this discrete logarithm is to just iterate over all possible 2^L values of l_i . More precisely, to recover l_i from a signature $\sigma_i = (c_i, s_i)$, we first unblind the first signature component $s = f(b_i l_i G)$ and compute the point $V_i \leftarrow b_i^{-1} s_i G = l_i G$. From V_i , we extract l_i as $l_i \leftarrow \text{Extract}(V_i)$. This procedure simply iterates over all possible l_i until finding the value. We describe the computational optimizations to speed up this process below in §5.

Discovery resistance. Assume the backdoor implementation gets leaked. This includes the secret b . Everyone who knows b can recover the leaked seed S . If this is a concern, then the backdoor implementation should leak a public-key encryption $E(S)$ of S instead of bare S . A good choice for the encryption functionality E is ECIES [Sho01] over secp256k1. This greatly simplifies the implementation as all the elliptic curve machinery is already in the signing implementation. The only drawback of this approach is ciphertext expansion: ECIES roughly doubles⁵ the size of the subliminal leakage (512 bits), requiring more signatures to be leaked.

4 Discussion

4.1 How many signatures are needed to leak the full seed?

Tradeoff between L and number of signatures. There is a tension between the number of leaked bits per signature L and the number of required signatures to reconstruct the seed S . Obviously, we want to keep the number of required signatures as low as possible to leak the seed as soon as possible. That forces a high L , which in turns makes the recovery computationally expensive. The running time of `BSig.Recover` is exponential in L . We study this tradeoff in this section.

How much efficiency are we losing? The map from Eq. (1) is quite efficient: there is little redundancy across multiple l_i . Each l_i as generated per the mapping from Equation (1) leaks approximately L bits of S . This is because the rank of a random square matrix⁶ over $\text{GF}(2)$ is very close to its di-

⁵Pure trapdoor-based public key encryption do not yield a more efficient scheme. RSA encryption would add zero overhead, but obviously for the parameters in question RSA-256 would provide an unacceptable security level. Even state-of-the-art trapdoors with low overhead [DGH⁺19] still would be more expensive than KEM+DEM for such a short plaintext size (256-bit). For example, if the construction from [DGH⁺19] is asymptotically perfect (rate-1 ciphertext expansion), for 64-byte messages the ciphertext is 274-byte long [DGH⁺19, §6]. Another alternative is to use ECIES over a smaller curve.

⁶This naturally assumes the matrix \mathbf{M} is constructed as per §B which can be considered uniform essentially random in $\{0, 1\}$.

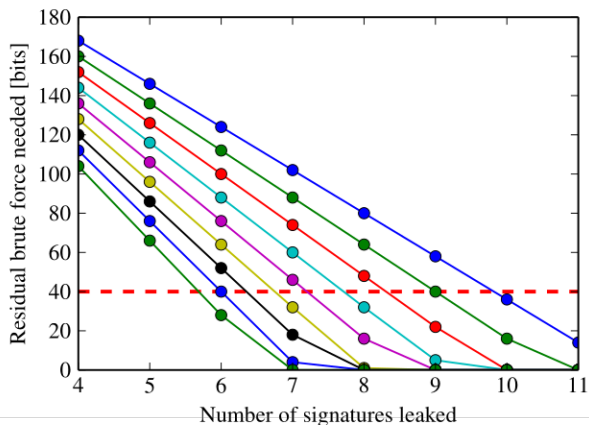


Figure 2: Remaining brute force effort needed to recover a unique S after solving the linear system from (1), for varying number of signatures leaked and bits per signature L . From top to bottom, each line corresponds to $L = 22, 24, \dots, 38$. We plot a horizontal line at 40-bit effort level (representing a feasible brute force effort).

mension [Kol99, §3.2]. To get full rank with high probability, it suffices to add a few additional rows. This means that we need to leak around $256/L$ signatures to have a system of equations with almost unique solution for S .

Using less than $256/L$ signatures. Typically, we have access to an oracle that allows us to distinguish a correct guess for S from an incorrect one. (For example, when S is a seed, we can quickly tell if a guess for S is correct by checking if S unlocks some outputs.) This means that we do not need a fully determined system of equations to solve for S . When the system is not fully determined, the remaining few bits can be bruteforced by exhaustively generating all solutions for the system and checking each candidate. Note that generating solutions is a very efficient linear algebra operation (span the null space). This process only works when the remaining bits to be brute force is kept low (e.g. under 2^{40}).

Empirical results. In Figure 2 we empirically study how many signatures we need to leak the full secret S . As an example, we can see that if we allow $L = 35$ bits leaked per signature, after leaking 7 signatures there is only 11 bits left to brute force. This means that we can leak the whole 256-bit HD wallet seed with a handful of signatures. Setting $L = 35$ bits per signature is feasible, as detailed below in §5.

4.2 Deterministic signatures

ECDSA signatures come in two broad flavors: randomized and derandomized (aka *deterministic signatures* [MNPV98, Por13]).⁷ A backdoor signature scheme should mimic the existing wallet behavior to remain stealthy. Otherwise, the backdoored scheme is trivially distinguishable.

Deterministic backdoored signatures The scheme as described in Section 3 is deterministic, except for the ECIES encryption E for discovery resistance. To make ECIES deterministic, we can use a Encrypt-with-hash variant⁸ from Bellare et al. [BBO07, §5.1]. This is possible in our case since the plaintext input to ECIES comes from a space with large min-entropy (a 256-bit seed S). This reduces to using a hash of the plaintext S as the randomness required for ECIES.

Randomized backdoored signatures. If the backdoor designer wishes to emulate a “randomized” version of ECDSA, they can randomize the backdoored nonce r by multiplying by a small integer v as $r \leftarrow v \cdot h(b, m) \cdot l_i$. This comes at an increased cost at recovery time (exponential in the bitsize of v). The random factor v should be large enough so that collisions are below a threshold backdoor detection probability. This is not a fully randomized ECDSA, since typically to be indistinguishable from a real random ECDSA signature, the value v would need to be very large (in the order of 128 bits). As a result, this backdoored ECDSA is not fully randomized, but may be useful to avoid light detection.

4.3 Recovery discussion

Variant: lighter recovery. To improve recovery speed, set $r \leftarrow b \cdot 2^i$ instead of $r \leftarrow b \cdot l_i$. This will speed up recovery, since it replaces elliptic curve point additions by point doublings, which are typically faster.

Identifying backdoored signatures. By design, there is no “in-protocol shortcut” to determine which signatures on the blockchain the recovery procedure should be applied to. This means that the backdoor designer should apply the recovery procedure to every signature, and discard those signatures for which the recovery procedure fails (i.e. does not yield a subliminal message l_i). Note that the backdoor designer could make use of additional side-channel information outside the raw ECDSA signature (like the way the transaction looks in

⁷Derandomized ECDSA signatures uses a deterministic process to generate the (pseudo-)random value k needed at signing time. They are preferred in practice since they are more resilient to imperfect randomness (at the cost of slightly increased computation). Many bitcoin wallets implement this strategy, usually in the form of RFC6979.

⁸This deterministic public-key encryption construction does not attain the usual standard level for encryption (semantic security) but in our specific case this is acceptable [BBO07].

vulnerable wallets) to speed up this process, but this is not necessary and orthogonal to our case.

Outsourcing recovery We note here that when using the discovery resistance feature from §3.2, it is possible to outsource the computationally expensive process in recovery to a different party. This party does not learn the content of S (only $E(S)$). Thus, this party, without access to `bp.extract`, cannot steal funds, nor correlate to a specific transaction. This can be useful to externalize this computationally expensive process. This party can also amortize its computational effort across different back door users, potentially by doing a heavy pre-computation upfront and amortizing across different clients.

5 Recovery implementation

In this section, we focus on the `Extract` procedure, which is the most computationally demanding procedure from `Recover`. The `Extract(rP)` takes a curve point $Q = rP$ and outputs the discrete logarithm with respect to P , assuming r is bounded $0 < r < 2^L$.

Basic implementation. The basic implementation is just a linear search on l_i . `Extract(V_i)` essentially loops sequentially over candidate l'_i till it hits $l'_i G = V_i$. The cost is 2^L elliptic curve additions and point comparisons, which is manageable when we keep L small. We can optimize this search at different levels.

Optimizations: baby-step giant-step. First, we can apply a classic time-vs-memory tradeoff (TMTO) by precomputing a table $T[j]$ storing M multiples of G , evenly spread over the search interval (from G to $2^L G$). This speeds up the search since `Extract` just needs to compute L/M additions $V_i + G, V_i + 2G, \dots, V_i + \frac{L}{M}G$ and on each step check for inclusion on $T[j]$ to recover l_i . This is essentially baby-step giant-step algorithm to solve discrete logarithms.

Optimizations: point representation. Secondly, we can keep the points in Jacobian form (thus making point addition very fast) and perform the inclusion check on $T[j]$ after converting to affine representation. The speed-up comes from batching several points in this conversion and leveraging batched modular inversion.

Optimizations: compressed table. To lower memory requirements we can store compressed points in the table T (just the x -coordinate). This compression could be loopy (a short “fingerprint” of each point, such as some bits from the x -coordinate), at the cost of false positives (which can be filtered out easily).

Optimizations: parallelization. This search is amenable to parallelization at different levels. First, the search is embarrassingly parallel on the search interval $1, \dots, M$. Second, SIMD operations can speed-up the search by computing in parallel different chains of V_i . Note that in contrast with the usual context of elliptic-curve cryptography, the main objective in this search is to maximize *throughput* in point operations, not latency.

Implementation results. We wrote a prototype in Go featuring the TMTO optimization. This implementation tests about $2^{37.2}$ candidates l_i per second on a single core of a 2014 MacBook Pro with a table T holding 2^{22} points (compressed to 64 bits of the x -coordinate). The implementation is concise and takes around 30 lines of code. It is not particularly optimized for speed. It relies on `math.big` for multiprecision integers (field arithmetic is not optimized for `secp256k1`). For fast lookups, the table T is implemented as a hash map.

Real-time detection. In this section we see how quickly can an attacker leak a full 256-bit seed S if their computing power is the 2014 laptop from the section above. The fact that a single laptop can test about $2^{37.2}$ candidates per second means a single laptop can run `Extract` on real time on every transaction getting mined on the blockchain when each signature is leaking at most $L \leq 34$ bits. This rough estimate assumes the blockchain has a throughput of 8 transactions per second⁹. In turn, by looking up Figure 2, leaking $L = 34$ bits per signature means after just 7 signatures are leaked, there is enough information leaked to completely recover the seed S . (Leaking 7 signatures leaks about 238 bits, and the remaining 8 bits can be easily bruteforced).

Naturally, the estimations above are done with a single 2014 laptop as computing device. If the attacker has fancier hardware, they can use it to leak more bits per signature and accelerate the process, as they would need to leak less signatures.

Other techniques. It might be tempting to implement `Extract` based off generic techniques to solve the discrete logarithm problem in an interval. One such method is Pollard’s lambda algorithm (also called Pollard’s Kangaroo). This is left as future work.

6 Experiments

We implemented the backdoor end-to-end. This experimental backdoored wallet software runs on a laptop (rather than an actual hardware wallet) to make it easy to perform experiments. We implement the full backdoor except the “discovery

⁹This is a conservative estimation, the actual number is more between 3 and 7

resistance” feature (ECIES encryption) discussed in §3.2. The implementation is written in python for simplicity.

Results. We leak 19 bits per signature. The implementation signs 13 transactions. Signing overhead is negligible (an additional vector-matrix multiplication). The recovery process recovers the leaked arbitrary message in a matter of seconds. We set the leaked message to: 0x1234567890123456789012345678901234567890123456789012345678901234. We extract in total 247 bits from the signatures and the remaining 9 bits we just brute force by going through all the 2^9 solutions to the linear system of equations.

Transaction hashes. We put a chain of 13 transactions on signet (a Bitcoin staging network) starting with the transaction f019bd3461309ae48c48a9cee5edaefb8ff4ef4c921fbd9d43377ff64162d77b. The full list of transactions can be found in Appendix C and the example output of the recovery tool can be found in Appendix D.

Generalizations. Whilst these concrete transaction chain uses the same private key for every transaction, this is coincidental (to make the implementation easier) and not essential. Therefore, the backdoor is compatible with BIP32 wallets. Also, even if these concrete transactions are chained, this is not a requirement. The different backdoored signatures could come from unrelated, unlinked transactions.

7 Detection, deployment

7.1 Distinguishing backdoored signatures

We elaborate here on the requirement R2 from §2.1.

Unknown-key scenario. For an observer that does not know the secret key material (but only observes the black-box, input/output behavior of the wallet), backdoored signatures are indistinguishable from regular ones. This follows from the PRF security of the hash function $h(b, m)$.

Known-key scenario. For an observer that knows secret key material (or can choose the key), the backdoored scheme is trivially distinguishable. For example, the backdoored scheme will not pass test vectors. This observer can take the reference test vectors from a known-good implementation (or from the signature algorithm specification, such as RFC6979). Note that a backdoor could easily hide itself (by computing non-backdoored signatures) whenever it detects from its environment that it is running in test mode (for example, the backdoor could detect if it is being fed test vector inputs from a publication) or is running in a developer machine, or in the software build pipeline, or is using a test key, or the key was not generated internally at random inside the signer device.

7.2 Deployment aspects

Build pipeline. An attacker can plant the backdoor by compromising the software build pipeline. These systems are typically operated by different teams and could become an easy target.

Stealing firmware signing keys. Alternatively, this backdoor could also be planted by stealing the firmware signing keys. Conversely, note that the firmware writer has a *lifelong liability* to keep this signing key safe.

Evil maid. The evil maid is a particularly attractive vector for introducing this backdoor. Say you buy a Trezor or Ledger from Amazon. Replacing the whole hardware wallet with an evil, backdoored one is an option to deploy this backdoor. Ironically, the fact the code is open-source makes this process extremely easy. (Maybe by sending the wallet back to Amazon through the RMA process after having injected the backdoor.)

Implementing the backdoor at other abstraction levels.

In this paper, we assume we implemented the backdoor at the application firmware level. The backdoor could be implemented at other levels: at the OS level (detecting whenever the ECDSA nonce is generated), or at a hardware level (for example by tampering with the RNG peripheral on nonce generation.)

Lowering detectability. The retrieval strategy is out of scope of this document. One possibility is to plant this backdoor in many wallets, but steal funds only from a few well-funded wallets. This can be used to lower the suspicion on a systemic breach like a firmware compromise. For example, one could only siphon out the top 0.1% of the wallets.

Multi-user setting. The backdoor description in §3 assumes we want to leak a secret from a single wallet. We can extend this to the multi-user setting (multiple wallets) in several ways. We can diversify the backdoor recovery secret b on a per-wallet basis. This is a clean approach on the wallet side; the recovery effort increases linearly with the number of backdoored wallets. Alternatively, the w -th wallet can leak first a single, short subliminal message l_1^w using a global backdoor recovery secret b . This l_1^w encodes a “session” backdoor recovery secret $b_w = \text{KDF}(b, l_1^w)$. The w -th wallet uses this “session” recovery secret b_w for subsequent subliminal messages. This makes the complexity of recovery substantially smaller.

7.3 Comparison

Our construction relies on subliminal messages in ECDSA. Simmons already provided in the 1980s several constructions

for this [Sim83, Sim84, Sim85] that relied on manipulating the DSA secret nonces. Our construction adds a layer on top for message fragmentation.

The early construction of Young and Yung [YY97b] is very efficient, requiring only two signatures. However, it needs to set the same key for both signatures, and the signer must be stateful. In an actual hardware wallet, statefulness may be hard to guarantee since this requires write capabilities to non-volatile storage (which may be not even present).

The construction of Ateniese et. al [AMV15] is tangentially related to ours. However, the running time for the backdoored signer is exponential in the number of bits leaked, which makes the backdoor easy to detect by measuring execution time.

8 Mitigations

Split trust: multisignatures. Many protocols (including Bitcoin) accept multisignatures natively. This consideration can be taken at design time to generate a 2-of-2 wallet between the host and the hardware wallet. This technique can be used to avoid the consequences of a backdoored signature, but comes at the price of longer transaction (hence more expensive) and more complexity.

As with any technique that relies on splitting trust, diversity and heterogeneity are critical to actually gain security. In this case, if both the software running in the host *and* the firmware running in the hardware wallet are developed by the same teams, the cost of mounting an attack against both is not much higher.

Split trust: firewalled signatures. As noted by Dauterman et al [DCM⁺19], this problem can be solved with cryptographic reverse firewalls [MS15]. This is a general technique that protects against cryptographic subversion and assumes some system parts (the reverse firewall) are trusted and behave correctly. Cryptographic reverse firewalls can be built from zero-knowledge techniques, but the performance is typically much worse than custom designs such as [DCM⁺19].

Split trust: multi-party computation. Firewalled signatures can be implemented also with multi-party computation. There are a myriad of threshold ECDSA designs that could be used [Lin17, GG18, CCL⁺19, MPS19, CGG⁺20].

On the more practical side, Dauterman et al [DCM⁺19] design a lightweight 2-party protocol for firewalled ECDSA signatures between a signer and a firewall. By construction, the signer cannot exfiltrate any message via bits of the signature. At first sight, it is easy to fall into this circular reasoning: what does this buy us if we anyways have to trust the firewall? This construction is appealing since the firewall itself does not require to store any secret key material (thus making it

easy and cheap to manufacture with commercial, off-the-shelf parts, and hence trust).

We see an opportunity in standardizing the protocol the signer and the firewall (potentially, multiple firewalls) so that interoperability between different manufacturers for the firewall and signer is possible. Diversity here is beneficial for security.

At run time: attestation. One way to mitigate evil-maid style attacks is by using attestation: the hardware wallet could prove its authenticity to the host before the host trusts the hardware wallet.¹⁰ This requires setting some kind of PKI between the hardware wallet manufacturer and host software; and heavy modifications to the hardware wallet (quote generation functionality and provisioning secrets or certificates in the hardware wallet.)

General supply-chain mitigations. A backdoored wallet is a hardware and software supply chain problem, hence, generic mitigations against supply chain threats apply. These are not specific to the problem of a backdoored wallet, but apply to every security-critical hardware or software product. Without being exhaustive, generic mitigations like code audit, code signing, build system hardening, artifact store hardening, reproducible builds, artifact signing and secure boot will help.

Strengthening firmware signing. A take away is that we need to make firmware signing more robust. Firmware signing keys protect too much value, so it is convenient to diffuse this pressure. One way for example is by using multiple firmware signing keys, each owned by a different party that performs independent authorization of the signing action. This is essentially a straight multi-signature scheme. One can use more complex multi-signature schemes like MuSig2 [NRS21]. Potentially some of the firmware signing keys could be stored offline with a tight, verifiable log of key utilization.

At validation time: known-answer tests. A mitigation strategy could be cross-validation with a known-good implementation. As noted in §7.2, the backdoor could recognize the test vector inputs and react accordingly to hide the backdoor. This can be mitigated by using random, unpredictable inputs. In addition, the backdoor could sense the environment that is currently running on, and only get activated in a production environment, while the debug/development builds hide the backdoor behavior.

Applicability to other wallets. We focus here on hardware wallets, but the same principles could be applied to hot wallets or purpose-specific cold storage appliances.

¹⁰We need some mental gymnastics when laying out the threat model here since the *raison d'être* of hardware wallets is the host is untrusted.

9 Lessons learned

We collect here lessons we learned that could be useful in the threat modeling process. The lessons here can assist the security architect in gauging the risk level when developing Bitcoin wallets.

Firmware signing key: lifetime responsibility. The technique presented in this paper can be used to turn an existing, uncompromised wallet into a backdoored one. This means the wallet designer has a lifetime responsibility of safeguarding the firmware signing key (provided the wallet has some kind of firmware update mechanism).

Firmware signing key: value. The monetary value of the firmware signing key can be roughly estimated as the sum of the wallet balances where the signed firmware runs. This is typically much larger than the firmware signing key of consumer electronics. Thus, significant more resources should go to protect this key.

Build system: value. The previous observation carries also to the build system: the cost of sneaking a backdoor anywhere in the supply chain should be commensurate to the reward of backdoor wallets. Otherwise, there's an opportunity for an attacker to make a profit.

The previous two points make clear the need for making firmware signing robust. We described mitigations in §8.

Eliminating exfiltration channels is not enough. A common good practice is to reduce or eliminate the “free slots” to exfiltrate data from the signer. This could take the shape of exfiltrating data through unused data fields (in case the API allows that), or by using representations that are not deterministic (i.e. admit several different representations for the same input). Whilst this is generally a good idea, it is not enough, since just the signature is enough to exfiltrate seeds in a backdoored wallet.

Physical exfiltration ranks lower in priority. For the security architect designing a Bitcoin wallet, physical exfiltration concerns rank lower than supply-chain security, since attacks using physical exfiltration are strictly harder to mount than supply-chain ones.

Compromise of the signer is enough. A relevant metric in assessing the attack difficulty or chances to get detected is how many different components need to be compromised. In this case, we do not need to compromise any other system upstream of the signer. (The attack works if the host computer that the hardware wallet is connected to remains uncompromised.)

Air-gap may give false sense of security. The backdoored wallet described here works also in “air-gapped” systems. By definition, in an air-gapped wallet the signature will always need to cross the air gap, and hence air-gapping is not enough to prevent this attack (even if it can help to reduce the attack surface.)

RMA process can be a can of worms. The reverse order fulfilling system is also a good opportunity to inject backdoors: a malicious user may return a wallet claiming it does not work properly, in the hope that the wallet gets later sold to another customer as a refurbished device. The wallet manufacturer is thus forced to inspect the wallets for backdoors if they want to sell later refurbished devices. This is a very hard problem.

10 Conclusion

In this paper, we engineered a bitcoin backdoored wallet. The backdoor is very efficient and can leak a full seed in about a dozen signatures. We demonstrated the feasibility of our approach by implementing an end-to-end demo. We hope some of our observations help the development of hardware wallets.

Future work. This backdoor is efficient, but could scale better in the number of injected backdoors. The effort required to recover exfiltrated seeds is linear in the number of injected backdoors and this could be improved.

Ethical considerations. The observations in this paper could be used to cause harm. We have performed our experiments in a non-production network using mock values. We believe the recommendations in this paper for mitigating backdoored wallets outweigh the potential harm.

A ECDSA definition

We adopt the same notation and exposition as [FKP16]. We have a prime order group \mathbf{G} of order q generated by $g \in \mathbf{G}$, $H : \{0, 1\}^* \rightarrow \mathbf{Z}_q$ a hash function, and f a “conversion function” $f : \mathbf{G} \rightarrow \mathbf{Z}_q$ (in ECDSA f typically returns the x -coordinate of the input curve point). We define ECDSA as the following three algorithms:

- ECDSA.KeyGen() \rightarrow (pk, sk). Sample $x \in_R \mathbf{Z}_q$ at uniform random. Set x as the private key and $X = g^x \in \mathbf{G}$ as the public key.
- ECDSA.Sign(sk, m) \rightarrow σ . Pick a random nonce $r \in_R \mathbf{Z}_q^*$. Compute $c \leftarrow f(g^r)$ and $s \leftarrow r^{-1} \cdot (H(m) + c \cdot x)$. Output $\sigma \leftarrow (c, s) \in \mathbf{Z}_q^2$. When the caller picks the random nonce r , we write ECDSA.Sign(sk, m ; r).

```
f019bd3461309ae48c48a9cee5edaefb8ff4ef4c921fbd9d43377ff64162d77b
32d31db85be4f21c0d1761f4871ec1dde6cd9b2bede0b83aa850262fb401ba32
a2a40cbf8e85cf315ef9e83efc148b67103e92feb6d52403bee62e0cdb2aede8
8d42358580badccd94407d279225829ffaf2215dcc508877bc12e3ae36967673
1db369cc29eee5330265563460ba939c19c0bc61ae7dd155b98329b482fc382b
5a63c79eca5de1fd901be18f98d41ce1a1e4dc5acc2fa0bb5a440b6c0ee4ca4a
71cf01fdfe9f3566a9e07b9535fff9d173f51a7e7f68dff253e9fafba1930a58
83b0d44575af054e035c42bb4cef8d1e23440c490d383fa6c0e4bc32b14888cb
303a9ab5e5763e66b2cee395de05db15648888b48ac7f19dfc5b85496804a50
55929d6b5f7c1973f1f44dce5bc2b687c6e667cd26edf56a968b03eac0abf9d7
a8563f37969ac19ce41137bac194f69169ba82670c96cdfad25405f720ddf100
350b8ce7186015283cd09fbc2cbbd546bc824c7d7b8518fd95e5e7af9c4be13
a0eb336a63e3eb246dfd89f9eb7d128f54bc980fa5443b2fbd3c828e5760f9db
```

```
>> recovering 19 bits per sig
recovered 0x050942 from sig 0
extracted 0x02af5b from sig 1
extracted 0x05d30b from sig 2
extracted 0x013ab6 from sig 3
extracted 0x03d2d2 from sig 4
extracted 0x043926 from sig 5
extracted 0x05e140 from sig 6
extracted 0x07e246 from sig 7
extracted 0x0742ac from sig 8
extracted 0x03ea4e from sig 9
extracted 0x0185a0 from sig 10
extracted 0x05915a from sig 11
extracted 0x01af06 from sig 12
>> exhaustive search
bits to guess: 9
brute force successful!
number_signatures=13
leaked_bits_per_sig=19
dim(ker(A))=9
recovered=12345678901234567890123456789012345678901234567890
123456789012345678901234
```

Figure 3: Transaction hashes for 13 transactions that leak a backdoor

- $\text{ECDSA.Verify}(pk, m, \sigma) \rightarrow \{0, 1\}$. Write X for the public verification key. The purported signature $\sigma = (c, s)$ is valid if and only if $f(R')$ is equal to c , where $R' = (g^{H(m)}X^c)^{1/s} \in \mathbf{G}$.

B Definition of M

The matrix \mathbf{M} is a $256 \times L$ random-looking matrix spanned by the public message m . We compute L different 256-bit hashes $h_1(m), \dots, h_L(m)$ of m and stack them to construct \mathbf{M} . (For example, $h_i(m) := h(i || \mathbf{M}\text{-matrix} || m)$) This ensures the matrix has balanced statistics and we can model it as uniformly random bits for the purposes of Section 4.1.

C Transaction hashes

In Figure 3 we write hashes for 13 transactions. We leak 19 bits per transaction.

D Recovery output

In Figure 4 we put an exemplary output of the recovery process. The tool recovers the leaked seed from 13 signatures after trying 2^9 candidates.

References

[AMV15] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signature schemes. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 364–375. ACM, 2015.

Figure 4: Recovery tool output

[AMV20] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signatures: Definitions, constructions and applications. *Theor. Comput. Sci.*, 820:91–122, 2020.

[BBO07] Mihir Bellare, Alexandra Boldyreva, and Adam O’Neill. Deterministic and efficiently searchable encryption. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 535–552. Springer, 2007.

[BLN16] Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: A standardized back door. In Peter Y. A. Ryan, David Naccache, and Jean-Jacques Quisquater, editors, *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, volume 9100 of *Lecture Notes in Computer Science*, pages 256–281. Springer, 2016.

[CCL⁺19] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Two-party ECDSA from hash proof systems and efficient instantiations. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 191–221. Springer, 2019.

[CGG⁺20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti,

- Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1769–1787. ACM, 2020.
- [DCM⁺19] Emma Dauterman, Henry Corrigan-Gibbs, David Mazières, Dan Boneh, and Dominic Rizzo. True2f: Backdoor-resistant authentication tokens. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 398–416. IEEE, 2019.
- [DGH⁺19] Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, Kevin Liu, and Giulio Malavolta. Rate-1 trapdoor functions from the diffie-hellman problem. In Steven D. Galbraith and Shihō Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 585–606. Springer, 2019.
- [FKP16] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (EC)DSA signatures. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1651–1662. ACM, 2016.
- [GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1179–1194. ACM, 2018.
- [GMME15] Mordechai Guri, Matan Monitz, Yisroel Mirski, and Yuval Elovici. Bitwhisper: Covert signaling channel between air-gapped computers using thermal manipulations. In Cédric Fournet, Michael W. Hicks, and Luca Viganò, editors, *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 276–289. IEEE Computer Society, 2015.
- [GSE20] Mordechai Guri, Yosef A. Solewicz, and Yuval Elovici. Fansmitter: Acoustic data exfiltration from air-gapped computers via fans noise. *Comput. Secur.*, 91:101721, 2020.
- [KA98] Markus G. Kuhn and Ross J. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In David Aucsmith, editor, *Information Hiding, Second International Workshop, Portland, Oregon, USA, April 14-17, 1998, Proceedings*, volume 1525 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 1998.
- [Kol99] Valentin F. Kolchin. *Random graphs*, volume 53 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 1999.
- [Lin17] Yehuda Lindell. Fast secure two-party ECDSA signing. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 613–644. Springer, 2017.
- [LU02] Joe Loughry and David A. Umphress. Information leakage from optical emanations. *ACM Trans. Inf. Syst. Secur.*, 5(3):262–289, 2002.
- [MNPV98] David M’Raïhi, David Naccache, David Pointcheval, and Serge Vaudenay. Computational alternatives to random number generators. In Stafford E. Tavares and Henk Meijer, editors, *Selected Areas in Cryptography '98, SAC'98, Kingston, Ontario, Canada, August 17-18, 1998, Proceedings*, volume 1556 of *Lecture Notes in Computer Science*, pages 72–80. Springer, 1998.
- [MPS19] Antonio Marcedone, Rafael Pass, and Abhi Shelat. Minimizing trust in hardware wallets with two factor signatures. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11598 of *Lecture Notes in Computer Science*, pages 407–425. Springer, 2019.
- [MS15] Ilya Mironov and Noah Stephens-Davidowitz. Cryptographic reverse firewalls. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part*

- II*, volume 9057 of *Lecture Notes in Computer Science*, pages 657–686. Springer, 2015.
- [MSST05] Anil Madhavapeddy, Richard Sharp, David J. Scott, and Alastair Tse. Audio networking: the forgotten wireless technology. *IEEE Pervasive Comput.*, 4(3):55–60, 2005.
- [Nak08] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. <https://bitcoin.org/bitcoin.pdf>, 2008.
- [NRS21] Jonas Nick, Tim Ruffing, and Yannick Seurin. Musig2: Simple two-round schnorr multi-signatures. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I*, volume 12825 of *Lecture Notes in Computer Science*, pages 189–221. Springer, 2021.
- [Por13] Thomas Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). *RFC*, 6979:1–79, 2013.
- [SAS22] Ledger SAS. . <http://www.ledger.com/>, 2022.
- [Sat22] SatoshiLabs. Trezor. <http://trezor.io/>, 2022.
- [SF07] D. Shumow and N. Ferguson. On the possibility of a back door in the NIST SP800-90 dual EC PRNG. rump2007.cr.yt.to/15-shumow.pdf, 2007.
- [Sho01] Victor Shoup. A proposal for an ISO standard for public key encryption. *IACR Cryptol. ePrint Arch.*, page 112, 2001.
- [Sim83] Gustavus J. Simmons. The prisoners’ problem and the subliminal channel. In David Chaum, editor, *Advances in Cryptology, Proceedings of CRYPTO ’83, Santa Barbara, California, USA, August 21-24, 1983*, pages 51–67. Plenum Press, New York, 1983.
- [Sim84] Gustavus J. Simmons. The subliminal channel and digital signature. In Thomas Beth, Norbert Cot, and Ingemar Ingemarsson, editors, *Advances in Cryptology: Proceedings of EUROCRYPT 84, A Workshop on the Theory and Application of Cryptographic Techniques, Paris, France, April 9-11, 1984, Proceedings*, volume 209 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 1984.
- [Sim85] Gustavus J. Simmons. A secure subliminal channel (?). In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO ’85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 33–41. Springer, 1985.
- [VP09] Martin Vuagnoux and Sylvain Pasini. Compromising electromagnetic emanations of wired and wireless keyboards. In Fabian Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 1–16. USENIX Association, 2009.
- [Wui12] Pieter Wuille. Hierarchical deterministic wallets. *Bitcoin Improvement Proposal (BIP)*, 32, 2012.
- [YY96] Adam L. Young and Moti Yung. The dark side of "black-box" cryptography, or: Should we trust capstone? In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO ’96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 1996.
- [YY97a] Adam L. Young and Moti Yung. Kleptography: Using cryptography against cryptography. In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT ’97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1997.
- [YY97b] Adam L. Young and Moti Yung. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO ’97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 264–276. Springer, 1997.
- [YY04] Adam L. Young and Moti Yung. *Malicious cryptography - exposing cryptovirology*. Wiley, 2004.

Oh No, My RAN! Breaking Into an O-RAN 5G Indoor Base Station

Leon Janzen^{id}, Lucas Becker^{id}, Colin Wiesenäcker, Matthias Hollick^{id}
Technical University of Darmstadt (TUDa)
{ljanzen, lbecker, cwiesenaecker, mhollick}@seemoo.de

Abstract

Indoor base stations are expected to play a crucial role in 5G and beyond, as they are required to provide millimeter wave connectivity in buildings. However, they are a prime target for attacks, as they are difficult to secure against physical access attacks and highly connected within the RAN, especially for Open Radio Access Network (O-RAN) indoor base stations. In this work, we develop and introduce a threat model for indoor base stations. We conduct a security analysis of a proprietary O-RAN Radio Unit and present four novel vulnerabilities. Further, we analyze the Radio Unit regarding its hardware, software, and services, highlighting deviations from the O-RAN standards. The vulnerabilities we discover lead to remote code execution on the Radio Unit, highlighting security issues arising from the novel attack surface introduced by indoor base stations.

1 Introduction

Two trends in the fifth-generation technology standard for cellular networks (5G) make indoor base stations (BSs) a prime target for attacks, especially in the Open Radio Access Network (O-RAN): (1) Achieving physical access control for indoor BSs is hard, if not infeasible, and (2) indoor BSs are highly connected within the O-RAN.

While mobile network operators (MNOs) have thoroughly designed policies regulating the security of outdoor BSs, *physical access control is impractical* for indoor BSs. Unlike outdoor BSs, typically secured with fences, security cameras, and stringent physical access control measures [14], indoor BSs are often deployed on walls or ceilings, similar to enterprise Wi-Fi routers [40]. As a result, only some of the outdoor BS policies apply to indoor BSs. This lack of access control exposes indoor BSs to potential physical port access by attackers, significantly expanding the attack surface of the Radio Access Network (RAN) and the cellular network. The aspect of cellular network security has received limited attention in the research community so far, which motivates us to **introduce a threat model for indoor BSs**.

O-RAN BSs are highly connected using various interfaces to communicate with other cellular network components. In O-RAN, the BS is disaggregated into several components [52], leaving only the Radio Unit (RU) deployed at the cell site (Figure 1). Within the O-RAN ecosystem, the RU directly interfaces a Distributed Unit (DU) and the Service Management Orchestration Framework (SMO) featuring one of the RAN Intelligent Controllers (RICs) [52]. The O-RAN Alliance has released specifications for the Open Fronthaul Interface surrounding the RU [46, 47]. In this work, **we conduct a security analysis of a proprietary O-RAN RU** to evaluate how vendors implement the specifications in the real world.

Physical access to indoor BS makes adjacent attacks on the RAN feasible, drawing attention to the security of RAN hardware. We deem this novel attack surface one of the major security challenges for RAN vendors and MNOs. To highlight this issue, **we present four vulnerabilities we discovered on a proprietary O-RAN RU**, two exploitable to achieve Remote Code Execution (RCE). In summary, our key contributions are as follows:

- We develop a threat model for indoor BSs (Section 3).
- We conduct a security analysis of a real-world O-RAN RU, highlighting deviations from the Open Fronthaul standards (Section 4).
- We present four vulnerabilities we discovered on a proprietary O-RAN RU (Section 5), which we classify as high or critical.
- We discuss our findings in the context of research trends for future cellular networks, including mitigation of the found exploits (Section 6).

We responsibly disclosed all identified issues to *Airspan Networks Inc.* and are in the process of requesting Common Vulnerabilities and Exposures (CVE) entries for our findings.

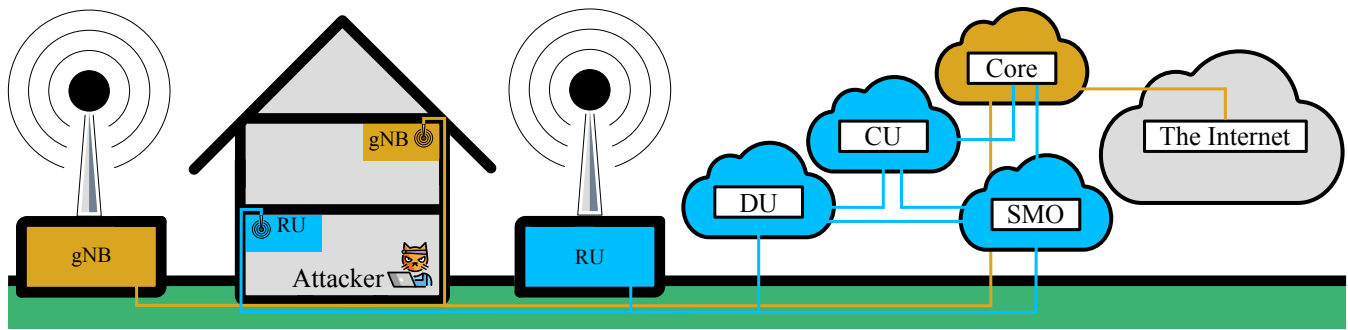


Figure 1: In **conventional cellular networks**, users connect to indoor or outdoor Next Generation NodeBs (gNBs) that directly connect to the core network (CN), from where traffic is forwarded to the Internet. The **Open Radio Access Network (O-RAN)** disaggregates the gNB into Radio Unit (RU), Distributed Unit (DU), and Central Unit (CU) with an additional Service Management Orchestration Framework (SMO). DU, CU, and SMO are typically virtualized and deployed remotely.

2 Background and Related Work

This section introduces relevant concepts and terminology of 5G networks (Section 2.1), the Open Radio Access Network (O-RAN) (Section 2.2), and indoor base stations (BSs) (Section 2.3) before summarizing related work (Section 2.4).

2.1 5G Cellular Networks

As depicted in Figure 1, in conventional Radio Access Networks (RANs), the user equipment (UE) connects to a Next Generation NodeB (gNB) that handles all layers of the 3rd Generation Partnership Project (3GPP) stack [2] from the physical layer (PHY) to the Radio Resource Control (RRC) and Service Data Adaptation Protocol (SDAP) and sends user traffic to the core network (CN) [3]. The CN is the central point of the cellular network, providing numerous core network functions (NFs), e.g., user authentication, session management, access- and mobility management, and policy control [5]. When users access the Internet via 5G, the CN converts user plane (U-Plane) traffic from the 3GPP stack to the Internet Protocol (IP) stack and forwards it to the Internet.

2.2 O-RAN and Open Fronthaul

The O-RAN-specific parts of the cellular network are highlighted in blue in Figure 1. One of the innovative ideas of the O-RAN is that the gNB is disaggregated into three components [41], as depicted in Figure 2: The Radio Unit (RU) handles the radio frequency (RF) connectivity and lower PHY [4] before sending user traffic via the Open Fronthaul interface [46, 47] to the Distributed Unit (DU) [41]. The DU handles the remaining upper PHY, the medium access control (MAC) layer, and the Radio Link Control (RLC) layer. Finally, the Central Unit (CU) handles the Packet Data Convergence Protocol (PDCP) and RRC layers before forwarding the traffic to the CN [41, 52]. In contrast to the RU, which is deployed physically at the cell site, DU and CU are typically

virtualized [10]. The CN, CUs, DUs, and RUs often build a tree topology where multiple RUs connect to one DU, multiple DUs connect to one CU, and multiple CUs connect to the CN [41, 52]. The O-RAN Alliance uses the data modeling language Yet Another Next Generation (YANG) to model Network Configuration Protocol (NETCONF) configuration and state data of O-RAN components and interfaces. Thus, NETCONF and YANG models facilitate standardization and interoperability between O-RAN vendors.

The Open Fronthaul interface is one of the numerous interfaces in O-RAN, connecting RU and DU. While control and user plane (CU-Plane) traffic is sent via enhanced Common Public Radio Interface (eCPRI), synchronization plane (S-Plane) traffic is sent via Precision Time Protocol (PTP) [46]. Management plane (M-Plane) traffic is sent via NETCONF [21, 47].

The above description suits the O-RAN Split 7.2x [52], where the RU/DU split is within the PHY. Other popular O-RAN splits are Split 6 below the MAC layer [1], also referred to as network functional application platform interface (nFAPI) [56] and preferred by the Small Cell Forum (SCF) [57] or Split 8 above the analog-to-digital and digital-to-analog converter (ADC/DAC) [2].

2.3 Indoor Base Stations

Indoor BSs are expected to play a crucial role in 5G and beyond to utilize the extremely high frequency (EHF) band for millimeter waves (mmWave) communications [2, 62]. Vendors of indoor BSs include *Airspan Networks Inc.* (Airspan), *Nokia Corporation* (Nokia), *Telefonaktiebolaget LM Ericsson* (Ericsson), and *Hon Hai Precision Industry Co., Ltd.* (Foxconn). **This paper focuses on the Airspan AirVelocity 2700 (AV2700)** because it is intended for indoor deployments, supports mmWave communications, and is compatible with O-RAN Split 7.2x [7].

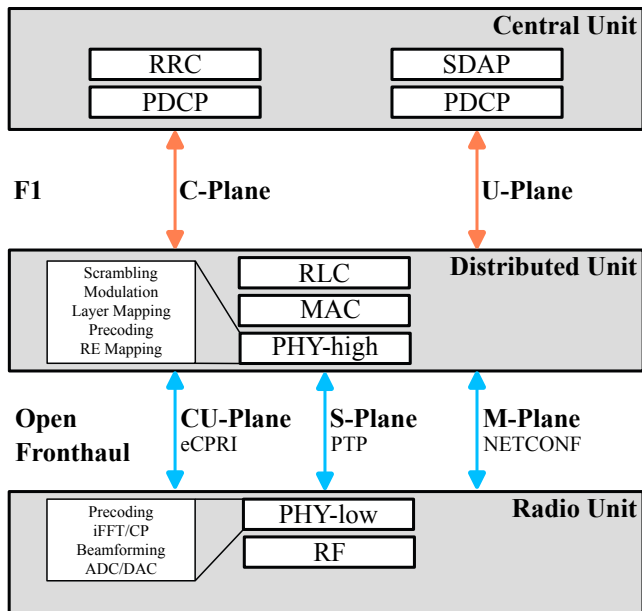


Figure 2: Open Radio Access Network (O-RAN) disaggregation of a Next Generation NodeB (gNB) into a Central Unit (CU), Distributed Unit (DU), and Radio Unit (RU). This figure depicts O-RAN Split 7.2x, where the control and user plane (CU-Plane) of the Open Fronthaul is split within the physical layer (PHY). The F1 interface connects DU and CU. This figure is adapted in parts from [52].

2.4 Related Work

We summarize general O-RAN-security-related publications (Section 2.4.1), address existing work related to the Open Fronthaul and RU security (Section 2.4.2), and distinguish our work from the aforementioned publications (Section 2.4.3).

2.4.1 O-RAN Security

Liyanage et al. [36] analyze security risks and challenges within the O-RAN ecosystem by classifying security-related risks. They offer a detailed overview of various threat categories, including descriptions and evaluations of their applicability to the O-RAN ecosystem. They discuss potential security solutions derived from Cloud Radio Access Network (C-RAN) and delve into design errors while exploring their consequences and available mitigation options for O-RAN. Klement et al. [33] investigate the O-RAN environment, evaluating the security status of its deployed components and proposing measures to ensure their secure operation. They identify critical stakeholders in the O-RAN context and list best practices to enhance O-RAN security. Groen et al. [30] investigate the security aspects of O-RAN systems, adopting a holistic approach, including the O-RAN interfaces and the overall platform. They identify potential threats and offer solutions to address security issues in these areas.

Without a specific focus on the O-RAN architecture, Farooqui et al. [23] present a threat model for 5G-based systems, defining a layered architecture and mapping threats to the respective applicable layers. Sattar et al. [54] model the threats arising from small cells in Long-Term Evolution (LTE) networks. They define trust boundaries including physical security as one aspect.

2.4.2 Open Fronthaul and Radio Unit Security

Abdalla et al. [6] delve into the standardization efforts of the O-RAN Alliance, focusing on network threats with a specific emphasis on the Open Fronthaul. They identify end-to-end security threats affecting the interface and recommend countermeasures and best practices against the identified threats. They detail an attack scenario involving unauthorized access to the physical layer of the Open Fronthaul by compromising the physical connection between the DU and the RU. Liao et al. [35] developed a Denial-of-Service (DoS) attack tool for the Open Fronthaul control plane (C-Plane) by generating C-Plane packets that initiate DoS attacks. Dik et al. [16, 17] contribute two consecutive works on the security of the Open Fronthaul. In their first work [16], the researchers examine the transport security of the Open Fronthaul by investigating threats that can impact the interface. They survey the data types transported over the different data planes and derive necessary security measures. In their second work [17], Dik et al. conduct a more in-depth analysis of the transport network security in the Open Fronthaul. They discuss threats and vulnerabilities of the interface and their network impact. They provide a threat protection solution in MACsec as a layer two security mechanism implemented on field-programmable gate arrays (FPGAs) to secure the Open Fronthaul.

2.4.3 Distinction from Related Work

The publications presented in this section are relevant to our work as they introduce overarching challenges, threats, and vulnerabilities associated with O-RAN, showing the larger attack surface of the ecosystem and shedding light on various approaches attackers can take when attacking the O-RAN components and interfaces. The presented papers all take a theoretical approach to analyzing O-RAN security. In contrast, our work focuses on the security of a single O-RAN component, i.e., the RU. **We investigate the AV2700 as an example of a proprietary RU and present real-world vulnerabilities and security issues of the AV2700.**

3 Threat Model

In contrast to wireless access points [37, 59, 65], RUs pose an especially interesting attack surface with their multiple interfaces to other RAN components. Our threat model is consistent with existing publications [6, 23, 31, 36, 42, 45, 54, 55]

and applicable standards [22, 43] tailored towards indoor BSs. It aligns with existing threat models of indoor BSs in conventional RANs, including femtocells [28], for all non-O-RAN aspects. This section defines our system model (Section 3.1) and discusses an adversary’s motivation (Section 3.2) and their assumed capabilities (Section 3.3).

3.1 System Model

Figure 1 depicts our system model. We assume an O-RAN infrastructure with one or more RUs deployed indoors. The RUs connect to a corresponding DU via Ethernet to handle control, user, synchronization, and management plane (CUSM-Plane) communication. The RU also connects to the Service Management Orchestration Framework (SMO), where one of the RAN Intelligent Controllers (RICs) is deployed [42, 43].

In contrast to physically protected outdoor cell towers [14], RUs are installed akin to prevalent enterprise Wi-Fi routers, implying that they are accessible from within the building [15]. We consider an RU affixed to a wall. The RU might be located within or without the reach of an adversary. The RU might be secured with anti-theft protection means, e.g., a Kensington lock. Surveillance measures might be implemented to mitigate undiscovered interactions with the RU. The network infrastructure might be configured so an adversary can achieve Ethernet access from an adjacent Ethernet port connected to the RU. Depending on these deployment options, the adversary can gain different capabilities (Section 3.3). Possible scenarios enabling such access include installations in shared or multi-tenant buildings, e.g., office complexes, shopping centers, or universities.

3.2 Adversary Motivation

The adversary we consider aims to attack the 5G network, using the O-RAN RU for their initial foothold. Note that the cellular network is classified as a critical infrastructure [24] and, hence, is particularly interesting to adversaries. In the context of this work, **the adversary aims to gain complete control of an RU to facilitate further attacks.**

While any attacks beyond controlling the RU are outside this work’s scope, the adversary’s next steps might include local operation or lateral movement: (1) On the RU, the adversary might manipulate in-transit traffic by recording, manipulating, or redirecting, potentially targeting UEs [12, 34, 51]. Additionally, the adversary might extract sensitive configuration data. (2) The adversary might prepare attacks for lateral movement in the O-RAN by escalating attacks from the controlled RU to the DU [6] or SMO [58, 60].

3.3 Adversary Capabilities

We consider an adversary targeting the RAN by abusing physical access to an indoor RU, directly achieving physical access,

Table 1: Summary of the adversary’s potential actions achievable with capabilities $C_1 - C_4$. Checkmarks imply that the adversary is capable of the potential action and crosses imply that the adversary is not.

Potential Action	C_1	C_2	C_3	C_4
Access to the RU’s Ethernet ports	✓	✓	✓	✓
Access to the RU’s power socket	✗	✓	✓	✓
Access to the RU’s debug ports	✗	✓	✓	✓
Evaluation in own environment	✗	✗	✓	✓
Redeployment of a modified RU	✗	✗	✗	✓

or connecting to an adjacent Ethernet port connected to the RU’s Open Fronthaul interfaces. In doing so, our assumed adversary achieves a subset of the following four capabilities summarized in Table 1:

Ethernet Access With access to the RU via Ethernet (C_1), the adversary can communicate with the RU’s Open Fronthaul interface (Figure 3). This access enables the adversary to take the logical position of another RAN component, e.g., the DU or SMO, to target exposed services on the RU and any attack surface provided by the Open Fronthaul interface. By exploiting vulnerabilities in this attack surface, they attempt to obtain control over the RU. The adversary can achieve C_1 with access to an adjacent Ethernet port connected to the RU, regardless of surveillance and anti-theft protection means in place.

Full Interface Access Access to all of the RU’s interfaces (C_2) grants the adversary all of C_1 and access to the RU’s power socket and, potentially, to debug ports. With the power socket, the adversary can shut down and restart the RU, e.g., for trivial DoS attacks and to activate the RU’s start-up procedure. Additionally, the adversary can inspect the RU’s High-Definition Multimedia Interface (HDMI) debug port, potentially facilitating gaining control of the RU. Capability C_2 requires physical access to the RU, the feasibility of which depends on surveillance and access control means in place.

RU Theft If the adversary can remove the RU (C_3), they can conduct further attacks in a prepared environment. This enables the adversary, on top of C_1 and C_2 , to perform more intrusive operations that require disassembly. If no hardware security features exist, they can use this access to extract secrets from the device. In addition, they can inspect and modify the firmware running on the device. After probing the RU, e.g., to extract non-default secrets and potentially tamper with the software and hardware of the device, the adversary can use the findings to attack other RUs. Capability C_3 requires direct physical access to the RU and an unguarded deployment.

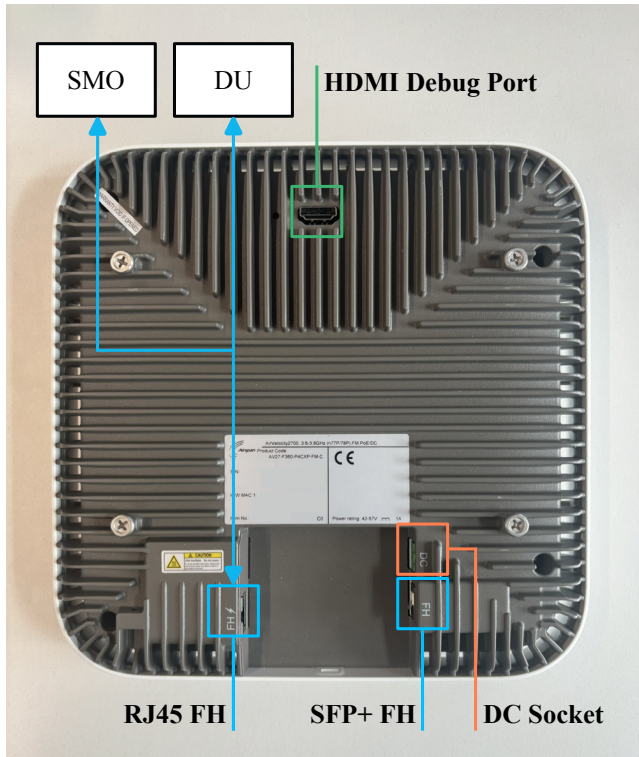


Figure 3: The AV2700’s interfaces feature a **power socket**, an **HDMI debug port**, and two **Open Fronthaul interfaces** (Small Form-factor Pluggable+ (SFP+) and RJ45; blue) that connect to the Distributed Unit (DU) and Service Management Orchestration Framework (SMO).

RU Redeployment Redeploying the RU (C_4) grants the adversary all of $C_1 - C_3$ and the option to set up a manipulated RU into the O-RAN. After probing, modifying, and possibly gaining control over the removed RU, the adversary can re-deploy the device to its designated spot. Taking the position of the RU enables interaction with the other components in the RAN and lays the basis for further attacks. Capability C_4 requires physical access to the RU and a deployment environment that is unguarded and unsecured over an extended period.

4 Analysis

The AV2700’s hardware and software structure is not publicly disclosed, so we decided to learn as much as possible about its inner workings to understand its attack surface. We connected a computer to the AV2700 (Section 4.1) to understand the network interfaces, remotely connected to the AV2700 to explore the file system, and reverse-engineered firmware parts. We report insights into the AV2700’s hardware and software structure (Section 4.2) and its exposed services (Section 4.3).

4.1 Setup

Our hardware setup comprises two components interconnected via an Ethernet cable: A commercial off-the-shelf (COTS) computer and an AV2700. We utilize the computer to communicate with the AV2700, investigate the device, and capture network traffic for analysis. The AV2700 connects to the computer via Ethernet. In this connection, we observed unencrypted traffic between the AV2700 and the PC for different reasons: (1) During start-up, the RU initiates a call-home procedure to the DU, and (2) some services running on the RU’s host system are not directly related to O-RAN. In normal operation, RU and DU communicate over an encrypted channel. Apart from the AV2700, our setup solely comprises COTS hardware, highlighting that only minimal resources are necessary to replicate our findings.

4.2 Hardware and Software Structure

The AV2700 hardware (Figure 3) is based on the *Mercury+XU8* System-on-Chip (SoC) [20] containing a *Xilinx Zynq UltraScale+*, which includes an FPGA [64], an *ARM Cortex A53* [8] and an *ARM Cortex R5F* [9].

We investigated the AV2700 with firmware 19.6.3 of System Release 1.6.37. The operating system (OS) on the AV2700 is an embedded Linux solution built and deployed using *PetaLinux 2019.1*. *PetaLinux* is an embedded software development kit (SDK) for Xilinx FPGA-based SoC designs that includes auxiliary functions for building Linux solutions for embedded systems [63]. Further, *BusyBox v.1.29.2*, a Unix software suite for embedded systems and mobile devices [11], provides Unix functionality on the AV2700.

Besides a power socket, the AV2700 has three physical interfaces (Figure 3), two connecting to other RAN components, while we assume the third to be a physical debug interface:

SFP+ port The first physical port is an SFP+ Ethernet port, providing high-speed connectivity, which is ideal for the Open Fronthaul control, user, and synchronization plane (CUS-Plane). It requires an SFP+ module and connector.

RJ45 port The second physical interface is an RJ45 Ethernet port, allowing communication to the AV2700.

Debug port The third physical interface is an HDMI port, which we suspect to be an HDMI-muxed debug port similar to [50] and compatible with HDMI-muxed debug cables [49].

4.3 Services

Figure 4 outlines the service architecture deployed on the AV2700. Seven ports are open, out of which four are unauthenticated. The most notable components are:

Table 2: Summary of our findings. The impact of $\mathbb{F}_1 - \mathbb{F}_4$ is a combination of reconfiguration (Reconf.), Denial-of-Service (DoS), and Remote Code Execution (RCE). The CVSS scores refer to the Common Vulnerability Scoring System (CVSS) version 4.

	Finding	Impact	CVSS ¹	Affected Services	Mitigation
\mathbb{F}_1	Exposed TCF Agent	RCE	9.3	tcf-agent	Remove before deployment
\mathbb{F}_2	Missing Access Control	DoS, Reconf.	8.4	clish_agentd, mosquitto	Implement authentication
\mathbb{F}_3	Memory Corruption	DoS/RCE	8.3	All management daemons	Secure coding best practices, bound checking
\mathbb{F}_4	Command Injection	RCE	9.3	itf-mgmt	Sanitize user input

¹ We self-assigned the CVSS scores to vulnerabilities $\mathbb{F}_1 - \mathbb{F}_4$ according to [25].

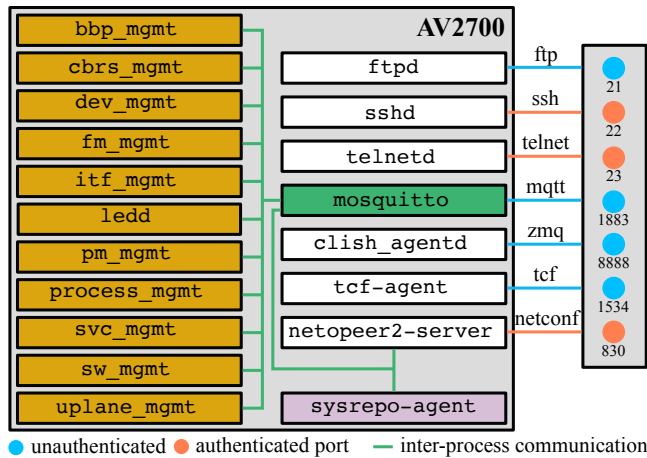


Figure 4: Service architecture of the AV2700. The central point is the mosquitto service communicating to the management daemons via inter-process communication (IPC). Three of the ports are authenticated and four unauthenticated.

FTP Server The AV2700 runs the ftpd of BusyBox started as an inetd server. The firmware misconfigures the File Transfer Protocol (FTP) server using an unsupported argument without a directory to serve the supplied files, preventing successful file transfers. Consequently, its purpose is unclear, especially when considering that the O-RAN standard mandates Secure File Transfer Protocol (SFTP) and File Transfer Protocol Explicit-mode Secure (FTPEs), which secure FTP with Secure Shell (SSH) and Transmission Control Protocol (TCP), respectively, see Section 5.1 of [47]. Furthermore, per standard, the FTP service is located on the DU, where the RU is supposed to connect for file uploads.

SSH server The sshd service is provided by the OpenSSH 7.8 SSH server. This service is required for secure M-Plane connections; see Section 5.4 of [47]. However, the SSH server with enabled shell access also allows command execution on the RU, which is not a functionality described in the standard.

Telnet The built-in telnetd of BusyBox provides Teletype Network (Telnet) remote access. It offers functionality similar to the SSH server without the confidentiality or integrity protection of the transmitted data. Identical to the SSH

server, the remote access capabilities offered by Telnet are not mandated by any standard covering the RU, nor are they part of the M-Plane. However, when considering Telnet as a vendor extension of the M-Plane specification, it violates the end-to-end security requirement stated in [47].

Mosquitto MQTT server The mosquitto Message Queuing Telemetry Transport (MQTT) server listens on all interfaces and is externally reachable. In extension, the IPC functionality to the manager daemons can also be called from outside, indirectly exposing the manager daemons. The MQTT server appears to be unrelated to any O-RAN standard. We assume it to be a leftover implementation detail of the IPC mechanism that the internal manager daemons (described below) use to communicate.

Clish-agent service The clish-agentd implements the functionality of a local oru-shell over ZeroMQ [66]. It dispatches commands sent to the shell to the corresponding internal manager daemon by publishing them on the MQTT topic. No O-RAN standard describes the oru-shell, but it is directly derived from the NETCONF YANG models. Consequently, supported options overlap with settings configured over NETCONF. The clish-agentd violates the mandatory end-to-end security of the M-Plane because it uses an unauthenticated plain-text protocol [47].

TCF debugger The tcf-agent exposes debugger functionality for the FPGA. The Target Communication Framework (TCF) is an open-source network protocol to communicate with embedded devices [18]. The tcf-agent is unrelated to any O-RAN specification and appears to be a leftover development artifact. We checked recent Board Support Packages (BSPs) for the Zynq UltraScale+ and found the tcf-agent enabled by default. Therefore, its presence might be unintentional and not the result of a deliberate decision during development.

NETCONF The netopeer2-server implements the NETCONF protocol over SSH. NETCONF over SSH is required by the M-Plane specification [47]. In addition, the specification also requires NETCONF over Transport Layer Security (TLS), which we found missing (Section 5.5.1).

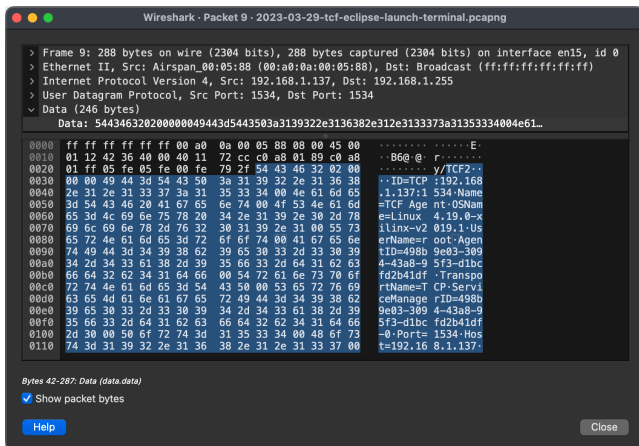


Figure 5: Target Communication Framework (TCF) packet broadcasted periodically by the TCF agent on port 1534, disclosing the TCF agent with ID, port, and version, and the operating system Linux 4.19.0-xilinx-v2019.1 with user root, leading to \mathbb{F}_1 .

Sysrepo agent The `sysrepo-agent` is another central component that implements the NETCONF protocol’s internal YANG datastore as part of the O-RAN M-Plane [47]. Furthermore, it connects to the MQTT server to trigger manager functions as a reaction to NETCONF remote procedure calls (RPCs). This component directly results from the M-Plane specification, which requires NETCONF support for device management.

Manager daemons The AV2700 software is structured into multiple manager daemons (the leftmost box in Figure 4), which are used to configure and monitor device aspects related to the O-RAN: The `sw_mgmt.d`, e.g., is used to update, install, and activate firmware archives. These managers use the publish-and-subscribe-based MQTT protocol for IPC to exchange messages encoded with JavaScript Object Notation (JSON) via the shared `mosquitto` [26] server.

5 Findings

This section presents four novel vulnerabilities we discovered on the AV2700, which are summarized in Table 2: An exposed TCF agent (Section 5.1), missing access control (Section 5.2), multiple memory corruption vulnerabilities (Section 5.3), and an OS command injection vulnerability (Section 5.4). We also discuss deviations from the O-RAN standards identified on the AV2700 (Section 5.5). **All vulnerabilities $\mathbb{F}_1 - \mathbb{F}_4$ are exploitable for adjacent adversaries with capability \mathbb{C}_1 using low-complexity attacks without user interaction, special privileges, or additional attack requirements (Figure 6).**

```

1 import zmq
2
3 context = zmq.Context()
4
5 # Socket to talk to server
6 print("Connecting to remote server...")
7 con = context.socket(zmq.DEALER)
8 con.connect("tcp://o-ran-ru-ip:8888")
9
10 con.send(b"view=system-view subview=all reboot\n")
11
12 while 1:
13     message = con.recv()
14     print(message.decode(), end="")

```

Listing 1: Python code to interact with ORU shell views. In this example, the `reboot` command in the system view is invoked to disrupt the device (\mathbb{F}_2).

5.1 Exposed TCF Agent

In the context of the AV2700, TCF enables developers to communicate with the built-in FPGAs [19], e.g., from within Eclipse with the TCF debugger add-on that opens a terminal for debugging. In the background, TCF opens a terminal on the device and runs commands as `root`.

The AV2700’s TCF agent periodically sends out User Datagram Protocol (UDP) packets on port 1534 (Figure 5) while it waits for connections. The O-RAN standards do not describe the use of TCF to communicate to the RU, so we assume it to be a leftover service that the vendor unintentionally enabled during development. As the TCF agent was not removed before deployment, adversaries can abuse its functionality with at least capability \mathbb{C}_1 , yielding \mathbb{F}_1 . The TCF packets disclose detailed information about the device, including the host and port. After recording the handshake between Eclipse and the AV2700, we reconstructed the TCF messages required to execute arbitrary shell commands on the embedded device.

Finding \mathbb{F}_1 gives an adversary full control over the RU, which we further discuss in Section 6.1. While mitigation is straightforward, i.e., removing the TCF agent before deployment, we assign \mathbb{F}_1 a *critical* CVSS score of 9.3 (Figure 6a) with a high impact regarding all security goals with low subsequent impact on confidentiality and integrity and a high subsequent impact on availability.

5.2 Missing Access Control

While the NETCONF, Telnet, and SSH interfaces require authentication, the `mosquitto` MQTT and `clish-agentd` services can be accessed unauthenticated, yielding \mathbb{F}_2 . In contrast to NETCONF, these services are not required by the applicable O-RAN standards. The `clish-agentd` is a more user-friendly way to access the NETCONF settings and can, therefore, be considered a vendor-specific O-RAN extension. The MQTT server is an exposed implementation detail with-

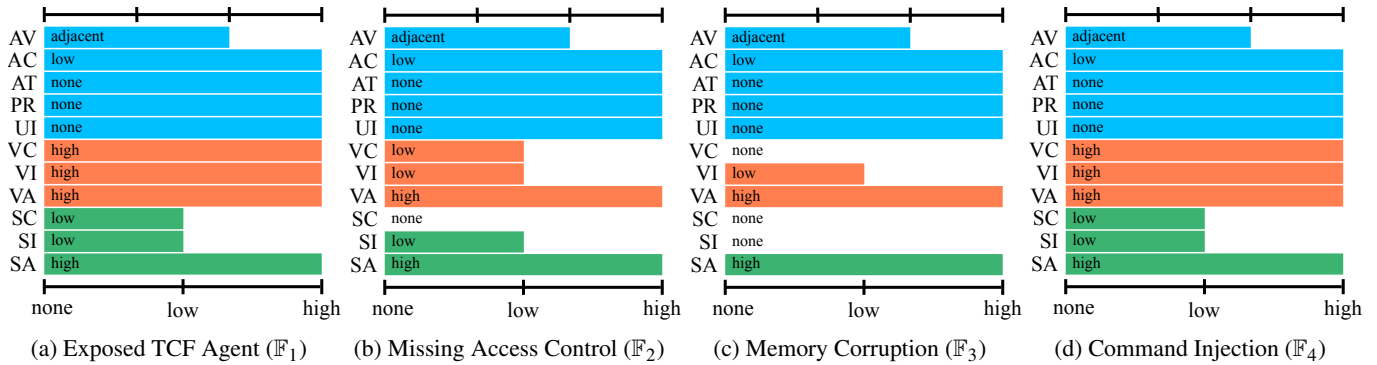


Figure 6: The Common Vulnerability Scoring System (CVSS) scores of $\mathbb{F}_1 - \mathbb{F}_4$. The colored sectors depict the three CVSS metrics: **Exploitability**, **vulnerable system impact**, and **subsequent impact metrics**. Each figure shows, starting from the top, the following items: **attack vector (AV)**, **attack complexity (AC)**, **attack requirements (AT)**, **privileges required (PR)**, **user interaction (UI)**, **confidentiality (VC)**, **integrity (VI)**, **availability (VA)**, **confidentiality (SC)**, **integrity (SI)**, **availability (SA)**. The individual scores of the exploitability metric are as follows: None, low, high for PR; none, passive, active for UI; low, high for AC; none, present for AT; and network, adjacent, local, physical for AV.

out significant user benefits. As a result of the missing authentication, the entire interface of the `clish-agentd` is available to a remote adversary with at least \mathbb{C}_1 . It can be used to set relevant configuration options of the AV2700 (Listing 1). While there is no built-in option to execute arbitrary commands, the shell can be abused to misconfigure the device. Since the available configuration options include vital system parameters such as the sending power, which affects the transmission of user data, this attack vector endangers the system’s availability. Communication with the `mosquitto` server grants comparable capabilities to accessing the `clish-agentd` since the majority of commands implemented by this agent are also dispatched via MQTT. However, it poses a higher risk because direct access to the underlying MQTT broker allows the adversary to control message contents fully. A remote adversary possessing at least \mathbb{C}_1 can exploit vulnerabilities in the supposedly internal management daemons by carefully crafting messages, which we discuss further in Sections 5.3 and 5.4. We assign \mathbb{F}_2 a *high* CVSS score of 8.4 (Figure 6b) with a high impact on availability, low impact on confidentiality and integrity, high subsequent impact on availability, and low subsequent impact on integrity.

5.3 Memory Corruption Vulnerabilities

The custom-written management daemons of the AV2700 (the leftmost box in Figure 4) are most likely written in C, judging by the libraries used, some strings referring to filenames with a `.c` extension, and the overall observable programming paradigms in place. Consequently, these components suffer from a lack of language-based memory safety, leading to \mathbb{F}_3 . Examples of such issues include multiple null pointer de-references crashing the affected components.

In the custom components, bounds checking is done im-

```

1 void* buffer = calloc(1, 0x102c);
2 void* build_id = cJSON_GetObjectItem(json_obj,
3                                     "build_id");
4 if (build_id != 0) {
5     // Fortified version of strcpy (safe)
6     __strcpy_chk(buffer, *(build_id + 32), 64);
7     [...]
8     void* buffer_ptr = buffer + 0x188;
9     void* filename_field = cJSON_GetObjectItem(
10        json_obj, "file-name");
11     if (filename != 0) {
12         // strcpy call with indirect pointer,
13         // not fortified (unsafe!)
14         strcpy(buffer_ptr - 0x84,
15              *(file_name_field + 32));
16     }

```

Listing 2: Reverse-engineered code surrounding the heap buffer overflow in Line 13 due to unfortified functions (\mathbb{F}_3).

PLICITLY using the fortified versions of security-relevant functions such as `__strcpy_chk` instead of `strcpy()` [27]. These functions perform checks to ensure sufficient buffer sizes and prevent the exploitation of buffer overflows. However, this means that all services using such functionality immediately crash when encountering an out-of-bounds error, resulting in straightforward attacks on availability. We found this a problem in almost every case where input is copied from an MQTT message containing a user-supplied JSON payload. While one can argue that in our threat model, the adversary can always attack availability by pulling the plug that supplies the unit with energy, this attack vector allows for the disruption of specific sub-services in a stealthier manner.

Fortified functions can only be used if the length of the target buffer is known during compile time. This requirement

```

1 /* Function is called with user supplied input
2    extracted from the JSON payload in an MQTT
3    message. */
4 void create_interfaces(char *inf, int vlan_id) {
5     char if_name[10];
6     char cmd_buff[100];
7
8     /* This formatting call limits the size of
9        'inf' to 7. 'if_name' is not used for the
10       system() call, but the program crashes
11       if the size is exceeded. */
12     __sprintf_chk(if_name, 1, 10,
13                  "%s.%d", inf, vlan_id);
14     if(!check_if_inf_exists(if_name)) {
15         __sprintf_chk(cmd_buff, 1, 100,
16                      "vconfig add %s %d",
17                      inf, vlan_id);
18         system(cmd_buff);
19     }
20 }

```

Listing 3: Reverse-engineered code excerpt of the operating system (OS) command injection vulnerability in Line 18 (F₄).

is violated when accessing dynamically allocated memory areas such as heap buffers with pointers, and their fortified counterparts cannot replace unsafe functions. We attribute the existence of F₃, a heap buffer overflow in the `sw_mgmt.d` daemon, to this fact (Listing 2): Surrounding code relies on `__strcpy_chk()` in combination with stack-located buffers, while the problematic code copies data from an MQTT message into a heap buffer (Line 13). Here, the unfortified `strcpy()` function is used, thus producing an overflow bug when copying from untrusted input. The impact of such issues goes beyond simple DoS attacks against services on the RU and can lead to full RCE [32] with severe consequences for the whole O-RAN (Section 6.1). At least C₁ is required to exploit this finding since the service is exposed on the Open Fronthaul interface. We assign F₃ a *high* CVSS score of 8.3 (Figure 6c) with a high impact on availability, a low impact on integrity, and a high subsequent impact on availability. We base this score on the conservative assumption that exploitation for full RCE might be infeasible due to insufficient primitives.

5.4 Command Injection Vulnerabilities

Memory-related issues are not the only area where user input sanitization is lacking. Generally, we noticed that commands executed by the `system` function were built using string formatting techniques. A review of associated input parameters uncovered an exploitable command injection vulnerability in one of the management daemons (Listing 3), resulting from the passing of untrusted user input to `system` (F₄). Since the management daemons run as `root`, this enables the execution of arbitrary commands in the context of the super-user. Similar to memory corruption issues, this

vulnerability is also externally exploitable with at least C₁ and no authentication due to the missing access control on the MQTT server. As described in Section 4.3, no O-RAN standard mandates the MQTT server. Instead, it is an implementation detail of the RU vendor. The exposure of internal services that implement O-RAN-specific functionality leads to additional attack surfaces that could have been avoided. The specific command injection vulnerability we found gives an adversary-controlled buffer of seven bytes (Line 12). Only five usable bytes remain after accounting for two bytes to terminate the previous command and cut off trailing characters. Although this length restriction prevents straightforward execution of arbitrary code, known techniques exist to exploit exactly such scenarios to gain full RCE [61], with effects on the whole O-RAN, which we describe in Section 6.1.

We adapt this idea to create empty files with controlled filenames by using the shell's output redirection operator (`>`). After creating the necessary files, we use `ls *> 0` to create a file containing the chosen payload. Note that we get the trailing zero for free due to the virtual local area network (VLAN) ID appended to the injectable interface name (Line 15), allowing us to stay within the payload length constraints. We force `ls` to list the files in the order of most recent creation by creating a file called `-tx` beforehand, which is parsed as an argument to `ls`, controlling the sorting precedence of the output. This order allows our files to appear first in the directory listing, enabling us to ignore trailing characters. As `ls` adds whitespace between filenames, we facilitate a combination of `tr` and `sed` invocations to remove whitespaces and construct arbitrary payloads, inserting `${IFS}` whenever we require spaces in our payload. We assign F₄ a *critical* CVSS score of 9.3 (Figure 6d) with a high impact on confidentiality, integrity, and availability, a low subsequent impact on confidentiality and integrity, and a high subsequent impact on availability.

5.5 Open Fronthaul Standard Deviations

The investigation of the AV2700 RU revealed several deviations from the concepts and functionalities introduced in the O-RAN M-Plane specification for the Open Fronthaul. Specifically, various features specified in the standard for the RU startup procedure were absent in the AV2700. This section addresses the missing TLS option (Section 5.5.1) and the persistent creation of users (Section 5.5.2) before discussing the use of default credentials (Section 5.5.3). **While none of these deviations are exploitable, they can facilitate follow-up attacks.**

5.5.1 Missing NETCONF via TLS Option

The RU performs a call-home procedure during start-up, which leads to the DU establishing a NETCONF connection to the RU. The M-Plane specification mandates TLS encryption as an alternative to SSH for establishing the NETCONF

connection [47]. However, we discovered that the NETCONF via TLS option is missing in the AV2700. As a result, only NETCONF via SSH is available during the initiation of the call-home procedure. To use this deviation in combination with a flaw affecting the SSH implementation, an adversary will need C_2 to restart the RU and trigger the start-up procedure. $F_1 - F_4$ would still be exploitable when using TLS encryption.

5.5.2 Persistent Creation of Users

The second discrepancy occurs when creating a new user account with super-user privileges on the AV2700. The M-Plane specification states that upon creating a new user account and assigning it super-user privileges, the default `root` account on the device should be deactivated, and the active NETCONF connection should be disconnected [47]. However, we found that after creating a new user account and assigning super-user privileges on the AV2700, the device neither disconnects the active NETCONF connection with the default account nor deactivates the default `root` account. This behavior deviates from the specification and can facilitate follow-up attacks, e.g., for adversaries that manage to create a user via the debug port (C_2).

5.5.3 Default Credentials

The O-RAN Alliance has identified the use of default credentials on RUs as a main security issue [44]. Considering the prevalence of default password lists [38] and the associated risks in network equipment [13], the failure to deactivate the default account poses severe security risks. Adversaries can gain access to deployed AV2700s by brute-forcing devices with default password lists as long as the default super-user remains active, which can be attacked by adjacent adversaries with access to a connected Ethernet port (C_1).

6 Discussion

This section discusses the requirements of our findings and their impact on the operation of the O-RAN (Section 6.1). We outline mitigation means for $F_1 - F_4$ (Section 6.2). We discuss the security implications of our findings considering technological trends related to indoor BS (Section 6.3), 5G and beyond (Section 6.4), and the O-RAN ecosystem (Section 6.5). Finally, we address limitations of our work, future work (Section 6.6), and the responsible disclosure process (Section 6.7).

6.1 Impact on the Cellular Network

In Section 3, we defined the goal of our presumed adversary as full control of an RU running in an O-RAN. This section summarizes the exploitation requirements of findings $F_1 - F_4$

(Section 6.1.1) and their impact (Section 6.1.2) on the cellular network. Figure 7 depicts which capabilities are required to exploit $F_1 - F_4$ and what level of control they enable. Finally, we point out follow-up attacks (Section 6.1.3).

6.1.1 Requirements

Findings $F_1 - F_4$ are all exploitable via the RU's Open Fronthaul interface. Thus, adversaries with access to an adjacent Ethernet port connected to the RU can exploit them (C_1). The adversary is not required to have specific knowledge of any credentials.

6.1.2 Impact

In the following, we discuss what adversaries can achieve with $F_1 - F_4$ and how close that brings them to fully controlling an RU running in O-RAN.

Reconfiguration With C_1 , adversaries can reconfigure the running RU with F_2 . Note how F_1 and F_4 also enable reconfiguration of the RU.

Denial of Service A DoS attack on the RU leads to users losing access to the cellular network. An adversary that is limited to C_1 can exploit F_2 to achieve DoS by reconfiguration of settings in the `oru-shell`: (1) data transmission can be interrupted by modifying RF parameters, (2) access to the RU can be hindered by configuring a VLAN tag unknown to the network operator, or (3) the device can be rebooted repeatedly with the `reboot` option to disrupt availability. Note how, with C_2 , C_3 , or C_4 , adversaries can trivially achieve DoS by repeatedly restarting or shutting down the RU.

Full Access Findings F_1 and F_4 both grant the ability to execute arbitrary code on the RU. Notably, both findings only require C_1 and allow RCE in the security context of the `root` user. Thus, **F_1 and F_4 give the adversary full control of the RU**. Assuming it is feasible to gain RCE with F_3 , that finding also gives the adversary full control of the RU.

6.1.3 Follow-Up Attacks

Figure 1 depicts to which O-RAN components the RU is connected. With full control over an RU, there are three potential follow-up goals: (1) targeting users via their UEs, (2) attacking the O-RAN DU on the Open Fronthaul CUSM-Plane, or (3) attacking the O-RAN SMO on the Open Fronthaul M-Plane. Adversaries can target users by injecting downlink traffic to attack UEs. While no known attacks targeting users from an O-RAN RU exist, similar attacks exist for LTE [12, 34, 51]. Lateral movement in the O-RAN is possible towards the DU and the SMO. Adversaries can conduct the Open Fronthaul C-Plane DoS attack against the DU described

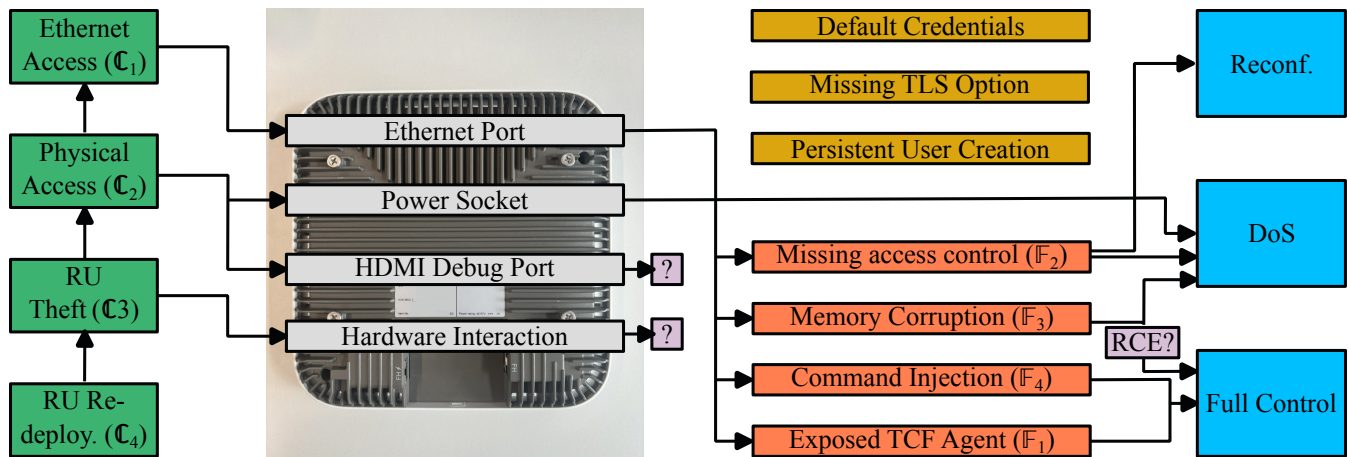


Figure 7: The Requirements and impact of our findings $\mathbb{F}_1 - \mathbb{F}_4$. From left to right, the adversary’s capabilities $\mathbb{C}_1 - \mathbb{C}_4$ determine the level of access to the Radio Unit (RU). Our findings enable the adversary to achieve attacker goals, i.e., reconfiguration, Denial-of-Service (DoS), or full control of the RU. While not required for the shown attacks, the Open Fronthaul standard deviations facilitate further attacks. We also highlight angles for future work.

by Liao et al. [35]. They can also attempt to get in control of a DU [6]. Adversaries can attack the SMO on the Open Fronthaul M-Plane [58, 60].

6.2 Mitigating the Discovered Vulnerabilities

Mitigating \mathbb{F}_1 is straightforward by removing the exposed TCF agent before deployment. To mitigate \mathbb{F}_2 , we recommend limiting internal services to local addresses to avoid exposing them to external threats. Regarding \mathbb{F}_3 , we recommend performing explicit bound checking on all untrusted user input and considering switching away from the programming language C [48]. Vulnerability \mathbb{F}_4 is addressable by sanitizing user input before passing it to functions that evaluate commands, such as the `system()` function. Limiting the internal services to local addresses, as suggested for \mathbb{F}_2 , also restricts the exploitability of this issue but still enables a low-privileged user to escalate their privileges.

Generally, we recommend applying a reasonable threat model (Section 3) during the software design phase to limit the external attack surface from an architectural point of view. Furthermore, we identified several deviations from the O-RAN and Open Fronthaul specifications (Section 5.5). Coherency to these standards, especially regarding security, ensures the implementation of the best practices, thus mitigating vulnerabilities in general.

6.3 Indoor Base Stations

The high number of security-related issues emphasizes the need for an updated threat model for indoor BSs. This shift voids the assumption that only trusted entities can directly communicate with the RU. Combined with a system architecture that exposes many services without authentication,

as described in Section 4.3, the AV2700 presents a vast attack surface. Large parts of the AV2700’s internal code are probably written in C, requiring high security awareness and rigorous security testing [53]. Without such precautions, memory safety bugs that lead to vulnerabilities are very likely.

As we show in Section 5, the security weaknesses affecting the AV2700 spread beyond memory corruption issues, including missing access control for dangerous services and an OS command injection. These problems can also occur in software written in memory-safe languages. Therefore, it is necessary to follow security best practices and apply a proper threat model during the development phase, reflecting the reality that adversaries might have physical access to the RU when deployed as an indoor BS in a public space.

6.4 Technologies of 5G and Beyond

The intended application areas of 5G, namely enhanced Mobile Broadband (eMBB), Ultra Reliable Low Latency Communications (URLLC), and Massive Machine Type Communications (mMTC), incur high requirements on the RAN. 5G facilitates novel technology, such as FPGAs for mmWave beamforming to fulfill these requirements. However, using novel technology in gNBs and O-RAN RUs introduces new challenges for RAN vendors and mobile network operators (MNOs), e.g., more complex hardware in indoor BSs (Section 4.2) and other RAN components.

While \mathbb{F}_1 is not a vulnerability within the cellular network itself, it is exploitable by an adjacent adversary to gain access to the AV2700’s host system, from where escalation to the AV2700 is trivial with `root` privileges. The exposed TCF agent vulnerability (Section 5.1) is a direct cause of developers not removing the TCF agent from the AV2700 before

deployment. As FPGAs are included in O-RAN RUs to fulfill the requirements of 5G in the application areas, \mathbb{F}_1 is a consequence of the novel technologies of 5G and beyond. Additionally, with the TCF debugger enabled by default for the widespread *Zynq UltraScale+*, \mathbb{F}_1 can likely be reproduced on other RUs.

6.5 Complexity of the Open RAN Ecosystem

The O-RAN ecosystem is complex with its new open interfaces and introduced features. The different components are highly interconnected through the various interfaces, and research identified that adversaries can use the interconnectivity to their advantage to escalate attacks [36, 39]. As detailed in [6], establishing control of an RU provides an adversary with the means to escalate attacks upwards, penetrating the O-RAN through the DU and beyond, consequently impacting the entire O-RAN ecosystem, including the CU, the SMO, and the RICs. The security implications of this intrusion into the O-RAN are critical as adversaries might access user- and other sensitive data. They might manipulate the O-RAN to transmit malicious packets and data to users, potentially affecting user devices. Additionally, an adversary might bring down the entire O-RAN with a DoS attack, leading to a large-scale outage in 5G, classified as a critical infrastructure.

6.6 Limitations and Future Work

We did not fully evaluate the RU's HDMI debug port. An adversary with access to all interfaces (\mathbb{C}_2) might use the RU's debug port to perform a DoS attack or prepare follow-up attacks that lead to RCE, e.g., creating a new super user. An adversary capable of removing the RU (\mathbb{C}_3) can perform more intrusive operations to achieve full control of the RU, e.g., firmware modifications or hardware fault injection. However, if these attacks lead to RCE in the security context of `root`, the adversary still needs to redeploy the RU into the running O-RAN to achieve their goal, requiring \mathbb{C}_4 .

We analyzed the AV2700 as an example of a proprietary indoor O-RAN RU. We focused on the capabilities of an adversary abusing physical access to an indoor RU, potentially stealing, modifying, and redeploying the RU. As we did not analyze an RU in a live O-RAN, future work might provide valuable insights into how much CU-Plane traffic an adversary with full control of the RU can access.

While we aimed to highlight general issues with indoor O-RAN RUs, our evaluation considered only one product, the AV2700. Future work might reproduce our findings on other indoor RUs and assess to which extent our findings are generalizable.

6.7 Responsible Disclosure

We privately reported \mathbb{F}_1 to Airspan on April 19, 2023. After waiting for an acknowledgment or response, we sent a follow-up email on February 13, 2024, with a revised deadline of April 13, 2024, marking 360 days from the initial reporting. On February 14, 2024, an Airspan executive responded to our email, who acknowledged dismissing our initial email as a phishing attempt. We were assured that the responsible team at Airspan had been informed about our report and that they would contact us regarding the vulnerability and next steps. On February 21, 2024, we privately reported $\mathbb{F}_2 - \mathbb{F}_4$ to Airspan. We set a deadline for May 21, 2024, marking 90 days from the day of reporting, which complies with recommended industry practice [29]. To the best of our knowledge, Airspan is now working on patches for $\mathbb{F}_1 - \mathbb{F}_4$.

7 Conclusions

With this paper, we contribute to the RAN security of 5G and beyond, especially regarding the deployment of indoor BSs. We introduce a threat model for indoor BSs, considering they are more easily accessible than outdoor BSs. Our security analysis of the Airspan AirVelocity 2700 (AV2700) results in multiple deviations from the O-RAN and Open Fronthaul standards. We find four vulnerabilities on the AV2700 that we, due to the lack of official scores, self-assign high or critical CVSS scores ($\mathbb{F}_1 - \mathbb{F}_4$) and recommend mitigation means for all of them. Our findings show that vulnerabilities in the host system of a state-of-the-art indoor BS are exploitable to Remote Code Executions (RCEs), which facilitate follow-up attacks on the RAN. This highlights the importance of securing not only the RAN-related implementations of a RAN component but also the underlying host.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their helpful suggestions. This work had been co-funded by the Federal Ministry of Education and Research of Germany in the project Open6GHub (grant number: 16KIS014) and the German Research Foundation (DFG) in the project CRUST (grant number: 503199853).

References

- [1] 3rd Generation Partnership Project (3GPP). Study on CU-DU lower layer split for NR, Technical Report (TR) 38.816, version 15.0.0. Technical report, December 2017.
- [2] 3rd Generation Partnership Project (3GPP). Study on New Radio Access Technology: Radio Access Archi-

- ecture and Interfaces, Technical Report (TR) 38.801, version 14.0.0. Technical report, March 2017.
- [3] 3rd Generation Partnership Project (3GPP). NR; NR and NG-RAN Overall description; Stage-2, Technical Specification (TS) 38.300, version 18.0.0. Technical specification, December 2023.
- [4] 3rd Generation Partnership Project (3GPP). NR; Physical Layer; General Description, Technical Specification (TS) 38.201, version 18.0.0. Technical specification, September 2023.
- [5] 3rd Generation Partnership Project (3GPP). System architecture for the 5G System (5GS), Technical Specification (TS) 23.501, version 18.4.0. Technical specification, December 2023.
- [6] Aly Sabri Abdalla and Vuk Marojevic. End-to-End O-RAN Security Architecture, Threat Surface, Coverage, and the Case of the Open Fronthaul, 2023.
- [7] Airspan Networks Inc. 5G Products. <https://airspan.com/5g-products/>, 2024. Accessed: 2024-03-10.
- [8] ARM Ltd. Cortex-a53 – the most widely used low-power processor. <https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a53>, 2024. Accessed: 2024-03-10.
- [9] Arm Ltd. Cortex-r5 – seamless, real-time embedded processors. <https://www.arm.com/products/silicon-ip-cpu/cortex-r/cortex-r5>, 2024. Accessed: 2024-03-10.
- [10] Leonardo Bonati, Michele Polese, Salvatore D’Oro, Stefano Basagni, and Tommaso Melodia. Open, Programmable, and Virtualized 5G Networks: State-of-the-Art and the Road Ahead. *Computer Networks*, 182:107516, December 2020.
- [11] BusyBox Developers. BusyBox. <https://busybox.net/>. Accessed: 2024-03-04.
- [12] Merlin Chlosta, David Rupprecht, Christina Pöpper, and Thorsten Holz. 5g suci-catchers: Still catching them all? In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 359–364, 2021.
- [13] Min-kyu Choi, Rosslyn John Robles, Chang-hwa Hong, and Tai-hoon Kim. Wireless Network Security: Vulnerabilities, Threats and Countermeasures. *International Journal of Multimedia and Ubiquitous Engineering*, 3(3):77–86, 2008.
- [14] Jeffrey Cichonski, Joshua M Franklin, and Michael Bartock. Guide to LTE Security.
- [15] Charles Clancy and Scott G. Kelly. Control And Provisioning of Wireless Access Points (CAPWAP) Threat Analysis for IEEE 802.11 Deployments. Informational RFC 5418, Internet Engineering Task Force.
- [16] Daniel Dik and Michael Stübert Berger. Transport Security Considerations for the Open-RAN Fronthaul. In *2021 IEEE 4th 5G World Forum (5GWF)*, pages 253–258, 2021.
- [17] Daniel Dik and Michael Stübert Berger. Open-RAN Fronthaul Transport Security Architecture and Implementation. *IEEE Access*, 11:46185–46203, 2023.
- [18] Eclipse Foundation. Target Communication Framework (TCF). <https://wiki.eclipse.org/TCF>. Accessed: 2024-02-15.
- [19] Eclipse Foundation. TCF/RISC-V: Connect the TCF Debugger. https://wiki.eclipse.org/TCF/RISC-V#Connect_the_TCF_Debugger, 2020. Accessed: 2024-02-15.
- [20] Enclustra FPGA Solutions. Mercury+ XU8 | Xilinx Zynq UltraScale+ MPSoC Module. <https://www.enclustra.com/en/products/system-on-chip-modules/mercury-xu8/>, 2024. Accessed: 2024-03-10.
- [21] Rob Enns, Martin Björklund, Andy Bierman, and Jürgen Schönwälder. Network Configuration Protocol (NETCONF). RFC 6241, June 2011.
- [22] European Telecommunications Standards Institute (ETSI). 5G; Security architecture and procedures for 5G System (3GPP TS 33.501 version 17.12.0 Release 17). Technical specification, January 2024.
- [23] Muhammad Najmul Islam Farooqui, Junaid Arshad, and Muhammad Mubashir Khan. A Layered Approach to Threat Modeling for 5G-Based Systems. 11(12):1819.
- [24] Federal Office for Information Security (BSI). Open-RAN Risk Analysis. Technical report, Federal Office for Information Security, Germany, 2 2022. Accessed: 2024-03-10.
- [25] Forum of Incident Response and Security Teams (FIRST). Common Vulnerability Scoring System Version 4.0 Calculator. <https://www.first.org/cvss/calculator/4.0>, 2024. Accessed: 2024-03-10.
- [26] Eclipse Foundation. Eclipse Mosquitto. <https://mosquitto.org/>.
- [27] Free Software Foundation, Inc. Source Fortification (The GNU C Library). https://www.gnu.org/software/libc/manual/html_node/Source-Fortification.html.

- [28] Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. Weaponizing femtocells: The effect of rogue devices on mobile telecommunications. In *NDSS*, 2012.
- [29] Google. About google’s app security. <https://about.google/appsecurity/>, 2024. Accessed: 2024-03-10.
- [30] Joshua Groen, Salvatore DOro, Utku Demir, Leonardo Bonati, Michele Polese, Tommaso Melodia, and Kaushik Chowdhury. Implementing and Evaluating Security in O-RAN: Interfaces, Intelligence, and Platforms, 2023.
- [31] Sebastian Haas, Mattis Hasler, Friedrich Pauls, Stefan Köpsell, Nils Asmussen, Michael Roitzsch, and Gerhard Fettweis. Trustworthy computing for o-ran: Security in a latency-sensitive environment. In *2022 IEEE Globecom Workshops (GC Wkshps)*, pages 826–831. IEEE, 2022.
- [32] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic Heap Layout Manipulation for Exploitation. pages 763–779.
- [33] Felix Klement, Stefan Katzenbeisser, Vincent Ulitzsch, Juliane Krämer, Slawomir Stanczak, Zoran Utkovski, Igor Bjelakovic, and Gerhard Wunder. Open or not open: Are conventional radio access networks more secure and trustworthy than Open-RAN?, 2022.
- [34] Gyuhong Lee, Jihoon Lee, Jinsung Lee, Youngbin Im, Max Hollingsworth, Eric Wustrow, Dirk Grunwald, and Sangtae Ha. This is your president speaking: Spoofing alerts in 4g lte networks. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 404–416, 2019.
- [35] Shu-Hua Liao, Chih-Wei Lin, Fransiscus Asisi Bimo, and Ray-Guang Cheng. Development of C-Plane DoS Attacker for O-RAN FHI. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, page 850–852. Association for Computing Machinery, 2022.
- [36] Madhusanka Liyanage, An Braeken, Shahriar Shahabuddin, and Pasika Ranaweera. Open RAN security: Challenges and opportunities. *Journal of Network and Computer Applications*, 214:103621, 2023.
- [37] Ahmed Redha Mahlous. Threat model and risk management for a smart home iot system. *Informatica*, 47(1), 2023.
- [38] Daniel Miessler, Jason Haddix, and g0tmilk. SecLists. <https://github.com/danielmiessler/SecLists>, 2024. Accessed: 2024-03-07.
- [39] Dudu Mimran, Ron Bitton, Yehonatan Kfir, Eitan Klevansky, Oleg Brodt, Heiko Lehmann, Yuval Elovici, and Asaf Shabtai. Evaluating the Security of Open Radio Access Networks, 2022.
- [40] David Muirhead, Muhammad Ali Imran, and Kamran Arshad. Insights and approaches for low-complexity 5g small-cell base-station design for indoor dense networks. *IEEE access*, 3:1562–1572, 2015.
- [41] O-RAN Working Group 1. O-RAN Architecture Description. Technical Specification OAD-R003-v11.00, O-RAN ALLIANCE, 2024. Available online at <https://www.o-ran.org/specifications>.
- [42] O-RAN Working Group 11. O-RAN Security Threat Modeling and Risk Assessment. Technical Report TR.0-R003-v02.00, O-RAN ALLIANCE, 2024.
- [43] O-RAN Working Group 11. Security Requirements and Controls Specifications. Technical Report TR.0-R003-v08.00, O-RAN ALLIANCE, 2024.
- [44] O-RAN Working Group 11. Study on O-RU Centralized User Management. Technical Report TR.0-R003-v01.00, O-RAN ALLIANCE, 2024.
- [45] O-RAN Working Group 11. Study on Security for Shared O-RU. Technical Report TR.0-R003-v04.00, O-RAN ALLIANCE, 2024.
- [46] O-RAN Working Group 4. Control, User, and Synchronization Plane Specification. Technical Specification CUS.0-R003-v14.00, O-RAN ALLIANCE, 2023.
- [47] O-RAN Working Group 4. Management Plane Specification. Technical Report MP.0-R003-v14.00, O-RAN ALLIANCE, 2024.
- [48] Office of the National Cyber Director. Back to the Building Blocks: A Path Toward Secure and Measurable Software. Technical report, White House, Washington, DC, 2 2024. Accessed: 2024-03-10.
- [49] Mark Alan Overby. HDMI-muxed debug cable methods and apparatuses, October 7 2014. US Patent 8,856,744.
- [50] Mark Alan Overby. HDMI-muxed debug port methods and apparatuses, April 7 2015. US Patent 9,003,369.
- [51] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyoo Lee, Insu Yun, and Yongdae Kim. {DoLTEst}: In-depth downlink negative testing framework for {LTE} devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1325–1342, 2022.
- [52] Michele Polese, Leonardo Bonati, Salvatore D’oro, Stefano Basagni, and Tommaso Melodia. Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges. *IEEE Communications Surveys & Tutorials*, 2023.

- [53] Alex Rebert and Christoph Kern. Secure by Design: Google's Perspective on Memory Safety. Technical report, March 2024.
- [54] Danish Sattar, Ashraf Matrawy, Troy Bryant, and Marc Kneppers. Threat Modeling in LTE Small Cell Networks. In *2018 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE)*, pages 1–5. IEEE.
- [55] Chih-Ting Shen, Yu-Yi Xiao, Yi-Wei Ma, Jiann-Liang Chen, Cheng-Mou Chiang, Shiang-Jiun Chen, and Yu-Chuan Pan. Security threat analysis and treatment strategy for oran. In *2022 24th International Conference on Advanced Communication Technology (ICACT)*, pages 417–422. IEEE, 2022.
- [56] Small Cell Forum. 5G nFAPI Specifications. https://scf.io/en/documents/225_5G_nFAPI_specifications.php. Accessed: 2024-03-11.
- [57] Small Cell Forum. Small Cell Forum. <https://www.smallcellforum.org/>. Accessed: 2024-03-11.
- [58] Kashyap Thimmaraju, Altaf Shaik, Sunniva Flueck, Christian Werling, and Jean-Pierre Seifert. Security testing the o-ran near-real time ric & a1 interface. In *Proceedings of the 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'24)*, 2024.
- [59] Aju Mathew Thomas, Gowtham Akshaya Kumaran, R Ramaguru, R Harish, and K Praveen. Evaluation of wireless access point security and best practices for mitigation. In *2021 5th International Conference on Electrical, Electronics, Communication, Computer Technologies and Optimization Techniques (ICEECCOT)*, pages 422–427. IEEE, 2021.
- [60] Walter Tiberti, Eleonora Di Fina, Andrea Marotta, and Dajana Cassioli. Impact of man-in-the-middle attacks to the o-ran inter-controllers interface. In *2022 IEEE Future Networks World Forum (FNWF)*, pages 367–372. IEEE, 2022.
- [61] Orange Tsai. BabyFirst Revenge v2 Challenge Writeup. <https://github.com/orangetw/My-CTF-Web-Challenges/blob/325ab4e4b4888a7ca73092b8f9e4af70844a09e9/README.md#babyfirst-revenge-v2>. Accessed: 2024-02-21.
- [62] Sichao Wen and Yuandan Dong. A Low-Profile Wideband Antenna With Monopolelike Radiation Characteristics for 4G/5G Indoor Micro Base Station Application. *IEEE Antennas and Wireless Propagation Letters*, 19(12):2305–2309, 2020.
- [63] Xilinx Inc. *PetaLinux Tools Documentation*. Xilinx Inc., 2019. Accessed: 2024-03-04.
- [64] Xilinx, Inc. Zynq ultrascale+ mpsoc. <https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html>, 2024. Accessed: 2024-03-10.
- [65] Bin Yang, Yue Hou, Yefeng Zhang, Shiyong Feng, and Yong Zhang. Security architecture of wireless private networks for smart grid. *Electrical Engineering and Computer Science (EECS)*, 2:95–98, 2019.
- [66] ZeroMQ authors. ZeroMQ. <https://zeromq.org/>.



RIPencapsulation: Defeating IP Encapsulation on TI MSP Devices

Prakhar Sah
Virginia Tech
sprakhar@vt.edu

Matthew Hicks
Virginia Tech
mdhicks2@vt.edu

Abstract

Internet of Things (IoT) devices sit at the intersection of unwieldy software complexity and unprecedented attacker access. This unique position comes with a daunting security challenge: how can we protect both proprietary code and confidential data on a device that the attacker has unfettered access to? Trusted Execution Environments (TEEs) promise to solve this challenge through hardware-based separation of trusted and untrusted computation and data. While TEEs do an adequate job of protecting secrets on desktop-class devices, we reveal that trade-offs made in two of the most widely-used commercial IoT devices undermine their security.

This paper uncovers two fundamental weaknesses in IP Encapsulation (IPE), the TEE deployed by Texas Instruments for MSP430 and MSP432 devices. We observe that lack of call site enforcement and residual state after unexpected TEE exits enable an attacker to reveal all proprietary code and secret data within the IPE. We design and implement an attack called RIPencapsulation, which systematically executes portions of code within the IPE and uses the partial state revealed through the register file to exfiltrate secret data and to identify gadget instructions. The attack then uses gadget instructions to reveal all proprietary code within the IPE. Experiments with commodity devices and a production compiler show that—even after following all manufacturer secure coding practices—RIPencapsulation reveals, within minutes, both the code and keys from third-party cryptographic software, as well as allowing unrestricted writes to TEE memory.

1 Introduction

The global IoT industry is projected to become a trillion-dollar industry by 2027 [2]. IoT devices are widely deployed in both safety- and mission-critical roles in government, healthcare, transportation, manufacturing, defense, and telecommunications industries. Thus, these devices are a treasure trove of sensitive information. Data security concerns are a major obstacle to the growth of the IoT sensor market as data breaches

continue to rise with the advancement of technology. In addition to data security, the growing software complexity of IoT devices and proliferation of Artificial Intelligence mandate code security, i.e., the protection of proprietary algorithms and models. Failing to protect proprietary code and secret data puts both consumers and companies at risk.

“Trying to design information security solutions without due consideration of the complex human nature may prove to be an Achilles heel” [9]. Cryptographic algorithms like AES and RSA provide confidentiality of data, but the key still ends up in device memory. This leaves keys vulnerable to exfiltration by an attacker with physical access. Unfortunately, physical access is the common case for IoT devices. To address the threat of co-resident attackers, manufacturers provide a Trusted Execution Environment (TEE). TEEs bifurcate hardware (either physically or virtually) into security domains, where code and data in the high-security domain are protected from the low-security domain. For IoT-class devices, Texas Instruments provides a TEE called IP Encapsulation (IPE), which physically partitions device memory into a protected region and an unprotected region.

Texas Instruments is the world’s second-largest manufacturer of microcontrollers [7]. Their MSP family of devices consists of over 2000 unique devices [3] making them one of the most widely deployed microcontrollers [4, 5]. MSP430s are 16-bit industrial-grade microcontrollers with low power consumption at a low cost. The MSP432 line of microcontrollers extends the capabilities of the MSP430 with a 32-bit, ARM-based architecture. Both devices have TEE support in the form of IPE, making IPE the most widely deployed TEE.

While the community continues to probe the security of TEEs provided by higher-end devices, the security of IoT-class TEEs remains under-explored. This paper fills that gap by analyzing the security of TI’s IPE. IPE protects code and data within the IPE zone from all non-IPE zone read and write accesses [37, 45, 48]. IPE is enforced by the Memory Protection Unit (MPU) for MSP430 and the System Controller (SYSCTL) module for MSP432, which restricts direct external IPE zone accesses by checking the origin of mem-

ory accesses. In addition, the MPU/SYSCTL also restricts all JTAG/DMA accesses to the IPE zone. Thus, only memory accesses from within the IPE are allowed to the IPE zone, making it ideal for storing secret data (e.g., keys) and proprietary code (e.g., AI models). Our exploration of IPE security reveals that—despite following the TI-recommended secure programming practices—the state-of-art compilers produce assemblies that leak information from and allow unrestricted writes to the IPE zone via the unprotected register file. These weaknesses render IP encapsulation insecure.

IPE has two weaknesses that undermine security by enabling **arbitrary read and write access to secure memory**:

- IPE allows all code outside the IPE zone to branch to arbitrary instructions within the IPE zone.
- When execution leaves the IPE zone unexpectedly (e.g., via an interrupt) the contents of the register file remain.

We construct an attack leveraging these weakness called RIPencapsulation, which exfiltrates all code and data protected by the IPE—within minutes. Inspired by SGX-Step [49] and Interrupt-oriented Bugdoor Programming [43], but extending them to the IoT domain, RIPencapsulation combines interrupt-based control flow attack patterns, with data-oriented attack patterns, and side-channel attack patterns to break IPE. **The result of RIPencapsulation is unrestricted read, write, and execute access to IPE-protected memory.**

We implement and evaluate RIPencapsulation on TI MSP430 and MSP432 devices. Our evaluation shows that RIPencapsulation reveals the entire contents of the IPE zone with third-party implementations of AES, SHA256, and RSA, using a variety of optimization settings on a production compiler. RIPencapsulation exfiltrates all keys and code from both devices, automatically, within minutes—even when following all of TI’s secure coding best-practices.

This paper makes the following technical contributions:

- **Create a side channel:** We design and implement an interrupt-based side channel that reveals the state within the IPE zone of TI MSP devices (§4, §5, §6).
- **Reconstruct firmware:** We use partial state, timing, and size information to reconstruct the IP-encapsulated assembly (§4.2, §5.2).
- **Demonstrate generality:** We show the problem is pervasive across TI MSP devices, memory types, cryptographic implementations, and compiler optimizations (§7, §7.6).
- **Modify firmware:** We uncover interrupt-based data-oriented gadgets to modify the IP-encapsulated code and data (§7.7).
- **Discuss mitigation:** We qualitatively analyze various mitigation techniques with respect to their impact on an IoT device (§8).
- **Open-source RIPencapsulation.** We open-source all attack and evaluation code [8].

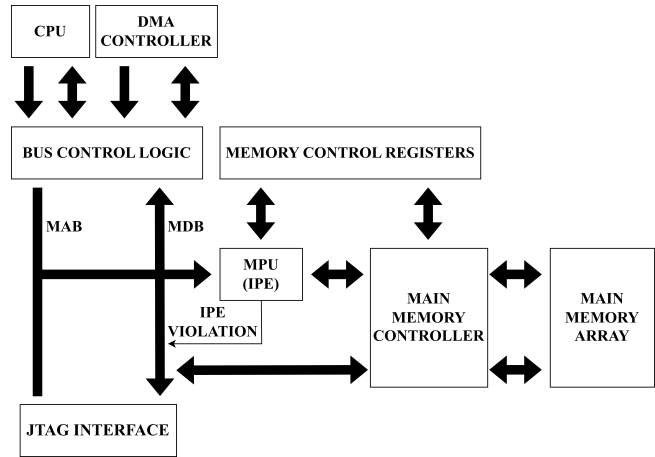


Figure 1: Block diagram illustrating how the MPU enforces IP Encapsulation on MSP430 devices.

Responsible disclosure. We have disclosed RIPencapsulation and the threat it poses to the security guarantees of IP Encapsulation on MSP-class devices to Texas Instruments.

2 Background

This section covers topics underpinning RIPencapsulation.

2.1 Trusted Execution Environment

A Trusted Execution Environment (TEE) is a secure processing area that provides isolated execution and secure storage of code and data inside a tamper-resistant module. TEEs include process isolation, the integrity of applications running inside the TEE, and the confidentiality of data associated with it. TEEs achieve this by using a memory protection mechanism restricting access to the security-critical software module [39].

Intel Software Guard Extensions (SGX) and AMD Secure Encrypted Virtualization (SEV) are multi-tenant systems running with operating systems and full-fledged commodity desktop/server-class systems. They use hardware memory-based encryption to isolate sensitive code and data. This type of deployment scenario is not available in low-end microcontroller devices like the MSPs due to the hardware overhead of the encryption engine and the run time overhead of encryption in an environment with single-cycle main memory access.

ARM TrustZone is a microcontroller-focused TEE that splits the physical memory into secure and normal regions, with specifically assigned computational units like Secure Attribution Unit (SAU), ensuring safe context switches between secure and normal processes, or lightweight remote attestation schemes like SMART [21] and VRASED [35], which provide user trust with minimal HW/SW modifications. SANCUS [34] is another secure architecture that provides

isolated execution and privacy of code and data in low-level networked embedded systems.

The only TEE for the low-end devices common to large-scale IoT deployments is TI’s Intellectual Property Encapsulation (IPE). IPE protects the encapsulated memory region from all direct non-IPE accesses, whether it be from on-chip execution or off-chip via the debugger, i.e., only program code executed from the IPE region itself has access privileges to IPE code and data. Figure 1 illustrates the IPE implementation for MSP430 devices: the Memory Protection Unit (MPU) verifies IPE region accesses by snooping the Memory Address Bus (MAB) and the program counter to check whether the access request is made by code in the IPE zone. Any unauthorized access to the IPE zone causes the MPU to drive the Memory Data Bus (MDB) with $0x3FFF$. To execute code stored inside the IPE segment, the program must call functions within the IPE zone. The MSP432’s IPE mechanism is slightly different technically, but works in a similar manner and has the same flaws as the MSP430 IPE (§4).

Existing attacks demonstrate ways to undermine deployed TEEs. CLKscrew [44] exploits software-exposed energy management mechanisms to introduce faults in the secure part of the memory, exfiltrating IP from TrustZone. Volt Boot [32] is another attack that demonstrates the vulnerability of on-chip volatile memories due to the physical separation common to modern system-on-chip power distribution networks. Recent research reveals vulnerabilities in TEEs like ARM TrustZone [15] and AMD SEV [30, 31], calling into question the reliability of these protection mechanisms. **Our work adds to the TEE attack literature with an attack on TI’s IPE.**

2.2 Code-reuse Attacks

Attackers turn to code reuse attacks when injecting their own code is prevented. Instead of injecting new code, the attacker constructs malicious functionality by chaining existing code snippets, called “gadgets” found in the target program. These gadgets are short sequences of instructions, typically ending with a “return” instruction. By crafting a chain of gadgets and manipulating the program’s control flow, the attacker is able to execute arbitrary commands. Many techniques exist in the literature that utilize various aspects of the memory control plane [11–13, 16, 40] and data plane [18, 25] to create Turing-complete gadgets for malicious purposes. We take inspiration from such techniques, replacing a `ret` with a carefully crafted interrupt and leveraging data-oriented attack principles.

2.3 Context Switches and Call Site Verification

Context switching is the process of transitioning the execution from the untrusted zone to the secure/trusted zone and vice versa. When a context switch occurs, the current state of the regular execution environment, including registers, memory contents, and program counter is saved, and the system

transitions between zones. Secure context switching in TEEs is crucial for ensuring the isolation of trusted code and data from the rest of the system as any data that remains serves as a side channel to untrusted software.

In addition to context clearing, a TEE must enforce call site verification to ensure that untrusted code interfaces with trusted code in acceptable ways. When a program interacts with the TEE, it makes calls to procedures within the enclave. These calls are known as “call sites”. Call site verification verifies that the callee is a function allowed to be called from untrusted code. This prevents attackers from hijacking the control flow and executing arbitrary gadgets within the enclave. We show that call site verification is a necessary component of any secure TEE.

3 Threat Model

The defender’s code is bug-free. The defender follows TI’s secure coding practices: (1) they clear IPE state on exit and (2) disable interrupts upon IPE entry. The defender uses commodity cryptographic algorithms and compilers.

We assume the attacker has the capability to run any code in the untrusted world, this can either come from co-resident software or from having physical access [9]. This means the attacker can configure the timer and interrupt service routines. We assume the attacker’s software has a way to communicate with the attacker; The attacker wishes to have arbitrary read, write, and execute access to IPE-protected code and data. Note that this level of access is in-keeping with how TI expects users to interface with the IPE, i.e., the IPE is a place for protected, third-party code that untrusted user software can use as a library.

4 RIPencapsulation Design

RIPencapsulation is an interrupt-based side-channel memory exfiltration attack on TI MSP IP-Encapsulated (IPE) memory. As Figure 2 illustrates, the attack consists of three phases. After the first two phases of the attack, we are able to reverse engineer 80% of the IPE-protected code. If the attacker’s goal is only to get the keys from commodity cryptographic implementations, this generally suffices, as we prove in §7. However, completing all three phases enables the attacker to exfiltrate 100% of IPE-protected memory.

4.1 Creating a Side Channel (Phase 1)

The fundamental weakness in the design of IPE is that it does not protect the register file when context switches outside the IPE zone. This first phase exploits this weakness as a side channel that reveals the internal state of the IPE zone. The naive approach is to single-step IPE code using a debugger, but the MPU/SYSCTL monitors the memory address bus and

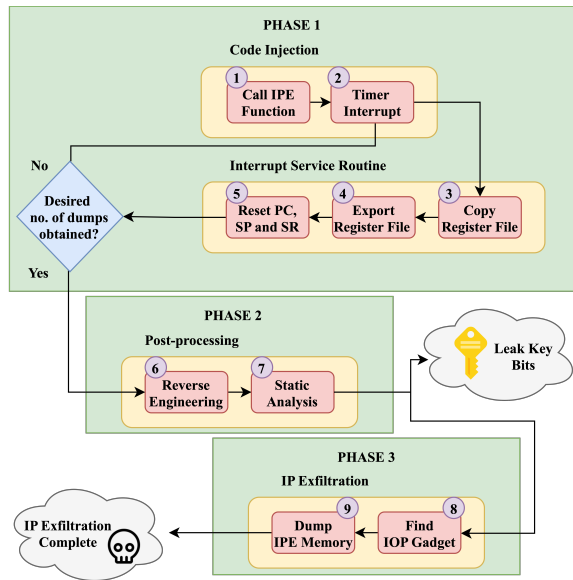


Figure 2: High-level IPE attack flow.

prevents JTAG or DMA from accessing the IPE-protected memory zone. This prevents breakpoints from working inside the IPE zone. While we observe that it is still possible to send HALT signals while in the IPE zone, it only provides timing granularity of 250 ms—not single-instruction precision.

The ability to process signals indicates that, in general, interrupts are processed while executing in the IPE zone. Thus, we can leverage a more precise interrupt source for single-instruction precision: the timer. Timer module interrupts have the capability to interrupt a program at single-cycle granularity (i.e., sub-instruction level).

Figure 2 shows the flow of the RIPencapsulation attack. In ①, attacker-controlled code sets up the timer module to interrupt the processor at a counter value of one clock cycle into the IPE code. Then the attacker code jumps directly to the instruction after the instructions which disables external interrupts.¹ In ②, the victim IPE code executes until the timer module interrupts it and passes control to the attacker’s ISR. Note that on an interrupt, any currently executing instruction is completed, and the PC of the next instruction, along with the SR, is pushed onto the stack. The context of the IPE code at the time of the interrupt remains intact.

Inside the ISR, ③ we copy the register states to attacker-controlled memory. This includes all the general-purpose registers as well as the stack pointer. In ④, the memory space available on MSP430/MSP432 is limited, with memory sizes around 256KB. So saving the register states of the entire

¹TI’s secure coding practice recommends disabling interrupts while executing in the IPE zone. We leverage the lack of call site verification to bypass any interrupt disabling code. It is also possible to use external interrupts or even a power reset in place of timer interrupts.

IPE-analysis process on the target MSP microcontroller is not feasible and we use its UART interface to transmit the exfiltrated register state to a workstation for analysis.² In ⑤, the attacker’s ISR resets the PC, SP, and SR registers to restore the program state to attacker-controlled code. The attacker code increments the timer counter value by one cycle and calls the victim code again. We repeat Phase 1 until the state-change due to all IPE code is captured. The end result is similar to single-stepping through the IPE code.

4.2 IPE State Dump Post-processing (Phase 2)

Phase 2 consists of post-processing the register state dumps and does not require access to the device. The goal is to decode the IPE assembly, extracting as much information as possible from the state dumps. In ⑥ we Look at the changes in PC deduce instruction boundaries. Comparing its PC with the PC of the next instruction, we are able to infer the size of the completed instruction unless it is a control-flow discontinuity. The frequency of successively repeating PCs in the state dumps provides instruction cycle count.

In the MSP430 instruction set architecture, the instruction size and cycles depend only on the operands and not the operation. An MSP430 instruction has broadly two types of operands, register or memory, with the possibility of combining these two for the source and destination operands. Identifying the operation is straightforward when both operands are direct register values. In such cases, the critical piece of information is the change in values of the general-purpose registers (GPRs) and how those changes correlate to the previous values of the registers. For instance, an ADD instruction with two register operands causes a difference in the value of the destination operand. By doing so, we decode the operation as well as the operands. However, we need to probe memory-to-register or register-to-memory instructions further in order to decipher them, as we detail in §5.2. We also observe that memory-to-memory operations are indecipherable just by looking at the register file, as at least one of the GPRs must change for identification.³

Status bit changes are also helpful in resolving the instruction guesses. A carry bit set implies that the result of the operation produces a carry. Zero or negative bit changes help us infer whether the result of an operation is zero or negative. The overflow bit signifies an overflow in the signed-variable range [46]. It also helps resolve certain operations, e.g., an AND instruction resets the overflow bit. These distinctions are most beneficial when the destination is not a register.

After reverse engineering, we are able to decode all the register and register-indirect addressing mode operations, when

²We also experimentally verify that it is possible to use the debugger to transmit the exfiltrated register state—albeit 22x slower than the UART.

³The ARM instruction set of the MSP432 only allows for register-to-register state-changing operations, which increases the power of Phase 1.

both operands are in one of those addressing modes.⁴ ⑦ The next step in post-processing depends on whether the attacker aims to obtain the key or exfiltrate the entire IPE memory.

4.3 IPE Memory Access Gadgets (Phase 3)

A ROP gadget is a security exploit that allows an attacker to execute code in the presence of code protection mechanisms [16, 38]. For TI’s IPE, there is no context switch gateway between calls from untrusted code to trusted code. We observe that an attacker has the capability to branch/jump to any instruction inside the IPE code without breaching the IPE security. Taking inspiration from ROP, we devise an interrupt- and data-oriented exploit for memory access purposes, which we call *Interrupt-Oriented Programming (IOP)*. Interrupt-oriented Bugdoor Programming (IOBP) [43] first introduced the idea of IOP in the context of complete read/write access locked microcontrollers. However, these IOBP gadgets are rare in real-world firmware. The fundamental idea of our IOP is that if we setup the registers for a specific IPE instruction, branch directly to that IPE instruction, wait for it to complete execution, interrupt execution before the following instruction executes, and look at the results in the attacker’s ISR, we get a Turing-complete set required for return-oriented programming [16, 38]. Register-indirect addressing mode instructions in the IPE code that directly modify the value of a destination register are ideal candidates for memory exfiltration as these instructions enable the attacker to read an IPE memory location by modifying the memory address value in the source register, followed by a readout of the value returned in the destination register. The same is true for register-based memory store instructions. Henceforth, we refer to this class of instructions as *IOP gadgets*.

⑧ The first step of arbitrary IPE memory access is to find an IOP gadget. ⑨ Next, the attacker injects code into the non-IPE memory that sets up the timer module with a counter value, allowing the execution of the targeted instruction to complete. The malicious function also writes the desired IPE segment base address to the source register of the instruction. Then the program jumps to the IOP gadget inside the IPE zone. After execution of the gadget instruction(s), the timer module interrupts IPE execution, invoking the attacker’s ISR. In the case of exfiltration, the ISR sends the value of the destination register to the attacker workstation and restores the PC, SP, and SR of the program to the attacker’s control code. For memory writes, the host uploads a new malicious value. Repeating this process, the attacker is able read and write the entire IPE-protected memory.

⁴An attacker may leverage the lack of call-site verification to control register values and memory addresses to further disambiguate instructions as part of Phase 1, but none of our attacks required it.

Platform	MSP430FR5994	MSP432P401R
ISA	MSP430 CPUX	ARMv7-M
NV Memory Type	FRAM	Flash
NV Memory Size	256 KB	256 KB
Max Clock Frequency	16 MHz	48 MHz
IP Encapsulation	Yes	Yes
UART Interface	Yes	Yes
Timer Module	Yes	Yes

Table 1: RIPencapsulation target platforms.

Algorithm 1 Interrupt-based side-channel routine for exfiltrating IPE process register states

```

1: ...Set timer ISR routine...
2:  $i \leftarrow 0$ 
3:  $TIMER\_COUNT \leftarrow 0$ 
4: while  $i \neq DESIRED\_DUMPS$  do
5:    $TIMER\_COUNT \leftarrow TIMER\_COUNT + 1$ 
6:   ...Set timer and UART parameters...
7:   ...Start timer counter...
8:   ...Enable interrupts...
9:   CALL IPE_Function()
10:  // Secure process gets interrupted
11:  ...Copy all core registers to unprotected memory...
12:  ...Copy saved state to UART TX Buffer...
13:   $i \leftarrow i + 1$ 
14:  ...Reset SP, SR and PC registers...
15: end while

```

5 RIPencapsulation Implementation

We implement and evaluate RIPencapsulation on Texas Instruments MSP430FR5994 and MSP432P401R launchpads. Table 1 details the devices’ relevant specifications. Since the MSP430 and MSP432 are based on different instruction sets, there are some implementation differences in RIPencapsulation on the two microcontrollers; §6 covers these differences.

5.1 Capturing IPE Register State

RIPencapsulation’s interrupt-based side-channel procedure transfers the single-cycle register state dumps to the attacker’s machine. Algorithm 1 provides the details of RIPencapsulation’s side channel routine. MSP430FR5994 comes with two asynchronous general-purpose timer modules, each with four operating modes, and supports multiple captures/compares.⁵ The three operating modes besides Stop mode are – Up, Continuous, and Up/down; they have similar behavior: the timer counts to a value before overflowing and restarting the

⁵We use the timer module instead of the watchdog timer because it has finer-grain control over timer interval ranges and supports interrupt intervals as low as one clock cycle.

Source Operand	Destination Operand				
	Rm	PC	x(Rm)	TONI	&TONI
Rn	✓	✓	✓	×	×
@Rn	✓	✓	✓	×	×
@Rn+	✓	✓	✓	×	×
#N	-	-	-	×	×
x(Rn)	✓	✓	✓	×	×
EDE	×	×	×	×	×
&EDE	×	×	×	×	×

Table 2: MSP430 decoded instructions. EDE and TONI are generic labels and have no special meaning. Certain instructions cannot be guessed accurately as there is ambiguity when the source operand is an immediate value or label (-).

count. It generates interrupts when the timer counter overflows or reaches the `capture/compare` register value, depending on the operating mode.

After setting up the timer parameters, we start the timer and enable interrupts. Then we branch to the IPE code, where the timer interrupts it, invoking the interrupt service routine (ISR). Inside the ISR, we copy the latent IPE state from the register file and sent it to the workstation via the UART. Next, the ISR restores PC, SP, and SR to the attacker’s state machine code, where we reset the timer parameters and restart the counter with the incremented `TIMER_COUNT` value so that the device goes one cycle deeper into the IPE code.

5.2 Reverse Engineering and Heuristics

MSP430 consist of the 27 implemented instructions of the MSP430 CPU [46]. Besides this, some MSP430 devices also support additional `CPUX` instructions that are used when a greater than 16-bit address space is required. The instructions are categorized into three types based on the operand sizes: byte, word, and 20-bit. The MSP430 instructions support seven different addressing modes. Table 2 summarizes the various source and destination addressing modes that the MSP430 instruction set architecture supports and highlights the instructions RIPencapsulation is able to guess correctly. RIPencapsulation successfully decodes the instructions that use register/indexed/indirect register mode source and destination operands. It can also decipher the indirect autoincrement mode instructions, where the source operand is `@Rn+`. In contrast to the indirect register mode instructions, these instructions increment the source register value by 1, 2, or 4 depending on whether the operand size is byte, word, or 20-bit respectively. It is important to note that these four types of instructions constitute the bulk of the assembly in our benchmarks. We are unable to decipher instructions with symbolic or absolute addressing mode source or destination operands

as they are IP encapsulated and cannot be directly modified to attacker-controlled non-IPE address locations for disambiguation. Also, we do not consider guesses for instructions with immediate mode source operand and register/indexed/indirect register mode destination operand in our evaluation of decoded assembly instructions, as it is ambiguous whether the source operand is an immediate value or a value stored in some IPE symbol or absolute address.

Algorithm 2 Sorting algorithm for register mode instructions

```

1:  $i \leftarrow 0$ 
2: while  $i \neq TOTAL\_DUMPS - 1$  do
3:   if  $PC[i + 1] - PC[i] \neq 0$  then
4:     if  $Rm[i + 1] - Rm[i] \neq 0$  then
5:        $R_{dest} \leftarrow Rm$ 
6:     end if
7:     if  $R_{dest}[i + 1] \leftarrow f(R_{dest}[i], Rn[i])$  then
8:        $R_{source} \leftarrow Rn$ 
9:        $OPERATION = f$ 
10:    end if
11:  end if
12:   $i \leftarrow i + 1$ 
13: end while

```

Algorithm 2 details the sorting algorithm for register mode instructions where the source and destination operands are register contents. For indexed or indirect register mode instructions, the source or destination operand contains a register value used as a pointer to the actual operand value. Algorithm 3 highlights the procedure for identifying these instructions. Before branching to the suspected instruction, the attacker sets all the general-purpose registers (GPRs) to controlled non-IPE memory addresses. Based on the addressing mode, the suspected instruction will modify either the corresponding destination register or the memory address value it points to. Observing which register or memory address value changes, we guess the destination operand and its addressing mode (register/indexed/indirect). Analyzing the new value at the destination location and comparing it with the older GPR values, we are able to figure out the source operand, its addressing mode, and the operation performed. For instance, a "MOV @R10, R15" loads the value at the memory address pointed by R10 to R15. For indexed instructions, we perform multiple tests with different memory address inputs to get a complete resolution, as we also need to identify the offset to the address value stored in the register operand. Besides this, we also perform multiple tests to disambiguate unintended clashes between operations. For instance, XOR between `0xFE` and `0x00` produces the same output as MOV with `0xFE`.

Single operand instructions such as `PUSH`, `POP`, `CALL`, and `RETI` modify the stack and have unique effects on the stack pointer, giving away their identity. We also perform heuristic analysis on the obtained register dumps to clarify the victim program assembly state better. Figure 3 illustrates the register

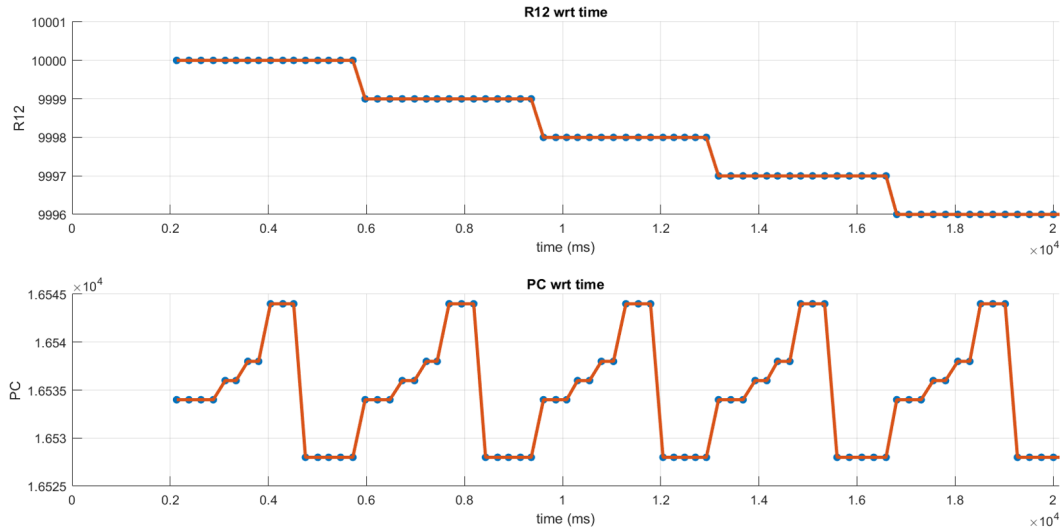


Figure 3: Register state plot of a decrementing loop-counter showing the change of R12 register with the PC.

Algorithm 3 Routine for identifying indexed and indirect mode instructions

```

1: ...Set timer ISR routine...
2:  $i \leftarrow 0$ 
3:  $TIMER\_COUNT \leftarrow CYCLES\_REQUIRED$ 
4:  $ADDR \leftarrow SUSPECTED\_INSTRUCTION$ 
5: while  $i \neq NUMER\_OF\_TESTS$  do
6:   ...Set timer and UART parameters...
7:   ...Start timer counter...
8:   ...Enable interrupts...
9:   ...Set all GPRs to controlled memory addresses...
10:   $JUMP\ ADDR$ 
11:  // Secure process gets interrupted
12:  ...Copy all core registers to unprotected memory...
13:  ...Copy saved state to UART TX Buffer...
14:   $i \leftarrow i + 1$ 
15:  ...Reset SP, SR and PC registers...
16: end while

```

state plot of a decrementing loop-counter program. Here the R12 register holds the counter value decremented at the end of each loop. The return of the program counter to the lower memory address signifies JUMPing back to the start of the loop. We interpret PC value discontinuities which do not modify the SP as JUMP instructions.

5.3 IPE Memory Exfiltration

Instructions which use indexed/indirect register mode source operand are ideal for exfiltrating the IPE memory. However, any indexed/indirect register value also works as we modify the memory address pointed to by that register

Algorithm 4 MSP430 IP exfiltration routine using IOP gadget

```

1: ...Set timer ISR routine...
2:  $i \leftarrow 0$ 
3:  $TIMER\_COUNT \leftarrow CYCLES\_REQUIRED$ 
4:  $ADDR \leftarrow IPE\_START - OPERAND\_SIZE$ 
5: while  $i \neq DESIRED\_DUMPS$  do
6:   ...Set timer and UART parameters...
7:   ...Start timer counter...
8:   ...Enable interrupts...
9:    $ADDR \leftarrow ADDR + OPERAND\_SIZE$ 
10:   $R_{source} \leftarrow ADDR$ 
11:   $JUMP\ IOP\_Gadget$ 
12:  // Secure process gets interrupted
13:  ...Copy  $R_{dest}$  value to unprotected memory...
14:  ...Copy saved state to UART TX Buffer...
15:   $i \leftarrow i + 1$ 
16:  ...Reset SP, SR and PC registers...
17: end while

```

to some attacker-controlled memory location. After the attacker identifies an IOP gadget, they inject malicious code that writes the IPE segment base address to the source register of the victim instruction and branches to it. Following the timer interrupt, the ISR transmits the saved destination operand containing the IPE memory value over the UART. The malicious code then increments the victim address before the ISR transmits the next destination operand value to the attacker's machine. Algorithm 4 details RIPencapsulation's MSP430 IP exfiltration routine using the IOP gadget.

6 RIPencapsulation on MSP432

We also test the efficacy of RIPencapsulation on MSP432 with the Cortex-M4 Flash-based microcontroller MSP432P401R. This device has a different ISA, different system-on-chip, and different non-volatile memory, so by showing that RIPencapsulation works for the MSP432, we show that the flaws in TI's IPE are fundamental.

6.1 CPU Halts Inside IPE Zone

MSP432 IP Encapsulation has some key differences in its implementation, besides the fact that it is only configurable using the bootloader (due to Flash vs. FRAM differences). MSP432 IPE is more secure in that it disables all kinds of CPU halts inside the IPE zone. The SYSCTL security control monitors all debug accesses from the DAP (debug) port inside the IPE zone. In fact, SYSCTL does not permit any CPU halts inside the IPE zone, including breakpoint addresses pointing inside the IPE memory (unlike MSP430 IPE). Even with this precaution, we find that we are able to trigger timer interrupts inside the IPE zone. Thus, like for the MSP430, we use timer interrupts to build our attack, however, there are some differences in the interrupt handler routine.

6.2 Return from Interrupt Handler

The setup procedure for timer interrupts is the same as on the MSP430. However, the working of interrupts is slightly different. On the MSP432, all interrupts and exceptions are handled by the Nested Vectored Interrupt Controller (NVIC). When the NVIC detects that an interrupt signal is HIGH, it changes the state of the interrupt to pending. Interrupts remain pending until the processor enters the interrupt service routine (ISR), upon which the NVIC changes the interrupt status to active. When the interrupt is serviced by the ISR, the processor loads an EXC_RETURN value stored in the link register (LR) to the program counter (PC). In our test code using timer interrupts, an EXC_RETURN value of 0xFFFFF9E9 is loaded into PC, which returns to the original thread using the state from the main stack pointer (MSP). On return from the ISR, the NVIC changes the interrupt status from active to inactive. As such, when we try to replicate RIPencapsulation on the MSP432 by jumping to the malicious code, like we do on the MSP430, the interrupt status remains active, and the NVIC does not allow re-entry into the ISR even when the timer interrupt becomes pending again. To resolve this, we exit the ISR using the EXC_RETURN value. So, instead of jumping directly to the malicious code, we write the malicious return state on the MSP. This way, when the EXC_RETURN value is loaded into the PC, the NVIC changes the interrupt status to inactive and the processor returns to the malicious code instead of the original thread, where we reset the timer interrupts and

Algorithm 5 MSP432 IP exfiltration routine using IOP gadget

```
1: ...Set timer ISR routine...
2:  $i \leftarrow 0$ 
3:  $TIMER\_COUNT \leftarrow CYCLES\_REQUIRED$ 
4:  $ADDR \leftarrow IPE\_START - OPERAND\_SIZE$ 
5: while  $i \neq DESIRED\_DUMPS$  do
6:   ...Set timer and UART parameters...
7:   ...Start timer counter...
8:   ...Enable interrupts...
9:    $R_{source\_STORE} \leftarrow 0X695A$ 
10:   $R_{dest\_STORE} \leftarrow SYS\_SECDATA\_UNLOCK$ 
11:   $R_{dest\_BRANCH} \leftarrow LOAD\_INSTRUCTION\_ADDR$ 
12:   $ADDR \leftarrow ADDR + OPERAND\_SIZE$ 
13:   $R_{source\_LOAD} \leftarrow ADDR$ 
14:  JUMP IOP_Gadget
15:  // Secure process gets interrupted
16:  ...Stop timer and clear interrupt flags...
17:  ...Copy  $R_{dest\_LOAD}$  value to unprotected memory...
18:  ...Copy saved state to UART TX Buffer...
19:   $i \leftarrow i + 1$ 
20:  ...Write malicious return address to MSP...
21: end while
```

increment the timer counter for single-step execution of the victim IPE code.

6.3 Clearing Interrupt Flags

Even though the program control returns to the malicious code upon completion of the ISR, as we desire, we observe that the program immediately re-enters the ISR, not executing any victim code. We discover that this is happening because the interrupt signal stays asserted even after completion of the ISR. The NVIC detects the asserted signal and immediately changes the interrupt status to pending, leading to re-entry inside the ISR. Clearing the interrupt flags de-asserts the interrupt signal and the NVIC waits for the next timer interrupt upon completion of the ISR.

6.4 Unlocking Read Access

Exfiltrating the data from the MSP432 IPE region is not so straightforward. The attacker must unlock data access each time the control goes inside the IPE zone. If we try to jump to an arbitrary load instruction inside the IPE zone without unlocking the data access first, the processor throws an exception. Writing 0x695A to the memory-mapped SYS_SECADATA_UNLOCK register unlocks the data access for the IPE region writing to that register (since MSP432 allows the creation of up to four isolated IPE regions). To bypass this, we need to construct a more sophisticated read IOP gadget. Algorithm 5 details RIPencapsulation's MSP432 IP exfiltration routine using the IOP gadget. In our implementation, we

Optimization Level	Instructions decoded	Reveal key bits?	Contains IOP Gadget?
-O0	60.5%	✓	✓
-Og	66.6%	✓	✓
-O1	68.4%	✓	✓
-O2	68.4%	✓	✓
-O3	68.4%	✓	✓
-Os	67.9%	✓	✓
-Ofast	68.4%	✓	✓

Table 3: Test cases with different compiler optimizations for AES (`tiny AES`). The second column depicts the percentage of assembly instructions we are able to reverse engineer after Phase 2.

jump inside the IPE zone to a store instruction to write the unlock value to the memory-mapped `SYS_SECDATA_UNLOCK` register followed by an indirect register branch instruction to branch to the actual load instruction which we then use to exfiltrate the IPE firmware. The attacker is capable of leveraging other sophisticated IOP gadgets to perform this read exploit.

7 Evaluation

We evaluate RIPencapsulation against four commodity cryptographic algorithms, namely AES (`tiny AES`), SHA256 (`saddi`), SHA256 (`gladman`), and RSA (`codebase`). We select these specific benchmarks in order to evaluate RIPencapsulation across different forms of cryptographic algorithms like symmetric-key and public-key cryptography and cryptographic hashing. We compile all benchmarks using the open-source MSP430 GCC toolchain developed by Texas Instruments [47], testing them against a range of optimization levels. Our evaluation answers the following questions:

- Are cryptographic implementations generally susceptible to the RIPencapsulation attack?
- What effect does the compiler optimization level have on the vulnerability of these cryptographic implementations?
- How long does it take to carry out the RIPencapsulation attack?
- Is IPE vulnerable across device types?

7.1 AES

Our static analysis results show that the AES implementation leaks key bits to the registers in its `KeyExpansion` function. At optimization levels `-O1` and above, the `KeyExpansion` function is embedded inside the `AES_init_ctx` function. All the instructions leaking the key bits are of the form `MOV.B @Rn, Rm`. The destination registers in these instructions leak

Optimization Level	Instructions decoded	Reveal 'pt' bits?	Contains IOP Gadget?
-O0	59.4%	✓	✓
-Og	56.7%	✓	✓
-O1	53.9%	✓	✓
-O2	49.7%	✓	✓
-O3	50.1%	✓	✓
-Os	53%	✓	✓
-Ofast	50.1%	✓	✓

Table 4: Test cases with different compiler optimizations for SHA256 (`saddi`). The second column depicts the percentage of assembly instructions we are able to reverse engineer after Phase 2.

the last four bytes of the secret key and all the remaining round keys. Reverse engineering the AES secret key using the round keys is a deterministic process, and literature exists describing the same [19]. We take the second-round key and the last four bytes of the secret key to obtain the original 128-bit key using the following formula:

If $i\%4 \neq 0$,

$$\omega_{i-4} = \omega_i \oplus (\omega_{i-1})$$

Else,

$$\omega_{i-4} = \omega_i \oplus \text{sbox}(\text{shift}(\omega_{i-1})),$$

followed by XOR of ω_{i-4} 's 1st byte with $Rcon[j]$

Here ω_i is the i th word in the complete AES key. $\text{sbox}(x)$ represents the byte substitution using the S-Box lookup table, $\text{shift}(x)$ is a cyclical shift of the bytes of ω_i , and $Rcon[j]$ is the round constant for round j , whose key we reverse engineer here. This would be the first round in our case. It is also worth mentioning that the `-O0` optimization level assembly reveals the secret key location in the stack. Assemblies produced at all optimization levels contain an instruction of the form `MOV.B @Rn, Rm`. We use them as IOP gadgets to exfiltrate the AES IPE code and data. Table 3 summarizes the results for our AES-128 implementation.

7.2 SHA256

Table 4 highlights the results of the RIPencapsulation attack on `saddi SHA256` implementation. Not only do all the assemblies leak the plaintext location to the registers, but also all the plaintext bits. The plaintext bits are leaked in the `SHA256Guts` function, which is embedded inside the `sha256_update` function at optimization levels `-O1` and above. The instructions containing the leaking registers are word instructions, and

Optimization Level	Instructions decoded	Reveal 'pt' bits?	Contains IOP Gadget?
-O0	59.2%	✓	✓
-Og	60.9%	✓	✓
-O1	58.3%	✓	✓
-O2	60.4%	✓	✓
-O3	60.7%	✓	✓
-Os	57.7%	✓	✓
-Ofast	60.7%	✓	✓

Table 5: Test cases with different compiler optimizations for SHA256 (gladman). The second column depicts the percentage of assembly instructions we are able to reverse engineer after Phase 2.

Optimization Level	Instructions decoded	Reveal key bits?	Contains IOP Gadget?
-O0	58.3%	✓	✓
-Og	58.4%	✓	✓
-O1	57.3%	✓	✓
-O2	54.3%	✓	✓
-O3	54.3%	✓	✓
-Os	60.8%	✓*	✓
-Ofast	54.3%	✓	✓

Table 6: Test cases with different compiler optimizations for RSA (codebase). The second column depicts the percentage of assembly instructions we are able to reverse engineer after Phase 2. *Readout from the stack is required for some bits of the private key.

the plaintext bytes are present in reverse order within each word as the memory model that MSP430FR5994 uses is little endian. All our SHA256 assemblies contain instructions of the form `MOV.W x(Rn), Rm`. We use them as IOP gadgets to exfiltrate our SHA256 IPE code and data.

We also evaluate RIPencapsulation on the gladman SHA256 implementation. Table 5 summarizes its evaluation results. All assemblies of this SHA256 implementation leak the plaintext bits to the registers inside the `sha_end1` function. The compiled code also contains a prologue instruction inside the calling function for SHA256, which performs a SUBA operation on the stack pointer. We bypass this instruction and, as such, include it in our calling routine for the IP-encapsulated SHA256 function. Assemblies produced at all optimization levels contain IOP gadgets that look like `MOV.W @Rn, Rm`.

7.3 RSA

The codebase RSA implementation is the smallest code of all the benchmarks evaluated in this paper. Table 6 summa-

rizes the evaluation results for our RSA implementation. Assemblies produced at optimization levels -O0, -Og, and -O1 store the private key location in the stack and leak the private key bits to the registers inside the `modexp` function, which is called by the `rsaDecrypt` function. Level -O2, -O3, and -Ofast leak both the private key location and the key itself to the general-purpose registers inside the `rsaDecrypt` function. The -Os assembly stores the private key location in the stack and only partially leaks the private key bits to the registers inside the `modexp` function. The assembly directly leaks the first two words to the registers but accesses the next two words in the `BIS.W x(SP), R12` instruction, which performs an OR between the two operands. Since OR is lossy, we cannot reverse-engineer the operands from the result of the operation. So, we need to read out the last two words of the private key from the stack. The -Os optimization assembly contains an IOP gadget of the form `MOV.W @SP, R12`, that is used to exfiltrate the private key bits from the stack. The -O0 level produces assembly with a prologue consisting of a PUSH operation on the stack followed by a SUBA operation on the stack pointer. All other assemblies of RSA contain only the SUBA operation on the stack pointer in the calling function.

7.4 Attacking Real Firmware

-Opt	O0	Og	O1	O2	O3	Os	Ofast
IOP Gadget?	✓	✓	✓	✓	✓	✓	✓

Table 7: Test cases with different compiler optimizations for a flight controller GPS/GPRS tracker.

While cryptographic algorithms are a natural target in TEEs, we demonstrate the generalizability of our exploit on real-world firmware. To this end, we use an existing GPS/GPRS tracker application from a drone flight controller [6]. We verify that this application is vulnerable to RIPencapsulation attacks as it contains several indirect load and store instructions which serve as our IOP gadgets. Table 7 shows that assemblies produced at all optimization levels contain IOP gadgets. Indirect load instructions serve as read IOP gadgets which enable the exfiltration of all the IPE protected code and data. We also verify the presence of indirect store instructions in the IPE code which serve as write IOP gadgets. Such write IOP gadgets give the attacker the ability to plant bugs in the GPS/GPRS tracker code.

7.5 Attack Time

IPE exfiltration rate is characterized by the delay in transmitting the data packets to the external device over the UART communication channel. At a baud rate of 115200 and clock

frequency of 1 MHz, it takes 10 seconds on average to obtain the register states leaking the second round key of the tiny AES encryption process. `saddi` SHA256 directly leaks the plaintext bits to the register states in under 75 seconds for the `-O0` optimization level, whereas `-Ofast` optimized code leaks the plaintext in 23 seconds. Although the entire `gladman` SHA256 register state exfiltration process takes the longest time due to its large code size (2x in size compared to `saddi` SHA256 and 4x compared to `codebase` RSA), it leaks the plaintext bits to the registers in roughly the same amount of time as `saddi`, ranging from 30 to 70 seconds, with the `-O0` level assembly being the most time-consuming. `Codebase` RSA is the simplest benchmark of all and leaks the private key to the registers within 5 seconds. However, since `-Os` optimized RSA code requires exfiltration of some key bits from the stack, `Phase 3` is necessary and the attacker must use an IOP gadget for exfiltrating the private key. In general, we are able to receive 10,000 dumps over the UART channel in under 2 minutes. The time taken to execute `Phase 2` of the attack depends on the attacker's approach to analyzing the exfiltrated register state. To accelerate `Phase 2`, we automate the reverse engineering process on our workstation.

7.6 MSP432 Results

We were able to replicate all of the attacks from the MSP430 on the MSP432. Even though the MSP432 is a newer ISA with a more security-oriented system-on-chip, the RIPencapsulation is easier on the MSP432 due to its Thumb architecture set, which predominantly uses register mode operations. Thus, the reverse engineering capability of RIPencapsulation on MSP432 increases to 80% without `Phase 3`.

7.7 Write Exploit

Besides leaking the IPE memory to the outside world, specific IOP gadgets are also able to write to the IPE region, giving the attacker the power to modify the IPE code and possibly get access control over it that way. This breaks the integrity and authenticity guarantees of code and data inside the IPE region. Register direct/indirect addressing mode instructions in the IPE code with register indirect destination addressing are ideal candidates for such an exploit as they allow deploying a payload to any desired IPE memory location. For the MSP430 instruction set these instructions look like `MOV Rn, x(Rm)`, `MOV @Rn, x(Rm)`, `MOV x(Rn), x(Rm)`, `MOV Rn, x(SP)` and many more variations of the same. We find that all our MSP430 benchmarks contain one of these instructions across all compiler optimizations. Meanwhile, the MSP432 uses Flash for its non-volatile memory hence we need the Flash Controller for writes. However, MSP432 IPE only allows storing code and constants in its IPE region and the Flash Controller is disabled inside the IPE zone.

8 Mitigation Strategies

We find two fundamental shortcomings in Texas Instruments's (TI's) IP Encapsulation (IPE) design, which make it vulnerable to RIPencapsulation. First, it does not clear the IPE state on context switches to non-IPE execution. This enables the attacker to invoke an Interrupt Service Routine (ISR) outside the IPE zone but retain access to the latent state of IPE execution through the register file. Secondly, IPE has no call site verification. This allows the attacker to jump anywhere inside the IPE zone, opening the door to control- and data-oriented attacks. Even though TI recommends the IP author to disable interrupts from non-IPE code as a secure coding practice, we show that this is futile as attackers can bypass any protections in the IPE zone by jumping to the instruction right after. Thus, a defense must address both the fundamental IPE flaws.

"The privacy of register and on-chip caches should be protected by the trusted computing base from software attacks" [42]. In the AEGIS Architecture [42], a secure context manager (SCM) stores all the process's register values in the SCM table on interrupt and clears the register states before invoking the ISR so that the ISR cannot access the internal state of the secure process. The SCM restores the register states from the SCM table on return from the ISR. Clercq et al. [20] provide a hybrid implementation of AEGIS for MSP430 devices that improves SANCUS [34]. When extended to include clearing the residual state present in the shared memory region of SRAM (e.g., the stack), AEGIS-based system should address the problem of latent IPE state upon unexpected IPE exits—at the cost of hardware modification.

The lack of valid call site verification allows the attacker to orchestrate data-oriented control flow attacks. Almost 90% of exploit-based software attacks use some form of Return-Oriented Programming (ROP) [27]. Address Space Layout Randomization (ASLR) is a well-known class of code security techniques [22, 26, 28, 52, 53] that randomize the memory address of a program's sections in order to reduce the chances of code reuse exploits that rely on knowing the exact location of process objects. Not only does ASLR entail a heavy overhead, but it also supposes a higher level of ability to intrude into the code and introspect its insecure uses. The facilities available on MSP430 and MSP432 devices do not lend themselves to an efficient implementation of such a high overhead approach.

A better defense for control- and data-oriented attacks is a call gateway veneer. In the embedded space, ARM TrustZone enforces this gateway veneer by introducing a Non-Secure Callable (NSC) memory region. All calls from a normal program to a secure function must go through the gateway veneer residing in the NSC memory. Calls to invalid entry points inside the secure code cause a hardware exception which always traps into a secure state. Fault injection [44] and short-term data remanence [32] attacks break the security guarantees of ARM TrustZone by pausing the trusted execution in a

controlled manner to reveal its internal state.

We believe that ARM TrustZone, combined with fault attack protection, in conjunction with AEGIS-style secure context-switching represents the best solution to defend IoT-class devices against RIPencapsulation. Given the many trade-offs at play for ultra-constrained devices like MSPs, we believe that the design, implementation, and evaluation of such a defense is important future work that this paper motivates.

9 Related Work

Trusted Execution Environments (TEEs) are process isolation and secure storage solutions that are finding their way into IoT-class embedded devices. Recent work indicates a rise in the trend of secure process preemption or exception-based exploits, which infer the program's internal state by studying their effects on the unprotected areas of the compromised device. In light of this trend and given RIPencapsulation is a TEE attack, we cover attacks against other TEEs on devices ranging from desktop-class to embedded devices.

9.1 Interrupt-based Attacks

SGX-Step [49] is an interrupt-based side-channel attack on Intel Software Guard eXtensions (SGX), which builds on previous kernel-level SGX exploits that preempt the enclave execution to leak information from page tables (PTE) [51, 54] or branch prediction units [29]. SGX-Step exploits the Advanced Programmable Interrupt Controller (APIC) timer to interrupt the secure process at several-instruction granularity in order to gain fine-grained control of the side channels, improving the temporal resolution of previous enclave preemption attacks [24, 29, 51, 54]. Nemesis [50] extends the SGX-Step by using an interrupt-based side channel to leak instruction-level information from TEE execution. Nemesis requires precisely timed interrupts to capture per-instruction latency differences.

RIPencapsulation extends the idea of SGX-Step and Nemesis to IoT class devices. RIPencapsulation provides single-cycle granularity and uncovers 100% of TEE-protected memory by combining interrupt-based attacks with data-oriented attack patterns. RIPencapsulation also demonstrates an exploit to modify the TEE-protected code and data for its target device class. Note that, the compiler-based defenses that reduce the effectiveness of SGX-Step, apply to heavyweight TEEs and rely on detecting high rates of page faults or interrupts by leveraging the x86 Transitional Synchronization eXtensions (TSX) and as such do not apply to the IoT-class devices targeted by RIPencapsulation [17, 41].

The Page-Fault Weird Machine [10] has a similar high-level idea to Interrupt-Oriented Programming (IOP) for using interrupted execution with pre-existing gadgets in the device to get useful computation. However, RIPencapsulation's IOP differs in the context of gadget aims, target device class, and security

violation. We use gadgets on the firmware level (instruction set architecture) as opposed to microarchitecture gadgets. Our target class devices are bare-metal resource-constrained microcontrollers as opposed to x86 full-fledged desktop class devices (for example our device class does not have virtual memory). Finally, we use our gadgets to create read/write exploits and break the security of TEEs on our target devices.

Interrupt-oriented Bugdoor Programming (IOBP) [43] is conceptually the same as RIPencapsulation's IOP. The difference lies in the adversarial setting. IOBP exploits require some a priori knowledge to find useful IOBP gadgets in read/write access blocked microcontrollers. We too consider our protected firmware (IPE protected library) to be read/write access blocked, however a part of the memory is unlocked to read/write accesses. This is a realistic threat model for IP theft/imitation in cases such as flight controllers, where a competitor (potential adversary) uses the licensed security critical library (flight controller) APIs in their development code. When IOP is used in such a context, a) we do not require a priori knowledge of the IPE firmware, as we can use IOP gadgets to dump CPU states and reverse engineer partial information about the underlying code, enough to detect read/write IOP gadgets and b) find more usefully exploitable IOP gadgets for protected state extraction/modification.

9.2 Debugger-based Attacks

Shedding too much Light on a Microcontroller's Firmware Protection [36] analyzes the security of protected Flash memory on STM32 microcontrollers. The paper uses fine-grain CPU resets and a vulnerability in the Flash protection logic protocol to extract the entire firmware by accessing iterative addresses of the firmware via the debugger and capturing latent Flash data in unprotected SRAM. While this attack requires optical fault injection and chemical etching to manipulate particular Flash bits, RIPencapsulation can leverage their controlled CPU resets in the event that timer interrupts are not available.

Brosch [14] presents a firmware dumping technique for an ARM Cortex-M0 SoC that uses the debugger to manipulate the CPU register values at single-step intervals. By single-stepping through the program code and observing the CPU register changes, they find load instructions to exfiltrate the protected memory. Their attack is based on the debugger's single-stepping capability inside protected memory, which is prevented by TI's IPE implementation. RIPencapsulation side-steps TI's protection through a combination of interrupts and data-oriented attacks, achieving single-step IPE execution. Additionally, RIPencapsulation presents a write exploit, breaking the integrity and authenticity of firmware inside the protected memory.

9.3 Blind Attacks

Code reuse attacks require varying degrees of information on the target. Blind attacks aim to create exploits when neither the source nor binary code is available. Half-Blind attacks [23] presents a stack overflow/ROP gadget attack to gain privileged access in the bootstrap loader (BSL) of an MSP430 device, which can then be used to extract the firmware image. Hacking Blind [11] is a more generic and advanced version of half-blind attack which is fully blind and presents techniques to find and chain multiple, different gadgets. While these attacks do not work with TI's IPE protection, they serve as inspiration for attacking unknown binaries with RIPencapsulation.

9.4 Other attacks on TEEs

CipherLeaks [31] is a ciphertext side-channel attack on AMD's Secure Encrypted Virtualization (SEV) TEE. SEV protects Virtual Machines (VM) from an untrusted hypervisor by using hardware-enforced memory encryption. On domain switch between the guest and host VM, SEV stores the encrypted register states in the Virtual Machine Save Area (VMSA). The CipherLeaks attack model assumes that the attacker has read access to the VMSA but no write access. CipherLeaks uses the hypervisor to monitor specific offsets of the VMSA to infer changes of any 16-bit plaintext. Non-Automatic VM Exits (NAE) expose some plaintext register values to the hypervisor. In essence, the attacker triggers an NAE to collect a dictionary of plaintext-ciphertext pairs for these registers stored in the VMSA. Cipherleaks then uses this plaintext-ciphertext dictionary to crack the entire OpenSSL RSA key in 410 rounds with 100% accuracy. In order to patch this vulnerability, AMD added randomization in stored register values when encrypting and saving them into the VMSA during VMEXITs [1]. This fix is available in the AMD SEV-SNP TEE. Unfortunately, encrypting the register file is not a complete defense against RIPencapsulation, as the ability to enter the IPE zone at arbitrary points allows the attack to create IPE zone altering and exfiltration gadgets.

CLKscrew [44] is an energy management-based exploit that manipulates the voltage and frequency of the processor to induce faults. Dynamic Voltage and Frequency Scaling (DVFS) is an energy management scheme, ubiquitous on commodity devices, that trades off processing speed for energy savings. CLKscrew is able to break the confidentiality and integrity of ARM TrustZone using software-only control of the regulators. In essence, the attacker increases the frequency of the processor beyond the limits dictated by the operating voltage to induce instability and halt the TrustZone process from the normal world. Performing Differential Fault Analysis on the correct and faulty decrypted plaintext pair, the attacker is able to infer the AES key. We envision an exploit that combines the Interrupt-Oriented Programming described

in this paper with CLKscrew to get very fine-grained control over fault injections inside the TEE.

During preparation of the camera-ready version of this paper, we discovered a paper citing a previous version of this paper that leverages our arbitrary read/write/execute access to IPE-protected memory to strengthen their attack on IPE [33]. While the practical variant of their attack relies heavily on RIPencapsulation, they provide their own unique capabilities worth noting. Their main contribution is an attack primitive that they refer to as *controlled call corruption* which exploits a microarchitecture bug in the IPE access control mechanism. If an adversary sets the `SP` to some protected memory address, followed by a `call` to the IPE zone, the return address is pushed to the stack pointer location, subverting the IPE access control check. Their defensive landscape analysis agrees with ours that a hybrid solution is required for a comprehensive defense.

10 Conclusion

Texas Instruments MSP IP Encapsulation (IPE) aims to provide confidentiality of data stored inside the IP-encapsulated memory zone; this includes proprietary code and keys. RIPencapsulation breaks this guarantee by leveraging two fundamental drawbacks in MSP430 IPE design: residual state on context switches and lack of call site verification. We exploit these flaws to create an interrupt-based side channel to gain cycle-accurate control IPE execution and exfiltrate all IPE secrets. The evaluation shows that this attack works using production tools and settings of popular open-source cryptographic implementations.

This paper shows that Trusted Execution Environment (TEE) designers must pay careful attention to unexpected TEE entries and exits. Without guarding the entries to IPE code, attackers can bypass defenses and create gadget-like instruction sequences. Without cleaning up residual state shared across security domains on every possible exit, attackers have access to secret-revealing side-channel information. These requirements extend beyond any single TEE implementation, serving as necessary conditions for ensuring code and data confidentiality by all TEEs.

Acknowledgments

The project depicted is sponsored by the Defense Advanced Research Projects Agency. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. Approved for public release; distribution is unlimited. This material is based upon work supported by the National Science Foundation under Grant No. 2240744.

References

- [1] AMD Secure Encryption Virtualization (SEV) Information Disclosure. <https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1013.html>. Accessed: 2023-4-14.
- [2] IoT market size, trends and global forecast to 2032. <https://www.thebusinessresearchcompany.com/report/iot-global-market-report>. Accessed: 2023-4-13.
- [3] MSP430 microcontrollers. <https://www.ti.com/microcontrollers-mcus-processors/msp430-microcontrollers/overview.html>. Accessed: 2023-4-13.
- [4] Embedded Survey: The Current State of Embedded Development. <https://www.embedded.com/embedded-survey/>, September 2016. Accessed: 2023-7-18.
- [5] 2019 Embedded Markets Study, 2019.
- [6] GPS/GPRS tracker. <https://github.com/wudidashao/tracker>, 2019.
- [7] Microcontroller (MCU) Market Analysis. <https://www.utmel.com/blog/categories/microcontrollers/microcontroller-mcu-market-analysis>, February 2022. Accessed: 2023-2-3.
- [8] RIPencapsulation Code. <https://github.com/ForTE-Research/RIPencapsulation>, 2023.
- [9] Adnan Akhuzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015.
- [10] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.
- [11] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.
- [12] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.
- [13] Erik Bosman and Herbert Bos. Framing signals—a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258. IEEE, 2014.
- [14] Kris Brosch. Firmware dumping technique for an ARM Cortex-M0 SoC. <https://blog.includesecurity.com/2015/11/>, November 2015. Accessed: 2023-7-27.
- [15] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.
- [16] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.
- [17] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017.
- [18] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.
- [19] William R Cordwell. Aes key recovery from round keys. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2008.
- [20] Ruan De Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. Secure interrupts on low-end microcontrollers. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 147–152. IEEE, 2014.
- [21] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: secure and minimal architecture for (establishing dynamic) root of trust. In *Ndss*, volume 12, pages 1–15, 2012.
- [22] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pages 67–72. IEEE, 1997.
- [23] Travis Goodspeed and Aurélien Francillon. Half-blind attacks: mask ROM bootloaders are dangerous. In *Proceedings of the 3rd USENIX conference on Offensive technologies. USENIX Association*, pages 6–6, 2009.

- [24] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference*, volume 17, 2017.
- [25] Hong Hu, Shweta Shinde, Sendriou Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.
- [26] Vivek Iyer, Amit Kanitkar, Partha Dasgupta, and Raghunathan Srinivasan. Preventing overflow attacks by memory randomization. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 339–347. IEEE, 2010.
- [27] Ayman Jarrous. Anti-ROP: A moving target defense. <https://securityintelligence.com/anti-rop-a-moving-target-defense/>, September 2016. Accessed: 2023-4-14.
- [28] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348. IEEE, 2006.
- [29] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium*, volume 19, pages 16–18, 2017.
- [30] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A systematic look at ciphertext side channels on amd sevsnp. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 337–351. IEEE, 2022.
- [31] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. Cipherleaks: Breaking constant-time cryptography on amd sev via the ciphertext side channel. In *USENIX Security Symposium*, pages 717–732, 2021.
- [32] Jubayer Mahmud and Matthew Hicks. Sram has no chill: exploiting power domain separation to steal on-chip secrets. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1043–1055, 2022.
- [33] Cas Magnus Marton Bognar, Frank Piessens, and Jo Van Bulck. Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers. In *33rd USENIX Security Symposium*, 2024.
- [34] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herreweghe, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, 2013.
- [35] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. {VRASED}: A verified {Hardware/Software}{Co-Design} for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, 2019.
- [36] Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller’s firmware protection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.
- [37] Katie Pier. *MSP Code Protection Features*. Texas Instruments, December 2015.
- [38] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.
- [39] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.
- [40] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.
- [41] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.
- [42] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003.
- [43] Samuel Junjie Tan, Sergey Bratus, and Travis Goodspeed. Interrupt-oriented bugdoor programming: A minimalist approach to bugdooring embedded systems firmware. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 116–125, 2014.

- [44] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Clkscrew: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, volume 2, pages 1057–1074, 2017.
- [45] Texas Instruments. *MSP432P4xx Family Technical Reference Manual*, December 2016. Rev. E.
- [46] Texas Instruments. *CPUX*, March 2018. Rev. F.
- [47] Texas Instruments. *MSP430 GCC Toolchain*, June 2020. Rev. F.
- [48] Texas Instruments. *MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User’s Guide*, April 2020. Rev. P.
- [49] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.
- [50] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195, 2018.
- [51] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*, pages 1041–1056. USENIX Association, 2017.
- [52] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*, volume 10, pages 3026877–3026906, 2016.
- [53] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, pages 260–269. IEEE, 2003.
- [54] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.



Reverse Engineering the Eufy Ecosystem: A Deep Dive into Security Vulnerabilities and Proprietary Protocols

Victor Goeman
victor.goeman@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Dairo de Ruck
dairo.deruck@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Tom Cordemans
tom.cordemans@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Jorn Lapon
jorn.lapon@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Vincent Naessens
vincent.naessens@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Abstract

The security of Internet-of-Things (IoT) is a growing concern, with IP cameras like those from Eufy promising robust security through military-grade encryption. While Eufy's claims are strong, independent verification of these claims is crucial to confirm the integrity and resilience of its systems against potential vulnerabilities and extend the lessons learned to the broader IoT landscape, ensuring practices keep pace with technological advancements.

We unveiled the inner workings and security measures in the Eufy ecosystem through reverse engineering, particularly focusing on its smart doorbell and Homebase, and evaluated the proprietary peer-to-peer protocol and encryption methods.

This paper offers a comprehensive analysis of the Eufy ecosystem, offering insights into the broader implications of IoT device security. Our investigation revealed critical vulnerabilities within the ecosystem, which were responsibly disclosed and confirmed by Eufy. The vulnerabilities could compromise end-user privacy by allowing unauthorized access to the end users' private network within seconds. A key tool in our research was `dAngr`, a symbolic debugger we developed to facilitate the reconstruction of encryption keys in intricate cross-architecture binaries, thus enabling a more efficient reverse engineering process.

The research revealed vulnerabilities in Eufy's ecosystem, leading to serious privacy and security concerns, and suggests effective countermeasures, stressing the need for continued vigilance in IoT device security.

1 Introduction

Smart doorbells have experienced a remarkable surge in sales, reaching a market value of USD 16.2 Billion in 2023, and are expected to grow at an annual rate of 33.4% from 2023 to 2030 [1]. To give an idea of their popularity, Amazon sold more than 400.000 smart doorbell devices and accessories during the pandemic [34]. Smart doorbells empower end users to monitor and interact with people at their doorstep remotely,

enhancing both convenience and physical security. However, this surge in popularity has brought concerns regarding security, safety and privacy to the forefront, forcing doorbell manufacturers to invest in bolstering their security measures.

Eufy [12], a rapidly emerging player, is a part of Anker Innovations. Anker is one of the leading electronics brands in America. Founded in 2016, Eufy is already among the ten most popular IP Camera brands in 2022 [23].

Eufy distinguishes itself with its emphasis on security, offering secure local storage (as it eliminates the need for cloud storage subscriptions), as well as promising military-grade encryption and end-to-end encryption [13].

In this work, we present an in-depth security analysis of the Eufy ecosystem which was studied for more than 9 months. Our research is based on the Eufy Homebase 2 in combination with the Eufy video doorbell 2K. However, our findings extend beyond those devices, affecting a whole array of Eufy devices (including its IP Cameras). Our security analysis includes several techniques including *network analysis*, *symbolic execution*, *static and dynamic analysis of the firmware and reverse engineering*. This combined effort of analysis methods enabled us to dissect the complete ecosystem. We exposed a series of critical vulnerabilities across various areas of the ecosystem, encompassing the peer-to-peer protocol, authentication, networking, encryption and the pairing process. These vulnerabilities pose a substantial threat to the ecosystem's confidentiality, integrity and availability.

We present a major attack on the Eufy Homebase, exploiting distinct vulnerabilities, and endangering the network and privacy of all end users using the Eufy ecosystem. The only requirement of the attack is being in proximity (i.e., up to miles away using specialized hardware) of an Eufy device and the attack takes no longer than 20 seconds. No network connection is required. The result of the attack is **unrestricted access** to the end user's home network. Prompt and thorough remediation of these vulnerabilities was of the utmost importance considering the gravity and scale of the attack.

Furthermore, we present `dAngr`, a debugger built upon the symbolic execution engine `angr` [40]. It simplifies and en-

hances the manual analysis of cross-architecture binaries, abstracting away the complexities of the symbolic execution engine. `dAngr` was used to reconstruct the media AES encryption keys allowing us to recover all video and media sent from the Homebase. Additionally, the ease with which the encryption keys for proprietary peer-to-peer communication can be recovered undermines the assertion of providing military-grade encryption.

To address these vulnerabilities and shortcomings, we propose countermeasures and best practices for each specific issue within the Eufy ecosystem. Following responsible disclosure, Eufy has acknowledged the identified vulnerabilities, and we have provided input to mitigate the various vulnerabilities.

Outline. In the remainder of this paper, we start an overview of the Eufy Ecosystem in Section 2. Section 3 presents the attacker model and methods used for the attacks performed in Section 5. Countermeasures are presented in Section 6 followed by Section 7 with general insights and recommendations. Related work is discussed in Section 8, and we conclude with Section 9.

2 The Eufy Ecosystem

The Eufy ecosystem encompasses several components. First, the smart devices, including the smart-doorbells, lights, cameras, vacuums, entry sensors and more, are connected through a closed and dedicated wireless Eufy network with the Eufy Homebase, a central component of the Eufy ecosystem. This core element of the ecosystem acts as a local hub for the management of connected smart devices and handles encryption, networking, firmware updates, and connects to the cloud through the end users' wired local network. The end user can interact with the Homebase using a mobile App which connects to the Homebase (either through the cloud or via the local network). Alternatively, the Eufy web application can connect to the Homebase through the cloud. The Eufy ecosystem studied in this work is depicted in Figure 1. During our study, we focus on two Eufy devices, namely the Homebase 2 and the video doorbell 2K. However, our findings go beyond these devices, affecting the complete Eufy ecosystem.

Before explaining the details of the various findings and vulnerabilities, we provide essential context for understanding and interpreting the subsequent findings and vulnerabilities related to the Eufy ecosystem.

2.1 Video Streaming and Communication

Commands and video streams are transferred at various points in the ecosystem. Commands consistently use a proprietary peer-to-peer protocol (P2P). Video streams use the P2P protocol or other protocols depending on the network location and application retrieving the stream.

2.1.1 Doorbell Communication

The doorbell solely communicates with the Homebase, and this communication occurs over a dedicated *hidden* Eufy Wi-Fi network. The traffic is secured at the data link layer using wireless communication security. Authentication to this network is established using WPA2-PSK, a pre-shared key of eight characters generated during the initial setup of the Eufy Homebase. This wireless network's SSID follows a pattern consisting of the string "*OCEAN_XXXXXX*" where *XXXXXX* represents the last 24 bits of the Homebase's MAC address.

Pairing mechanism. The pairing mechanism, as depicted in Figure 2, leverages soundwaves as an out-of-band channel to pass sensitive information to the doorbell. The Eufy App instructs the end user to bring the doorbell in close proximity to the Homebase. Next, the Homebase emits a soundwave carrying both SSID and WPA2-PSK of the dedicated Eufy network. The doorbell retrieves the information contained in the soundwave and connects to the Eufy wireless network. The Homebase and doorbell are subsequently connected via a wireless network protected with a pre-shared key. Afterwards, the Homebase and doorbell exchange their P2P connection information (serial number, licenses, etc.) through the Eufy network, and store this information in flash memory, completing device pairing. Finally, the Homebase and doorbell can exchange commands and pass the camera feed.

Streams and messages. Within this wireless network, various streams and messages are transmitted in the clear. P2P commands allow, for instance, to notify the end user when someone rings the doorbell, or to control the camera of the doorbell. We identified a UDP stream containing a continuous stream of JFIF video data and a smaller TCP stream containing JFIF image data. JFIF can be considered as a successor of the original JPEG format [16].

Once the doorbell and the Homebase are paired, both JFIF streams are persistent, even when the user is not watching the feed. Upon reaching the Homebase, the feed undergoes analysis by a motion detection and facial recognition module. Upon detecting movement, the Homebase promptly notifies the user and subsequently encrypts and stores the video and images locally. Optionally, the user may choose to backup the encrypted media in the cloud.

2.1.2 Communication with the End User

Eufy employs several methods to communicate with the end user depending on the location and application in use. While commands are always sent encrypted over the P2P protocol using a symmetric P2P AES key, the protocols used for transmitting video and images between the Homebase and the end user differ. Figure 3 illustrates the scenarios, which are discussed below in more detail.

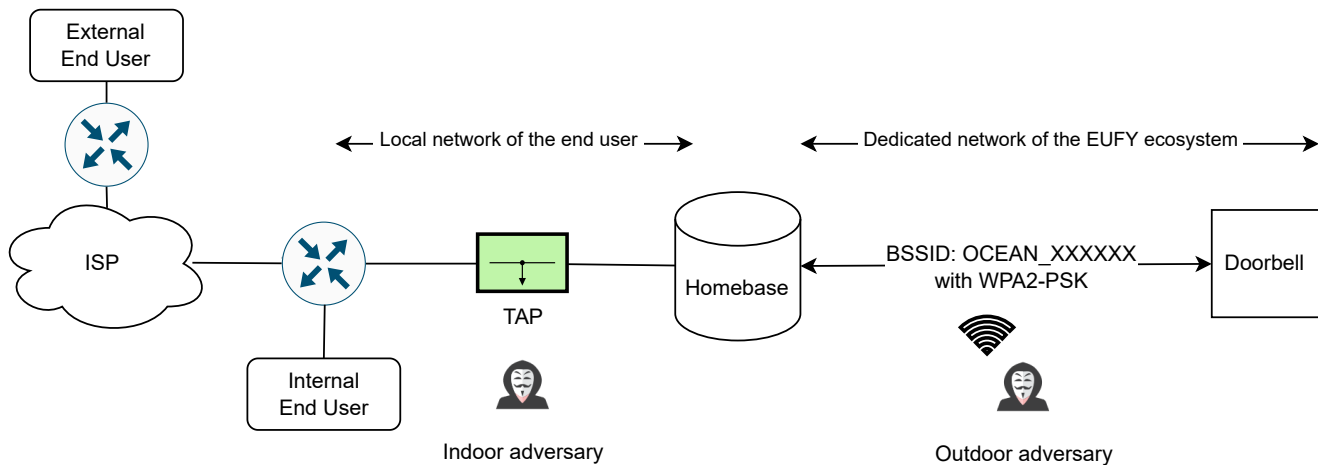


Figure 1: Eufy ecosystem

Mobile App. The first scenario, depicted at the bottom of Figure 3, occurs when the end user uses the mobile App to view the stream. The Homebase employs the proprietary peer-to-peer protocol (P2P) over UDP to communicate the media to the App. The JFIF stream is encrypted with a symmetric P2P AES key. When the end user uses the mobile App to view the stream while connected to the end user’s home network (i.e., not using the Internet), communication between both devices is direct. When, on the other hand, the user wants to connect remotely (i.e., a direct connection to the Homebase is infeasible), communication between both devices is relayed through a custom cloud server.

Web viewer. When the end user uses the web viewer (see at the top of Figure 3) independent of the user’s location, media is sent using Web Real-Time Communication (WebRTC) [2], an open-source web-based application technology. It is pri-

marily used for establishing real-time, peer-to-peer communication. It is always encrypted using vetted algorithms (e.g., DTLS, SRTP [7, 39]) and leverages standard protocols for NAT Traversal (ICE, STUN and TURN [18, 33, 38]) using another cloud server.

The more secure WebRTC is only used in the communication between the Homebase and the web application. The mobile App always relies on the proprietary P2P protocol to propagate the media stream, and as we will discuss in Section 5.4, these AES keys are insecure, endangering the confidentiality of the ecosystem.

2.2 Homebase Firmware

To understand the functioning of the Homebase, its firmware was thoroughly analyzed. The platform uses a MIPS architecture, running a custom Linux built with Buildroot [3]. Multiple binaries developed by Eufy are present on the firmware. The main binary called `home_security` handles all major functionalities of the Homebase. This binary has multiple instances running concurrently.

3 Attack Vectors & Methods

Our analysis involved various techniques to analyze the Eufy ecosystem. Studying the device from distinct angles allows for a comprehensive analysis of the ecosystem. Our research encompasses three methods of analysis: network analysis, firmware analysis and symbolic analysis. While we conducted an in-depth analysis of the smart doorbell and Homebase, the mobile App and web viewer were only used during network analysis.

Attacker model: In our analysis, we consider two types of adversaries, as illustrated in Figure 1:

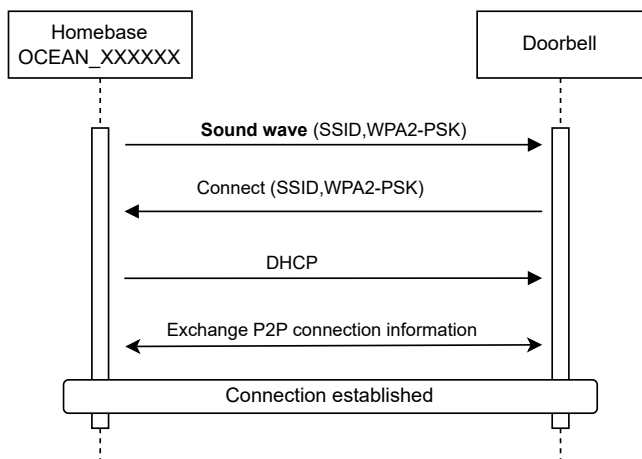


Figure 2: Pairing process

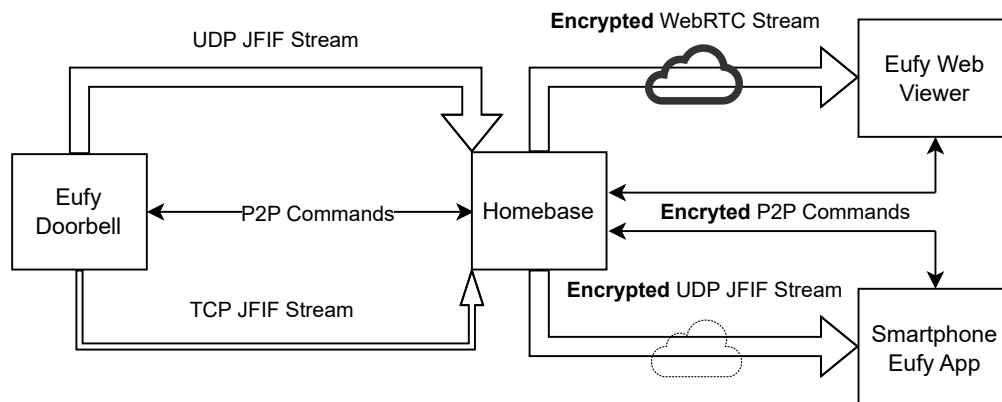


Figure 3: Video streaming in the Eufy ecosystem

1. The *outdoor adversary* operates within reach of the wireless Eufy network but is not connected to either the Eufy or the end user’s home network.
2. The *indoor adversary* has access to the home network, including the Homebase. It allows interception and manipulation of communication between the Homebase and other devices in the home network, including the Internet gateway.

Network analysis. The initial phase of our investigation involves the analysis of the network traffic within the Eufy ecosystem. The objective is to uncover the ecosystem’s functionality, employed protocols, interactions with external entities, and identification of cleartext communication. This process involves capturing and analyzing the communication between the Homebase and the home gateway along with testing man-in-the-middle attacks (MITM) from an indoor adversary’s perspective. From an outdoor adversary’s standpoint, the Eufy wireless communication is analyzed using a WiFi dongle.

Firmware analysis - reverse engineering. The process of firmware analysis involves disassembling the devices and scrutinizing debug ports and other hardware peripherals that may facilitate firmware extraction. Upon successful extraction, the firmware undergoes a series of tests, encompassing both automated and manual analysis. Initially, automated analysis of the firmware is conducted using EMBA, an open-source firmware analyzer [24].

Following the automated analysis, an exhaustive manual analysis is performed, selecting proprietary, custom-built binaries for in-depth inspection. To facilitate this reverse engineering step, we employ Ghidra, an open-source reverse engineering tool [11]. Leveraging the Ghidra decompiler to provide insights into the program logic, focusing on areas such as the proprietary peer-to-peer protocol, authentication

mechanisms, encryption and decryption processes, and cloud communications.

Selective execution. During our analysis of the key generation mechanism employed for encrypting media, challenges arose during the decompilation of the embedded MIPS binary. Ghidra faced difficulties in generating proper decompiled code for the more intricate functions. While other decompilers may have better support for these types of embedded binaries, we opted for a distinctive approach. We aimed to execute the embedded code to recreate the media keys. However, executing binaries of an embedded device presents its own set of challenges. In the upcoming Section, we introduce the use of dAngr to execute a specific function with chosen inputs and retrieve the media encryption key.

4 Selective and Platform Independent Execution with dAngr

Running a selected function in an embedded binary allows for several opportunities for testing and analysis. Potential benefits include analysis of the behaviour of the function under various conditions, testing for vulnerabilities, and automated testing procedures. In this Section, we discuss existing techniques’ benefits and issues and introduce a novel technique.

4.1 Existing Approaches

Selective execution of embedded functions. The aim of selective execution is to execute a function in a binary of an embedded device, leveraging known inputs to reconstruct, for instance, AES encryption keys. Several techniques exist, each with specific shortcomings. The most challenging approach would be to reconstruct a binary by extracting code using `objdump`. However, this approach is complex and may face difficulties, particularly when dealing with global static variables. A simpler approach involves using a debugger such

as `radare2` or `gdb` to execute the required function with the chosen inputs [31]. While with `radare2` this may still be complex, requiring the correct memory and register settings, calling a function with chosen arguments in `gdb` is straightforward.

Unfortunately, a major disadvantage of these solutions is platform dependency. Both binary reconstruction and debuggers require a matching platform and dependencies (e.g., `libc`) to execute the function.

Platform-dependent execution. Most solutions require either full or partial execution of the binary, necessitating a platform and libraries that match the binary. This requirement poses challenges, particularly for binaries from IoT devices that may use less common platforms. Executing, for instance, a MIPS binary on a standard platform is not straightforward.

To overcome the challenges some solutions propose to run a debugger on the physical device. However, access to such a device may not always be feasible. Another approach involves using a hardware platform with a matching architecture, but this requires a bootable binary that simulates all peripheral initialization and hardware communication, which can be complex and time-consuming. Likewise, emulating the hardware platform using, for instance, `QEMU`, in which either the binary or the full device may be emulated [29], suffers from this same issue.

An alternative approach is to perform binary lifting into an intermediate representation (IR) and either recompile to another platform or perform virtual execution (interpretation) of the IR code. However, both approaches suffer from issues with simulating actual devices.

Our approach. To address the shortcomings of the existing solutions, we combine both *Selective execution* and *Platform-agnostic execution* using `dAngr`, a debugger for `angr` (see Section 4.2). Platform independence is achieved through the execution of VEX IR. Furthermore, to overcome the configuration and hardware initialization challenges, we use selective execution to simulate only the necessary functionality to execute the selected functions.

Table 1 summarizes the main points of comparison between the traditional methods for selective and platform-independent execution of embedded functions versus our approach using `dAngr`. The `dAngr` approach combines the advantages of both supporting selective execution and being platform agnostic, offering a more streamlined and versatile solution for testing and analyzing embedded devices.

4.2 `dAngr`: a Debugger for `angr`

To support our research, we developed `dAngr`, a debugger built on top of `angr`. `angr` is a symbolic execution engine implemented in Python. `angr` handles the complexities of

binary lifting and interpretation, while the debugger interfaces (a command line, and JSON interface) simplify its use, requiring minimal knowledge about the underlying engine.

While for our attack we use `dAngr` for concrete execution (i.e., with concrete inputs instead of symbolic inputs), it also supports symbolic execution. The debugger contains common debugging commands such as adding, removing, enabling/disabling breakpoints, stepping and running. Note that since we use `angr` as an interpreter, stepping occurs per basic block instead of per instruction as in other debuggers. Similar to other debuggers, the `run` command performs execution until the next breakpoint or the end of the binary. However, in the case of symbolic execution, the debugger may stop if a forking state is reached, allowing the user to choose the branch to take.

Moreover, `dAngr` supports setting and retrieving registers or memory at specified addresses. Instead of starting the execution upon the binary's entry point, it is possible to relay the start of the execution to a selected address. Combining memory and registry control with starting execution at a chosen address enables selective execution of a function in the binary.

However, this approach still requires some in-depth knowledge of the platform e.g., to specify the function arguments using the correct registers and calling convention. To simplify executing functions and make our tool more accessible, we support three additional commands: `set_function_prototype`, to specify the function prototype, `set_function_call` to set the debugger to the function address and correctly set the arguments, and `get_return_value` to retrieve the return value with the type as specified in the prototype. Listing 1 shows an example of the commands to execute a function with specified arguments.

Listing 1: `dAngr` example commands for calling a function func with arguments

```
> set_function_prototype int func(char*, int)
> set_function_call func({"abc"}, 2)
> run
> get_return_value
```

In addition, testers can pass hooks to the debugger to replace or implement specific functions called inside the executed code. This feature was used to debug the binary and attest certain parameters.

This novel approach simplifies the execution process and aligns seamlessly with our goal of recovering the key generation algorithm, allowing the execution of a specific function with concrete inputs without the need for an exhaustive and complex setup.

`dAngr` is made open-source and available on GitHub ¹.

¹<https://github.com/angr-debugging/dAngr>

Table 1: Comparison of Execution Approaches

Attribute/Approach	Existing Techniques	dAngr Approach
Selective Execution	Requires binary reconstruction or debugger (e.g., radare2, gdb)	Enables execution of specific functions with chosen inputs
Platform Independence	Highly dependent, requiring matching platform and libraries	Utilizes VEX IR for platform independence
Complexity of Setup	Complex, may involve binary reconstruction or setting memory and registers	Simplifies simulation, avoiding deep knowledge of platform specifics
Simulation of Hardware/Peripheral Setup	Requires accurate simulation or emulation for execution	Only simulates the necessary functionality for executing selected functions
Suitability for Testing and Analysis	Limited by platform dependency and setup complexity	Enhanced by ease of executing specific functions and platform independence
Approach to Execution	Direct execution on hardware or through emulation/simulation	Virtual execution of intermediate representation (IR) code

5 Attacking and Identifying Vulnerabilities in the Ecosystem

After analyzing the Eufy doorbell and Homebase, we uncovered several flaws that undermine the security of the system. In the following, we introduce the steps taken during the analysis of the Eufy Homebase and smart doorbell.

5.1 Firmware Acquisition

To gain access to the Eufy ecosystem, our first step involves acquiring the firmware.

Homebase. For the Homebase, the firmware acquisition process commences with the disassembly of the device, revealing UART debug ports. Connecting a USB-to-TTL reader to these ports, we discovered a password-protected UART shell. However, during the boot process, a temporary recovery shell can be accessed without authentication. Within this recovery shell, we analyzed the filesystem, unveiling the password of the root user. This password coincided with the WPA2-PSK securing the dedicated Eufy network, highlighting a vulnerability in *password reuse*. These particular flaws had been previously identified by other researchers conducting security analysis on the Eufy ecosystem [5]. By using this password, we gain entry to the password-protected UART shell, thereby accessing the system. Consequently, the firmware is obtained through a firmware dump in this shell.

Doorbell. Unlike the Homebase, the doorbell lacks obvious UART ports. However, upon carefully analyzing the hardware of the doorbell, SPI NOR flash is detected. Reading this NOR flash chip is accomplished using an SPI reader. Specifically,

we used a CH314a flash programmer to acquire the firmware of the doorbell [22].

While the doorbell primarily handles only essential functions, such as capturing and sending media to the Homebase, the Homebase manages more intricate operations such as media processing, encryption and authentication. Hence, the majority of the security analysis is concentrated on the Homebase. Notably, an automated analysis of the firmware using EMBA did not uncover any significant vulnerabilities.

5.2 Cracking the WPA2-PSK

To understand the construction of the WPA2-PSK key, we located, through reverse engineering, the responsible functions in the `home_security` binary found in the Homebase firmware.

The WPA2-PSK of the Homebase has a fixed length of eight characters, comprising both lowercase letters, uppercase letters, plus, slash and numbers (64 possible values), yielding a theoretical entropy of approximately 48 bits. This is already a weak key strength to protect wireless network communication. Nevertheless, as shown in Table 2, brute-forcing a key with *only commodity hardware* may take several years.

However, weaknesses in the key generation process lower the entropy of the WPA2-PSK even further. The *WPA2-PSK has a one-to-one mapping* with a non-secret variable (i.e., the serial number), making it susceptible to exploitation. Learning or brute-forcing the serial number compromises the security of the dedicated Eufy network.

The serial number can be discovered in at least three distinct ways: it is printed on the device casing, it can be intercepted in the LAN or WAN network when using the App, and it can be brute forced.

The WPA2-PSK is derived by (1) computing the MD5 hash

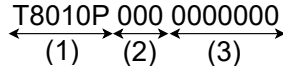


Figure 4: Serial number format

of this serial number, (2) encoding the result using Base64, and (3) taking the first eight bytes.

$$\text{WPA2-PSK} = \text{B64}(\text{MD5}(\text{serial})) [0 : 7]$$

An example of a key generation is shown below:

$$\begin{aligned} \text{SN} &= T8010P23224107B0 \\ \text{MD5}(\text{SN}) &= c4732772b06902fe671689ef92946675 \\ \text{B64}(\text{MD5}(\text{SN})) &= YzQ3MzI3NzJiMDY5MDJmZTY... \\ \text{WPA2-PSK} &= \text{B64}(\text{MD5}(\text{SN})) [0 : 7] \\ &= YzQ3MzI3 \end{aligned}$$

To brute force the WPA2-PSK by guessing serial numbers, we can take advantage of the structure of the serial number. Upon scrutinizing the serial numbers of over 30 distinct Homebases found online, a pattern emerges. Figure 4 shows the various parts of the serial number: (1) device-type identifiers, (2) batch identifiers and (3) device-specific identifiers.

The device-type identifiers are uniform across distinct Homebases. The only difference we detect is the character following the device-type identifiers (i.e., T8010), which may be either "P" or "N". For the batch identifiers, we consistently observe numbers ranging from zero to three. The device-specific identifiers consist of seven hex numbers. However, extrapolating the entire range based on only 30 analyzed devices may lead to inaccuracies. Therefore, we adopt a pragmatic approach, considering both a best-case (BC) scenario, where only batch identifiers are considered to be in the range of zero to three, and a worst-case (WC) scenario, where batch identifiers consist of all possible hexadecimal values. While the former, using our commodity hardware takes approximately 11 hours, the latter may need 30 days.

Upon further investigation of the WPA2-PSK key generation process, we discovered an even worse vulnerability allowing the recovery of the WPA2-PSK in only 20 seconds.

A closer look at the code revealed that only the first six characters of the MD5 hash affect the WPA2-PSK: by definition of Base64 encoding, the first eight characters of the Base64 encoded string only depend on the first six characters of the MD5 hash. To make things worse, the MD5 hash function outputs hexadecimal characters. Hence, the WPA2-PSK is based on only six hexadecimal characters (16 possible values each). The example above simplifies to the following:

	Possibilities	Entropy (Bits)	Time to crack*
WPA2PSK	$64^8 = 2.18 * 10^{15}$	48	17 years
Serial number (WC)	$2 * 16^3 * 16^7 = 2.2 * 10^{12}$	41	30 days
Serial number (BC)	$2 * 4^3 * 16^7 = 34 * 10^9$	35	11 hours
Shortened MD5	$16^6 = 16.8 * 10^6$	24	20 seconds

Table 2: Entropy of distinct WPA2-PSK phases
* (Using our commodity hardware testing @ $\pm 800\text{K}$ keys/sec)

$$\begin{aligned} \text{SN} &= T8010P23224107B0 \\ \text{MD5}(\text{SN}) &= c4732772b06902fe671689ef... \\ \text{MD5}(\text{SN}) [0 : 5] &= c47327 \\ \text{B64}(\text{MD5}(\text{SN}) [0 : 5]) &= YzQ3MzI3 \\ &= \text{WPA2-PSK} \end{aligned}$$

This vulnerable key derivation process diminishes the WPA2-PSK's entropy to a mere 24 bits. This low entropy allows the creation of a custom password list containing all ± 16.8 million potential eight-character passwords for Eufy's dedicated networks. By exploiting this vulnerability, an attacker can leverage a dictionary attack on the WPA2-PSK.

Exploit: Executing a brute force attack on a dedicated Eufy network protected with a WPA2-PSK involves cracking the four-way handshake between the client and access point (i.e., the Homebase). To obtain this handshake, a de-authentication attack on the doorbell is executed [17]. Simultaneously de-authenticating the doorbell and monitoring the wireless network enables us to capture the handshake. Such attacks are common for leveraging an offline attack on WPA and WPA2 security protocols. While brute-forcing WPA3 is more challenging, the key's limited entropy of only 24 bits makes a brute-force attack feasible, even in the case of WPA3.

The Aircrack-ng tool suite is employed for this purpose [26]. Subsequently, the captured handshake is cracked offline using Hashcat [30], a password-cracking tool, and our custom password list. Using commodity GPU hardware², Hashcat successfully cracks the WPA handshake within 20 seconds.

The vulnerability in the WPA2-PSK generation has been assigned CVE-2023-37822.

5.3 Lack of Network Security

Insecure communication. Armed with the password list crafted earlier, the previously deemed secure dedicated Eufy network now faces a significant threat. Engaging in wardriving for hidden wireless networks with an SSID starting with "OCEAN_", we gain unrestricted access to the network of

²We reach about 800k tests per second using the AMD Ryzen 9 5950X, NVIDIA RTX 3080 10GB, 64GB DDR4 3600Mhz

any Eufy Homebase 2 ecosystem. Furthermore, all communication, including commands, video streams and images, is sent unprotected, in *cleartext* on this dedicated network. Compromising the confidentiality and integrity of the Eufy ecosystem.

Lack of isolation. Despite the insecure communication, the most critical networking flaw of the Homebase lies in its use as a pivot. The *lack of isolation* between the dedicated network and the end user's home network through the Homebase is a significant vulnerability. Acting as a proxy, the Homebase permits traffic to flow from the dedicated Eufy network to the end user's home network without any restrictions. Since the end user lacks visibility into the dedicated Eufy network, a malicious actor joining this network could go unnoticed. In such a scenario, the complete private network of the user becomes accessible to the outdoor attacker, turning the Eufy ecosystem into an easy and stealthy entry point for adversaries into the private networks of end users. When combined with the vulnerabilities discussed earlier, the potential consequences of this flaw become enormous.

5.4 Breaking the Encryption Schemes

Eufy states to ensure the users' privacy and promises military-grade encryption. This Section delves into the subsequent measures taken by Eufy to achieve these goals and how we compromised them.

AES encryption in Eufy's ecosystem. Eufy predominantly uses the weaker ECB version of the AES encryption as its cryptographic foundation for data protection. Within the ecosystem, two distinct symmetric AES encryption keys play pivotal roles in safeguarding user information: the P2P AES encryption key dedicated to *securing P2P communication* (commands and messages), and the media key used to *encrypt media* (images and videos for both storage and communication). Both the Homebase and client applications reconstruct the AES keys using obscure key generation processes.

Before discussing the distinct key generation methods, it is imperative to introduce the so-called PPCS identifier (also denoted as *PPCS_ID*). Next to the serial number, this device-specific identifier is stored in flash memory. It consists of three distinct parts separated by dashes, among others, used to derive encryption keys. The 20-character string contains three parts: the first part consists of uppercase characters that identify the device type, the second part is made up of unique numbers related to the device, and the third part contains uppercase characters. It takes the following format:

AAAAAAA — 000000 — BBBB

5.4.1 Breaking Encrypted P2P Traffic

Encrypted P2P traffic exchanged between the Homebase and the mobile App (local or remote) uses AES ECB encryption. The AES key is created containing device-specific information as follows:

$$Key = PPCS_ID[0 : 15] + serial[9 : 15]$$

Here, the key is a combination of the first 16 characters of the PPCS identifier, and the last seven characters of the serial number. It is crucial to note that all parameters used in the key derivation process can be observed in the network traffic between the Homebase and the App (local or remote). This information is transmitted in plain text. Given that all key material is pre-shared over the same network, the encryption of the P2P traffic brings no additional security.

5.4.2 Generating the Media Encryption Key

In the encryption of media, Eufy adopts a more intricate method for generating the AES encryption keys. Although the encryption of videos and images slightly differ (i.e., storage format), the encryption key is the same. This Section focuses on the encryption process of images to demonstrate how the Eufy ecosystem secures its media.

Encrypted image format. Before delving into the encryption process of an image, it is essential to understand the format of an encrypted image. An encrypted image is a file that starts with a plaintext Eufy header containing the serial number of the Homebase (*serial*) and a random value (*rand*). Both are used in recovering the encryption key. The format of the Eufy header is represented as follows:

eufysecurity : < *serial* > : 01 < *rand* > :

The cleartext Eufy header is succeeded by 256 encrypted bytes, being the encrypted JFIF header. Subsequently, this encrypted header is followed by the remainder of the unencrypted JFIF image. Since only the JFIF header is encrypted, leaving the rest of the image unencrypted, there is a potential for information leakage.

Media AES encryption key generation algorithm. The reconstruction of the media encryption key is more complex. Since the Homebase binaries do not contain code to decrypt media, we focus on the encryption process. The primary function responsible for encrypting images is denoted *jpg_encrypt*. This function first constructs the encryption key using *create_pic_code_v1*, and next, encrypts the image.

The generation of the media key entails three steps, depicted in pseudocode in Algorithm 1. First, a Homebase unique *baseCode* is created based on the serial number and

Algorithm 1 Critical key generation functions in the `create_pic_code_v1` algorithm (pseudo code)

```
1: function GETHOMEBASECODE(serial, PPCS_ID)
2:   sfx = getPPCSSuffix(PPCS_ID) ▷ See Alg. 2
3:   baseCode = concat(serial[0 : l], str(sfx))
4:   return baseCode
5: end function
6: function GETRANDSEED(PPCS_ID)
7:   sfx = getPPCSSuffix(PPCS_ID)
8:   rndStr = "01" || str(random()) || str(1000 - sfx)
9:   seed = Obfuscate1(MD5(rndStr))
10:  return (seed, rand)
11: end function
12: function CREATEIMAGEKEY(baseCode, seed)
13:  h = SHA256("01" + baseCode + seed)
14:  encKey = Obfuscate2(h)
15:  return encKey
16: end function
```

PPCS identifier. Next, a *seed* is generated from the same PPCS identifier along with a freshly generated random integer. Finally, the encryption key is derived from the combination of *baseCode* and the *seed*.

Both `genHomebaseCode` and `genRandSeed` use *PPCS_ID* to compute a suffix *sfx*. Then, the *baseCode* is constructed by concatenating a substring of the serial (where the length *l* depends on the last byte) with this suffix. Similarly, the *seed* is computed by concatenating the random integer with a value derived from the suffix (i.e., $1000 - sfx$). The resultant string is then hashed, followed by an obfuscation step. This obfuscation is essentially transforming bytes. Finally, in `createImageKey`, the *baseCode* and the *seed* are hashed and again obfuscated with a custom deterministic algorithm. It is evident that each of these steps, including the obfuscations, is reproducible given the *serial*, *PPCS_ID* and *rand* are known.

5.5 Reconstructing the Media Encryption Key Using dAngr

As outlined earlier, the intricacies involved in the encryption key derivation pose significant challenges to manual reverse engineering. The first attempts were ineffective due to the complexity and inaccuracies found in the decompiled Ghidra code. Specifically, the Ghidra decompiler encountered difficulties with certain sections of the MIPS code, resulting in unreliable decompiled output.

As discussed in Section 4.1, we adopt a novel approach to recover the encryption keys. We leverage dAngr for a concrete and platform agnostic execution of the `create_pic_code_v1` function required to reconstruct the key. Listing 2 shows the commands passed to dAngr to reconstruct encryption keys given the correct inputs.

Since the binary only contains the encryption part, we use

this function to reconstruct the keys. However, in this case, during the actual key generation, a fresh random value is generated to create a unique key for each image. To be able to decrypt an encrypted image, we need to reconstruct the key given a specific random (included in the Eufy header in the encrypted image). Therefore, we take advantage of the hooking functionality of dAngr to replace the call to generate a random (i.e., `random`) with a stub that returns the random number included in the encrypted image.

Listing 2: dAngr commands to reconstruct a media encryption key

```
> load_hooks hooks.py
> set_function_prototype void
   create_pic_code_v1(char*, int, char*,
   char*, char *)
> set_function_call create_pic_code_v1('
   T8010P123DEADBEA', 0x10, 'ZYXABCD
   -456789-TSRQP', '0'*10, '0'*32)
> run
> get_string_memory 0x5000
```

In Listing 2, a function is set up and configured with the correct parameters. The first argument is the serial number, the third is the *PPCS_ID* and the final two parameters are character arrays of 10, respectively 32 characters for the returned random value and the encryption key. The first 16 characters of the latter hold the actual key. To read out the key, we need to access the memory of the last parameter of which the address (0x5000) is printed during debugging.

To decrypt a given image, we extract the *serial* and *rand*, and together with the *PPCS_ID*, we can easily recover the encryption key. The PPCS identifier can be intercepted from the network traffic between the Homebase and the mobile App.

Alternatively, we can eliminate the dependency on the *PPCS_ID*. After further investigation, we successfully recovered the algorithm to generate the *sfx* suffix derived from the PPCS identifier (see Algorithm 2).

Algorithm 2 `getPPCSSuffix(PPCS_ID)`

```
1: s = PPCS_ID[0 : 15].split('-')[1]
2: sfx = int(s[0]) + int(s[1]) + int(s[3]) + int(s[5])
3: if sfx < 5 then
4:   sfx = sfx * 2
5: end if
6: return sfx
```

This function calculates the sum of four of the six digits in the middle part of the PPCS identifier. Next, the result is doubled when the sum is smaller than five, further diminishing the already limited entropy. Thus, instead of monitoring the network traffic and waiting for the *PPCS_ID* to leak, we can opt for a brute-force approach on the parts of the *PPCS_ID* being used.

We simply hook the `getPPCSSuffix` function and generate the 480 potential *sfx* values output by `getPPCSSuffix`. We can easily verify the correctness of a key based on the presence of the magic bytes (i.e., `0xFFD8` for JFIF images) in the decrypted JFIF header. Once we find a match, we also have a valid *sfx* which can be used to decrypt further images.

Using this brute-force approach, we can decrypt any encrypted image without requiring any additional information beyond the encrypted image itself.

The media key derivation process is clearly flawed, enabling an indoor attacker or the cloud server to decrypt all media. It is important to note that other researchers independently uncovered the encryption mechanism while reverse engineering the mobile App [6]. Their motivation primarily focused on facilitating access to the ecosystem through open-source tools. In contrast, our objective was to identify weaknesses in their encryption process. Notably, we achieved this goal, even generating Eufy keys leveraging the lack of entropy without requiring the device identifiers.

6 Countermeasures

Considering the vulnerabilities outlined in the previous section, defining countermeasures for fortifying the Eufy ecosystem is crucial. For each vulnerability, we propose countermeasures:

- *Password reuse*: Avoid reusing the WPA2-PSK. To protect the UART boot sequence, the debug port should be disabled.
- *Password has one-to-one mapping*: Ensure passwords do not have a one-to-one mapping with *public* variables. These variables should be kept secret. Alternatively, passwords should be randomly chosen using a secure random generator.
- *Low entropy password*: Although rectifying low-entropy passwords presents a challenge. A solution would be to discreetly transmit a new high-entropy WPA2-PSK to each paired device after updating each Eufy device. This must be done before changing the Wireless network, allowing background updates without user interaction or breaking the connection with the smart devices.
- *Lack of isolation*: Prevent attackers from pivoting between networks by implementing Linux `iptables` functionality on the Homebase. The Homebase should act solely as an Internet gateway restricting traffic to flow between isolated networks. If necessary, only essential ports should be forwarded.
- *Cleartext traffic*: Augment WPA2-PSK as a protection mechanism with additional protection. Encrypt network communication using established protocols such as TLS

to introduce an extra layer of security and end-to-end encryption. This should be implemented for all communication, including P2P traffic.

- *Bad encryption keys*: Enhance the key derivation process, by adopting standard and secure key derivation and encryption schemes. Refrain from using proprietary DIY algorithms and AES ECB mode. Furthermore, a proper key management solution must be implemented such that keys must not be derived from non-secret information such as serial numbers.

Additionally, instead of only encrypting the media headers, the entire payload should be encrypted.

7 General Insights & Recommendations

Conducting an in-depth security analysis has provided valuable insights into the Eufy ecosystem, unveiling both its vulnerabilities and strengths. Several key lessons can be learned from this comprehensive examination.

To evaluate the impact of our research, we initially assessed the alignment of the Eufy Homebase and doorbell with the OWASP IoT Top 10 [28] before and after our investigation. Initially, the Eufy ecosystem showcased strong compliance with the OWASP IoT Top 10, boasting standardized AES encryption and secure WPA2-PSK-protected network communication. Only an unprotected UART recovery shell and having open UART debug ports were left unaddressed. However, as revealed in our analysis, critical flaws in data encryption, network architecture weaknesses and the use of weak guessable passwords are uncovered. Consequently, Eufy's standing on the Top 10 shifted after our in-depth analysis, now failing in several key areas.

Our analysis revealed that in IoT, particularly in the realm of consumer IoT, security is still often treated as an afterthought, especially in teams lacking security expertise. This often results in reliance on security by obscurity and do-it-yourself (DIY) solutions.

For instance, our work revealed several weak key generation methods. Notably seeking compliance with security standards such as EN 303 645 (ETSI Consumer IoT) may not have been sufficient to prevent the flaws discovered in this study. The recommendation stemming from these insights is clear: IoT manufacturers should invest in comprehensive IoT security training. They must adopt industry-standard, vetted protocols to comply with established security standards. Rather than resorting to custom solutions, strict adherence to best practices is crucial.

Enforcing unique keys per device has become mandatory for compliance with prominent security standards. In turn, security compliance will be required to enter markets worldwide. For instance, embedded devices can only be sold on the EU market after having received a CE label, and demonstrating security compliance will be part of the certifying

process from August 2024. The aforementioned requirement – i.e. unique device keys – is imposed by the realistic attacker model in which a malicious stakeholder with physical access to one IoT device cannot undermine the whole ecosystem. Well-established mechanisms and protocols exist and many standards point to very concrete tactics (without enforcing a specific solution or technology).

However, many developers still develop proprietary solutions instead of relying on widely recognised mechanisms. The major reason is the often recurring complex tension between security and manageability. To decrease the key management burden, obscure mechanisms are often constructed in which keys are unique per device but can still be derived by having knowledge of the device firmware. This implies that an attacker with physical access to an IoT device no longer directly undermines the security of the whole ecosystem (as keys are no longer shared across devices) but can indirectly derive the keys of other devices by inspecting the device firmware. This is possible if an attacker has physical access to one device and can rely on firmware inspection tools which are becoming easily accessible. To tackle this evolution, standards and even legislation should become stricter in the sense that they do not only impose requirements concerning general characteristics of the device but also on feasible and non-feasible mechanisms to enforce it. Although this may restrict the degrees of freedom at design and development time and may result in more advanced key management (ultimately resulting in a more expensive lifecycle), it will result in improved security.

The community, particularly in consumer IoT, would benefit from the availability of *reference architectures and proof of concepts* that depict commonly encountered use cases and scenarios. These should encompass essential aspects such as the proper use of STUN, TURN and ICE services for remote access; secure pairing of smart devices, gateways and mobile devices; correct use and implementation of public key infrastructure; secure update procedures; and the secure use of cloud services and APIs. Such resources would deter developers from resorting to DIY strategies and obscure solutions.

Securing IoT devices is undoubtedly a substantial endeavor, requiring expertise across various domains, including embedded hardware and software, network security, cloud communication, and mobile or web development. The commitment to strong security practices is essential for the sustained integrity of IoT ecosystems.

8 Related Work

We discuss in this Section relevant and previous research and studies that influenced our approach to analyzing the Eufy ecosystem.

State of the art of IoT security. Costin et al. performed the first large-scale analysis on IoT devices [10], examining over 683 firmware images, unveiling vulnerabilities on 123 distinct products. Another large-scale analysis is done by Neshenko et al. [25], they focus on discovered IoT vulnerabilities and classify the various vulnerabilities and weaknesses inherent to IoT devices. Performing new large-scale analyses has become increasingly challenging, due to a recent trend where manufacturers strive to maintain the secrecy of their device’s firmware. This approach may result in fostering security through obscurity, which fails to deter attackers equipped with sufficient resources.

In a more targeted study, Schwartz et al. analyze the security of 16 popular IoT devices leveraging reverse engineering techniques [41]. Their systematic application of reverse engineering techniques uncovered distinct vulnerabilities, emphasizing the importance of this method in identifying security weaknesses. Obermaier et al. focus on cloud-based video surveillance systems, analyzing four distinct IP Cameras [27] through a combination of network and firmware analysis. This approach led to the uncovering of various vulnerabilities related to authentication, proprietary encryption algorithms and weak certificate validation. Rondon et al. delved into the security of E-IoT systems scrutinizing proprietary protocols used in E-IoT settings [36]. Collectively, these studies indicate the urgent need for enhanced security measures in IoT devices.

WPA attacks. The landscape of wireless security, particularly in the context of Wi-Fi networks, has been a subject of extensive research and exploration. Lorente et al. scrutinized WPA2 password-generation algorithms in Wireless routers and discovered that many routers used weak password-generation algorithms [21]. Reversing the algorithms, Lorente et al. discovered that in most algorithms known parameters were used as input and that they had a simple deterministic password-generation process. One of the vulnerabilities we uncovered is similar to this work. However, we go further than discovering a one-to-one mapping, identifying multiple weaknesses in Eufy’s password generation algorithm.

Reversing engineering. Several studies discuss methodologies for discovering and analyzing vulnerabilities [14, 19, 20, 42]. In this related work, reverse engineering is considered an efficient but exhaustive method for analyzing embedded devices. Techniques for more efficient reverse engineering and methodologies for performing a complete device analysis are discussed. Thomas et al. present a framework to reduce the upfront effort in analyzing and reverse engineering using static and dynamic analysis techniques [45].

In case studies, Casagrande et al. applied reverse engineering methodologies to unveil vulnerabilities in the Xiaomi ecosystem. Their work exposed issues in both the pairing process and in applications developed by Xiaomi [8, 9]. In their work, they reverse-engineered the Xiaomi companion App

and the Bluetooth Low-Energy communication. The reverse engineering led to the uncovering of various vulnerabilities in the pairing process of the Xiaomi Fitness tracking system and the Xiaomi E-scooters. The vulnerabilities they discovered are cleartext keys, unauthenticated pairing and modifying the password without authentication. Ullrich et al. reversed the Neato vacuum and discovered an attack leveraging weak secret keys and a buffer overflow via the cloud to break the Neato ecosystem [46]. Giese et al. reverse engineer using hardware hacking the Amazon Echo Dot and perform IoT forensics to uncover bad practices that lead to personal data leakage [15]. Other examples of high-impact attacks uncovered by reverse engineering are the Zigbee worm exploiting Philips Hue lamps [37], and breaking glucose monitoring systems thanks to weak proprietary protocols [35]. In our research, we employ a similar methodology to uncover vulnerabilities, focusing on reversing the binaries, networking, and internal operations of the Eufy ecosystem. Contrary to the above work, we present a novel approach leveraging a new cross-platform debugger to assist manual reverse engineering in embedded devices.

Symbolic execution. Yadegari et al. and Banescu et al. emphasize symbolic execution as a potent mechanism to circumvent obfuscation techniques [4, 48]. Symbolic execution proves invaluable in identifying weaknesses and vulnerabilities. Nevertheless, symbolic execution faces its own set of challenges, notably in the analysis of cryptographic functions, which is inherently complex. Vanhoef et al. demonstrate that simulating cryptographic primitives during symbolic execution can be done to find weaknesses in cryptographic functions [47]. Ramos et al. develop an under-constrained symbolic execution framework to analyze individual functions rather than whole programs, bypassing several weaknesses of symbolic execution engines [32]. Contrary to prior work, this work leverages the binary lifting and interpretation provided by `angr` to make debugging platform-independent. While our attack only requires concrete execution, our debugger also supports symbolic execution.

Case studies including the Eufy doorbell. P. Moore analyzed the web interface of the Eufy doorbell [44]. Moore proved that Eufy uploads images to the cloud without authorization. Moore also discovered that the video stream of the Eufy doorbell was sent unencrypted. These vulnerabilities were confirmed and patched by Eufy, ensuring that now all video and images are end-to-end encrypted.

M.A. Stanislav examined various security frameworks to determine the overall security posture of internet-connected devices [43]. An analysis was performed on 40 internet-connected cameras. This analysis includes information gathering, disassembling the device, analyzing the various interfaces of the device, and more. Eufy is one of the cameras being

analyzed, and comes out as one of the more mature brands, having overall good security and conforming to best practices.

The open-source community also reverse-engineered the Eufy App and reconstructed the P2P protocol. Allowing them to replace the App or web interface, and connect it to a home automation system [6]. The project primarily focuses on a specific aspect of the P2P protocol related to communication between App and Homebase. However, the P2P protocol within the ecosystem extends further than App Homebase communication, the communication between the Homebase and other Eufy devices is a critical part of the P2P protocol. To build upon and expand the existing research of the Eufy ecosystem, we dissect the firmware of the Eufy devices and the Eufy ecosystem internals, while actively seeking vulnerabilities and weaknesses.

9 Conclusion

The reverse engineering and analysis of the Eufy ecosystem provided insights into the intricate workings of its devices. This investigation uncovered multiple weaknesses, highlighting critical areas of one of the top players in the IP Camera domain. The core of our work involved the analysis of the proprietary peer-to-peer protocol, dissecting the encryption mechanisms, and understanding the internal network's behaviour through a combination of reverse engineering, binary interpretation, and network traffic analysis.

We introduce a novel approach for key reconstruction in embedded devices. We developed `dAngr` a symbolic debugger that augments manual reverse engineering. By leveraging `angr`, a symbolic execution engine that implements binary lifting and interpretation of the lifted code, our tool enables platform-agnostic execution of specific functions, allowing us to execute isolated cryptography functions in an embedded cross-architecture binary without a complex or time-consuming process. We demonstrate our novel approach by reconstructing the AES keys for media encryption in the Eufy ecosystem.

Our findings culminated in an attack on the Eufy ecosystem requiring no network connectivity. The sole prerequisite is proximity to the Homebase's dedicated network. Leveraging two vulnerabilities uncovered during our analysis, the attack serves as a potential entry point into the end user's private home network. The ease and severity of this attack deem it highly critical. We proposed appropriate countermeasures for each identified flaw.

Eufy has confirmed the vulnerabilities and initiated security patches, further bolstering their security.

Responsible disclosure. In June 2023, we responsibly disclosed all newly identified vulnerabilities to Eufy. The comprehensive disclosure process was conducted through Anker’s channels. In addition to providing a detailed write-up of the vulnerabilities, we included recommendations for effectively mitigating these issues. A condensed version of the recommended mitigations can be found in Section 6.

References

- [1] MarkNtel Advisors. Smart Doorbell Market Achieves USD 16.2 Billion 2023, Braces for 16.7% CAGR Elevate Until 2030. <https://www.marknteladvisors.com/research-library/smart-doorbell-market.html>, 2024.
- [2] Harald T. Alvestrand. Transports for WebRTC. RFC 8835. <https://www.rfc-editor.org/info/rfc8835>, January 2021.
- [3] Buildroot Association. Buildroot. <https://buildroot.org/docs.html>, 2024.
- [4] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200. ACM.
- [5] Gerhard Hechenberger Bernhard Gründling and Steffen Robertz. Sec consult - the eufycam long-term observation. <https://sec-consult.com/blog/detail/the-eufycam-long-term-observation/>, 2024.
- [6] Patrick Broetto. eufy-security-client. <https://github.com/bropat/eufy-security-client>, 2024.
- [7] Elisabetta Carrara, Karl Norrman, David McGrew, Mats Naslund, and Mark Baugher. The Secure Real-time Transport Protocol (SRTP). RFC 3711. <https://www.rfc-editor.org/info/rfc3711>, March 2004.
- [8] Marco Casagrande, Riccardo Cestaro, Eleonora Losiouk, Mauro Conti, and Daniele Antonioli. E-spoofers: Attacking and defending xiaomi electric scooter ecosystem. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 85–95. ACM, 2023.
- [9] Marco Casagrande, Eleonora Losiouk, Mauro Conti, Mathias Payer, and Daniele Antonioli. Breakmi: Reversing, exploiting and fixing xiaomi fitness tracking ecosystem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):330–366, Jun. 2022.
- [10] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. SEC’14, page 95–110, USA, 2014. USENIX Association.
- [11] NSA’s Research Directorate. Ghidra. <https://ghidra-sre.org/>, 2024.
- [12] Eufy. Eufy. <https://us.eufy.com/>, 2024.
- [13] Eufy. Privacy Commitment. <https://us.eufy.com/pages/privacy-commitment>, 2024.
- [14] Aurélien Francillon, Sam L. Thomas, and Andrei Costin. Finding software bugs in embedded devices. In Gildas Avoine and Julio Hernandez-Castro, editors, *Security of Ubiquitous Computing Systems: Selected Topics*, pages 183–197. Springer International Publishing.
- [15] Dennis Giese and Guevara Noubir. Amazon echo dot or the reverberating secrets of IoT devices. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 13–24. ACM.
- [16] Eric Hamilton. JPEG File Interchange Format Version 1.02. <https://www.w3.org/Graphics/JPEG/jfif3.pdf>, 1992.
- [17] Stefan Savage John Bellardo. 802.11 denial-of-service attacks: real vulnerabilities and practical solutions. page 95–110, USA, 2003. USENIX Association.
- [18] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445. <https://www.rfc-editor.org/info/rfc8445>, July 2018.
- [19] Xixing Li, Qiang Wei, Zehui Wu, and Wei Guo. A comprehensive survey of vulnerability detection method towards linux-based IoT devices. In *Proceedings of the 2023 2nd International Conference on Networks, Communications and Information Technology*, pages 35–41. ACM.
- [20] Muqing Liu, Yuanyuan Zhang, Juanru Li, Junliang Shu, and Dawu Gu. Security analysis of vendor customized code in firmware of embedded device. In Robert Deng, Jian Weng, Kui Ren, and Vinod Yegneswaran, editors, *Security and Privacy in Communication Networks*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 722–739. Springer International Publishing.
- [21] Eduardo Novella Lorente, Carlo Meijer, and Roel Verdult. Scrutinizing WPA2 password generating algorithms in wireless routers. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

- [22] mantech. EEPROM CH341A 24 25 Series Flash BIOS USB Programmer. <https://www.mantech.co.za/Datasheets/Products/CH341B-Programer.pdf>, 2024.
- [23] MarkWideResearch. Smart Doorbell Market Analysis. <https://markwideresearch.com/smart-doorbell-market/>, 2024.
- [24] Michael Messner. EMBA. <https://github.com/e-m-b-a/embra>, 2022.
- [25] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. Demystifying iot security: An exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations. *IEEE Communications Surveys and Tutorials*, 21(3):2702–2733, 2019.
- [26] The Aircrack ng Project. Aircrack-ng. <https://www.aircrack-ng.org/>, 2024.
- [27] Johannes Obermaier and Martin Hutle. Analyzing the security and privacy of cloud-based video surveillance systems. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security*, IoTPTS '16, page 22–28, New York, NY, USA, 2016. Association for Computing Machinery.
- [28] OWASP. Owasp iot top 10. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Top_10, 2024.
- [29] Qemu project. QEMU. <https://www.qemu.org/>, 2024.
- [30] The Hahscat Project. Hashcat. <https://hashcat.net/hashcat/>, 2023.
- [31] radare org. Radare2. <https://rada.re/n/index.html>, 2024.
- [32] David A. Ramos and Dawson Engler. Under-Constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., August 2015. USENIX Association.
- [33] Tirumaleswar Reddy, K. Alan Johnston, Philip Matthews, and Jonathan Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 8656. <https://www.rfc-editor.org/info/rfc8656>, February 2020.
- [34] Grand View Research. Smart Doorbell Market Size, Share and Trends Analysis Report By Product Type (Wired Doorbell, Wireless Doorbell), By End-user (Residential, Commercial), By Region, And Segment Forecasts, 2023 - 2030. <https://www.grandviewresearch.com/industry-analysis/smart-doorbell-market-report>, 2024.
- [35] Luca Reverberi and David Oswald. Breaking (and fixing) a widely used continuous glucose monitoring system. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.
- [36] Luis Puche Rondon, Leonardo Babun, Ahmet Aris, Kemal Akkaya, and A. Selcuk Uluagac. LightningStrike: (in)secure practices of e-IoT systems in the wild. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 106–116. ACM.
- [37] Eyal Ronen, Colin O’Flynn, Adi Shamir, and Achi-Or Weingarten. IoT goes nuclear: Creating a ZigBee chain reaction. *IEEE Security and Privacy*, pages 54–62.
- [38] Jonathan Rosenberg, Christian Huitema, Rohan Mahy, and Joel Weinberger. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489. <https://www.rfc-editor.org/info/rfc3489>, March 2003.
- [39] Henning Schulzrinne, Anup Rao, Rob Lanphier, Magnus Westerlund, and Martin Stiemerling. Real-Time Streaming Protocol Version 2.0. RFC 7826. <https://www.rfc-editor.org/info/rfc7826>, December 2016.
- [40] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.
- [41] Omer Shwartz, Yael Mathov, Michael Bohadana, Yuval Elovici, and Yossi Oren. Opening pandora’s box: Effective techniques for reverse engineering IoT devices. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications*, volume 10728, pages 1–21. Springer International Publishing. Series Title: Lecture Notes in Computer Science.
- [42] Omer Shwartz, Yael Mathov, Michael Bohadana, Yuval Elovici, and Yossi Oren. Reverse engineering IoT devices: Effective techniques and methods. 5(6):4965–4976. Conference Name: IEEE Internet of Things Journal.
- [43] Mark A. Stanislav. *Multi-dimensional Security Integrity Analysis Of Broad Market Internet-connected Cameras*. PhD thesis, Dakota State University, 2022.

- [44] Nicholas Sutrich. Security researcher says Eufy has a big security problem. <https://www.androidcentral.com/accessories/smart-home/security-researcher-says-eufy-has-a-big-security-problem>, 2022.
- [45] Sam L. Thomas, Jan Van den Herrewegen, Georgios Vasilakis, Zitai Chen, Mihai Ordean, and Flavio D. Garcia. Cutting through the complexity of reverse engineering embedded devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021:360–389, Jul. 2021.
- [46] Fabian Ullrich, Jiska Classen, Johannes Eger, and Matthias Hollick. Vacuums in the cloud: Analyzing security in a hardened IoT ecosystem. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.
- [47] Mathy Vanhoef and Frank Piessens. Symbolic execution of security protocol implementations: Handling cryptographic primitives. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association.
- [48] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 732–744. ACM.

SoK: Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems

Xi Tan
CactiLab, University at Buffalo

Zheyuan Ma
CactiLab, University at Buffalo

Sandro Pinto
Universidade do Minho

Le Guan
University of Georgia

Ning Zhang
Washington University in St. Louis

Jun Xu
University of Utah

Zhiqiang Lin
Ohio State University

Hongxin Hu
University at Buffalo

Ziming Zhao
CactiLab, University at Buffalo

Abstract

Arm Cortex-M processors are the most widely used 32-bit microcontrollers among embedded and Internet-of-Things devices. Despite the widespread usage, there has been little effort in summarizing their hardware security features, characterizing the limitations and vulnerabilities of their hardware and software stack, and systematizing the research on securing these systems. The goals and contributions of this paper are multi-fold. First, we analyze the hardware security limitations and issues of Cortex-M systems. Second, we conducted a deep study of the software stack designed for Cortex-M and revealed its limitations, which is accompanied by an empirical analysis of 1,797 real-world firmware. Third, we categorize the reported bugs in Cortex-M software systems. Finally, we systematize the efforts that aim at securing Cortex-M systems and evaluate them in terms of the protections they offer, runtime performance, required hardware features, etc. Based on the insights, we develop a set of recommendations for the research community and MCU software developers.

1 Introduction

Microcontroller units (MCUs) are small computers designed for embedded and Internet of Things (IoT) applications in contrast to microprocessors used in smartphones, personal computers, and servers. They operate at frequencies ranging from several kHz to several hundred MHz. The sizes of their ROMs and RAMs are small and usually fall into the range of several hundred bytes to several megabytes. Even though MCUs are general-purpose computers, they are commonly employed for running specialized software and firmware tailored to specific applications.

The Arm Cortex-M family, which has three major architectures and 12 processors as of 2023, is the most popular 32-bit MCU architecture without a memory management unit (MMU) on the market. More than 80 hardware vendors have licensed Cortex-M cores [1]. 4.4 billion Cortex-M MCUs were shipped in the 4th quarter of 2020 alone [2], and it is es-

timated that Cortex-M MCUs account for almost 100 billion deployed embedded and IoT devices in 2021 [3].

Given the sheer volume of deployed Cortex-M systems, one would anticipate that the security of their hardware and software stack has been thoroughly studied and systematized. Unfortunately, this is not the case. To bridge the knowledge gap that hinders the users and researchers, we seek to answer the following questions regarding their security states:

- *Q1 - What are the security features, limitations, and issues at the Cortex-M microarchitecture, instruction set architecture (ISA), and beyond?* The answer helps understand the constraints in securing software on Cortex-M.

To address this question, we analyze the hardware security limitations of Cortex-M by comparing its offerings with microprocessors. *Our main observation (§3) is that Cortex-M processors lack support for memory virtualization and provide only basic memory protection mechanisms. Additionally, their other security features, e.g., TrustZone, are streamlined compared to their Cortex-A counterparts and introduce new vulnerabilities.*

- *Q2 - What are the security mechanisms and flaws of Cortex-M based software systems?* The answer helps understand the status of Cortex-M software security in real-world systems.

To answer this question, we compile a dataset of 1,797 real-world Cortex-M firmware samples, including 1,003 newly collected ones, and perform by far the largest empirical analysis on the adoption of security mechanisms on real-world Cortex-M systems. In particular, we summarize the software architectures found in these samples and other research projects. We develop binary analysis tools to verify if the collected samples leverage the security mechanisms that have been widely deployed on microprocessor-based systems, e.g., privilege separation and stack canaries. *We uncovered that (§4) despite extensive research on more secure architectures for microcontroller-based systems, these advancements are rarely implemented in real-world firmware. Moreover, the hardware security features offered by Cortex-M processors are seldom utilized in the majority of the assessed firmware; hence, where is the "up"?!. Furthermore, existing compiler-based mitiga-*

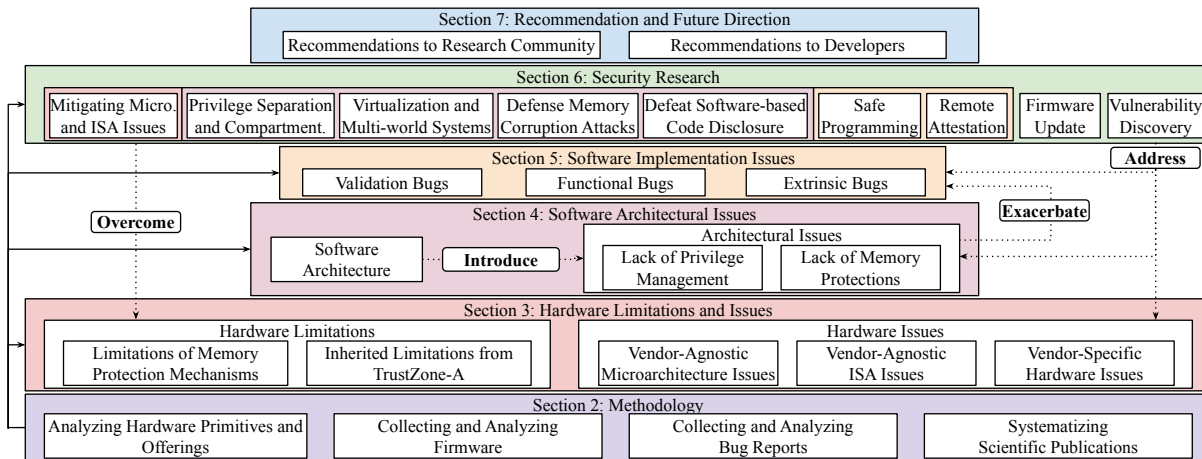


Figure 1: Overview of the organization and contributions of this paper

tions designed for process-based operating systems (e.g., stack canaries) prove ineffective when operating within a single physical address space.

- *Q3* - What are the nature and severity of the publicly disclosed vulnerabilities in the Cortex-M based software systems? The answer helps find out software bugs that are more likely to be exploited in such systems.

To tackle this question, we analyze 310 Cortex-M related software bug reports spanning nearly six years, from 2017 until 2023. Our analysis includes systems developed by nine hardware vendors, e.g., Nordic and NXP, and seven real-time operating systems (RTOS), e.g., FreeRTOS. We further categorize the software implementation issues into validation, functional, and extrinsic bugs, a taxonomy adopted in a recent work studying the vulnerabilities in Cortex-A systems [4]. Our insights (§5) include that these systems not only exhibit memory corruption vulnerabilities but also display weaknesses in their protocol and cryptographic implementations.

- *Q4* - What defenses for Cortex-M systems have been explored in the literature, and what are their limitations? Together with the previous answers, this helps shed light on new research directions to secure Cortex-M systems.

To address this question, we create a taxonomy and comparative evaluation of over 50 papers spanning nearly nine years. Our evaluation framework considers the defenses each solution offers, the hardware units it relies on, and their runtime overhead in terms of memory size, performance, etc. Our major observations (§6) include the research community not only shifts the exact same defenses from microprocessor-based systems on Cortex-M systems, e.g., enforcing isolation and confinement, stack integrity, and control flow integrity, but also develops solutions intrinsically linked to the MCU characteristics, e.g., peripheral-oriented fuzzing.

Based on the insights, we develop a set of recommendations for the research community and MCU software developers (§7). Figure 1 provides an overview of the organization and contributions of this paper. We have open-sourced our source

code, dataset, and supplementary materials ¹.

2 Methodology

2.1 Adversarial Model

In general, we consider the security limitations and issues of the microarchitecture, ISA, and above. In particular, we assume an adversary can perform (i) microarchitecture side-channel attacks, e.g., bus interconnect; (ii) glitching, e.g., voltage fault injection; (iii) remote attacks via a network; (iv) nearby wireless attacks via BLE, ZigBee, etc.; (v) local attacks through peripherals and debug ports; and (vi) software side-channel attacks. On systems without TrustZone-M, we consider an adversary with one or more of the following objectives: (i) to obtain secrets from the flash, e.g., intellectual property (IP) theft and RAM; (ii) to tamper sensitive data; (iii) code execution and privilege escalation, e.g., control-flow hijacking. On systems with TrustZone-M, we assume all components in the non-secure state are untrusted and consider an adversary with all aforementioned goals as well as compromising the secure state.

2.2 Analyzing Hardware Offerings

We provide a detailed analysis of the hardware security limitations and issues. Due to the page limit, a detailed walk-through of the Cortex-M architecture is not included in this paper. Interested readers please refer to our supplementary materials, which consolidate information from various official sources [5–14]. To aid in research for the community, we have developed an open-source code suite, demonstrating the use of Cortex-M security features.

2.3 Collecting and Analyzing Firmware

Collecting Firmware. The process of collecting and decoding Cortex-M firmware was far from straightforward and re-

¹<https://github.com/CactiLab/SoK-Cortex-M>

Table 1: Manufacturer distribution of the compiled real-world firmware dataset. *Italic* represents newly collected sample that were not publicly released before.

HW Vendor	Nordic [15]	<i>Other Nordic</i>	TI [15]	<i>Telink</i>	<i>Dialog</i>	<i>NXP</i>	<i>Cypress</i>	ST [16]	Total
# Firmware	768	690	22	192	53	1	67	4	1,797
# Devices	513	-	20	120	36	1	-	-	689

sulted in the accumulation of significant amounts of unusable data. We used three approaches to collect firmware: (i) we filtered Cortex-M firmware from publicly available embedded system datasets [15, 17–21]; (ii) adopting an analogous methodology as described in [15], we developed scripts to analyze/unpack mobile apps and extract potential Cortex-M firmware. Using this approach, we collected 4,693 potential samples from six silicon vendors. These samples are in various formats, e.g., S-record for NXP, cyacd format for Cypress, and proprietary format of Qualcomm; (iii) we crawled websites for 25 silicon and device vendors known for embedded and IoT devices. This effort resulted in 1,687 potential samples, but none of them turned out to be Cortex-M firmware. This aligns with the findings in FirmXRay [15], which noted that vendors seldom make their firmware available online.

As shown in Table 1, our firmware collection endeavor ended up with 1,797 unique Cortex-M firmware from seven hardware vendors. Among these, the FirmXRay dataset includes 790 firmware samples, representing 533 distinct devices from two vendors (768 from Nordic [22] and 22 from Texas Instruments [23]). Additionally, the HEAPSTER dataset [16] encompasses four Cortex-M binaries from STMicroelectronics (ST) [24]. Furthermore, we have gathered 1,003 firmware from other vendors, including Nordic (690), Telink [25] (192 firmware for 120 unique devices), Dialog [26] (53 firmware for 36 devices), NXP [27] (1), and Cypress [28] (67). These samples have not been publicly shared before. The firmware in our collection is in raw binary format, lacking symbolic information.

Analyzing Firmware. We used FirmXRay [15] to recognize the base address of each firmware. Scripts were then developed to identify the Cortex-M vector table and perform recursive disassembly with Ghidra [29]. We also applied scripts to filter a portion of firmware samples that contain device information, ensuring that they are from distinct devices. We conducted an analysis of the disassembled samples using the following heuristics: (i) to identify if firmware uses any RTOS, we performed binary function recognition [30] and string searches for ten popular RTOSs; (ii) for firmware that uses an RTOS, we analyzed if task stack overflow checks are performed. To this end, we checked if the task stack overflow handling functions, e.g., `osRtxKernelErrorNotify()` with the parameter `osRtxErrorStackOverflow` in CMSIS RTOS2 [31], are called by other functions in the firmware; (iii) we analyzed if and how the `CONTROL` register is changed and how the `SVC` instruction is used to determine privilege separation and stack usages; (iv) to check if there are stack

canaries, we analyzed function prologues and epilogues for specific instruction patterns derived from canary-protected functions generated by three compilers. In addition, we searched if the firmware has the hard-coded libc error message “*** stack smashing detected ***” and whether the function printing out this message is called by other functions, which is a practice used before [32].

2.4 Collecting and Analyzing Bug Reports

We retrieved over 500 hardware and software bug reports related to Cortex-M systems from 2017 to 2023 [33], which shows a growing trend. Besides “Arm”, we included in our list of keywords the names of top hardware vendors [34], popular RTOSs [35], and embedded SSL libraries, e.g., Mbed TLS [36] and wolfSSL [37]). We manually confirmed the bug reports indeed affect Cortex-M systems, including verifying the affected chips and inspecting the source code. Two researchers worked together to categorize each bug into a relevant subclass, which was verified by a third researcher.

2.5 Systematizing Scientific Publications

We collected over 30 papers on Cortex-M security from top conferences². In addition, we supplement our list of surveyed papers with another over 20 articles that are highly relevant to the topic but published in other venues. Note that our systematization focuses on the works explicitly designed for and implemented on Cortex-M. Nevertheless, we discuss related works that were designed for or implemented on other architectures but may be applied to Cortex-M in §6.10.

2.6 Threats to Validity

Our analysis of firmware may be subject to biases and imprecision due to the limited number of firmware. There is a risk of over-representing systems from specific vendors. Most firmware in our dataset (57.3%) are raw binaries and lack detailed device and architecture information, making it difficult to confirm their intended use cases and resulting in a potential bias in analyzing similar firmware samples. Additionally, the lack of proof-of-concept exploits and vague CVE descriptions introduces imprecisions in the classification of vulnerabilities. Furthermore, our analysis focuses on publicly disclosed vulnerabilities. Undiscovered vulnerabilities could unveil additional fundamental issues in Cortex-M systems.

3 Hardware Limitations and Issues

3.1 Hardware Limitations

Hardware limitations are missing or constrained hardware security features, which are typically non-patchable. Compared

²<https://csrankings.org/>

with Cortex-A, Cortex-M features distinct design elements, particularly in its memory protection mechanisms and the TrustZone extension (TrustZone-M versus TrustZone-A).

Limitations of Memory Protection Mechanisms

L01. No memory virtualization: No hardware-supported memory virtualization is available on Cortex-M due to the absence of a memory management unit (MMU). Instead, software modules share the same physical address space. Such lack of memory virtualization also implies a small address space (4GB), which presents challenges to effective address space layout randomization (ASLR) due to low entropy.

L02. No input-output memory management unit: Besides MMU, input-output memory management unit (IOMMU) or its equivalents, i.e., IOMPU, that provide memory protection from malicious direct memory access (DMA)-capable peripherals are also missing on Cortex-M. Some hardware vendors implement their own IOMPU, i.e., the resource domain controller on NXP i.MX RT [38, 39], but they are only found in some of the latest devices.

L03. A small number of MPU regions and limited sizes: Cortex-M only supports a small number of memory protection unit (MPU) regions, and the size of regions must be a multiple of 32 bytes. Compared to the page-based memory access control on microprocessors, the granularity of MPU-based is coarse-grained, and it is insufficient to implement fine-grained isolation that requires a large number of regions.

L04. A small number of secure/non-secure memory regions: The number of regions supported by secure attribute unit (SAU) is small, e.g., up to 8 regions on Cortex-M33, resulting in limited design choices in splitting the secure and non-secure address space. To alleviate this issue, silicon vendors use the implementation defined attribution unit (IDAU), which supports up to 256 regions, to create more partitions. However, if more than 256 partitions are needed or the device has many peripherals, this may not be enough [40].

Inherited Limitations from TrustZone-A

L05. No intrinsic encryption to protect the secure state memory: TrustZone-M does not encrypt the secure state memory. Consequently, cold boot attacks [41] can dump the secure state memory. There could also be information leakage when a memory protection controller (MPC) assigns a memory region from the secure state to the non-secure state at run-time, which we will discuss in [I05](#).

L06. Lack of intrinsic support for multiple trusted execution environments: TrustZone-M only provides *one* isolated execution environment in which the trusted firmware executes, resulting in a large software trusted computing base (TCB). For instance, TF-M [42] has over 117K lines of code.

L07. Lack of hardware-based remote attestation in TrustZone-M: Same as Cortex-A [4], Cortex-M TrustZone lacks a hardware-based integrity reporting mechanism, so it

cannot provide a hardware-based remote attestation as Intel software guard extensions (SGX) does. For example, the Arm platform security architecture (PSA) introduces a weakened software-based attestation method [43, 44].

Insights

- The Cortex-M architecture offers weaker memory management interfaces than popular microprocessors, creating challenges to enforce memory isolation and security.
- TrustZone-M inherits hardware limitations of TrustZone-A and introduces more constraints.

3.2 Hardware Issues

Hardware issues discuss vulnerable hardware components and hardware-supported operations.

Vendor-Agnostic Microarchitecture Issues

I01. Vulnerable to microarchitectural side-channel attacks: Although most Cortex-M processors lack a cache or branch predictor at the microarchitectural level, there are other side channels that can leak information.

Information leakage through power analysis: ELMO [45] demonstrates the feasibility of reversing AES S-Box output code sequences through power analysis on the Cortex-M0 processor. Furthermore, Vafa et al. [46] successfully applied a power analysis attack to recover running instructions on the Cortex-M3 processor.

Information leakage through timing side-channels: MCU bus interconnect arbitration logic involves delays when multiple bus masters, such as the CPU and DMA, simultaneously access a shared secondary port, like a memory controller. As demonstrated in BUSTed [47], the attacker can successfully bypass protections provided by the MPU and TrustZone by exploiting these timing differences.

Information leakage through long-term data remanence: UnTrustZone [48] reveals that static random-access memory (SRAM) can be manipulated to imprint and expose on-chip secrets by accelerating analog-domain changes in SRAM. Using this method, UnTrustZone successfully extracts AES keys and proprietary firmware from various Cortex-M devices protected by TrustZone.

I02. Vulnerable to fault injections: A fault injection attack involves deliberately causing errors in a system's hardware (e.g., voltage, clock, electromagnetic) to disrupt its normal operations of a digital circuit and exploit these induced faults for malicious purposes. Johannes Obermaier and Marc Schink et al. discussed how to escalate the debug interface permissions or execute arbitrary code by injecting faults into voltage [49], Quad-SPI bus [50], and electromagnetic [51] at boot time on Cortex-M0/3/4 devices. μ -Glitch [52] entails injecting multiple, coordinated voltage faults into Cortex-M devices to bypass the TrustZone protection, allowing leaking secrets stored in secure memory into non-secure code.

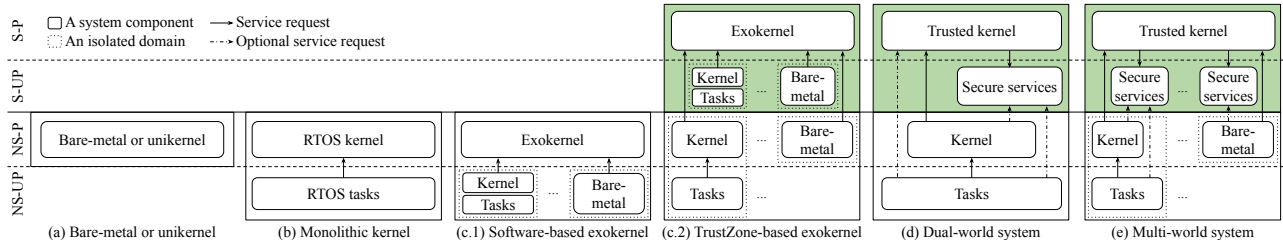


Figure 2: Identified Cortex-M software architectures in the collected dataset and in the literature. NS-UP: non-secure unprivileged, NS-P: non-secure privileged, S-UP: secure unprivileged, S-P: secure privileged.

Vendor-Agnostic ISA Issues

I03. Fast state switch mechanism exploitable for privilege escalation: Cortex-M TrustZone uses the fast state switch technique to allow direct cross-state transitions from any privilege level without the need for a higher privileged secure monitor mode like Cortex-A TrustZone. Although this feature makes cross-state transitions more efficient, it exposes vulnerabilities to a recently discovered attack known as *ret2ns* [53]. This attack leverages critical system registers and instructions used by the fast state switch to escalate privilege in the non-secure state, potentially leading to arbitrary code execution.

I04. Improper privilege management for inter-processor debugging: CVE-2018-18068 reports that the debugging host’s privilege level is ignored in the inter-processor debugging mode, allowing the non-secure state on both TrustZone-M and TrustZone-A to gain access to the secure state resources via the ETM [54, 55].

I05. Information leakage to the non-secure state due to state switches: This could happen through memory and general-purpose and special registers: (i) if a region used by the secure state is re-mapped by MPC into the non-secure state without proper sanitization, sensitive information will be leaked; (ii) information leakage could happen if the general-purpose registers are not cleared when switching to the non-secure state. To address this issue, Arm recommends general-purpose registers that are not used to pass arguments should be cleared before state switches [7]; (iii) CVE-2021-35465 reports an issue of the floating-point lazy load multiple (VLLDM) instruction, which allows the non-secure code to access secure state floating-point registers.

Vendor-Specific Hardware Issues

I06. Improper privilege management in vendor-specific hardware features: Some hardware vendors introduce over-powerful hardware features that can be exploited to gain full control of the system. For example, NXP LPC55S6x MCUs include a ROM patch controller to fix bugs in the ROM after fabrication. CVE-2021-31532 reports that even attackers in the non-secure state and unprivileged level can utilize the ROM patch controller to reconfigure the SAU regions to gain privilege escalation. CVE-2022-22819 shows that the ROM patch controller firmware also has a buffer overflow bug that can lead to arbitrary code execution at the privileged level.

I07. Bypassable vendor-specific readback protection:

Only M55 and M85 have the execute-only memory (XOM) feature, which prevents software or a hardware debugger from reading execute-only memory [56]. For MCUs before M55, some hardware vendors implement their own hardware units to prevent reading from the debug interface, a feature known as readback protection. For instance, the Nordic nRF51 series implements a mechanism to prevent debuggers from directly accessing flash and RAM address ranges. Notwithstanding, we found that only 32 out of the 1,458 Nordic samples in our dataset enable this feature. This protection, however, can be easily bypassed through arbitrary register read and write and single stepping in debugging [57]. Though the mechanism was improved in the nRF52 series [58], CVE-2020-27211 reports that a voltage glitch attack can still bypass it [51]. Similar mechanisms implemented by ST [59], NXP [60], and TI [61] are also bypassable by inferring instructions from the observed state transitions [62].

Insights

- Streamlined hardware mechanisms in Cortex-M, e.g., fast state switch, lead to new privilege management vulnerabilities and information leakage.
- The fragmentation of the Cortex-M ecosystem has brought in new security challenges: vendors aggressively introduce over-powerful hardware, which can undermine Cortex-M systems security if not properly designed.

4 Software Architectural Issues

4.1 Software Architectures

As shown in Figure 2, we identified two (i.e., a and b) software architectures in the collected firmware dataset and another three (i.e., c, d, and e) in the literature. **Bare-metal systems and unikernels** (a) run directly on the hardware at the highest (non-secure) privilege level. The RTOSs in such systems are only linked as a library OS, e.g., Mbed OS bare-metal profile [63]. We will discuss in I08 that over 99.44% of the 1,797 firmware belong to this category, including 66 firmware samples that use FreeRTOS and another 13 firmware use Mbed OS. **Monolithic kernels** (b) are the most common organization in microprocessor-based systems, e.g., Linux and Windows. Such systems run the kernel entirely at the

Table 2: Empirical Analysis of Security Features Adopted in Real-world Firmware

Hardware Vendor	Nordic (FirmXRay)		Other Nordic	TI		Telink		Dialog		NXP	Cypress	ST	Total
	#F	#D	#F	#F	#D	#F	#D	#F	#D	#F	#F	#F	#F
Readback Protection (I07)	17 2.21%	9 1.75%	15 2.17%	-	-	-	-	-	-	-	-	-	32 1.78%
Privilege Separation (I08)	8 1.04%	5 0.97%	2 0.29%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	10 0.56%
SVC for Library Call (I09)	753 98.04%	500 97.47%	690 100%	2 9.09%	1 5%	17 8.85%	17 14.17%	0 0%	0 0%	0 0%	2 2.99%	2 50%	1,466 81.58%
Stack Separation (I10)	49 6.38%	34 6.63%	82 11.88%	0 0%	0 0%	0 0%	0 0%	3 5.66%	1 2.78%	0 0%	0 0%	0 0%	134 7.46%
Stack Limit Register Usage (I10)	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%
Task Stack Ovf. Guard* (I10)	59 96.72%	4 80%	9 32.14%	-	-	-	-	-	-	-	-	-	68 76.40%
Memory Access Control (MPU) (I12)	0 0%	0 0%	4 0.58%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	1 100%	0 0%	0 0%	5 0.28%
Memory Access Control (sMPU) (I12)	19 2.47%	17 3.31%	0 0%	-	-	-	-	-	-	-	-	-	19 1.10%
Stack Canaries (I13)	0 0%	0 0%	1 0.14%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	1 0.06%
Proper Instruction Sync. Barriers† (I14)	30 36.59%	16 27.12%	68 40%	-	-	-	-	0 0%	0 0%	-	-	-	98 34.88%

#F: Number of firmware, #D: Number of devices, -: Not applicable, *: The percentage is only based on firmware that use RTOS, †: The percentage is only based on firmware that update CONTROL with the MSR instruction.

privileged level, and applications run in (unprivileged) user space. However, only 0.56% of the firmware samples in our dataset fall into this category. **Exokernels** (c) run at the highest privilege level, virtualizing and allocating resources to RTOSs or bare-metal applications running at a lower privilege level. We will discuss two software-based exokernel projects, Hermes [64] and MultiZone [65], and two Cortex-M TrustZone-based exokernel projects, ILTZVisor [66, 67] and SBIs [68], in **D05. Dual-world systems** (d), which are enabled by TrustZone-M, run RTOSs and applications in the non-secure state, whereas secure OS/services run in the secure state. The Trusted Firmware for Cortex-M (TF-M) [69] is a reference implementation of this architecture. **Multi-world systems** (e) enable multiple equally-secure TEEs. We will discuss uTango [70], one prominent example of a multi-world TEE implementation leveraging TrustZone-M in **D06**.

Insights

- Despite the research progress towards more secure architectures for Cortex-M systems, a large number of the real-world firmware in our dataset are simply bare-metal systems and unikernels.

4.2 Architectural Issues

Software architectural issues refer to common limitations and flaws we found in real-world firmware.

Lack of Privilege Management

I08. No or weak privilege separation: As shown in Table 2, only 10 out of 1,797 samples in our dataset execute some code at the unprivileged level, and the others execute entirely at the privileged level. Due to the lack of spatial isolation and privilege separation, a bug anywhere may compromise the whole system, even reverting MPU settings.

I09. SVC repurposing: The SVC instruction is designed to escalate the execution level; however, executing this instruction at the privileged level also transfers the control to the SVC handler. Surprisingly, we find that 1,466 (81.58%) samples run everything at the privileged level and repurpose this feature to call library APIs, e.g., Nordic SoftDevice [71], in-

stead of privilege escalation. The behavior is consistent across vendors, e.g., Nordic, TI, Telink, Cypress, and ST.

Lack of Memory Protections

I10. No or weak stack separation: RTOSs, such as FreeRTOS [72] and Zephyr [73], support multi-tasking, so each task has its own stack. However, stack separation between the kernel and application is rarely used in bare-metal firmware. Armv8-M also introduces stack limit registers (PSPLIM and MSPLIM) to delimit the boundaries of stacks. However, no firmware in our dataset has been used them.

RTOS Implementations: We found that only a few RTOSs protect tasks' stacks, and only Zephyr optionally supports using stack limit registers. When stack guard is enabled, FreeRTOS [74] and Mbed OS [75] insert a predefined delimiter to mark the boundary of each task's stack. Zephyr can use either PSPLIM or an MPU-configured memory guard to prevent overwriting beyond a task's stack [76].

Empirical Analysis on Real-world Firmware: 10 samples that adopt privilege separation (discussed in **I08**) leverage both the MSP- and PSP-based stacks. In addition, another 124 samples use both the MSP- and PSP-based stacks without privilege separation. All other samples (1,663; 92.54%) only adopt a single MSP-based stack. 59 of the 66 FreeRTOS-based firmware samples and 7 of the 13 Mbed OS-based firmware samples use task stack overflow guards.

I11. Secure state exception stack frame manipulation: CVE-2020-16273 shows that the non-secure state software may manipulate the secure stacks and hijack the secure control flow if the secure software does not properly initialize the secure stacks. To this end, an attacker creates a fake exception return stack frame to deprivilege an interrupt.

I12. No or weak memory access control; executable stack: Despite the presence of MPU, previous research suggests that it is rarely utilized in most real-world systems [77-79]. We confirm that 1,773 of the 1,797 firmware in our dataset do not use MPU, which means the *code*, *SRAM*, and *RAM* regions are executable and malicious code can read and write arbitrary memory. Out of the 24 firmware that use MPU in our dataset, five use the MPU defined by Arm. The remaining 19 use a vendor-specific implementation, i.e., Nordic's simplified

MPU (sMPU) [80], which only supports a subset of MPU features. Specifically, sMPU only supports read and write permissions with two protection domains.

I13. No or weak stack canary: Stack canary implementation involves initializing the canary value, runtime verification, and handling mismatches. The compiler and libraries manage the latter two, with the system initializing the canary value. In the standard C libraries (libc), the value of the stack canary is taken from a global variable `__stack_chk_guard`. In modern OSs, the value of the canary is randomly initialized when a process is created. However, embedded systems often use a fixed canary value post-compilation or boot [81]. Notably, there is only one `__stack_chk_guard` for the entire physical address space. We found that only *one* of the 1,797 firmware samples in our dataset adopts it.

I14. Missing barrier instructions: Barrier instructions, including data memory barrier (DMB), data synchronization barrier (DSB), and instruction synchronization barrier (ISB), guarantee that system configurations take effect before any memory operations [82]. The omission of them is unlikely to cause any issues on most Cortex-M MCUs because they do not have out-of-order execution and branch prediction capabilities. For MCUs that do have such capabilities, e.g., M7, this may lead to similar vulnerabilities that were discovered on microprocessors [83–85]. To check if barriers are set in firmware, for any CONTROL register update, we verify if there is an ISB instruction in its ten subsequent instructions. Our analysis shows that only 98 of the 281 firmware samples (34.88%) that update the CONTROL register use the ISB instruction thereafter. However, as we cannot confirm which architecture those firmware are using, it is unclear whether the missing barrier instructions will cause issues or not.

Insights

- The real-world firmware samples in our dataset barely use the security features of Cortex-M and largely lack the security mitigations that are widely deployed on modern microprocessor-based systems.
- Some software- and compiler-based mitigations, e.g., stack canaries, are less effective on MCU-based systems and should be redesigned.

5 Software Implementation Issues

Table 3 presents the numbers of Cortex-M related CVEs affecting nine hardware vendors, seven RTOSs, and two TLS libraries. We break down the number based on CVSS scores [86]. As shown in Table 3, the majority of CVEs (53.85%) affecting hardware vendors are classified as “medium” severity, while the majority of CVEs affecting RTOSs (78.07%) are categorized as either “critical” or “high”. We use a bug classification system proposed in [4] to characterize them into three major classes, i.e., validation, functional, and extrinsic. We summarize the results in Table 4, where we

Table 3: Distribution of disclosed Cortex-M related CVEs (2017 - 2023)

HW Vendor/RTOS/Lib	Critical	High	Medium	Low	Total
Arm	0 0%	4 57.14%	2 28.57%	1 14.29%	7 1.99%
Microchip Technology	1 14.29%	2 28.57%	4 57.14%	0 0%	7 1.99%
Silicon Labs	6 40.00%	2 13.33%	6 40.00%	1 6.67%	15 4.27%
NXP Semiconductors	1 7.69%	6 46.15%	6 46.15%	0 0%	13 3.70%
ST Microelectronics	2 12.50%	2 12.50%	12 75.00%	0 0%	16 4.56%
Cypress Semiconductor	0 0%	6 50.00%	6 50.00%	0 0%	12 3.42%
Gigadevice	0 0%	0 0%	6 100.00%	0 0%	6 1.71%
Texas Instruments	0 0%	6 54.55%	5 45.45%	0 0%	11 3.13%
Nordic	0 0%	2 50.00%	2 50.00%	0 0%	4 1.14%
Subtotal (HW vendors)	10 10.99%	30 32.97%	49 53.85%	2 2.20%	91 25.93%
FreeRTOS	3 15.79%	9 47.39%	7 36.84%	0 0%	19 5.41%
CMSIS RTOS2	1 100.00%	0 0%	0 0%	0 0%	1 0.28%
Mbed OS	6 60.00%	4 40.00%	0 0%	0 0%	10 2.85%
Zephyr	17 23.61%	36 50.00%	18 25.00%	1 1.39%	72 20.51%
RIOT-OS	10 33.33%	18 60.00%	2 6.67%	0 0%	30 8.55%
Contiki-ng	16 39.02%	18 43.90%	7 17.07%	0 0%	41 11.68%
Azure	5 35.71%	3 21.43%	5 35.71%	1 7.14%	14 3.99%
Subtotal (RTOSs)	58 31.01%	88 47.06%	39 20.86%	2 1.07%	187 53.28%
Mbed TLS	6 20.69%	12 41.38%	11 37.93%	0 0%	29 8.26%
WolfSSL	10 22.73%	14 31.82%	20 45.45%	0 0%	44 12.54%
Subtotal (Libs)	16 21.92%	26 35.62%	31 42.47%	0 0%	73 20.80%
Total	84 23.93%	144 41.03%	119 33.90%	4 1.14%	351

further provide a breakdown of bugs based on the functional and the software components.

5.1 Validation bugs

Validation bugs refer to bugs that mishandle or improperly validate input and output data. Examples are out-of-bounds read and write and improper parameter validation. They are frequently exploited for arbitrary write and read, allowing attackers to steal/overwrite sensitive information, execute remote code, or cause a denial of service.

I15. Validation bugs in communication components: Table 4 shows that 57.78% of validation bugs affect communication stacks, e.g., Bluetooth and TCP/IP implementations. For instance, FreeRTOS has a DNS poisoning bug that does not check if a DNS answer matches an outgoing query (CVE-2018-16598). Open-source libraries that are heavily used by Cortex-M systems, such as Mbed TLS or WolfSSL, also have 42 validation bugs.

I16. Validation bugs in device drivers: Device drivers are exposed to attackers through physically-accessible peripherals, e.g., the USB interface. We found 25 bugs that affect two hardware vendors and two RTOSs in this category. For instance, the buffer overread bug of the NXP Kinetis K82 USB driver can be leveraged to access the flash (CVE-2021-44479). The USB driver in Zephyr also has a buffer overflow bug that allows a USB-connected host to cause possible remote code execution (CVE-2020-10019).

I17. Validation bugs in dynamic memory allocations: Embedded systems commonly implement custom allocators rather than using the standard heap implementations in the Libc [16]. Bugs in heap management can result in a system crash or arbitrary code execution. For example, NXP’s SDK, RIOT-OS, Mbed OS, and CMSIS RTOS are vulnerable to integer overflows in their allocator functions [87].

Table 4: Distribution of Cortex-M software CVEs in different classes

Bug Class	Functions	Affected HW Vendors' SDKs	Affected RTOSs / TLS libs	#Bugs	
Validation	Communication	NXP (2), Microchip (5), ST (1), TI (9), Cypress (10), Silicon Labs (8), Nordic (3)	FreeRTOS (11), RIOT-OS (24), Mbed OS (7), Zephyr (32), Contiki-ng (39), Mbed TLS (14), wolfSSL (28)	193	57.78%
	Device Driver	TF-M (1), NXP (4), ST (7)	Zephyr (8), Azure (5)	25	7.48%
	Memory Allocation	NXP (1)	FreeRTOS (2), RIOT-OS (2), Mbed OS (2), CMSIS RTOS2 (1), Zephyr (2)	10	2.99%
	Context Switch	TF-M (2)	FreeRTOS(1), Zephyr (3)	6	1.79%
	Others	Silicon Labs(5), NXP (2), Microchip (1)	Contiki-ng (1), Zephyr (10), Azure (9)	28	6.59%
Functional	Protocol Implementation	TI (1), Cypress (2), Silicon Labs (2)	FreeRTOS (3), RIOT-OS (4), Zephyr (13), Mbed OS (1), Mbed TLS (3), wolfSSL (9)	38	11.38%
	Memory Access Control	TF-M (1), NXP (1), ST (1)	FreeRTOS (2), Zephyr (4), Contiki-ng (1)	10	2.99%
	Cryptography Primitive	TF-M (2), Microchip (1), ST (1)	Mbed TLS (4), wolfSSL (4)	12	3.59%
Extrinsic	Software Side-Channel	ST (1)	Mbed TLS (8), wolfSSL (5)	14	4.19%

I18. Validation bugs in context switch components: Bugs in these components have been exploited for privilege escalation. Zephyr uses signed integer comparison to validate the syscall number, so a negative number leads to privilege escalation (CVE-2020-10027). TF-M has a bug allowing for out-of-bounds write in an NSC function, which can lead to data leakage from the secure state (CVE-2021-27562).

I19. Validation bugs in other components: As discussed in I08, many systems execute entirely at the privileged level, and bugs in any component could lead to severe consequences. For example, a buffer overflow in FreeRTOS' shell can cause privileged code execution (CVE-2020-10023). Microchip's SDK has integer overflows that can be leveraged to access flash memory (CVE-2019-16127).

5.2 Functional bugs

Functional bugs refer to programming errors that do not correctly implement the intended design.

I20. Functional bugs in protocol implementations: 11.38% of the functional bugs are related to protocol implementations. For instance, the Bluetooth controller in the Cypress SDK uses a much shorter random number (than 128 bits) as the paring number, allowing the brute force of the random number to perform a man-in-the-middle attack during BLE pairing (CVE-2020-11957).

I21. Functional bugs in memory access control: Incorrect memory access control configurations, including for MPU and TrustZone, compromise isolation. We found eight bugs affecting one hardware vendor and two RTOSs in this category. For example, FreeRTOS has a bug that allows any code to set the system privilege level (CVE-2021-43997).

I22. Functional bugs in cryptography primitives: We found four bug reports in this category. For instance, RIOT-OS has a nonce reuse bug in its encryption function (CVE-2021-41061) and TF-M has a functional bug when cleaning up the memory allocated for a multi-part cryptographic operation, resulting in a memory leak (CVE-2021-32032). The implementations of PKCS #1 v1.5 padding for RSA in the ST (CVE-2020-20949) and Microchip (CVE-2020-20950) SDKs are vulnerable to the Bleichenbacher attack [88]. This vulnerability relies on the use of error messages or responses

from the server to gain information about the validity of the padding after decryption attempts.

5.3 Extrinsic bugs

Extrinsic bugs refer to defects that do not belong to the validation bugs or functional errors.

I23. Software side-channels: The Lucky 13 attack in Mbed TLS (CVE-2020-16150 and CVE-2020-36423) enables an attacker to deduce secret key information by exploiting time variations in the decryption process. This vulnerability, specifically found in Cipher Block Chaining (CBC) mode, is based on the time differences associated with padding length.

Insights

- Most Cortex-M based production systems are written in memory-unsafe languages, e.g., C [89], and they suffer from memory corruption vulnerabilities.
- Microcontrollers lack security mechanisms present in microprocessors for decades, such as privilege separation. Microcontroller developers may not realize the absence of features like an MMU can pose greater risks than microprocessors. Without privilege separation, any bug can be critical and compromise the entire system.

6 Security Research

We present a taxonomy of the security research projects on Cortex-M systems. Figure 3 depicts and summarizes the relationships among limitations, issues, and mitigations at different layers. Table 5 presents a comparative evaluation.

Addressing Hardware Issues

6.1 Addressing Microarchitectural and ISA Issues

D01. Mitigating microarchitectural attacks: To mitigate information leakage through timing side-channels (I01), BUSted [47] recommends disabling DMA during sensitive execution, and introducing random delays. To counter information leakage through long-term data remanence, UnTrustZone [48] suggests initializing SRAM at startup. To mitigate

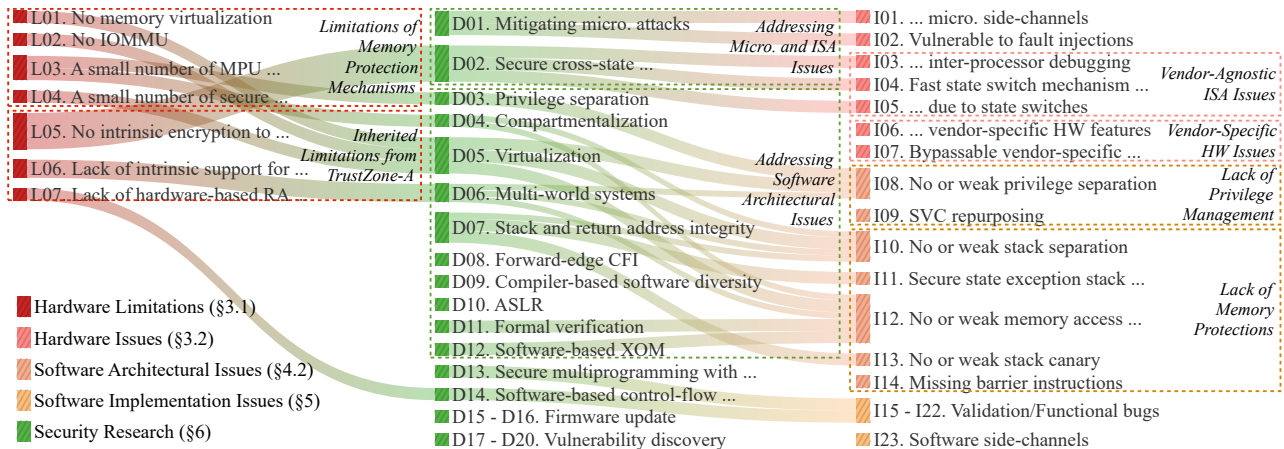


Figure 3: The relationships among the systematized Cortex-M related limitations, issues, and mitigations. The connections indicate the issues a research direction attempts to address and the limitations it needs to overcome. For instance, to address the issue of *no or weak privilege separation* (I08), mitigations (D03, D05, and D06) have been proposed, and they overcome some limitations (L01, L02, and L03). An interactive version of this figure can be accessed at our anonymized repo.

fault injection attacks (I02), one approach is the use of duplicate security-critical registers [131]. μ Glitch suggests introducing random delays in the execution code to complicate the parameter determination process for fault injections.

D02. Secure cross-state control and data interactions: One effective way to counteract privilege escalation through fast state switching (I03) is to add additional privilege checks. Ret2ns [53] suggests using address masking and MPU configuration checks to limit return targets from secure to non-secure state at the non-secure unprivileged level. In improving privilege management for inter-processor debugging (I04), Nailgun [55] employs MPU to restrict low-privilege access to debug registers. To mitigate information leakage during cross-state switches (I05), one approach is to implement authentication and authorization between the two states, as SeCRt [132] does for TrustZone-A. Secure Informer [95] and ShieldD [96] authenticate secure service calls from the non-secure state by verifying non-secure MPU configurations.

Addressing Software Architectural Issues

6.2 Separation of Privilege

Projects in this category provide different levels of granularity in isolating and confining software modules of *one* bare-metal system or *one* RTOS to address I08.

D03. Privilege separation: Solutions were proposed to automatically relegate RTOS tasks and bare-metal systems to the unprivileged level and use MPU to govern memory access. SAFER SLOTH [97] dispatches tasks as interrupt handlers and lowers the privilege level in the interrupt service routine. EPOXY [77] automatically identifies operations requiring privileged execution (e.g., MSR, move to system registers from general-purpose registers) in bare-metal systems. It then relegates the whole bare-metal system to the unprivileged level and instruments privilege escalation and relegation instructions around the operations requiring privileged execution.

These privilege separation approaches only introduce a small number of context switches, introducing low overhead.

D04. Compartmentalization: The projects on privilege separation (D03) only split a program into privileged and unprivileged parts. However, software modules at the same privilege level still reside in the same security and fault domain, resulting in coarse-grained memory access control (I12). Several compartmentalization solutions attempt to address this issue.

Compartmentalization with heavy context switches: uSFI compiler [98] instruments an entry function for each module and changes cross-module procedure calls to SVC instructions. ACES [79] instruments binaries to enforce inter-component isolation. MINION [99] automatically identifies the reachable memory regions of tasks through static analysis and enforces run-time memory access control. Because there are limited available MPU regions (L03), ACES and MINION propose schemes to merge the compartments. Compared to D03, compartmentalization introduces more context switches between modules; hence, the overhead is higher.

Compartmentalization with reduced context switches: To reduce the overhead introduced by compartmentalization, OPEC [100] leverages global variable shadowing to minimize the need for MPU regions and compartmentalizes programs to include only essential functions. EC [101] uses a formally verified microkernel and intra-kernel isolation to achieve compartmentalization. CRT-C [102] compartments an RTOS into kernel, threads, and device drivers and utilizes CheckedC [133] to restrict their programming capabilities.

DMA-enabled compartmentalization: The aforementioned compartmentalization solutions do not support DMA, leaving the system vulnerable to malicious DMA-capable devices due to the absence of an IOMMU (L02). D-Box [103] addresses this issue by introducing more secure MPU configurations and kernel extensions with explicit support for DMA operations.

Table 5: Comparative evaluation of system isolation and attack mitigation projects for Cortex-M (§6.2 - §6.8). The first column of the table lists the major defense mechanism proposed or adopted in a project.

Project	Year	Venue	Input (S: source code; B: binary) Target (B: bare-metal; R: RTOS) Prototype Implementation (ISA)	Limit. of Memory Prot. Mechanisms Inherited Limit. from TrustZone-A	§3.1	§3.2	§4.2	§5	Self-reported average or worst case overhead (%)																			
									Vendor-Agnostic Micro. Issues	Vendor-Agnostic ISA Issues	Vendor-specific Hardware Issues	Lack of Privilege Management	Lack Memory Protections	Validation Bugs	Functional Bugs	Extrinsic Bugs	MPU	Unprivileged Store/Load Instructions	TrustZone	DWT	FPB	Code, Binary Size Increase	Memory Overhead	Energy Consumption Overhead	Bare-metal Applications	RTOSs	BEBES [90]	CoreMark [91]
D01	BUSted [47]	2023	S&P	- - v8																								
	UnTrustZone [48]	2023	S&P	- - -																								
	μ Glitch [52]	2023	USENIX	- - -																								
D02	Nailgun [55]	2021	TDSC	S R v7																								
	ret2ns [53]	2023	DAC	S R v8																								
	Secure Informer [95]	2022	CPSS	S R v8																								
	ShieLD [96]	2022	TDSC	S R v8																								
D03	SAFER SLOTH [97]	2014	RTAS	S R v7																								
	EPOXY [77]	2017	S&P	S B v7																								
D04	uSFI [98]	2018	DATE	S R v7																								
	ACES [79]	2018	USENIX	S B v7																								
	MINION [99]	2018	NDSS	S R v7																								
	OPEC [100]	2022	EuroSys	S B v7																								
	EC [101]	2023	S&P	S B/R v7																								
	CRT-C [102]	2023	S&P	S R v7																								
	D-Box [103]	2022	NDSS	S R v7																								
	Hermes [64]	2018	MCSCA	S B/R v7																								
	MultiZone [65]	2020	EW	B B v7																								
	ILTZVisor [66,67]	2018	RTAS	S B/R v8																								
	SBIs [68]	2022	RTAS	S B/R v8																								
D05	RT-TEE [104]	2022	S&P	S R v8																								
	SafeTEE [105]	2022	DATE	S R v8																								
D06	uTango [70]	2022	Access	B B/R v8																								
	CaRE [106]	2017	RAID	B B v8																								
	Silhouette [107]	2020	USENIX	S B v7																								
	TZmCFI [108]	2020	IJPP	S R v8																								
	Kage [109]	2022	USENIX	S R v7																								
	SUM [110]	2023	C&S	S B v7																								
	SHERLOC [111]	2023	CCS	S B/R v8																								
	μ RAI [112]	2020	NDSS	S B v7																								
RIO [113]	2023	Access	S B v8																									
D07	Rendezvous [114]	2022	EuroS&P	S B v7/8																								
D08	HARM [115]	2022	EuroS&P	B B/R v8																								
	fASLR [116,117]	2022	ESORICS	S B v8																								
D09	Pip-MPU [118]	2023	IJESA	S B/R v7/8																								
D10	uXOM [119]	2019	USENIX	S R v7																								
	PicoXOM [120]	2020	SecDev	S B/R v7																								
D11	Tock [121]	2017	SOSP	S R v7																								
	DIAT [122]	2019	NDSS	S R v7																								
	LAPE [123]	2020	HPC	S B v7																								
	ISC-FLAT [124]	2023	RTAS	S B v8																								
D12	ARI [125]	2023	USENIX	S R v7																								
D13	ASSURED [126]	2018	TCAD	B R v8																								
	DisPatch [127]	2022	MobiSys	B R v7																								
	Shimware [128]	2023	RAID	B B/R v7																								
D14	HERA [129]	2021	NDSS	S R v7																								
	RapidPatch [130]	2022	USENIX	S B/R v7																								
D15																												
D16																												

v7: Armv7-M, v8: Armv8-M. -: Implemented defense techniques to address at least one issue or overcome one or more limitations in the corresponding category. +: Need specific hardware support. -: Not applicable. \downarrow and \uparrow represent small and big steps towards a similar goal, respectively.

6.3 Virtualization and Multi-world Systems

Solutions in this category enable or secure *multiple* bare-metal systems and RTOSs to run in an isolated fashion on *one* MCU.

D05. Virtualization: This technique can be used to support privilege separation (see 108).

Software-based virtualization: In those solutions, bare-metal systems and RTOSs execute at the unprivileged level

and the exokernel or an exception handler runs at the privileged level, as shown in Figure 2(c.1). A challenge is that the MSR and MRS (move to general-purpose registers from system registers) instructions fail silently without triggering any exceptions when executing at the unprivileged level, which can be addressed by replacing them with undefined instructions. Examples are Hermes [64] and MultiZone [65].

TrustZone-based virtualization: As shown in Figure 2(c.2), the exokernel or hypervisor runs at the highest privilege level (privileged secure state), and bare-metal systems and RTOSs can execute at the other three privilege levels. Prominent examples include ILTZVisor [66, 67] and SBIs [68].

D06. Multi-world systems: Multiple isolation environments enhance the isolation between system components.

Real-time and secure TrustZone-assisted dual-world system: De facto Cortex-M TEE solutions, e.g., TF-M [69], have availability and security issues, e.g., CVE-2021-32032. To address these issues, RT-TEE [104] ensures the real-time availability of both computation and I/O by adopting a policy-based event-driven hierarchical scheduler. SafeTEE [105] targets multi-core Cortex-M devices and isolates applications by assigning cores exclusively to them.

TrustZone-assisted multi-world system: As shown in Figure 2(e), TrustZone-assisted multi-world systems create multiple secure execution environments within the non-secure state to overcome L06. The uTango [70] kernel runs in the secure state at the privileged level, while other applications, services, and OSs are isolated in their non-secure state domains. Each domain has its own SAU configuration, which is only accessible by the uTango kernel.

6.4 Defeating Memory Corruption Attacks

The quest to defeat memory corruption attacks on Cortex-M systems (I15 - I19) largely includes adapting the security solutions for microprocessor-based systems to the resource and power constraint platforms.

D07. Stack and return address integrity: Stack and return addresses are a major attack vector (I10 and I11). Besides stack canaries (I13), there have been many attempts to maintain stack integrity on Cortex-M.

SafeStack: SafeStack [134] keeps unsafe local variables in a separate unsafe stack while keeping the return address in the regular stack. EPOXY implements an adapted SafeStack by (i) putting the unsafe stack on top of the RAM, (ii) making the stack grow up, and (iii) placing a region guard between the unsafe stack and other memory regions.

Shadow stack: Shadow stack [135] stores protected copies of return addresses. CaRE [106] and TZmCFI [108] use TrustZone-M and place the shadow stack in the secure state. To achieve low overhead, Silhouette [107] and Kage [109] restrict the writes to the shadow stack by transforming regular store instructions to unprivileged ones (STR*T). SUM [110] restricts unauthorized access to the shadow stack via the MPU.

Return address integrity: μ RAI [112] enforces the property of return address integrity by removing the need to spill return addresses to the stack. Rio [113] encrypts all return instructions in the firmware and instruments a runtime module to decrypt and execute these instructions. SHERLOC [111] introduces a reconstructed call stack (RCS) approach to ensure the matching of function calls and returns.

ROP gadget removal: Thumb-2 instruction set [136] allows the creation of ROP gadgets by jumping into the middle of a 32-bit instruction. To replace exploitable instructions, uSFI [98] and uXOM [119] convert all 32-bit instructions to equivalent 16-bit instruction sequences.

Stack sealing: To secure the secure world stack exception frame (I11), Arm recommends adding an integrity signature to the bottom of the secure exception stack frame [137].

D08. Forward-edge control-flow integrity (CFI): TZmCFI adopts LLVM's forward-edge CFI [138]. CaRE calculates the absolute target addresses, stores them in a branch table, and replaces all indirect branches with SVC instructions for runtime checking. Silhouette and Kage insert fixed CFI labels at the beginning of every address-taken function and check the label before the jump or the function call executes. SHERLOC maintains an indirect branch table to constrain the forward target within a predetermined CFG. InsectACIDE [139] retrieves a set of offline-computed legitimate transfer targets to validate the forward-edge transfers.

D09. Compiler-based software diversity: This technique randomizes the code and data of programs [140] to offer weakened probabilistic protection from code reuse attacks and data corruption attacks. However, the system memory layout remains the same after compilation. For instance, EPOXY [77] and Rendezvous [114] randomize the function order and add dummy variables to the .data and .bss regions.

D10. Address space layout randomization (ASLR): Without an MMU (L01) and the dynamic loading of programs, an ASLR solution on Cortex-M needs to increase entropy and decide when to perform the randomization. Both HARM [115] and fASLR [116] copy code from flash to RAM for execution and conduct randomization at the function level to increase entropy. HARM triggers randomization periodically by SysTick exceptions, while fASLR copies the function to a random location of RAM when it is called for the first time.

D11. Formal verification: Pip-MPU [118] introduces a formally verified kernel for Cortex-M. It features user-defined, MPU-guarded multiple isolation levels and is a refactored version of the MMU-based Pip protokernel [141]. It disables exceptions and puts the kernel inside the privileged level.

6.5 Defeating Software-based Code Disclosure

Projects in this category explore software-based XOM. Note that these efforts cannot address I07, in which a hardware debugger can disclose the contents in memory.

D12. Software-based XOM: uXOM [119] converts memory access instructions, excluding those that need privilege, into unprivileged ones (STR*T/LDR*T) and sets the code region as privileged access only. For the instructions that are not converted, uXOM instruments verification before them. PicoXOM [120] implements XOM by utilizing the address range matching feature of DWT with a much lower overhead.

The DWT, however, only has up to four comparators, which limits the number of configurable XOM regions.

Addressing Software Implementation Issues

6.6 Memory-safe Programming

Developing software in a manner that inherently reduces the likelihood of bugs and errors, thereby enhancing the overall safety and reliability of the system (I15 - I23).

D13. Secure multiprogramming with memory-safe languages: Tock [121] takes advantage of MPU and the type-safety features of Rust to build a multiprogramming system on Cortex-M. Rust encapsulates a large fraction of the Tock kernel with granular and type-safe interfaces.

6.7 Remote Attestation

Compared to the attack mitigation discussed in §6.4, remote attestation only detects adversarial presence.

D14. Software-based control-flow and data integrity attestation: Control-flow attestation (CFA) extends static attestation of code to run-time control-flow paths. DIAT [122] provides data integrity attestation and CFA of the code that generates and processes the data. LAPE [123] provides a coarse-grained CFA by grouping functions into compartments and attests the inter-compartment control-flow transfers. IS-FLAT [124] extends the aforementioned approaches to support interrupts, and ARI [125] formulates the property of real-time mission execution integrity.

Addressing Other Issues

6.8 Firmware Update

D15. Secure software update: ASSURED [126] allows a device to authenticate the source of firmware updates. DisPatch [127] allows end users to write patches in a domain-specific language, which DisPatch then automatically injects into the binary firmware. Shimware [128] investigates the challenges of updating monolithic firmware images with new security features. It automates finding safe injection locations and implementing self-checks to prevent modifications.

D16. Firmware hotpatching: While updating the whole firmware requires interrupting its normal execution (D15), hotpatching can fix minor issues at run-time. HERA [129] uses flash patch and breakpoint (FPB) to insert hardware breakpoints and redirects the instructions at breakpoints to the patch codes on RAM. However, FPB is only supported on M3 and M4 MCUs. To address this issue, RapidPatch [130] utilizes other hardware mechanisms, e.g., DWT.

6.9 Vulnerability Discovery

D17. Full firmware rehosting: One main challenge in emulating firmware on a desktop is how to model peripherals.

P²IM [142] observes the MMIO access pattern of each peripheral during firmware emulation. DICE [143] improves P²IM by additionally modeling DMA. Symbolic execution that models the return value of an MMIO read as a symbolic value has also been used in firmware emulation. Examples include Laelaps [144], μ Emu [145], Jetset [146], and Fuz-zware [21]. SEMu [147] extracts the condition-action rules to dynamically synthesize peripheral models. To sidestep the challenges in peripheral modeling, HALucinator [148] detects and replaces hardware abstraction layer functions of major chip vendors with host implementations. SAFIREFUZZ [149] executes embedded firmware as a Linux userspace process on systems sharing the same instruction set family as the targeted device. HOEDUR [150] employs multi-stream inputs, restructuring the traditional approach of firmware fuzzing into multiple, strictly typed, and cohesive streams, thereby enhancing mutation effectiveness and coverage.

D18. Hardware-in-the-loop rehosting: Full firmware rehosting techniques cannot accurately model more complex peripherals, such as the USB. Hardware-in-the-loop approaches address this challenge by redirecting I/O interactions to the physical hardware. The pioneer in this direction is Avatar [151], which is followed by its variants [152–155]. Instead of redirecting I/O interactions, Frankenstein [156] directly uses dumped firmware images from real devices to re-establish emulator states.

D19. On-device fuzzing: Existing rehosting solutions fall short in testing low-level drivers, either because they cannot provide the needed emulation fidelity or completely sidestep driver emulation. μ AFL [157] supports on-device fuzzing with the help of a debug dongle and ETM. Moreover, over-the-air fuzzing has been explored to find bugs in Bluetooth controllers [158, 159]. Lastly, to make bugs observable during fuzzing, μ SBS uses binary rewriting to instrument the firmware for sanitization checks [160]. SyzTrust [161] combines ETM for direct fuzzing on IoT devices with non-invasive state and code coverage tracking.

D20. Static methods: Static methods are typically geared toward detecting a particular type of bug. For instance, PASAN [162] considers concurrency issues with peripheral access. FirmXRay [15] aims to detect Bluetooth link layer vulnerabilities from bare-metal firmware. HEAPSTER [16] inspects common classes of heap vulnerabilities in Cortex-M monolithic firmware images.

6.10 Other research

Solutions and ideas for other architectures may be ported to or optimized for Cortex-M with proper modifications. For instance, the ideas of control-flow attestation (C-FLAT [163]) and operation execution integrity (OAT [164]) apply to Cortex-M naturally but were only implemented on Cortex-A. In addition, on Arm Cortex-A, pointer authentication code (PAC) has been utilized to enforce spatial (e.g., return ad-

dresses [165] and all pointers [166]) and temporal [167, 168] memory safety on userspace programs and the kernel [169].

7 Recommendations and Future Directions

7.1 Recommendations to research community

R01. Explore the pros and cons of new hardware features for security: The hardware features of Cortex-M exhibit streamlining and differences from its Cortex-A counterparts. This distinction spans from the microarchitectural layer to the ISA. For instance, TrustZone-M is a streamlined version of TrustZone-A, and the key management for PAC [14] on Cortex-M significantly differs from that on Cortex-A. All of these differences pose new challenges and opportunities in discovering their limitations and utilizing them for protections that were not possible before.

R02. Explore diverse IoT attack models and scenarios to identify new research problems and challenges: The application scenarios of Cortex-M systems, e.g., (i) deployed in the field and (ii) functionality implemented in privileged mode, present unique trust models and security research opportunities, which must be addressed with extra consideration for performance, memory, and energy cost [139, 170]. Future research should not only port the same defenses from microprocessor systems to Cortex-M systems but also address the challenges specific to MCUs.

R03. Investigate how to facilitate the practical adoption of academic research results: Compared to security research on Cortex-M, its deployment significantly lags behind. Operational research may focus on bridging the gap between security research outcomes and practical implementation. Such research may involve how to foster collaborations between researchers and industry practitioners, how to advocate for best practices, and how to promote educational programs to raise awareness about the importance of timely security deployment in Cortex-M systems.

7.2 Recommendations to developers

R04. Securing the network communications: As discussed in section §5, network protocol implementations often expose many vulnerabilities including validation and functional bugs. This is because these protocols are designed to work with microcontroller- and microprocessor-based systems, where developers may prioritize functionalities rather than security. Microprocessor-based systems have advanced security mechanisms like ASLR and DEP, which can handle most security issues. However, employing vulnerable protocols on microcontroller-based systems can lead to severe problems. Thus, microcontroller system developers should pay extra attention to security improvements, such as validating the input and output, utilizing security mechanisms discussed in section §6, and assessing the security of protocols before using them.

R05. Implement privilege separation or employ RTOSs with distinct privilege levels: We have observed that numerous real-world firmware was built upon vendor-supplied project templates, lacking privilege separation. We strongly recommend developers opt for templates incorporating essential security features or, alternatively, adopt RTOSs with different privilege levels as the foundational framework for their development.

R06. (Partially) Transition into memory-safe languages: A full transition into memory-safe languages, e.g., Rust, may not be immediately feasible for all Cortex-M projects due to factors like existing codebase, expertise, and project timelines [171]. Partial adoption of memory-safe languages, which provides a pragmatic and manageable approach toward embracing memory-safe languages' advantages within existing projects, can be highly valuable for enhancing the overall system robustness by mitigating memory-related issues like buffer overflows and null pointer dereferences.

R07. Enhance the synergy between developers and the security research community: During our efforts to systematize security research, we noticed that some issues lack corresponding defense mechanisms (Figure 3). This could be due to incomplete publication collections, as we primarily focused on security conferences. Nonetheless, similar to the varying levels of collaboration observed between the hacker community and academia [172], if developers and the security research community unite to share findings and insights, the security of microcontroller-based systems may be significantly improved.

8 Conclusion

We present a comprehensive systematization study of the hardware and software security of Cortex-M systems. It covers the Cortex-M hardware architectures, security-related features, limitations, and issues. The study includes by far the largest empirical analysis of real-world Cortex-M firmware, characterization of reported software bugs, and an overview of state-of-the-art security research in this area. Based on the insights, we develop a set of recommendations for the research community and MCU software developers.

Acknowledgment

This material is based upon work supported in part by National Science Foundation (NSF) grants (2237238, 2329704, 2207202, 2238264), a National Centers of Academic Excellence in Cybersecurity grant (H98230-22-1-0307), FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope UIDB/00319/2020, and a Cisco University Research Program Fund (71858473). Any opinions and findings expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

References

- [1] Arm, “Arm Partner Ecosystem Catalog,” [https://www.arm.com/partners/catalog/results#sort=date%20descending&f:armip=\[Cortex-M\]](https://www.arm.com/partners/catalog/results#sort=date%20descending&f:armip=[Cortex-M]).
- [2] Arm, “The Arm ecosystem ships a record 6.7 billion Arm-based chips in a single quarter,” <https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter>.
- [3] —, “Arm Partners Have Shipped 200 Billion Chips,” <https://www.arm.com/blogs/blueprint/200bn-arm-chips>.
- [4] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, “Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems,” in *IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [5] Arm, “Armv6-M Architecture Reference Manual,” <https://developer.arm.com/documentation/ddi0419/c/Application-Level-Architecture/The-Armv6-M-Instruction-Set/About-the-instruction-set>.
- [6] —, “Armv7-M Architecture Reference Manual,” <https://developer.arm.com/documentation/ddi0403/ed>.
- [7] —, “Armv8-M Architecture Reference Manual,” https://developer.arm.com/documentation/ddi0553/latest?_ga=2.1957362.2138159006.1623856318-792272022.1611588763.
- [8] J. Yiu, “ARMv8-M architecture technical overview,” *ARM white paper*, 2015.
- [9] Arm, “Armv8-M Memory Protection Unit,” <https://developer.arm.com/documentation/100699/0100>.
- [10] —, “Cortex-M23 Technical Reference Manual,” <https://developer.arm.com/documentation/ddi0550/>.
- [11] —, “Cortex-M33 Technical Reference Manual,” <https://developer.arm.com/documentation/100230/>.
- [12] —, “Cortex-M55 Technical Reference Manual,” <https://developer.arm.com/documentation/101051/>.
- [13] —, “TrustZone technology for the Armv8-M architecture Version 2.1,” <https://developer.arm.com/documentation/100690/latest/>.
- [14] —, “Armv8.1-M Pointer Authentication and Branch Target Identification Extension,” <https://community.arm.com/developer/ip-products/processors/b/processor-s-ip-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension>.
- [15] H. Wen, Z. Lin, and Y. Zhang, “FirmXRy: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware,” in *ACM Conference on Computer and Communications Security (CCS)*, 2020.
- [16] F. Gritti, F. Pagani, I. Grishchenko, L. Dresel, N. Redini, C. Kruegel, and G. Vigna, “HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images,” in *IEEE Symposium on Security and Privacy (S&P)*, 2022.
- [17] “ucsb-seclab/monolithic-firmware-collection,” <https://github.com/ucsb-seclab/monolithic-firmware-collection>.
- [18] “ThePBone/GalaxyBudsFirmwareDownloader,” https://github.com/ThePBone/GalaxyBudsFirmwareDownloader/tree/master/firmware_archive.
- [19] “grant-h/ShannonBaseband,” <https://github.com/grant-h/ShannonBaseband/tree/master/firmware>.
- [20] J. Friebertshäuser, F. Kosterhon, J. Classen, and M. Hollick, “Polypyus—the firmware historian,” in *Workshop on Binary Analysis Research (BAR)*, vol. 2021, 2021, p. 21.
- [21] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, “Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing,” in *USENIX Security Symposium*, 2022.
- [22] “Nordic semiconductor,” <https://www.nordicsemi.com/>.
- [23] “Texas instruments,” <https://www.ti.com/>.
- [24] “STMicroelectronics,” https://www.st.com/content/st_com/en.html.
- [25] “Telink Semiconductor,” <https://www.telink-semi.com/>.
- [26] “Dialog Semiconductor,” <https://www.dialog-semiconductor.com/>.
- [27] “NXP Semiconductors,” <https://www.nxp.com/>.
- [28] “Cypress Semiconductor,” <https://www.infineon.com/>.
- [29] “Ghidra,” <https://ghidra-sre.org/>.
- [30] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “ByteWeight: Learning to recognize functions in binary code,” in *USENIX Security Symposium*, 2014.
- [31] Arm, “ARM CMSIS RTOS2,” https://github.com/ARM-software/CMSIS_5/blob/2ccc9e92637fe80f50d5e8b9d503bb715112fe69/CMSIS/RTOS2/RTX/RTX5.scvd.

- [32] R. Yu, F. Del Nin, Y. Zhang, S. Huang, P. Kaliyar, S. Zalko, M. Conti, G. Portokalidis, and J. Xu, "Building Embedded Systems Like It's 1996," in *Network and Distributed System Security Symposium (NDSS)*, 2022.
- [33] MITRE, "CVE database," <https://cve.mitre.org/>.
- [34] "MCU Market Size In 2022 By Fastest Growing Companies," <https://www.marketwatch.com/press-release/iot-microcontroller-mcu-market-size-2022-industry-analysis-by-growth-share-trends-demand-segment-s-opportunities-and-forecast-2028-2022-09-19>.
- [35] Market Growth Reports, "United States IoT Operating Systems Market Report & Forecast 2021-2027," <https://www.marketgrowthreports.com/united-states-iot-operating-systems-market-19250528>.
- [36] Arm, "Mbed OS TLS," <https://tls.mbed.org/>.
- [37] wolfSSL, "wolfSSL," <https://www.wolfssl.com/>.
- [38] NXP Semiconductors, "i.MX RT Crossover MCUs," <https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus:IMX-RT-SERIES>.
- [39] —, "MCUXpresso SDK API Reference Manual," https://mcuxpresso.nxp.com/api_doc/dev/1411/a00057.html.
- [40] J. Y. Afonso Santos, "SAU, IDAU, MPC and PPC. What's the difference?" <https://community.arm.com/support-forums/f/architectures-and-processors-forum/12065/sau-idau-mpc-and-ppc-what-s-the-difference/34873>.
- [41] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, 2009.
- [42] Linaro, "Trusted Firmware M (TFM) v1.3.0 source code," <https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tag/?h=TF-Mv1.3.0>.
- [43] "Arm Platform Security Architecture Security Model," https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Architect/DEN0079-PSA_SM_ALPHA-02.pdf.
- [44] "PSA Attestation API," https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Implement/IHI0085-PSA_Attestation_API-1.0.1-2.pdf.
- [45] D. McCann, C. Whitnall, and E. Oswald, "ELMO: Emulating Leaks for the Arm Cortex-M0 without Access to a Side Channel Lab," *IACR Cryptol. ePrint Arch.*, 2016.
- [46] S. Vafa, M. Masoumi, and A. Amini, "An efficient profiling attack to real codes of PIC16F690 and Arm Cortex-M3," *IEEE Access*, 2020.
- [47] C. Rodrigues, D. Oliveira, and S. Pinto, "BUSSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.
- [48] J. Mahmood and M. Hicks, "UnTrustZone: Systematic Accelerated Aging to Expose On-chip Secrets," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.
- [49] J. Obermaier and S. Tatschner, "Shedding too much Light on a Microcontroller's Firmware Protection," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.
- [50] J. Obermaier, M. Schink, and K. Moczek, "One exploit to rule them all? on the security of drop-in replacement and counterfeit microcontrollers," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.
- [51] M. Schink, A. Wagner, F. Unterstein, and J. Heyszl, "Security and Trust in Open Source Security Tokens," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021.
- [52] X. M. Saß, R. Mitev, and A.-R. Sadeghi, "Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M," *USENIX Security*, 2023.
- [53] Z. Ma, X. Tan, L. Ziarek, N. Zhang, H. Hu, and Z. Zhao, "Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense," in *ACM/IEEE Design Automation Conference*, 2023.
- [54] Z. Ning and F. Zhang, "Understanding the Security of Arm Debugging Features," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [55] Z. Ning, C. Wang, Y. Chen, F. Zhang, and J. Cao, "Revisiting arm debugging features: Nailgun and its defense," *Transactions on Dependable and Secure Computing (TDSC)*, 2021.
- [56] Sultan Qasim Khan, "Whitepaper: Microcontroller Readback Protection: Bypasses and Defenses," *Technical Report*, 2020.

- [57] Kris Brosch, “Firmware dumping technique for an Arm Cortex-M0 SoC,” <https://blog.includesecurity.com/2015/11/firmware-dumping-technique-for-an-arm-cortex-m0-soc/>.
- [58] Nordic Semiconductor, “nRF52832 Objective Product Specification,” https://infocenter.nordicsemi.com/pdf/nRF52832_OPS_v0.6.3.pdf.
- [59] STMicroelectronics, “Proprietary code read-out protection on microcontrollers of the STM32F4 Series,” https://www.st.com/resource/en/application_note/an4701-proprietary-code-readout-protection-on-microcontrollers-of-the-stm32f4-series-stmicroelectronics.pdf.
- [60] NXP Semiconductors, “Using the Kinetis Flash Execute-Only Access Control Feature,” <https://www.nxp.com/docs/en/application-note/AN5112.pdf>.
- [61] Texas Instruments, “Tiv TM4C123GH6PM Microcontroller,” <https://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf>.
- [62] M. Schink and J. Obermaier, “Taking a Look into Execute-Only Memory,” in *Workshop on Offensive Technologies (WOOT)*, 2019.
- [63] “Mbed OS,” <https://os.mbed.com/mbed-os/>.
- [64] N. Klingensmith and S. Banerjee, “Hermes: A real time hypervisor for mobile and iot systems,” in *International Workshop on Mobile Computing Systems & Applications*, 2018.
- [65] S. Pinto and C. Garlati, “Multi zone security for arm cortex-m devices,” in *Embedded World Conference*, 2020.
- [66] H. M. E. Araújo, “ILTZVisor: a lightweight TrustZone-assisted hypervisor for low-end Arm devices,” Ph.D. dissertation, University of Minho, 2018.
- [67] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, “Virtualization on trustzone-enabled microcontrollers? voilà!” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.
- [68] R. Pan and G. Parmer, “SBIs: Application Access to Safe, Baremetal Interrupt Latencies,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.
- [69] “Trusted Firmware-M,” <https://www.trustedfirmware.org/projects/tf-m>.
- [70] D. Oliveira, T. Gomes, and S. Pinto, “uTango: an open-source TEE for IoT devices,” *IEEE Access*, 2022.
- [71] Nordic Semiconductor, “SoftDevices,” https://infocenter.nordicsemi.com/topic/ug_gsg_ses/UG/gsg/softdevices.html.
- [72] FreeRTOS, “RTOS Fundamentals - Context Switching,” <https://www.freertos.org/implementation/a00006.html>.
- [73] Zephyr Project Documentation, “Arm Cortex-M Developer Guide - Thread context switching,” https://docs.zephyrproject.org/3.0.0/guides/arch/arm_cortex_m.html#thread-context-switching.
- [74] “The FreeRTOS Kernel,” <https://www.freertos.org/RTOS.html>.
- [75] Arm, “API and RTX Reference Implementation - Configure RTX v5,” https://www.keil.com/pack/doc/CMSIS/RTOS2/html/config_rtx5.html.
- [76] Zephyr Project Documentation, “Arm Cortex-M Developer Guide - Memory protection features,” https://docs.zephyrproject.org/3.0.0/guides/arch/arm_cortex_m.html#memory-protection-features.
- [77] A. A. Clements, N. S. Almkhathub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, “Protecting bare-metal embedded systems with privilege overlays,” in *IEEE Symposium on Security and Privacy (S&P)*, 2017.
- [78] W. Zhou, L. Guan, P. Liu, and Y. Zhang, “Good Motive but Bad Design: Why Arm MPU Has Become an Outcast in Embedded Systems,” *arXiv preprint arXiv:1908.03638*, 2019.
- [79] A. A. Clements, N. S. Almkhathub, S. Bagchi, and M. Payer, “ACES: Automatic Compartments for Embedded Systems,” in *USENIX Security Symposium*, 2018.
- [80] Nordic Semiconductor, “nRF51 Series Reference Manual,” https://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.pdf.
- [81] X. Tan, S. Mohan, M. Armanuzzaman, Z. Ma, G. Liu, A. Eastman, H. Hu, and Z. Zhao, “Is the Canary Dead? On the Effectiveness of Stack Canaries on Microcontroller Systems,” in *ACM/SIGAPP Symposium On Applied Computing (SAC)*, 2024.
- [82] “Arm Cortex-M Programming Guide to Memory Barrier Instructions,” <https://developer.arm.com/documentation/dai0321/latest/>.
- [83] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, “Meltdown: Reading kernel memory from user space,” in *USENIX Security Symposium*, 2018.

- [84] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [85] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: attacking Arm pointer authentication with speculative execution," in *International Symposium on Computer Architecture (ISCA)*, 2022.
- [86] METRE, "Common Vulnerability Scoring System v3.1: User Guide," <https://www.first.org/cvss/v3.1/user-guide>.
- [87] "Multiple RTOS (Update E) | CISA," <https://www.cisa.gov/uscert/ics/advisories/icsa-21-119-04>.
- [88] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1," in *Annual International Cryptology Conference*. Springer, 1998.
- [89] Embedded by AspenCore, "2019 embedded markets study," https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.
- [90] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.
- [91] "CoreMark," <https://www.eembc.org/coremark>.
- [92] "CoreMark-Pro," <https://www.eembc.org/coremark-pro/>.
- [93] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, 1984.
- [94] "Embench: A Modern Embedded Benchmark Suite," <https://www.embench.org/>.
- [95] A. K. Iannillo, S. Rivera, D. Suciu, R. Sion, and R. State, "An REE-independent Approach to Identify Callers of TEEs in TrustZone-enabled Cortex-M Devices," in *ACM Cyber-Physical System Security Workshop (CPSS)*, 2022.
- [96] A. Khurshid, S. D. Yalaw, M. Aslam, and S. Raza, "ShieLD: Shielding Cross-zone Communication within Limited-resourced IoT Devices running Vulnerable Software Stack," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2022.
- [97] D. Danner, R. Müller, W. Schröder-Preikschat, W. Hofer, and D. Lohmann, "Safer Sloth: Efficient, hardware-tailored memory protection," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.
- [98] Z. B. Aweke and T. Austin, "uSFI: Ultra-lightweight software fault isolation for IoT-class devices," in *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.
- [99] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing Real-Time Microcontroller Systems through Customized Memory View Switching," in *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [100] X. Zhou, J. Li, W. Zhang, Y. Zhou, W. Shen, and K. Ren, "OPEC: operation-based security isolation for bare-metal embedded systems," in *European Conference on Computer Systems*, 2022.
- [101] A. Khan, D. Xu, and D. Tian, "Ec: Embedded systems compartmentalization via intra-kernel isolation," in *Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023.
- [102] —, "Low-cost privilege separation with compile time compartmentalization for embedded systems," in *Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023.
- [103] A. Mera, Y. H. Chen, R. Sun, E. Kirda, and L. Lu, "D-Box: DMA-enabled Compartmentalization for Embedded Applications," in *Network and Distributed System Security Symposium (NDSS)*, 2022.
- [104] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "RT-TEE: Real-time System Availability for Cyber-physical Systems using Arm TrustZone," in *IEEE Symposium on Security and Privacy (SP)*, 2022.
- [105] M. Schönstedt, F. Brasser, P. Jauernig, E. Stapf, and A.-R. Sadeghi, "SafeTEE: combining safety and security on ARM-based microcontrollers," in *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022.
- [106] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017.
- [107] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *USENIX Security Symposium*, 2020.

- [108] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M," *International Journal of Parallel Programming*, 2020.
- [109] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage," in *USENIX Security Symposium*, 2022.
- [110] W. Choi, M. Seo, S. Lee, and B. B. Kang, "SuM: Efficient Shadow Stack Protection on ARM Cortex-M," *Computers & Security*, 2023.
- [111] X. Tan and Z. Zhao, "SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.
- [112] N. S. Almahdhub, A. A. Clements, S. Bagchi, and M. Payer, " μ RAI: Securing embedded systems with return address integrity," in *Network and Distributed System Security Symposium (NDSS)*, 2020.
- [113] B. Kim, K. Lee, W. Park, J. Cho, and B. Lee, "RIO: Return Instruction Obfuscation for Bare-metal IoT Devices," *IEEE Access*.
- [114] Z. Shen, K. Dharsee, and J. Criswell, "Rendezvous: Making Randomization Effective on MCUs," in *Annual Computer Security Applications Conference (ACSAC)*, 2022.
- [115] J. Shi, L. Guan, W. Li, D. Zhang, P. Chen, and N. Zhang, "HARM: Hardware-Assisted Continuous Re-randomization for Microcontrollers," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2022.
- [116] L. Luo, X. Shao, Z. Ling, H. Yan, Y. Wei, and X. Fu, "faslr: Function-Based ASLR via TrustZone-M and MPU for Resource-Constrained IoT Systems," *IEEE Internet of Things Journal*, 2022.
- [117] X. Shao, L. Luo, Z. Ling, H. Yan, Y. Wei, and X. Fu, "faslr: Function-based aslr for resource-constrained iot systems," in *European Symposium on Research in Computer Security (ESORICS)*, 2022.
- [118] N. Dejon, C. Gaber, and G. Grimaud, "Pip-MPU: Formal verification of an MPU-based separation kernel for constrained devices," *International Journal of Embedded Systems and Applications*, 2023.
- [119] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uXOM: Efficient eExecute-Only Memory on Arm Cortex-M," in *USENIX Security Symposium*, 2019.
- [120] Z. Shen, K. Dharsee, and J. Criswell, "Fast Execute-Only Memory for Embedded Systems," in *IEEE Secure Development (SecDev)*, 2020.
- [121] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Panunto, P. Dutta, and P. Levis, "Multiprogramming a 64kb computer safely and efficiently," in *ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2017.
- [122] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter, "DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems," in *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [123] D. Huo, Y. Wang, C. Liu, M. Li, Y. Wang, and Z. Xu, "LAPE: A Lightweight Attestation of Program Execution Scheme for Bare-Metal Systems," in *IEEE HPC-C/SmartCity/DSS*, 2020.
- [124] A. J. Neto and I. d. O. Nunes, "ISC-FLAT: On the Conflict Between Control Flow Attestation and Real-Time Operations," 2023.
- [125] J. Wang, Y. Wang, A. Li, Y. Xiao, R. Zhang, W. Lou, Y. T. Hou, and N. Zhang, "Ari: Attestation of real-time mission execution integrity," 2023.
- [126] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "ASSURED: Architecture for secure software update of realistic embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.
- [127] T. Kim, A. Ding, S. Etigowni, P. Sun, J. Chen, L. Garcia, S. Zonouz, D. Xu, and D. Tian, "Reverse engineering and retrofitting robotic aerial vehicle control firmware using dispatch," in *International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2022.
- [128] E. Gustafson, P. Grosen, N. Redini, S. Jha, A. Continella, R. Wang, K. Fu, S. Rampazzi, C. Kruegel, and G. Vigna, "Shimware: Toward Practical Security Retrofitting for Monolithic Firmware Images," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2023.
- [129] C. Niesler, S. Surminski, and L. Davi, "HERA: Hotpatching of Embedded Real-time Applications," in *Network and Distributed System Security Symposium (NDSS)*, 2021.
- [130] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li, "RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices," in *USENIX Security Symposium*, 2022.

- [131] A. Barengi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, “Countermeasures against fault attacks on software implemented AES: effectiveness and cost,” in *Workshop on Embedded Systems Security (WESS)*, 2010.
- [132] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, “SeCRet: Secure Channel between Rich Execution Environment and Trusted Execution Environment,” in *Network and Distributed System Security Symposium (NDSS)*, 2015.
- [133] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, “Checked C: making C safe by extension,” in *Cybersecurity Development (SecDev)*. IEEE, 2018.
- [134] P. Larsen and A.-R. Sadeghi, *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, 2018, ch. Code-pointer integrity.
- [135] N. Burow, X. Zhang, and M. Payer, “SoK: Shining light on shadow stacks,” in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [136] Arm, “Arm Architecture Reference Manual Thumb-2 Supplement,” <https://class.ece.iastate.edu/cpre288/resources/docs/Thumb-2SupplementReferenceManual.pdf>.
- [137] —, “Armv8-M Stack Sealing Vulnerability,” <https://developer.arm.com/support/arm-security-updates/armv8-m-stack-sealing>.
- [138] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing forward-edge control-flow integrity in GCC & LLVM,” in *USENIX Security Symposium*, 2014.
- [139] Y. Wang, C. Lemieux Mack, X. Tan, N. Zhang, Z. Zhao, S. Baruah, and B. C. Ward, “InsectACIDE: Debugger-Based Holistic Asynchronous CFI for Embedded System,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024.
- [140] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “SoK: Automated software diversity,” in *IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [141] N. Jomaa, D. Nowak, and P. Torrini, “Formal Development of the Pip Protokernel,” *ENTROPY*, 2018.
- [142] B. Feng, A. Mera, and L. Lu, “P²IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling,” in *USENIX Security Symposium*, 2020.
- [143] A. Mera, B. Feng, L. Lu, E. Kirda, and W. Robertson, “DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis,” in *IEEE Symposium on Security and Privacy (S&P)*, 2021.
- [144] C. Cao, L. Guan, J. Ming, and P. Liu, “Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation,” in *Annual Computer Security Applications Conference (ACSAC)*, 2020.
- [145] W. Zhou, L. Guan, P. Liu, and Y. Zhang, “Automatic Firmware Emulation through Invalidity-guided Knowledge Inference,” in *USENIX Security Symposium*, 2021.
- [146] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, “Jetset: Targeted Firmware Rehosting for Embedded Systems,” in *USENIX Security Symposium*, 2021.
- [147] W. Zhou, L. Zhang, L. Guan, P. Liu, and Y. Zhang, “What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation,” in *ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [148] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, “HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation,” in *USENIX Security Symposium*, 2020.
- [149] L. Seidel, D. Maier, and M. Muench, “Forming faster firmware fuzzers,” in *USENIX Conference on Security Symposium*, 2023.
- [150] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, “HOEDUR: embedded firmware fuzzing using multi-stream inputs,” in *USENIX Conference on Security Symposium*, 2023.
- [151] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, “AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems’ Firmwares.” in *Network and Distributed System Security (NDSS)*, 2014.
- [152] M. Muench, A. Francillon, and D. Balzarotti, “Avatar²: A Multi-target Orchestration Platform,” in *Workshop on Binary Analysis Research*, 2018.
- [153] K. Koscher, T. Kohno, and D. Molnar, “SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems,” in *USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

- [154] Corteggiani, Nassim and Camurati, Giovanni and Francillon, Aurélien, “Inception: System-wide security testing of real-world embedded systems software,” in *USENIX Security Symposium*, 2018.
- [155] N. Corteggiani and A. Francillon, “HardSnap: Leveraging Hardware Snapshotting for Embedded Systems Security Testing,” in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.
- [156] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, “Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets,” in *USENIX Security Symposium*, 2020.
- [157] W. Li, J. Shi, , F. Li, J. Lin, W. Wang, and L. Guan, “ μ AFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware,” in *IEEE/ACM International Conference on Software Engineering*, 2022.
- [158] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, “SweynTooth: Unleashing Mayhem over Bluetooth Low Energy,” in *USENIX Annual Technical Conference*, 2020.
- [159] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, “BRAKTOOTH: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing,” in *USENIX Security Symposium*, 2022.
- [160] M. Salehi, D. Hughes, and B. Crispo, “ μ SBS: Static binary sanitization of bare-metal embedded devices for fault observability,” in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 381–395.
- [161] Q. Wang, B. Chang, S. Ji, Y. Tian, X. Zhang, B. Zhao, G. Pan, C. Lyu, M. Payer, W. Wang *et al.*, “SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices,” 2023.
- [162] T. Kim, V. Kumar, J. Rhee, J. Chen, K. Kim, C. H. Kim, D. Xu, and D. J. Tian, “PASAN: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications,” in *USENIX Security Symposium*, 2021.
- [163] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, “C-FLAT: control-flow attestation for embedded systems software,” in *ACM Conference on Computer and Communications Security*, 2016.
- [164] Z. Sun, B. Feng, L. Lu, and S. Jha, “OAT: Attesting operation integrity of embedded devices,” in *IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [165] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, “PACStack: an Authenticated Call Stack,” in *USENIX Security Symposium*, 2021.
- [166] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, “PAC it up: Towards pointer integrity using ARM pointer authentication,” in *USENIX Security Symposium*, 2019.
- [167] R. M. Farkhani, M. Ahmadi, and L. Lu, “PTAuth: Temporal Memory Safety via Robust Points-to Authentication,” in *USENIX Security Symposium*, 2021.
- [168] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, “PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication,” in *ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [169] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, “In-Kernel Control-Flow Integrity on Commodity OSES using ARM Pointer Authentication,” in *USENIX Security Symposium*, 2022.
- [170] Z. Zhao, M. Armanuzzaman, X. Tan, and Z. Ma, “Trusted Execution Environments in Embedded and IoT Systems: A CactiLab Perspective,” in *IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 2024.
- [171] A. Sharma, S. Sharma, S. Torres-Arias, and A. Machiry, “Rust for Embedded Systems: Current State, Challenges and Open Problems,” *arXiv preprint arXiv:2311.05063*, 2023.
- [172] H. Bos, “NDSS 2024 Keynote - Corruption of Memory: Those who don’t know history are doomed to repeat it,” <https://www.youtube.com/watch?v=vhj2We2vjqs>.

Appendix

Our open-source repository contains extra information for researchers:

- A Cortex-M firmware analysis tool (in the `firmware_analysis` folder).
- A Cortex-M firmware database (in the `firmware_analysis` folder).
- Cortex-M hardware feature test suites (in the `hw_feature_test_suites` folder).
- Supplementary Material 1: Cortex-M Architecture in a Nutshell (`Background.pdf`).
- An interactive figure showcasing the relationships between Cortex-M limitations, issues, and mitigations (download `relations_interactive_fig.html`).
- A collection of Cortex-M-related CVEs in Google Spreadsheet.

Not Quite Write: On the Effectiveness of Store-Only Bounds Checking

Adriaan Jacobs
DistriNet, KU Leuven

Stijn Volckaert
DistriNet, KU Leuven

Abstract

Compiler-based memory safety enforcement for unsafe C/C++ code has historically suffered from prohibitively high overhead. Despite regular advances in compiler optimization and increasing hardware resources and hardware support, most applications require too many checks to guarantee complete memory safety at an acceptable performance level. Consequently, researchers often propose relaxed policies where not all memory accesses undergo equally rigorous checking. One common suggestion is to omit pointer validity checks for memory loads. This omission significantly reduces the number of necessary checks and, thus, overhead. Moreover, it should *only* sacrifice the detection of pure information disclosure vulnerabilities through invalid reads, which are left unchecked.

This work challenges the perceived security benefits of store-only bounds checking. We show that invalid reads often suffice to take control of memory writes and bypass store-only validity checks. We empirically demonstrate the problem on SoftBound and qualitatively analyze the impact on a broad scope of other work. We also perform a large-scale evaluation on 1,000 popular C/C++ repositories and show that real-world code readily satisfies the necessary preconditions for store-only bypasses. Finally, we briefly discuss possible defenses and adaptations that let complete bounds checkers regain a part of the store-only overhead reduction potential without dramatically losing security.

1 Introduction

Memory-unsafe programming languages continue to dominate the composition of our modern software stack, from bootloaders, Operating System (OS) kernels, and system libraries to user-facing applications like web servers and browsers. Programs written in these languages often contain memory errors such as out-of-bounds (OOB) accesses [70, 72] or use-after-free bugs (UAF) [71], which can be exploited by attackers to leak or corrupt sensitive data, or to force the victim program to execute attacker-chosen code [102].

Owing to these security risks, government bodies [80] and industry leaders [89] are increasingly pushing for more memory safety in critical infrastructure and systems-level software, encouraging the use of safe languages instead, such as Rust [22]. Software vendors have already adopted these recommendations for new software projects [61, 99]. However, for a vast amount of already existing C and C++ code, translating it into safer languages is not feasible any time soon [86], leaving a mountain of unsafe code currently deployed in production environments for which no clear solution exists.

Researchers and practitioners from academia and industry alike have come up with many attempts to minimize the security impact of this unsafety through compiler transformations that automatically *harden* the code against memory error exploitation, e.g., by inserting checks on memory accesses or indirect control flow transfers. One such approach, which has been thoroughly investigated for decades [97, 102], is to retrofit memory safety into these languages by (semi-)automatically instrumenting memory accesses with runtime checks that validate pointer bounds (*spatial memory safety*) and object lifetimes (*temporal memory safety*) [6, 9, 12, 14, 18, 25–27, 30–32, 35, 36, 42, 49, 51, 54, 57–59, 63, 66, 74, 77–79, 90, 93, 94, 103, 116, 120, 121]. We broadly refer to these memory safety enforcement mechanisms as “bounds checkers” for short.

The design and implementation of bounds checkers has been a long-standing and highly active area of research, fueled by the promise of strong memory safety but plagued by prohibitive run-time overhead and compatibility issues. Despite steady advances over time, from optimizing the storage structure of bounds and lifetime metadata [18, 39, 41, 67, 68, 75, 78, 111], to avoiding the branch predictor pollution of typical compare-and-branch instrumentation [6, 18, 36, 58], or maximally reducing the number of redundant checks through static code analysis and optimization [12, 18, 44, 45, 52, 66, 100, 108, 112, 117, 122], the overhead of comprehensive memory safety enforcement remains well outside the stringent performance budget of typ-

ical production deployments [102]. For this reason, some prior work proposes to deliberately sacrifice *some* security coverage to reduce the performance impact by selectively eliding validity checks on memory accesses whose protection is explicitly scoped out [15, 63, 75, 77, 78], or whose performance impact is considered disproportional to their security benefit [38, 50, 107].

One commonly suggested strategy is to place validity checks on memory writes alone [36, 63, 75, 77, 78, 81, 94], which significantly reduces performance overhead, as most programs tend to read memory far more often than they write to it [65, 81, 84, 115]. Naturally, this comes at the cost of leaving pure information disclosure vulnerabilities out of scope. Major security crises like Heartbleed demonstrate that such confidentiality breaches are not necessarily of lesser impact [87]. Still, they only represent a minority of possible attacks, while their mitigation frequently requires more than double the amount of validity checks [78, 81]. Hence, *store-only* bounds checking is often heralded as a straightforward option to curb overhead while keeping the vast majority of memory vulnerabilities at bay by ensuring that attackers can never abuse invalid memory writes to corrupt program memory.

Store-only checking, which allows read operations on out-of-bound locations and with dangling pointers, is sufficient to prevent all memory corruption-based security vulnerabilities.

Nagarakatte et al. [77]

In this work, we argue that **the intended integrity assurance of store-only bounds checking does not hold in practice**, as a direct consequence of the lack of protection on memory reads. In short, the core issue is that store-only bounds checkers do not suffice to secure the data and pointer flow of the program, while their protection guarantees assume they do. As just one striking consequence of this, we show that attackers can corrupt arbitrary memory locations using *protected* writes by loading valid pointers through invalid memory reads. Our key finding is that the substituted, invalidly loaded pointer will always pass the store-only validity check, *regardless of the bounds checker design or implementation*. In summary, we make the following contributions:

- We outline four types of attacks that can corrupt memory under store-only bounds checking, including one that abuses protected writes.
- We empirically validate our attack on a SoftBound-hardened program [75], and qualitatively analyze the susceptibility of a representative selection of other work.
- We estimate the real-world feasibility of our attacks by analyzing a large corpus of open-source code.
- We reflect on the security assurances of store-only bounds checking and discuss possible improvements.

Listing 1: SoftBound’s pointer propagation instrumentation, adapted from [75].

```
ptr = *some_loc; // pointer load
bounds = lookup(some_loc)->bounds;

*other_loc = ptr; // pointer store
lookup(other_loc)->bounds = bounds;
```

2 Background

Bounds checkers check if the pointers a program dereferences still point to their *intended referent* [51]. This intended referent is usually the object whose address the program initially derived the pointer value from. Two main bounds-checking approaches can guarantee complete memory safety.

Pointer-based approaches explicitly track the intended referent for each pointer as run-time information, either in a disjoint metadata structure [25, 75, 76, 79, 81], or encoded as part of the pointer itself (so-called fat pointers¹) [9, 14, 49, 79, 109]. In the former case, the referent metadata is indexed using the address of pointers in memory, and the compiler explicitly instruments pointer copies in the program so they update the metadata. Listing 1 shows the way this explicit propagation happens in SoftBound [75].

Object-based approaches instead restrict pointer arithmetic such that the program can always recover the address of the original referent during memory accesses [26, 30, 51, 90]. Typically, this means not permitting a pointer to escape the original bounds of its allocation [30, 78]. The program associates safety metadata with every object’s base address and inspects the currently-pointed-to object’s metadata whenever it performs pointer arithmetic. Pointer propagation through memory requires no special handling, as the program can retrieve the allocation bounds based on the pointer value alone.

It is worth noting that many bounds checkers, especially recent ones [32, 58, 78, 93], do not perfectly fit either of these categories but instead appear more of a hybrid. For instance, Delta Pointers track the original referent per pointer through a relative distance metric in the pointer’s unused top bits (Pointer-Based) but can only do so for a limited range of pointer arithmetic, after which the original referent is lost (Object-Based) [58].

Secondly, as mentioned in Section 1, developers do not always operate bounds checkers in their most secure, full-coverage mode due to overhead concerns. Instead, bounds checkers sometimes omit some checks, allowing developers to accept a limited security risk to improve run-time performance. For instance, Wagner et al. argue that the most frequently executed memory accesses are the best candidates for

¹For brevity, we also include “diet” pointers that do not extend the native pointer width in this category.

bounds check elision [107], since they contribute to the overhead the most. Yet, the code that contains these accesses is likely the least bug-prone and best-tested code in the program, given its frequent execution.

A more popular way to deploy bounds checkers selectively is to restrict the checks to memory writes alone [63, 75, 77, 78, 81, 94], or to memory accesses that are permitted to access a certain amount of sensitive data [3, 15, 60, 101], or only certain regions of memory, e.g., the heap [30, 35, 45, 66]. The overhead reduction factor naturally depends on the amount of memory accesses that are left unchecked.

Store-only bounds checkers frequently report reduced overheads by a factor 2 or more [63, 75], while preventing all out-of-bounds or dangling pointer writes. Prior work has presented this as an attractive performance-security trade-off [77, 81], primarily due to the ease of converting any bounds checker design to a store-only working mode. Hence, although not all published memory safety enforcement work includes dedicated discussions and benchmarks of store-only operating modes, the prevailing notion seems that any bounds checker can readily be operated in a store-only mode when performance requirements dictate so, with limited security impact.

3 Risks of Store-Only Bounds Checking

The central thesis of this paper is that by leaving memory reads uninstrumented and freely exploitable, store-only bounds checkers give up much more security guarantees than “merely” the detection of pure information disclosure vulnerabilities such as Heartbleed [33, 87]. In this section, we describe several *additional* vulnerabilities and attack vectors spawned by the lack of protection on memory reads. In particular, we show that attackers can still arbitrarily corrupt memory *despite* passing all store-only validity checks.

3.1 Threat Model and Assumptions

Throughout this paper, we assume that (i) the program contains exploitable memory reads (e.g., out-of-bounds accesses or reads through dangling pointers), and (ii) the program uses a bounds checker of any type (i.e., pointer- or object-based, or a combination of both) to protect its memory writes. As we aim to break the intended integrity assurance of the store-only working mode, we do not rely on sub-object overflows [34] or vulnerabilities in external code or unprotected memory regions since prior work usually considers such vulnerabilities out of scope [30, 35, 45, 66]. We also assume that the attacker knows the details of the deployed store-only hardening and will adapt the attack to its design and implementation characteristics. Finally, as repeatedly demonstrated by previous work [73, 95, 98], we assume that any Address Space Layout Randomization (ASLR) [83] can readily be bypassed through information disclosure as a result of invalid memory reads.

3.2 Invalid Pointer Loads

A first, highly impactful security issue with store-only bounds checking appears when the program loads pointers from memory through exploitable memory reads such as the one shown in Listing 2. Attackers that can control the read on line 3 can choose which pointer to load from memory and, thus, which pointer gets dereferenced in the later memory write. Crucial here is that, as long as the loaded pointer points to a valid live object, the memory write will *always pass the store-only validity check* on line 5. The fundamental problem is that omitting the validity check for the memory read allows attackers to load a pointer value illegitimately, yet ensures that the pointer has valid bounds information when the program performs the store validity check. This is true even if the loaded pointer propagates through an arbitrary number of assignment statements before it reaches the final store instruction because the bounds checker will propagate the pointer metadata along the way if necessary.

Listing 2: A vulnerable code pattern under store-only hardening, with SoftBound instrumentation in red.

```
1 // exploitable pointer load
2 ptr = array[i];
3 bounds = lookup(&array[i])->bounds;
4 // ...
5 assert_in_bounds(ptr, bounds);
6 *ptr = ...;
```

Taking SoftBound as an example, Listing 2 shows that the bounds of the pointer at the `&array[i]` memory location are loaded and then checked against the value of the loaded pointer itself. Given control over `i`, attackers can choose which pointer is loaded, and due to the dynamic bounds propagation, SoftBound will look up the correct bounds associated with the accessed memory location. We stress that this is not a design or implementation issue with SoftBound; these are the intended bounds propagation rules for any bounds checker, regardless of object- or pointer-orientation. In Section 6, we describe the same issues against other types of bounds checkers.

To exploit this issue in practice, attackers must procure a valid pointer in the program to use as a substitute for (one of the) intended pointer values. Operating a bounds checker in store-only mode dramatically facilitates the search for these valid pointers since attackers can freely disclose large swaths of application memory through invalid reads, explicitly permitted by the threat model of these bounds checkers [77, 81]. Even without such capabilities, and depending on the type of victim application, offline analysis on a local binary may be sufficient to find useful pointers near the exploitable memory read location. Such an attack would not even require defeating ASLR in the first place, as a form of “Position-Independent

Address Reuse” [37].

Alternatively, attackers can *craft* valid pointers and inject them in attacker-controlled memory regions as part of the payload. This crafting option gives attackers even greater flexibility to meet the constraints of the invalid memory load. As far as we are aware, only pointer-based bounds checkers that maintain disjoint metadata keyed on pointer addresses, e.g., SoftBound [75], may be able to reject such crafted pointers during the store-only validity check, because the disjoint metadata will only contain entries for addresses of existing, valid pointers in the program. Any crafted pointers will not have corresponding entries in the metadata, and, as such, fail the metadata lookup itself. We note that modern pointer-based bounds checkers rarely use disjoint metadata, as it hurts cache locality [2, 81], can be a concurrency bottleneck in multi-threaded programs, and leads to compatibility issues when the bounds checker cannot reliably instrument all pointer copies that should update the metadata, e.g., in external code [42]. Hence, most modern bounds checkers [63, 78, 81] fail to detect the use of attacker-crafted pointers.

3.3 Arbitrary Code Execution Without Memory Corruption

No amount of validity checks on memory writes can help prevent exploits that solely use invalid memory reads. Existing store-only bounds checkers explicitly consider this in the case of information disclosure vulnerabilities [63, 75, 77, 78, 81, 94], but overlook the broader implications of memory-unsafe information flow. Consider the below snippet:

```
func = array[i];  
func(args);
```

As previous work also noted [60, 102], code patterns such as the above allow attackers to substitute `func` for other code pointers, including crafted ones, by merely abusing a single memory read. Such invalid function pointer reads suggest that developers should *at least* complement store-only bounds checks with defenses like Control Flow Integrity (CFI) [1]. In the original Code Pointer Integrity (CPI) paper [60], the protection against invalid code pointer reads is the precise difference between CPI and its less secure Code Pointer Separation (CPS) variant.

Store-only checking provides much better safety than control-flow integrity with similar performance overheads.

Nagarakatte et al. [77]

Interestingly, SoftBound also associates metadata with function pointers [75], much like data pointers, and checks on indirect calls whether the called address has a corresponding metadata entry. As acknowledged by the authors, any

valid function pointer can still be substituted, enabling expressive Whole-Function Reuse (WFR) [88, 92, 104]. More modern bounds checkers typically do not include any checks on indirect branches at all since the memory safety offered by the bounds checking itself should suffice to stop the initial memory error leading to a code pointer overwrite.

3.4 Invalidly Loading Non-Pointer Data

Further generalizing the implications of memory-unsafe information flow, attackers can also abuse invalid memory reads to load plain, non-pointer data from an attacker-controlled source. These invalid reads include pure information disclosure vulnerabilities like Heartbleed. However, they can also be used to create a *write-what* primitive where there previously existed none, as shown below:

```
1 int adminLvl = dangling_ptr->lvl;  
2 if (adminLvl > 2)  
3     system("/bin/bash");  
4 globalAdminLvl = adminLvl;
```

The use-after-free vulnerability on line 1 allows attackers to take control of the value of the `adminLvl` variable following the invalid load, typically by placing payload data at the `dangling_ptr` location. Because that memory load is left unchecked under store-only bounds checking, this snippet allows attackers to control a privilege flag without corrupting it, solely through an invalid read. In this case, the attack results in a privilege escalation. Note how this attack allows attackers to overwrite memory, e.g., the `globalAdminLvl` on line 4. Such an overwrite is entirely memory-safe.

3.5 Breaching Pointer Confidentiality

Some bounds checkers embed metadata in pointers (e.g., by writing a key tag into their top bits) but, for the sake of compatibility, still allow the program to perform arbitrary pointer arithmetic [42, 62, 63, 91, 105]. Unconstrained, this pointer arithmetic could overwrite the metadata. Any such design implicitly introduces a confidentiality requirement on pointer values. Consider the below snippet:

```
int* adminLvl = ...;  
ptr = &array[i];  
*ptr = ...;
```

If attackers can leak the `adminLvl` pointer value and the base address of the `array`, they can fill the difference between both in `as i`. The resulting `ptr` will then be equal to `array+(adminLvl-array) = adminLvl`, which will be a valid pointer to dereference, including all the necessary in-pointer metadata.

To defend against this type of attack, affected bounds checkers enforce the confidentiality of pointer values by checking memory reads to prevent information disclosure. In contrast, a

store-only deployment explicitly breaches this confidentiality by eliding checks on memory reads, massively exacerbating the applicability of this attack.

No matter how tempting it may sound to protect only writes, one must remember that buffer-overread vulnerabilities will slip away from such low-overhead checking.

Oleksenko et al. [81]

With the advent of low-latency cryptographic block ciphers in commodity hardware [10, 11, 56], we notice a growing trend towards such in-pointer metadata designs without pointer arithmetic restrictions [42, 62, 63, 105]. We want to stress that, even with full and store protection, these schemes still struggle to guarantee pointer confidentiality when the program is prone to sub-object overflows [34], or when it inadvertently leaks pointer values without violating memory safety. Concurrent work [40, 46] already exploits this precise weakness of the C³ defense [62]. On top of this, store-only checking grants attackers reliable access to confidential pointer values via information disclosure, thus presenting a clear security incompatibility with this emerging trend in low-overhead bounds checker design.

4 Ubiquity in Real-World Code

To assess whether existing code contains the necessary patterns to enable our store-only bypass techniques, we conducted an evaluation of the 1,000 most-starred C and C++ GitHub repositories. We tried to automatically identify the generic vulnerable patterns described in Section 3 using custom CodeQL² queries. We excluded two patterns from this search. We did not search for invalid loads of non-pointer data (Section 3.4), since its exploitable use, e.g., bypassing a privilege check, is highly application-specific and hard to infer automatically for a broad range of software. In addition, we also disregarded pointer arithmetic sites that are prone to the attack we described in Section 3.5 since we have no way of realistically estimating attacker control over the pointer offset.

Instead, we looked for loads of pointers that are later dereferenced in a memory write (Section 3.2), or called indirectly (Section 3.3). We excluded patterns where the load operation was obviously safe (e.g., direct loads from a scalar local variable). Instead, we focused on patterns (specifically on reads from arrays), of which we assume a substantial portion are exploitable. We then evaluated how many of them suit the requirements of store-only bypasses. This selection targets a large class of spatial C and C++ vulnerabilities but may miss potential Use-After-Free (UAF) issues, which can also appear without any indexing operations. However, these temporal safety issues are much harder to distinguish from obviously-safe pointer loads statically.

We match every pattern that contains direct data flow from a loaded pointer value to the pointer operand of a memory write (*unsafe data pointer loads*) or an indirect call (*unsafe funcptr loads*). Figure 1 shows that the former pattern occurs broadly across the entire suite of evaluated repositories. In addition, many repositories have frequent occurrences, e.g., 1,000 or more for over half of the evaluated programs. In contrast, the function pointer load pattern occurs less frequently, in large part because indirect calls occur less frequently than memory accesses. Hence, the store-only bypass based on invalidly loaded data pointers significantly increases the attacker's options when facing a store-only bounds-checked program.

5 Assurances of Store-Only Bounds Checking

Given the store-only bounds-checking risks we describe in Section 3, one may ask whether the utility of store-only bounds checking is defeated entirely. In this section, we analyze the expressiveness of the arbitrary write primitive granted through our store-only bypass, and discuss cases where store-only bounds checking is still useful.

After gaining some control over the target object of the memory write, attackers can corrupt address data to bootstrap a more powerful primitive [43], or corrupt key data structures directly, e.g., security-sensitive configuration data [16], syscall arguments [43], or syscall-guard variables [113]. Alternatively, attackers may seek arbitrary code execution by corrupting a code pointer in the program [13, 82, 85]. Most of these are already accessible through valid pointers in the program, so attackers can disclose the target corruption address more easily and obtain valid pointers to bypass the store-only validity checks. However, some objects never appear as valid overwrite targets in the bounds-checking metadata because no instrumented write should ever be able to target them. We describe a few examples here.

Return Addresses Overwriting return addresses can be difficult under store-only hardening since they are not part of any live object. In addition, some bounds checkers “heapify” [79] stack allocations to better control their memory layout [29], or to simplify instrumentation. This effectively leaves the return addresses on a safe stack [60], of which the location may be harder to disclose, and, in turn, complicates the task of crafting valid pointers. However, we find no such restrictions for the corruption of function pointers, i.e., forward-edge control flow hijacking, which is equally expressive [13].

Bounds Metadata An attractive option for adversaries looking to bootstrap an initial store-only bypass into a more expressive primitive may be to target the bounds or lifetime metadata itself. Once again, however, no pointers will naturally occur in the program for which any bounds checker

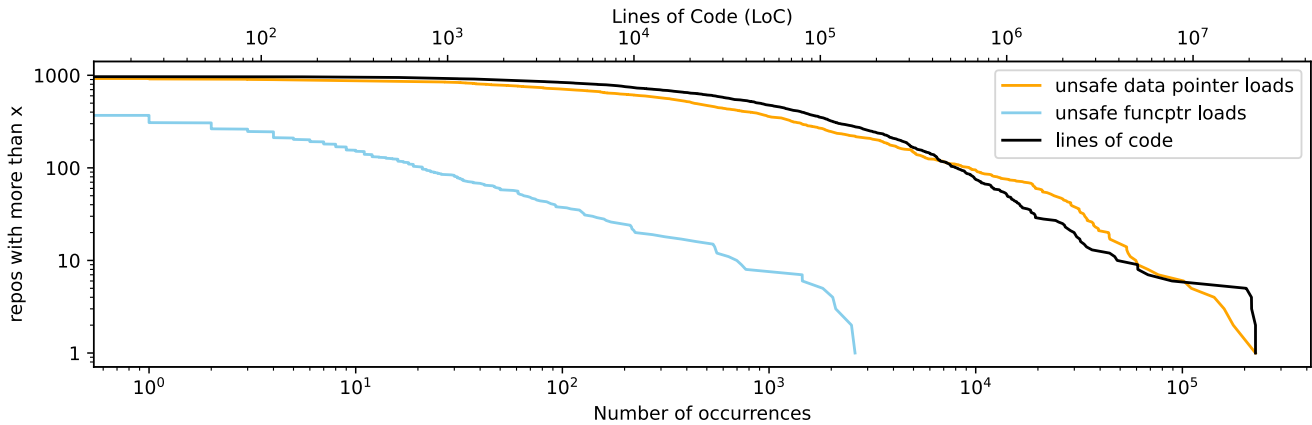


Figure 1: Occurrences of the enabling code patterns in the 1,000 most popular C/C++ GitHub repositories. The top axis counts the black graph, representing the distribution of the lines of code of each repository. The bottom axis counts the yellow and blue graphs, which show the distribution of unsafe data pointer loads, and unsafe function ptr loads respectively.

metadata is a valid target. Some defenses, e.g., CryptSan [42] and Mid-Fat Pointers [57], even include dedicated Software Fault Isolation (SFI) to explicitly outlaw invalid accesses to the metadata, as a defense-in-depth measure.

Safe Objects As a way to reduce run-time overhead [29, 106] or to provide isolation [44, 45, 60, 64], defenses statically identify objects that can never be the source of memory errors, because they are provably always accessed within their bounds. Bounds checkers do so too, e.g., to avoid heapifying many stack objects for performance reasons [29]. Similar to return addresses, this leaves them separated on a safer stack, which no bounds-checked memory writes can target.

For each of these cases, we notice a large difference between centralized/disjoint and decentralized/inline metadata. For a typical fat pointer approach, e.g., Austin et al.’s [9], attackers can craft fat pointers with any bounds attached to them, including those permitting access to the stack, bounds table metadata, or any other illegitimate targets, e.g., the Global Offset Table (GOT) [118]. After disclosing the address of the target object, such approaches pave the way for expressive exploitation.

On the other hand, many bounds checkers contain at least some metadata that is not kept inline with the pointer, and thus hinders straightforward crafting. Among “diet pointer” schemes, i.e., those that do not extend the native pointer width, many include a small *metadata key* as part of the pointer [18, 63, 78, 79, 91, 110], which can be used to retrieve complete metadata information during validity checks. These make it harder to craft arbitrary pointers to illegitimate targets, since it may require crafting a metadata entry, too. If all metadata is stored in a centralized, disjoint location [63, 91],

this is near impossible using store-only hardened writes alone. Alternatively, metadata can be stored inline with the objects too, e.g., as allocation headers or footers [18, 78, 79]. Adversaries must then be able to craft the metadata in the expected location, typically near the target object, to accompany the crafted pointer. For some illegitimate targets, e.g., return addresses, this can still be feasible when there are enough attacker-controlled regions available nearby.

Note that our discussion in this section primarily concerns *illegitimate* corruption targets, such as return addresses, to which the bounds checker will never create any valid pointers, as they are not supposed to be overwritten by application-level memory writes. All other corruption targets, e.g., function pointers, access control data, configuration data, etc., can generally be targeted through hardened memory writes using both crafted and reused pointers. In addition, store-only security risks that do not depend on invalid memory writes are not affected by any limitations of the store-only bypass primitive. For instance, loading attacker-chosen function pointers enables expressive control flow hijacking, with which these “illegitimate” targets can still be corrupted.

6 Analysis of Existing Store-Only Bounds Checkers

We reviewed several prominent bounds checkers that include a store-only mode and analyzed their susceptibility to the security risks we identified in Section 3. We summarize our findings in this section. Table 1 shows the condensed results, with the properties of each evaluated defense, and the bypass expressiveness it grants.

Property	SoftBound [75]	FRAMER [78]	PACMem [63]	Intel MPX [81]
Hardware Type	None	None	Commodity	Commodity
Per-Pointer Metadata	Pointer-based	Object-based	Pointer-based	Pointer-based
Per-Object Metadata	Disjoint	In-pointer	In-pointer	Disjoint
	None	Inline	Disjoint	None
Pointer Reuse	✓	✓	✓	✓
Pointer Crafting	✗	✓	✓	✓
Illegitimate Targets	✗	✗	✗	✓

Table 1: Comparison of selected bounds checkers that offer a store-only working mode. We highlight their respective design properties and the expressiveness of the store-only bypass technique under each.

Listing 3: Vulnerable program.

```

1 int* adminLvl = ...; // *adminLvl = 0
2 struct user {
3     int age;
4 }* users[NUM_USERS] = ...;
5
6 id = input_user();
7 age = input_user();
8 // exploitable memory read
9 struct user* user = users[id];
10 // checked memory write
11 user->age = age;
12
13 if (*adminLevel > 2) {
14     printf("Shell for admin: \n");
15     system("/bin/bash");
16 }

```

SoftBound [75] SoftBound is one of the most well-known spatial memory safety defenses in academic literature, with much derivative work reusing or extending its techniques [15, 100, 101]. The basic design is pointer-based with disjoint, centralized metadata. A large table, indexed by the storage locations of program pointers, contains information about the bounds of their intended referents. When pointers move around in memory, SoftBound updates the metadata to move around with them, and when they are loaded from memory, their bounds metadata is, too. This propagation mechanism allows SoftBound to check every pointer against the bounds of its intended referent on memory accesses without constraining or checking pointer arithmetic.

SoftBound, and the later review article by the same authors [77], includes an evaluation of a store-only working mode that reduces run-time overhead by a factor of 2 or more. Using SoftBound’s open-source prototype [23], we empirically validated our store-only bounds check bypass on a manually written vulnerable program, shown in Listing 3. On line 9, attackers can use the `id` variable to control the

loaded pointer from the `users` array. After defeating ASLR and disclosing the addresses of the objects involved, attackers can load the `adminLvl` pointer on line 9 by out-of-bounds indexing the `users` array, such that the bounds-checked write on line 11 overwrites the admin level, leading to privilege escalation in this case. We modeled this example after the IE God Mode bug [7], where a single variable controlled the privilege level of *VBScript* code executing in a sandbox.

We confirmed that we were able to successfully exploit the native program, without any hardening applied, by passing it the correct offset value for `id`, e.g., `&adminLvl - &users`, and supplying an `age` larger than 2. When we repeated this experiment on a fully hardened program version, SoftBound successfully detected the exploitation at the initial out-of-bounds memory read on line 9. We then turned off checks on memory reads and were able to exploit the program again, using the same technique as with the native program. During the memory read, SoftBound looked up the bounds associated with the actually-accessed memory location, i.e., `&adminLvl`, and enforced those at line 11. Naturally, these bounds were valid for the memory write to `*adminLvl`.

FRAMER [78] FRAMER is a spatial-only bounds checker which implements a mostly object-based design. Small in-pointer metadata keys track the location of per-object bounds information, which is typically located close to the object. FRAMER restricts pointer arithmetic to preserve the metadata key and, thus, to remember the intended referent at all times. FRAMER also supports a store-only working mode, which incurs less than a third of the performance overhead of its full instrumentation version.

As pointers store metadata keys in the unused top bits, they contain all the necessary information to pass the validity check. Naturally, pointer reuse is possible here to obtain valid substitute pointers, like in the previous SoftBound exploitation example. In addition, attackers can trivially craft pointers with arbitrary metadata keys in the upper bits. The possibility of pointer crafting makes FRAMER even more suitable for store-only attack bypasses than SoftBound.

PACMem [63] PACMem uses ARM’s Pointer Authentication (PA) feature [8] to bind pointers to their disjoint per-object metadata entries cryptographically. During allocation, PACMem generates a Pointer Authentication Code (PAC) based on the object’s full validity metadata (base pointer, allocation size, and a randomly generated temporal identifier called a “birthmark”) and places it into the top bits of the pointer. PACMem also stores per-object validity metadata in a linear table, indexed by the PAC of pointers during memory accesses. If the PAC does not match the looked-up validity metadata, PACMem knows the pointer is no longer tracking its intended referent, either due to out-of-bounds indexing or due to an intervening deallocation.

The authors also evaluate PACMem in a store-only working mode, which more than halves run-time overhead. From a store-only bypass perspective, PACMem behaves very similarly to FRAMER. The PAC is essentially a metadata key, protected from corruption through cryptographic integrity checks, for which FRAMER uses pointer arithmetic checks instead. In both cases, the metadata keys are revealed when the attacker can leak memory contents, and attackers can craft pointers using any metadata key to grant access to any bounds stored in the metadata table. Hence, this design permits both crafting and reuse to obtain valid pointers. To reiterate, what we describe as *pointer crafting* still requires the disclosure of authenticated pointers to the target object first to craft an identical copy in a different place. However, this is entirely in the scope of the store-only threat model, as mentioned in Section 3.1.

Finally, PACMem is the only one out of our evaluated schemes that suffers from the breach of its implicit pointer confidentiality. As discussed in Section 3.5, pointers contain their own metadata tags, and PACMem permits arbitrary pointer arithmetic. Hence, leaking two PACMem pointers and computing their offset gives attackers an index value with which they can construct pointer A from pointer B and vice versa.

Intel MPX [81] The now-deprecated Intel Memory Protection Extensions (MPX) were a hardware feature of select Intel CPU microarchitectures that included dedicated bounds registers, as well as bounds checking and management instructions that provided generic hardware acceleration for pointer-based bounds checking schemes [120]. Several papers additionally explored using MPX as a fast, coarse-grained intra-process isolation mechanism [15, 55, 60], for which it was arguably better suited.

MPX has architectural support for a centralized in-memory metadata structure that contains bounds entries for the location of every pointer in the program. In that regard, typical MPX-accelerated bounds checkers, such as those implemented by the GCC and ICC toolchains in the past [19, 81], are very similar to SoftBound, which itself was inspired by a hardware implementation of the same idea [25]. Indeed,

Oleksenko et al. analyzed the performance characteristics of Intel MPX when used for its intended bounds checking purpose [81], and included a performance comparison with, among others, SoftBound. They also evaluated a store-only working mode of such an MPX-based bounds checker and found that it reduces the performance overhead by a factor of 2.

A key difference between MPX’s design and SoftBound is that MPX redundantly stores the pointer’s value in the bounds entry that describes its intended referent. The goal is to allow the detection of external uninstrumented code that overwrites pointers in memory without updating their associated metadata entries, e.g., by re-assigning it to a different object. During loads of pointers, using the `BNDLDX` instruction [47], the processor checks whether the bounds table entry is present and holds a pointer value that matches its disjointly stored copy as a way to verify whether the metadata is still up to date. If it is not, MPX can take one of two implementation-defined actions. On the one hand, MPX can update the bounds table entry to cover the entire address space [47], i.e., the loaded pointer value can point to any object in the program, as a security concession that prioritizes compatibility with external code [77]. This compatibility mechanism allows MPX to gracefully handle calls to uninstrumented libraries, dynamically *unbounding* pointers when external code changes them instead of terminating the program. On the other hand, MPX can simply terminate the program. This latter option prioritizes security over compatibility.

From a store-only bypass perspective, the aforementioned compatibility option makes *pointer crafting* much easier than it is with SoftBound, which strictly distinguishes between valid pointer-holding locations and non-pointer data (cfr. Section 3.2). In its store-only mode, MPX would then graciously interpret any attacker-crafted pointer as a valid pointer for any object in the program, which bypasses previous pointer crafting limitations with SoftBound, yielding the single most expressive store-only bypass primitive we have observed in our review of the literature.

7 Discussion & Related Work

Until now, we described several attack vectors against store-only bounds checkers that go beyond information disclosure, in the hope of recalibrating the community’s expectations about the security guarantees of such defenses. In this section, we take a broader look at other types of store-only hardening, and the impact of our findings on other areas of memory safety enforcement.

Write Integrity Testing (WIT) [5] WIT is a notable memory safety hardening that solely provides store-only validity checks. However, its enforcement mechanism fundamentally differs from that of bounds checkers. At compile time,

WIT assigns the same *color* to all memory writes that may alias. This creates disjoint alias sets [53], each identified with a unique color, that hold all objects in the points-to sets of the aliasing writes. At run time, WIT tags each object with the color of its alias set, and queries the color of the actually-accessed object on memory writes. WIT’s store-only validity check verifies that the looked-up color matches the statically-assigned color of the write. This validity check ensures that the actually-accessed object is within the statically-computed points-to set of the memory write. Within that set, the memory write can corrupt all objects. Naturally, this permits clear memory safety violations, and has been regarded as strictly *weaker* than precise bounds checking for that reason. However, because WIT establishes the set of accessible objects statically, its validity checks cannot be fooled by our store-only bypass. Contrary to bounds checkers, WIT does not propagate any bounds or metadata information dynamically. As such, its security guarantees are not affected by any memory-unsafety from which the memory write operand originates; the same, statically-determined set of objects will be enforced regardless. WIT shows increased resilience over bounds checkers in the face of arbitrary memory reads, which makes it more suitable as a store-only hardening mechanism.

Impact on Static Analysis Bounds checkers typically include a range of compiler optimizations to suppress overhead [12, 18, 44, 45, 52, 66, 100, 108, 112, 117, 122]. A popular optimization is to check whether pointer operands of memory accesses are always in bounds of any object they could refer to [5]; if so, they are provably safe and do not require a dynamic bounds check. This in-bounds analysis typically requires statically tracing pointers backward to determine their origin, accumulating any offsets they garner along the way. Many pointers are loaded from memory eventually (Section 4), at which point thorough analyses perform a Reaching Definitions Analysis (RDA) [4] to determine the possible values of the loaded pointer. The in-bounds analysis can then continue investigating all these possible loaded pointer values. If all possible loaded values are in bounds, the analysis will consider the original memory access as safe, and leave it uninstrumented.

Again, a problem appears when the loaded pointer value originates from an exploitable memory read. Attackers can invalidly load a different pointer, and, due to the optimization, there will not even be a bounds check left to bypass. The underlying problem here is that many static analyses do not account for the memory-unsafety of C and C++ [69], but are still used to prove its safety properties. To avoid this specific issue, we recommend only performing RDA on memory loads which themselves are also provably in bounds.

Store-Only Testing In this paper, we have primarily discussed the weaknesses of bounds checkers as exploit mitigations, facing a sophisticated adversary that is motivated to

break the program’s protection through any means necessary. However, some bounds checkers simply aim to catch memory safety violations that are triggered during development or (fuzz) testing [17, 36, 67, 94]. The latter are commonly referred to as “sanitizers” [97], and tend to use less secure methods of catching memory errors, that nevertheless detect violations more precisely, e.g., at object bounds instead of allocation bounds [28]. Performance can still be important here, e.g., to improve throughput during automated fuzz testing [36, 48, 119, 122]. Indeed, the original AddressSanitizer (ASan) paper, now integrated into popular compilers [20, 21], included an evaluation of a “writes-only” instrumentation mode, which reduced the run-time overhead threefold. However, since ASan is not meant to run in production, despite a stint in the Tor browser [24], the impact of our attack is limited. Still, our work undermines the assumption that when a program is thoroughly sanitized/fuzzed for invalid write bugs, attackers will not be able to corrupt program memory or achieve arbitrary code execution.

Selective Bounds Checking Apart from store-only deployments, researchers have also proposed using bounds checkers to protect only a security-critical, sensitive part of the data space [3, 15, 60, 101]. These defenses generally include a coarse-grained isolation mechanism in the non-sensitive part to prevent access to the sensitive part, e.g. using SFI [114] or Intel MPK [47]. Typically, a pointer analysis determines which memory accesses are allowed to access the sensitive region and which are not. Depending on the way the analysis computes sensitivity, we believe that such selective bounds checkers carry a similar vulnerability to their store-only siblings. Consider the snippet below:

```
1 ptrToSens = nonSensArray[i];  
2 *ptrToSens = ...;
```

The `nonSensArray` is non-sensitive, and it contains non-sensitive pointers to sensitive objects. The load from the array on line 1 is only instrumented with coarse-grained bounds checks, since the pointer analysis correctly determined that it accesses a non-sensitive object (`nonSensArray`). The store on line 2 is bounds checked in a fine-grained way, since it is supposed to access sensitive data. When the load on 1 is exploitable, however, attackers can load any valid pointer to the sensitive region from the non-sensitive region, which will pass the validity check on line 2, in true store-only bypass fashion. Hence, attackers can choose which sensitive object gets written to on line 1, by abusing a memory error they were permitted to exploit (coarse-grained bounds check). Note that we bypass two layers of defense-in-depth at once here: attackers are not supposed to write to the sensitive region (inter-sensitive isolation), and sensitive memory accesses are not supposed to be exploitable, because they are bounds checked (intra-sensitive isolation).

One option to address this issue is to include pointers to sensitive objects in the sensitive region as well [15, 96], recursively. However, this can quickly lead to a very large sensitive region, with almost all memory accesses instrumented, and the associated performance overhead.

8 Conclusion

In this work, we uncovered fundamental weaknesses of store-only bounds checking, directly caused by the lack of protection on memory reads. In particular, we demonstrated that invalid loads of pointers give attackers control over *hardened* memory writes. We empirically validated our attack on a prominent bounds checker prototype, and characterized the same weakness in other bounds checker designs. Through automated code analysis, we showed that a large corpus of real software exhibits the vulnerable patterns that enable our store-only bypass.

Looking ahead, we discussed potential avenues to rebalance the security and overhead advantages of store-only hardening. To this end, we recognized the resilience of Data Flow Integrity (DFI) against malicious pointer loads. Given the broader importance of efficient memory safety enforcement, we encourage new research into store-only hardening, keeping in mind the subversive effects of attacker-controlled memory loads.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. In addition, we thank Silviu Vlasceanu and Mahmoud Ammar from Huawei Trusted System Security Lab Munich for the interesting conversations that led to this work, and Dairo de Ruck for providing access to much-needed computation resources. This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Program Flanders.

Availability

Our attack experiments and code analysis queries are available at <https://github.com/ku-leuven-msec/not-quite-write-experiments>.

References

- [1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS '05*, pages 340–353, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595932267. doi: 10.1145/1102120.1102165.
- [2] Masab Ahmad, Syed Kamran Haider, Farrukh Hijaz, Marten van Dijk, and Omer Khan. Exploring the performance implications of memory safety primitives in many-core processors executing multi-threaded workloads. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, HASP '15*, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334839. doi: 10.1145/2768566.2768572. URL <https://doi.org/10.1145/2768566.2768572>.
- [3] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N. Asokan, and Danfeng (Daphne) Yao. Not all data are created equal: Data and pointer prioritization for scalable protection against Data-Oriented attacks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1433–1450, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/ahmed-salman>.
- [4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.
- [5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pages 263–277, 2008. doi: 10.1109/SP.2008.30.
- [6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, page 96, 2009.
- [7] Anit Anubhav and Manish Sardiwal. The journey and evolution of god mode in 2016: Cve-2016-0189, 2017. URL <https://www.virusbulletin.com/virusbulletin/2017/01/journey-and-evolution-god-mode-2016-cve-2016-0189/>.
- [8] Arm Ltd. *Arm Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile*, 2022.
- [9] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94*, pages 290–301, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 089791662X. doi: 10.1145/178243.178446. URL <https://doi.org/10.1145/178243.178446>.

- [10] Roberto Avanzi. The QARMA block cipher family. *IACR Transactions on Symmetric Cryptology*, pages 4–44, 2017.
- [11] Yanis Belkheyar, Joan Daemen, Christoph Dobraunig, Santosh Ghosh, and Shahram Rasoolzadeh. Bip-bip: A low-latency tweakable block cipher with small dimensions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1): 326–368, Nov. 2022. doi: 10.46586/tches.v2023.i1.326-368. URL <https://tches.iacr.org/index.php/TCHES/article/view/9955>.
- [12] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davit. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 502–519, 2022. doi: 10.1109/EuroSP53844.2022.00038.
- [13] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305648. doi: 10.1145/1966913.1966919.
- [14] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 381–392, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355766. doi: 10.1145/3196494.3196540.
- [15] Scott A. Carr and Mathias Payer. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 193–204, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349444. doi: 10.1145/3052973.3052983. URL <https://doi.org/10.1145/3052973.3052983>.
- [16] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, 7 2005. USENIX Association. URL <https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats>.
- [17] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. MTSan: A feasible and practical memory sanitizer for fuzzing cots binaries. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 841–858, 2023.
- [18] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupe, and Gail-Joon Ahn. Vik: practical mitigation of temporal memory safety violations through object id inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284, 2022.
- [19] GCC Developers. Intel MPX support in the GCC compiler, June 2018. URL <https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler>.
- [20] GCC Developers. Program instrumentation options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#index-fsanitize_003daddress, 2024.
- [21] LLVM Developers. Addresssanitizer. <https://clang.llvm.org/docs/AddressSanitizer.html>, 2024.
- [22] Rust Developers. Rust programming language, 2024. URL <https://www.rust-lang.org/>.
- [23] SoftBoundCETS developers. softboundcets-34, 2014. URL <https://github.com/santoshn/softboundcets-34>.
- [24] Tor Developers. Tor browser 5.5a4-hardened is released, November 2015. URL <https://blog.torproject.org/tor-browser-55a4-hardened-released/>.
- [25] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939586. doi: 10.1145/1346281.1346295. URL <https://doi.org/10.1145/1346281.1346295>.
- [26] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi:

10.1145/1134285.1134309. URL <https://doi.org/10.1145/1134285.1134309>.

- [27] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933204. doi: 10.1145/1133981.1133999. URL <https://doi.org/10.1145/1133981.1133999>.
- [28] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. Baggy bounds with accurate checking. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, pages 195–200, 2012. doi: 10.1109/ISSREW.2012.24.
- [29] Gregory Duck, Roland Yap, and Lorenzo Cavallaro. Stack object protection with low fat pointers. In *NDSS Symposium 2017*, 2017.
- [30] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892212.
- [31] Gregory J Duck and Roland HC Yap. EffectiveSan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.
- [32] Gregory J. Duck, Yuntong Zhang, and Roland H. C. Yap. Hardening binaries against more memory errors. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 117–131, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519580. URL <https://doi.org/10.1145/3492321.3519580>.
- [33] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heart-bleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332132. doi: 10.1145/2663716.2663755. URL <https://doi.org/10.1145/2663716.2663755>.
- [34] Ronald Gil, Hamed Okhravi, and Howard Shrobe. There's a hole in the bottom of the c: On the effectiveness of allocation protection. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 102–109, 2018. doi: 10.1109/SecDev.2018.00021.
- [35] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. TAILCHECK: A lightweight heap overflow detection mechanism with page protection and tagged pointers. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.
- [36] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. FloatZone: Accelerating memory error detection using the floating point unit. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 805–822, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/gorter>.
- [37] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242, 2018. doi: 10.1109/EuroSP.2018.00024.
- [38] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 517–528, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978405. URL <https://doi.org/10.1145/2976749.2978405>.
- [39] Istvan Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. Metalloc: Efficient and comprehensive metadata management for software security hardening. In *Proceedings of the 9th European Workshop on System Security*, EuroSec '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342957. doi: 10.1145/2905760.2905766. URL <https://doi.org/10.1145/2905760.2905766>.
- [40] Mohamed Tarek Bnziad Mohamed Hassan. *Hardware-Software Co-design for Practical Memory Safety*. PhD thesis, Columbia University, 2022.

- [41] Konrad Hohentanner, Florian Kasten, and Lukas Auer. Hwasanio: Detecting c/c++ intra-object overflows with memory shading. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, pages 27–33, 2023.
- [42] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. Cryptsan: Leveraging arm pointer authentication for memory safety in c/c++. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC '23*, pages 1530–1539, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450395175. doi: 10.1145/3555776.3577635. URL <https://doi.org/10.1145/3555776.3577635>.
- [43] Hong Hu, Zheng Leong Chua, Sendriou Adrian, Praatek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium, SEC'15*, pages 177–192, USA, 2015. USENIX Association. ISBN 9781931971232.
- [44] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *Proceedings of the 2020 ISOC Network and Distributed Systems Security Symposium (NDSS)*, February 2022.
- [45] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. Top of the heap: Efficient memory error protection for many heap objects. *arXiv preprint arXiv:2310.06397*, 2023.
- [46] Mohamed Tarek Ibn Ziad, Evgeny Manzhosov, and Simha Sethumadhavan. C-4: Compromising cryptographic capability computing. 2022. Work in progress.
- [47] Intel Inc. *Intel 64 and IA-32 Architectures. Software Developer's Manual*, 2021.
- [48] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient sanitizer metadata design for fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 249–263. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL <https://www.usenix.org/conference/atc20/presentation/jeon>.
- [49] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference, ATEC '02*, pages 275–288, USA, 2002. USENIX Association. ISBN 1880446006.
- [50] X. Jin, X. Xiao, S. Jia, W. Gao, H. Zhang, D. Gu, S. Ma, Z. Qian, and J. Li. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *2022 IEEE Symposium on Security and Privacy (S&P)*, pages 473–488, Los Alamitos, CA, USA, May 2022. IEEE Computer Society. doi: 10.1109/SP46214.2022.00028. URL <https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00028>.
- [51] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, volume 97, pages 13–26, 1997.
- [52] Tina Jung, Fabian Ritter, and Sebastian Hack. Pico: A presburger in-bounds check optimization for compiler-based memory safety instrumentations. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(4):1–27, 2021.
- [53] Vineet Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. *SIGPLAN Not.*, 43(6):249–259, June 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375613.
- [54] Piyus Kedia, Rahul Purandare, Udit Agarwal, and Rishabh. Cguard: Scalable and precise object bounds protection for c. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1307–1318, 2023.
- [55] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *European Conference on Computer Systems (EuroSys)*, 2017.
- [56] Michael Kounavis, Sergej Deutsch, Santosh Ghosh, and David Durham. K-cipher: A low latency, bit length parameterizable cipher. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7, 2020. doi: 10.1109/ISCC50000.2020.9219582.
- [57] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *Proceedings of the 10th European Workshop on Systems Security, EuroSec'17*, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349352. doi: 10.1145/3065913.3065920. URL <https://doi.org/10.1145/3065913.3065920>.
- [58] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In

Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190553.

- [59] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnavtsov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221, 2017.
- [60] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, USA, 2014. USENIX Association. ISBN 9781931971164.
- [61] Michael Larabel. The first rust-written network phy driver set to land in linux 6.8, 12 2023. URL <https://www.phoronix.com/news/Linux-6.8-Rust-PHY-Driver>.
- [62] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. Cryptographic capability computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 253–267, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480076. URL <https://doi.org/10.1145/3466752.3480076>.
- [63] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 1901–1915, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3560598. URL <https://doi.org/10.1145/3548606.3560598>.
- [64] Hans Liljestrand, Carlos Chinaea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N. Asokan. Color my world: Deterministic tagging for memory safety, 2022. URL <https://arxiv.org/abs/2204.03781>.
- [65] Ankur Limaye and Tosiron Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, 2018. doi: 10.1109/ISPASS.2018.00028.
- [66] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. Camp: Compiler and allocator-based heap memory protection. In *USENIX Security Symposium*, 2024. URL <https://zplin.me/papers/CAMP.pdf>. To appear in USENIX Security 2024.
- [67] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. Giantsan: Efficient memory sanitization with segment folding. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)*, 2024.
- [68] Zhengyang Liu and John Criswell. Flexible and efficient memory object metadata. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 36–46, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350440. doi: 10.1145/3092255.3092268. URL <https://doi.org/10.1145/3092255.3092268>.
- [69] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhotak, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: A manifesto. *Communications of the ACM*, 58:44–46, 2015. URL <http://cacm.acm.org/magazines/2015/2/182650-in-defense-of-soundness/abstract>.
- [70] The MITRE Corporation (MITRE). CWE-125: Out-of-bounds read. <https://cwe.mitre.org/data/definitions/125.html>, 2024.
- [71] The MITRE Corporation (MITRE). CWE-416: Use after free. <https://cwe.mitre.org/data/definitions/416.html>, 2024.
- [72] The MITRE Corporation (MITRE). CWE-787: Out-of-bounds write. <https://cwe.mitre.org/data/definitions/787.html>, 2024.
- [73] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 167–182, 2018. doi: 10.1109/EuroSP.2018.00020.
- [74] Yeoul Na. -fbounds-safety. enforcing bounds safety for production c code. *EuroLLVM Developers' Meeting*, May 2023. URL <https://llvm.org/devmtg/2023-05/slides/TechnicalTalks-May11/01-Na-fbounds-safety.pdf>.

- [75] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542504.
- [76] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 189–200, USA, 2012. IEEE Computer Society. ISBN 9781450316422.
- [77] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything You Want to Know About Pointer-Based Checking. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 190–208, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-80-4. doi: 10.4230/LIPIcs.SNAPL.2015.190. URL <http://drops.dagstuhl.de/opus/volltexte/2015/5026>.
- [78] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. Framer: A tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, pages 612–626, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359799.
- [79] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, may 2005. ISSN 0164-0925. doi: 10.1145/1065887.1065892. URL <https://doi.org/10.1145/1065887.1065892>.
- [80] White House Office of the National Cyber Director (ONCD). Back to the building blocks: A path toward secure and measurable software. Technical report, ONCD, February 2024. URL <https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf>.
- [81] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhattotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.
- [82] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), November 1996. URL <http://www.phrack.com/issues.html?issue=49&id=14>.
- [83] PaX Team. Address space layout randomization (aslr). <https://pax.grsecurity.net/docs/aslr.txt>, 2001.
- [84] Tribuvan Kumar Prakash and Lu Peng. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor. *ISAST Trans. Comput. Softw. Eng.*, 2(1):36–41, 2008.
- [85] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security and Privacy*, 10(6):84–87, November 2012. ISSN 1540-7993. doi: 10.1109/MSP.2012.152.
- [86] Alex Rebert and Christoph Kern. Secure by design: Google’s perspective on memory safety. Technical report, Google Security Engineering, 2024.
- [87] Inc. Red Hat. CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., December 3 2014. URL <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160>.
- [88] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *NDSS*, 2017.
- [89] Mark Russinovich, September 2022. URL <https://twitter.com/markrussinovich/status/1571995117233504257>.
- [90] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 2004, pages 159–169, 2004.
- [91] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. Heapcheck: Low-cost hardware support for memory safety. *ACM Trans. Archit. Code Optim.*, 19(1), January 2022. ISSN 1544-3566. doi: 10.1145/3495152. URL <https://doi.org/10.1145/3495152>.
- [92] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P '15,

pages 745–762, USA, 2015. IEEE Computer Society. ISBN 9781467369497. doi: 10.1109/SP.2015.51.

- [93] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. ZOMETAG: Zone-based Memory Tagging for Fast, Deterministic Detection of Spatial Memory Violations on ARM. *IEEE Transactions on Information Forensics and Security*, 2023.
- [94] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC’12, page 28, USA, 2012. USENIX Association.
- [95] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 7, 2012.
- [96] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS 2016*, 2016.
- [97] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 1275–1295, 2019. doi: 10.1109/SP.2019.00010.
- [98] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections. *Black Hat USA*, 2008.
- [99] Jeffrey Vander Stoep. Memory safe languages in android 13, 12 2022. URL <https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html>.
- [100] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016. doi: 10.1109/TR.2016.2570538.
- [101] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (S&P)*, pages 1433–1449, 2020. doi: 10.1109/SP40000.2020.00042.
- [102] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi: 10.1109/SP.2013.13.
- [103] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications. *Proceedings of the ACM on Programming Languages*, 7(PLDI):124–147, 2023.
- [104] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 121–141, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23644-0.
- [105] Martin Unterguggenberger, David Schrammel, Lukas Lamster, Pascal Nasahl, and Stefan Mangard. Cryptographically enforced memory safety. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’23, pages 889–903, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3623138. URL <https://doi.org/10.1145/3576915.3623138>.
- [106] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-after-type: Practical and complete type-safe memory reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC ’18, pages 17–27, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365697. doi: 10.1145/3274694.3274705.
- [107] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, pages 866–879. IEEE, 2015.
- [108] Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen. Spindle: Informed memory access monitoring. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC ’18, pages 561–573, USA, 2018. USENIX Association. ISBN 9781931971447.
- [109] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014. doi: 10.1109/ISCA.2014.6853201.
- [110] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International*

- Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21*, pages 224–240, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446761. URL <https://doi.org/10.1145/3445814.3446761>.
- [111] Shengjie Xu, Eric Liu, Wei Huang, and David Lie. Mifp: Selective fat-pointer bounds compression for accurate bounds checking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 609–622, 2023.
- [112] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. SIMBER: Eliminating redundant memory bound checks via statistical inference. In *ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings 32*, pages 413–426. Springer, 2017.
- [113] Hengkai Ye, Song Liu, Zhechang Zhang, and Hong Hu. VIPER: Spotting Syscall-Guard variables for Data-Only attacks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1397–1414, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/ye>.
- [114] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009. doi: 10.1109/SP.2009.25.
- [115] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11*, pages 307–316, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137435. doi: 10.1145/940071.940113.
- [116] Yves Younan, Pieter Philippaerts, Lorenzo Cavallo, R. Sekar, Frank Piessens, and Wouter Joosen. Parichck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS '10*, pages 145–156, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589367. doi: 10.1145/1755688.1755707. URL <https://doi.org/10.1145/1755688.1755707>.
- [117] Yizhuo Zhai, Zhiyun Qian, Chengyu Song, Manu Sridharan, Trent Jaeger, Paul Yu, and Srikanth V Krishnamurthy. Don't waste my efforts: Pruning redundant sanitizer checks of developer-implemented type checks. 2024. To appear in USENIX Security 2024.
- [118] Chao Zhang, Lei Duan, Tao Wei, and Wei Zou. Secgot: Secure global offset tables in elf executables. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, pages 995–998. Atlantis Press, 2013/03. ISBN 978-90-78677-61-1. doi: 10.2991/iccsee.2013.250. URL <https://doi.org/10.2991/iccsee.2013.250>.
- [119] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in c/c++ programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494, 2021.
- [120] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, pages 631–644, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304017.
- [121] Yiyu Zhang, Tianyi Liu, Zewen Sun, Zhe Chen, Xuan-dong Li, and Zhiqiang Zuo. Catamaran: Low-overhead memory safety enforcement via parallel acceleration. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 816–828, 2023.
- [122] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association. URL <https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen>.



SoK: On the Effectiveness of Control-Flow Integrity in Practice

Lucas Becker 

Technical University of Darmstadt
lbecker@seemoo.de

Matthias Hollick 

Technical University of Darmstadt
mhollick@seemoo.de

Jiska Classen

Hasso Plattner Institute, University of Potsdam
jiska.classen@hpi.de

Abstract

Complex programs written in memory-unsafe languages tend to contain memory corruption bugs. Adversaries commonly employ code-reuse attacks to exploit these bugs. Control-flow Integrity (CFI) enforcement schemes try to prevent such attacks from achieving arbitrary code execution. Developers can apply these schemes to existing code bases by setting compiler flags, requiring less effort than rewriting code in memory-safe languages. Although many works propose CFI schemes and attacks against them, they paid little attention to schemes deployed to end-users. We provide a systematic categorization and overview of actively used CFI solutions. We then conduct a large-scale binary analysis on 33 Android images of seven vendors and two Windows builds for different hardware architectures to study CFI utilization in practice. We analyzed over 77,000 files on the Android images. We found that depending on the variant, up to 94% of binaries and 93% of libraries are unprotected. All analyzed binaries depend on unprotected libraries, therefore rendering CFI enforcement ineffective. Further, we look at the development of CFI coverage over time on Android and find it stagnating. CFI roll-out is closer to complete on the Windows builds, but not all files are protected yet (2.63% unprotected). Consequently, our results show that the adoption of CFI protection is lacking, putting devices at risk. Additionally, our results highlight a large gap between the state of the art in research and the reality of deployed systems.

1 Introduction

Memory safety vulnerabilities make up two thirds of security issues in large code bases across the industry [45]. Despite the ongoing effort to prevent and mitigate memory corruption attacks, adversaries exploit these memory corruption bugs to take over computer systems. Rewriting memory-unsafe code in memory-safe languages reduces this attack surface [101]. However, the tremendous engineering effort of, e.g., porting C/C++ code to Rust, will still take years and is often infeasible on a limited budget. As a generic solution fitting most

code bases, compiler toolchains add checks meant to prevent the exploitation of memory safety vulnerabilities. Control-flow Integrity (CFI) enforcement schemes are one instance of such checks. CFI checks prevent code-reuse attacks by limiting the allowed targets for indirect control-flow transfers. Ideally, this means that the program flow stays within the intended boundaries. Because the precise and sound points-to analysis required to enforce this property is generally undecidable [93], practical CFI schemes have to settle for less precise policies. Implementations must be efficient to be deployed on real-world systems while also granting sufficient security guarantees. As a result of this trade-off, coarse-grained CFI schemes can often be observed in practice, even though their ineffectiveness is well known [31]. We address the following research questions in this paper:

1. Which CFI schemes are found in practice?
2. Where and how consequently are they deployed?
3. What are their capabilities and limitations to prevent attacks?

In contrast to previous works comparing and benchmarking CFI schemes [19, 33, 65, 66, 78, 105, 114, 123], we study real-world ecosystems that deploy CFI mitigations. With this approach, we address how effectively CFI enforcement is deployed on actual systems rather than comparing academic research prototypes. For that, we study three different software- and four hardware-based CFI implementations on their corresponding platforms. We primarily focus on CFI schemes targeting user-space programs, even though most of them are used to protect the operating system kernel as well, since protecting OS kernels requires a different threat model. We also examine three shadow stack designs used to implement backwards-edge CFI. Numerous choices are involved in designing CFI enforcement schemes. These choices include which kind of control-flow transfers are protected, how the allowed Control-flow Graph (CFG) is derived, and whether special hardware features are required. CFI enforcement opens up a considerable research area, with a vast amount of different proposals [2, 24, 34, 46, 52, 59, 61, 62, 68, 72, 79, 82, 83,

87, 88, 110, 117, 118, 127]. Most of these proposals are not widely deployed in practice, as they depend on specialized hardware, require intrusive changes, come with a significant performance overhead, or are closed-source. Many promising academic solutions have not been adopted in practice and were not maintained over time. Following the approaches laid out by these prototypes, all of the most common operating systems [100] support some form of CFI enforcement in 2024. We identified the most notable solutions currently used in practice as:

- LLVM Clang CFI [107, 110], used primarily on Android and the Linux Kernel,
- Windows Control Flow Guard (WCFG) [76] and its successor eXtended Flow Guard (XFG) [120],
- ARMv8 Pointer Authentication (PA) [96] including Branch Target Identification (BTI), utilised by recent Apple Systems on a Chip (SoCs) starting with the A12, S4, and M1 chips [11], by Android, and by Windows on ARM [121]; and
- Intel Control-flow Enforcement Technology (CET) [54, 56], supported on Intel processors starting with the 11th Gen [55] and used by Windows and Linux.

There are also a few other commercial offerings, such as the Reuse Attack Protector (RAP) [49] and similar. We do not include them in this work, as it is difficult to reason about how frequently they are deployed.

We find that many binaries and libraries are missing appropriate protection, despite the compilation toolchains for these systems supporting them. On Android, we find that every investigated binary depends on at least one unprotected library. Overall, less than 17% of the binaries and libraries in recent firmware images are CFI protected. On Windows, CFI coverage is much higher, but a fine-grained CFI implementation is only available on preview builds. In summary, our main contributions are as follows:

- We systematize prevalent CFI solutions in practice, including LLVM’s CFI scheme and Microsoft’s closed-source implementations WCFG and XFG on Windows.
- We study CFI coverage, security characteristics, and effectiveness in practice by running a large-scale binary analysis on Android and Windows binaries.
- We analyse Android firmware releases of the same devices to get insights into the development over time.

2 CFI Design Space

Approaches to CFI Enforcement CFI schemes prevent deviation from a program’s control flow, assuming an attacker who can divert the control flow by exploiting memory corruption bugs. Figure 1 shows a simplified CFG example, where basic

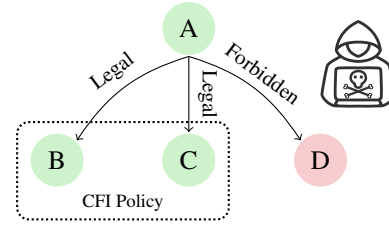


Figure 1: Simplified CFG under a CFI policy. Flows from A to D are unintended by the programmer and are only made possible by memory corruption attacks.

block A is allowed to call blocks B and C, but not block D. Calling into D from A violates the CFI policy. Block D could, for example, be the `system()` function on Unix-like systems.

To protect indirect control-flow transfers, most CFI enforcement schemes follow the same basic pattern: First, a program-specific CFG is derived from the policy specifying the rules for valid control-flow transfers. Then, during runtime, this CFG is enforced by guard code, which checks that a control-flow transfer abides by the CFG [123]. If a violation of the CFG is detected, the program can be terminated to prevent successful attacks. Some recent proposals also refine the CFG during runtime [34, 52, 83, 115]. This allows to increase the precision of the CFG, for example, to achieve forms of context sensitivity. Although CFI includes forward- and backward-edge protection, this approach is often only applied to forward-edge flows, while shadow stacks are the preferred method to protect backward-edge transfers [21]. They can leverage that the return address after a call instruction is known to be the address of the subsequent instruction, language features that require special stack unwinding aside.

Compile-time Instrumentation vs. Binary Rewriting Guard code can be added directly during a program’s compilation or by applying binary rewriting or instrumentation techniques. Hereby, there is a trade-off between applicability and precision: Compiler-based CFI implementations require the source code of applications to add protection, which implies that protection can only be added to commercial off-the-shelf software by the vendor itself. However, binary rewriting suffers from higher complexity and usually a loss of precision [85, 114]. Seemingly for this reason, we observed that all CFI schemes found in practice are compiler-based.

Policy Precision CFI schemes are often categorized into coarse- and fine-grained schemes. We adopt the definition from [83], wherein the number of supported *Equivalence Classes* is used as the decisive characteristic. Targets of indirect control-flow transfers are divided into classes so that if a target is reachable from a given control-flow transfer, every other target in the same equivalence class is a valid target as well, but others are not. Coarse-grained CFI schemes support only a program-independent and typically low number of equivalence classes. Fine-grained CFI schemes support a program-dependent number of equivalence classes, allowing

Table 1: Overview of CFI schemes used in practice

Scheme	Edge	Policy	Granularity	Impl.	Open-source	Platforms
LLVM CFI [107, 110]	→	Type-based	Fine	SW	✓	All LLVM supported
Control Flow Guard [76]	→	Marked function	Coarse	SW	✗	Windows
eXtended Flow Guard [120]	→	Type-based	Fine	SW	✗	Windows
Pointer Authentication [12, 96]	⇔	Implementation dep.	n/a	HW	●	ARMv8.3-A, ARMv8.1-M
Branch Target Identification [12]	→	Label-based (#l=3)	Coarse	HW	●	ARMv8.5-A, ARMv8.1-M
Indirect Branch Tracking [56]	→	Label-based (#l=1)	Coarse	HW	●	Intel 11th / 12th gen.
FineIBT [44]	→	Implementation dep.	n/a	Hybrid	●	Linux with IBT support
LLVM Shadow Call Stack [109]	←	Shadow Stack	n/a	SW	✓	ARM-based
SafeStack [64, 106]	←	Shadow Stack	n/a	SW	✓	All LLVM supported
CET Shadow Stack [56, 97]	←	Shadow Stack	n/a	HW	●	Intel 11th / 12th gen.

“Edge” specifies the protected control-flow transfers: backward-edge (←), forward-edge (→), and both (⇔). The “Impl.” column shows whether a scheme is implemented in software (SW) or hardware (HW). ● means that open-source implementations of the compiler and runtime components exist, but the hardware implementation is closed-source. For hardware schemes, “platforms” specifies the minimum CPU or instruction set.

each indirect control-flow transfer to have its own targets.

Evaluating CFI Effectiveness How to precisely quantify the effectiveness of CFI schemes is an open research question. To address this issue, several metrics to quantize security guarantees have been proposed, most notably Average Indirect target Reduction (AIR) [127], Average Indirect targets Allowed (AIA) [46], Relative Average Indirect target Reduction (RAIR) [117], Calltarget Reduction (CTR) [81], and Quantitative Security (QS) [19]. The common shortcoming of these metrics is that they only consider the target reduction while ignoring the *quality* of the corresponding targets. Consequently, good values in these metrics do not guarantee better security, as even with CFI, there can remain valid paths to divert the program flow maliciously. CFInsight [43] uses the length and number of such paths reaching syscalls to judge the ease of mounting attacks. We argue that this approach shares the same issue as the other metrics since it remains unclear which non-syscall gadgets are available and how path lengths correspond to exploitability.

Another approach is to collect gadgets useful to an adversary and measure their availability with and without CFI enforcement [30, 97]. In this case, it has to be defined which gadgets are considered useful. Multiple approaches exist to analyze gadget quality by determining the expressiveness of gadgets and their capabilities to set up function calls [18, 42]. We are unaware of any CFI-related work that uses such metrics for their evaluation.

3 Adversary Model and Known Attacks

CFI enforcement is a mitigation technique that aims to prevent code-reuse attacks by restricting the allowed targets of indirect control-flow transfers [2]. Therefore, CFI enforcement is intended to prevent even a strong adversary from executing arbitrary code [2, 64, 97, 110]. This adversary can read and write from/to arbitrary addresses in memory by ex-

ploiting already existing memory corruption vulnerabilities. The CFI adversary model assumes that by using these capabilities, the adversary can break Address Space Layout Randomisation (ASLR) [84, 128]. By common assumption, the adversary can perform arbitrary calculations, for example, by sending data to their server or by abusing existing scripting capabilities as present in web browsers. Since an adversary with arbitrary write capabilities could overwrite any checks, the enforcement of a Write ⊕ Execute (W⊕X) policy [104] is typically assumed to protect the integrity of code sections. Because CFI focuses on protecting individual control-flow transfers, CFI schemes generally cannot prevent data-only attacks, which only modify non-control data [21, 23].

Attacks Several generic attacks on CFI are known in the literature. The first category of attacks exploits imprecision in the enforced CFG. For instance, [31] studies Call-preceded Gadgets, assuming that the backward-edge protection only restricts returning to a legitimate call site but does not restrain the choice of call sites. This does not hold for shadow stacks, and only to some extent for PA, as discussed in Section 4.2.1, and is hence not fully applicable to programs that are adequately protected with either a hardware-based shadow call stack or PA. In the same category, [50] analyses the availability of so called Entry Point Gadgets, which are sequences of useful instructions that start at a function’s entry point and end with an indirect call or jump. Similarly, [38] introduces the notion of Argument Corruptible Indirect Call Site (ACICS) gadgets, which are pairs of indirect call sites and security-sensitive target functions that are reachable from the corresponding call sites. As the name suggests, a core property of these ACICS gadgets is that the attacker can control the arguments of the corrupted call site to gain additional capabilities (e.g., arbitrary code execution in the best case).

Fundamentally, the previously covered CFI schemes can only limit the number of available gadgets, not guarantee their absence. For coarse-grained schemes such as WCFG

and Indirect Branch Tracking (IBT), this means that the set of potential entry points of ACICS gadgets consists of all functions that are marked as valid call targets. Fine-grained CFI implementations like Clang’s schemes and XFG limit valid call targets per call site even more. Their protection implies that entry point gadgets must be chained so that the associated type of the call site at the end of the gadget matches the type of the next gadget or the gadget dispatching function. This exact scenario is covered by [41], which uses so-called Linker Gadgets to traverse the CFG in a policy-adhering fashion. Finally, the Counterfeit Object-oriented Programming (COOP) technique [95] chains fake objects with virtual function table pointers pointing to the functions to be called. This approach only works if C++ semantics are not adequately enforced, and hence is only applicable to WCFG, BTI or IBT, but not the type-based LLVM CFI and XFG schemes.

Besides these works, there are studies covering interactions between compilers, runtime, and CFI schemes leading to bypasses. Such interactions include the compiler spilling sensitive registers to the unprotected stack [29], compiler-introduced double-fetches that enable Time-of-check to Time-of-use (TOCTOU) attacks [122], and exception handling mechanisms that can be abused for control-flow hijacking [36]. Further works focus on data-only attacks to bypass CFI [21, 23, 58]. Such attacks break most CFI schemes since they fall outside the typical CFI adversary model.

4 CFI Scheme Internals

In this section we categorize existing schemes that we found relevant in practice and describe how they work. Refer to Table 1 for an overview.

4.1 Software-based Forward-edge CFI

CFI mechanisms for forward- and backward-edge protection can be implemented either purely in software [19, 66, 123] or based on hardware support [33, 105]. From the security perspective, we found that existing hardware-based forward-edge CFI mechanisms are not inherently more secure than schemes implemented entirely in software. Although Clerq et al. argue in [33] that software CFI instrumentation code can be bypassed if the adversary can change the page permissions of code to writable, this also applies to hardware-based schemes such as Intel’s CET or ARM’s PA and BTI. In addition, there is already the $W\oplus X$ policy to prevent such attacks, which is typically hardware-enforced [104]. Under it, an adversary must first overcome CFI to disable this policy, at which point CFI has already been broken.

4.1.1 LLVM Clang CFI

LLVM’s CFI implementation [107, 110] is part of the compiler front-end Clang and supports languages in the C fam-

ily, including C++. It protects indirect function calls, calls via pointers to member functions, virtual function calls, non-virtual function calls using polymorphic classes (i.e., classes declaring or inheriting virtual functions), and invalid casts of polymorphic classes¹.

The enforced policy follows the type system of the source language, e.g., a function pointer of a specific type is only allowed to call functions with a compatible signature. Consequently, all unique function signatures and class hierarchies form their own equivalence classes, and LLVM CFI is, therefore, a fine-grained CFI scheme. LLVM’s CFI checks can be divided into inlined local checks performed in the current module and Cross-Dynamic Shared Object (CDSO) checks crossing library boundaries.

Local CFI Local checks use a bit-vector-based approach. For indirect function calls involving function pointers or pointers to member functions, a jump table is generated during compilation for each unique function signature, which contains all related address-taken or exported functions. In addition, each call site is instrumented with instructions that check whether the call target is a member of the table belonging to the static type of the function pointer. Virtual and non-virtual function calls and casts to polymorphic classes are checked with a bit-vector, encoding valid vtable address points for the corresponding class type [108]. This more elaborate check is necessary because sub-classes may implement new virtual functions, resulting in vttables of different sizes, so a simple alignment and range check does no longer work.

Cross-DSO CFI When the program calls an exported function of another module, its type identifier must be derived to perform the CFI check. Hence, a direct table- or vtable- based check is infeasible in such a case, as the address of the correct table is unknown. To solve this problem, CDSO-compatible modules export the `__cfi_check` function, which is invoked by the calling module with the type identifier of the function pointer or class used in the checked call-site, and the address of the target function. This function can then check to confirm that the given target address has a matching type.

The corresponding module must be determined to find the correct `__cfi_check` function belonging to a target address. For that, CDSO-compatible programs maintain a CFI-shadow mapping that allows getting the `__cfi_check` address of the module a given address is located in. The lookup of the entry in the CFI-shadow and calling the correct `__cfi_check` function is handled by `__cfi_slowpath`. At runtime, functions affecting loaded modules such as `dlopen` must be intercepted to adjust the CFI-shadow mapping accordingly.

We find that the necessity to update the CFI-shadow mapping introduces a potential race condition, which we discuss further in Section A.2 in the appendix.

Unprotected Libraries LLVM’s CDSO CFI scheme allows loading unprotected libraries (i.e., without `__cfi_check`). In

¹We focus on control-flow transfers in this paper, as casts are not a typical concern of CFI. LLVM just uses the same mechanism to check them.

this case, the corresponding library is marked as unchecked in the shadow mapping, and indirect calls to targets in it always succeed. This principle applies even to protected indirect calls that are not intended to target functions in this library. If such calls are corrupted to transfer into an unprotected library, `__cfi_check` will dispatch them successfully, because no information is available regarding valid targets in this library. As a consequence, mixing protected and unprotected libraries diminishes CFI’s security guarantees, as large libraries are bound to contain useful gadgets. Our evaluation in [Section 5.1](#) shows that this is a serious issue across all major Android-based platforms.

4.1.2 Windows Control Flow Guard

Microsoft Windows has a proprietary CFI implementation, which is integrated into the operating system itself. It is called Control Flow Guard (WCFG) [76], and was first released in November 2014 [14]. WCFG enforces a CFI policy where indirect calls must target a known address-taken or exported function. This means there is only a single equivalence class, and WCFG is hence a coarse-grained CFI scheme. Indirect calls, including virtual calls using a vtable, are either protected with a call to a check function or entirely replaced with the call to a dispatch function that performs the WCFG check and dispatches the call afterward.

Implementation To mark functions that can be called indirectly, the `Load Configuration` structure that is part of the portable executable (PE) format is added to the executable during compilation. This structure contains various WCFG-related fields, including function pointers to the check/dispatch functions and the address of the table containing the relative addresses of all WCFG-protected functions [77]. Scanning this table when dispatching an indirect call is inefficient, which is why a bitmap marking valid functions is constructed when loading a program [124]. As the compiler aligns functions to 16-byte boundaries, a single bit per 16 bytes of address space would be sufficient to mark functions in the bitmap. Windows uses two bits to support unaligned functions, e.g., handwritten assembly.

Security Previous research identified various weaknesses in WCFG, such as gadgets that are contained in unaligned 16-byte blocks [15], or memory-based indirect calls via writable function pointers [102]. Independent of WCFG, multiple works raise issues of coarse-grained CFI schemes [31, 50, 95], implying that WCFG cannot prevent memory corruption attacks from achieving arbitrary code execution.

4.1.3 eXtended Flow Guard (XFG)

Microsoft is developing a WCFG successor called eXtended Flow Guard (XFG) [120], which is already available on Windows preview builds, even though undocumented. XFG uses a type-based policy similar to LLVM’s CFI implementation (cf.

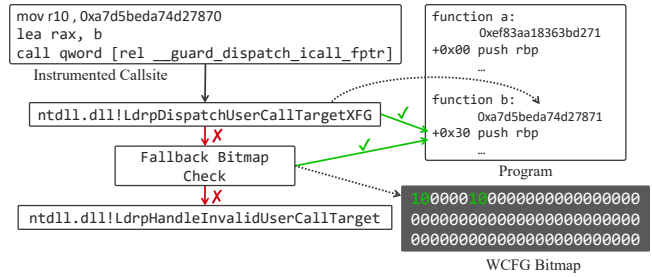


Figure 2: Reverse-engineered XFG check flow. Green and red arrows represent flows after a successful and failed check, respectively. The dotted arrows mark data fetches.

[Section 4.1.1](#)), but it is based on embedded labels to perform CFI checks. We extend existing third-party works treating XFG [39, 73] by reverse-engineering relevant XFG internals to compare its security properties with the other CFI schemes. In the implementation at the time of writing (Insider Preview build 23440), a 64-bit type hash precedes all XFG instrumented functions. During runtime, the XFG dispatch function checks whether a given call target has the expected type hash or else the program is terminated. The type hashes are derived from a combination of a function’s signature, its name, and the class hierarchy in case of virtual function calls. Consequently, cross-module calls to XFG-instrumented functions work without additional overhead since type hashes directly precede the functions. On some architectures such as x64, `MOV` instructions for loading the expected type hash contain the type hash itself as part of the instruction encoding. Such instructions would then produce unintended call targets. To address this issue, the instrumentation code loads the expected type hash with the last bit flipped, and the dispatch function undoes this bit flip before comparing it with the stored label. [Figure 2](#) depicts the whole XFG flow: First, an instrumented call site is redirected to the dispatch function. This function is configured by the loader, which sets the `__guard_dispatch_icall_fptr` function pointer depending on whether WCFG or XFG should be enforced. The XFG dispatch function loads the hash located at the quad-word prior to the target address, flips the last bit of the expected value, and compares them. If they match, the call is dispatched. Else, the WCFG bitmap is consulted to check if the target is a known function entry address. The target address is called if it is. Otherwise, a function is called to determine the consequences of this CFI violation.

XFG is backward-compatible with WCFG-protected programs. After a failed check for a matching type hash, the XFG dispatch function also consults the WCFG bitmap to check whether the target is a WCFG-protected function, and if so, may still allow the call. XFG-instrumented functions hence use the fourth remaining bitmap state to encode that they should not be valid WCFG call targets.

4.2 Hardware-based Forward-edge CFI

This section introduces four hardware-based CFI schemes targeting forward-edge protection. These schemes are coarse-grained, except for PA and FineIBT, which allow to implement different policies. It follows that they are less precise than LLVM CFI or XFG. However, due to their implementation in hardware, they are more efficient.

4.2.1 ARM Pointer Authentication

Pointer Authentication (PA) [12] is a security extension for the AArch64 architecture, which allows for protecting pointer integrity by inserting a cryptographic Message Authentication Code (MAC) called Pointer Authentication Code (PAC) into the unused upper bits of pointers. Unused bits are available because the virtual address space size does not occupy the full 64-bit of register width. Consequently, their exact number depends on the specific implementation. PA was introduced in ARMv8.3-A in 2016 [17], and later also for the microprocessor profile starting with the ARMv8.1-M architecture update, as announced in 2021 [80]. To operate on PACs, the PA extension adds a variety of instructions that can be divided into four categories [12]:

- *PAC** instructions to generate and insert a PAC,
- *AUT** instructions to authenticate and remove the PAC for subsequent use of a pointer,
- *XPAC** instructions to strip the PAC from a pointer without authenticating it, and
- (*no common prefix*) combined instructions that perform a PA-related operation and a related instruction together.

The MAC algorithm uses one of five keys and a 64-bit context value that allows tying pointers to a specific context. These keys are stored in CPU registers and are not accessible from exception level EL0 (user space).

Since PA is more of a building block for CFI schemes rather than a mitigation on its own, there are different PA-based implementations that differ in their respective characteristics. While the protection of forward-edge flows is covered in multiple research works such as [40, 57, 68, 94, 96, 126], Apple's `arm64e` ABI [10] is the only case where we observed a PA-based forward-edge scheme in practice.

4.2.2 ARM Branch Target Identification

ARM's BTI feature is a forward-edge CFI scheme and an alternative to custom PA-based schemes. It introduces the BTI instruction, which takes a target operand specifying what kind of control-flow transfer is allowed to target the instruction. The target operand can be `c`, `j`, or `jc`, indicating that the corresponding BTI instruction can be targeted by calls, jumps, or both respectively [12]. Jumps that target the registers X16 or

X17 are also compatible with the `c` target. This enables the use of jumps to these registers in Procedure Linkage Table (PLT) entries or for indirect tail-calls [92]. BTI allows configuring which memory page should be protected. Outside protected memory regions, the BTI executes as NOP [12].

4.2.3 Intel Indirect Branch Tracking

The IBT feature is the forward-edge control-flow transfer protection component of Intel CET. It is a coarse-grained CFI scheme using label instructions for marking valid call targets, and thus very similar to the proposal in the seminal work on CFI [2] and ARM's BTI feature. The two label instructions that IBT adds are `ENDBR32` and `ENDBR64`, for the 32-bit compatibility mode and the 64-bit mode, respectively.

The CFI policy enforced by IBT is straightforward: If an indirect call or jump is encountered, the next instruction executed must be a label instruction. If it is not, the control protection exception is raised [56]. There might be instances, such as switch-case constructs, where the control-flow transfer target resides in read-only memory or where IBT is undesired for some other reason. To support such instances, CET supports a no-track prefix that marks the subsequent `CALL` or `JMP` as not requiring a `ENDBR` instruction as the target. For backward compatibility, it is also possible to set up a bitmap that marks memory pages where the same exception applies [56].

4.2.4 FineIBT

FineIBT [44] is a hybrid CFI scheme, which improves the precision of coarse-grained hardware-based schemes while preserving their performance gains. While the general approach is mostly architecture-agnostic, their implementation targets Intel's BTI as suggested by the name of their scheme. The fundamental idea is that if a coarse-grained scheme like BTI or IBT protects a program, all indirect control-flow transfers are already limited to target particular instructions (i.e., `ENDBR64` for IBT), and instrumentation code only needs to be placed at these locations. This restriction means that the policy check can be executed *after* the control-flow transfer occurred since the hardware-based scheme guarantees that only such locations can be indirect call targets. Compared to full-software implementations of this approach like XFG, FineIBT avoids loading a label from memory before taking an indirect control-flow transfer. It follows that FineIBT is compatible with execute-only memory.

4.3 Software-based Backward-edge CFI

Shadow call stacks are a common approach to protect backward-edge control-flow transfers. They protect saved return addresses against memory corruption attacks by saving them to an isolated memory region. Since such metadata must be dynamically updated during runtime, it cannot be

protected by marking it read-only like CFI checks [20]. Solving this issue in software is challenging, as it either requires full Software Fault Isolation (SFI) or hardware-supported isolation [2, 20]. Often, software-based shadow call stacks rely on information hiding to protect the shadow stack area. However, it has been shown that due to information disclosure attacks [48, 84], this approach cannot withstand the CFI adversary [128]. Recent works propose re-randomization as a solution to such attacks [119, 129, 130]. They continuously re-randomize the addresses of protected areas or the addresses contained therein and thus limit the use of information leaks to an adversary. Another problem of software-based shadow stacks is that on some platforms such as x64, where call instructions directly push the return address on the stack, there is a timing window for a race condition between the call instruction and the return address being written to the shadow stack [2]. These shortcomings aside, a few software-based shadow stack approaches are found in practice.

4.3.1 LLVM Shadow Call Stack

LLVM implements a shadow stack scheme for the AArch64 architecture. The design supported by Clang is a compact shadow stack based on information hiding, i.e., the shadow stack maintains its own stack pointer in register X18, which must remain unknown to the attacker to offer any protection. This implies that this register must not be spilled onto the stack (e.g., when calling into unprotected code), or else an attacker could obtain the shadow stack location from there. Because of its nature as software-based shadow stack, which is only protected through information hiding, this design is inherently weak against attacks that try to uncover hidden locations in memory. Corresponding example attacks based on allocation oracles are proposed in [84]. A larger guard region can be allocated to increase the resistance against such attacks, containing the shadow stack itself. Thus, an allocation oracle will only find the whole guard region instead of the exact location of the shadow stack. Android implements this approach in its Libc [89].

4.3.2 SafeStack

Besides the shadow call stack, LLVM also implements SafeStack [64, 106]. The key idea of SafeStack is to separate safe and unsafe memory objects on the stack. Memory objects considered safe are return addresses, stack spills, and local variables that are not address-taken but only accessed via the frame pointer. Everything else is stored on the unsafe stack. The implementation relies on information hiding to protect the safe stack area and hence suffers the same associated weaknesses as the shadow call stack [47].

4.4 Hardware-based Backward-edge CFI

Hardware-based backward-edge CFI schemes address the weaknesses of software-based designs. They can implement atomic instructions to prevent race conditions and provide memory isolation for sensitive regions.

4.4.1 PA-based Approaches

One common scheme uses PA to protect saved return addresses on the stack by tying them to the stack pointer value at function entry [96]. This can efficiently be done by using the PACIASP and AUTIASP instruction pair, which sign and verify the link register with the current stack pointer value as context. Since the link register is used to store return addresses by BL instructions, these instructions can be placed at the start of the function prologue and epilogue, respectively. The PACIASP and PACIBSP instructions have implicit BTI behaviour, making them valid call targets [12] under BTI enforcement. Consequently, programs using PA to protect the return address with these instructions do not need an extra BTI instruction at the start of a function.

In comparison to a regular shadow call stack there is no memory overhead for the shadow call stack area. Neither loader nor operating system needs to do additional work besides the operating system managing the PA keys themselves, which is required for any PA-based scheme. Due to the nature of stack-based function calls, stack pointer values are not guaranteed to be unique to a specific function during program execution. This enables substitution attacks, where an adversary exchanges the saved return address with another unintended target that has been leaked earlier [96]. Consequently, compared to the hardware-based shadow call stack, the PA design offers weaker security guarantees.

4.4.2 Intel CET Shadow Call Stack

Intel CET features a hardware-based shadow call stack for backward-edge protection [97]. This shadow stack is implemented as a descending second stack designated for storing only return addresses. A new SSP CPU register holds the current shadow stack pointer. This register can only be modified by dedicated new instructions for shadow call stack management, which are intended for either the operating system or libraries that need to handle special stack unwinding cases. During normal program execution, the call and return instructions are shadow stack aware if the shadow stack feature is enabled. This means that the call instructions do not only push the return address to the unprotected program stack but also to the shadow stack. Similarly, the return instructions compare the return address stored on the shadow stack to the return address stored on the save stack and only continue if they match. Otherwise, the #CP exception is raised [56]. Adapting the semantics of these instructions means that existing programs do not need to be recompiled to benefit from shadow stack

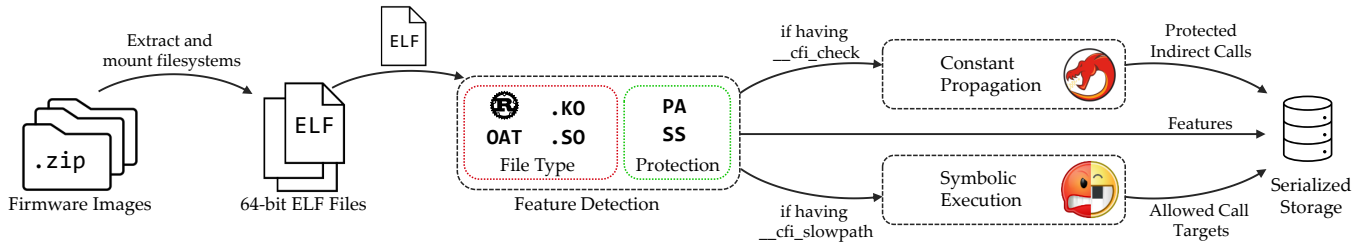


Figure 3: Android analysis pipeline.

protection as long as they do not implement custom unwinding logic. Only the runtime and standard libraries that handle unwinding must be modified to be shadow stack aware.

The shadow stack region is protected by adding a new attribute to the address translation to indicate a shadow-stack page. Pages marked as such cannot be modified by regular store instructions, protecting the integrity of the saved return addresses. In addition, call and return instructions fault if the page where they try to store or fetch the saved return address from the shadow stack, respectively, is not marked as shadow page [56]. This implies that even with complete control over a program’s virtual memory, an adversary cannot manipulate return addresses without access to either gadgets with management instructions or a primitive to create shadow-stack pages under their control.

5 Study on CFI Adoption in Platforms

In this section, we study the usage and effectiveness of the CFI schemes covered in the previous section on the Android, Linux, and Windows platforms.

5.1 Android Study

The Android Open Source Project (AOSP) [7] uses LLVM’s CFI implementation to enforce CFI since Android 8.1 for a set of components [4]. In user space, CDSO mode is used, enabling protected programs to perform indirect calls into shared libraries [108]. Since LLVM’s CFI implementation only protects indirect forward-edge control-flow transfers, Android (on AArch64) supports LLVM’s shadow call stack for backward-edge protection [6]. However, this shadow call stack is based on information hiding and is not designed to resist the typical CFI adversary. We also observed the PA-based return address protection scheme used as a more secure alternative. Bionic provides the necessary runtime support for the shadow call stack and LLVM CFI. Android 12 added support for ARM’s Memory Tagging Extension (MTE) [1, 8], a hardware-based memory safety mitigation which implements memory tagging. While the first phones supporting MTE have been released [16], it is not a CFI mitigation and disabled by default, and we do not include it in our study.

5.1.1 Android Image Analysis Setup

We analyze 33 Android images of popular flagship devices to compare the CFI usage on Android across multiple device manufacturers. Based on their market share, we select the smartphones Samsung Galaxy S22, Xiaomi 13, Vivo V25, and Oppo Reno 8 5G [9]. In addition, we include the pure AOSP Generic System Images (GSIs) for Android 10 to 14 and a system image from Google’s Pixel 7 phone, because Google is part of the driving force behind Android. We also include the GrapheneOS firmware for the Pixel 7, which promises increased security and privacy [27], to see if it has better CFI coverage than the Google Pixel 7 firmware. Based on the results of these images, we picked the Samsung Galaxy S20 and the Xiaomi Mi 10 for analysis over time. They both have publicly available firmware archives ranging from Android 10 to Android 13. The full versions and source URLs of all firmware images are specified in the appendix (Table 8).

Since all of these phones are ARM-based, we enumerate AArch64 ELF files on their Android firmware images and run an analysis on them. An overview of our analysis pipeline is shown in Figure 3. We extract the following characteristics:

ELF Type We distinguish between binaries (i.e., executable programs), shared libraries, and loadable kernel modules (with a .ko extension).

General LLVM CFI Usage To determine general LLVM usage, the analysis checks for the existence of the exported `__cfi_check` function, which is always present if the binary was compiled with LLVM CFI. We manually check samples to confirm that all vendors built their applications with CDSO.

Shadow Stack Usage The analysis searches for instructions unique to shadow-stack-protected binaries to detect shadow-stack usage. One such instruction is `ldr LR, [x18, #-8]!`, which is used to restore the link register from the shadow stack. We examine multiple random samples of files containing this instruction and found that it is only used for the shadow stack and does not appear in other contexts.

Pointer Authentication PA is used to protect return addresses in some of the files. We detected PA protection by scanning for related instructions. This approach introduces imprecision as it counts files where only certain functions are PA-instrumented as PA-protected. However, we deem this approach sufficient to understand the overall distribution of fully unprotected binaries. We also scanned for BTI related

Table 2: CFI coverage of Android firmware images.

Vendor & Devices	Files Total [count]			LLVM CFI Protected [%]				Shadow Stack Protected [%]			Pointer Auth. Protected [%]		
	Binaries	Libraries	Kernel Modules	Binaries	Libraries	Kernel Modules	Kernel	Binaries	Libraries	Kernel Modules	Binaries	Libraries	Kernel Modules
GSI 10	142	1240	0	4.93	9.19	n/a	n/a	0	0.24	n/a	0	0	n/a
GSI 11	154	1383	0	7.79	10.12	n/a	n/a	0	0.14	n/a	0	0	n/a
GSI 12	167	1708	0	8.98	10.6	n/a	n/a	0	0.18	n/a	12.57	5.62	n/a
GSI 13	173	1893	0	8.67	9.77	n/a	n/a	0	0.11	n/a	35.26	18.17	n/a
GSI 14	150	1729	0	9.33	8.04	n/a	n/a	0	0.12	n/a	39.33	22.15	n/a
Xiaomi 13	535	2601	270	47.85	65.24	100.0	✓	0	0.08	100.0	94.21	92.62	97.78
Google Pixel 7	234	998	241	7.26	16.83	100.0	✓	0	0.2	99.59	12.82	9.52	99.17
GrapheneOS Pixel 7	233	994	241	7.3	16.9	100.0	✓	0	0.2	99.59	12.88	9.56	99.17
Oppo Reno 8 5G	351	2316	9	6.27	8.25	0	✗	0	0.17	0	7.98	4.49	0
Samsung Galaxy S22	289	1775	6	7.27	12.23	100.0	✓	0	0.28	100.0	23.53	17.18	100.0
Vivo V25	361	2771	10	5.82	6.75	0	✗	0	0.14	0	11.36	4.19	0
S20 2020-02-19 10	255	1573	1	3.14	6.74	0	✗	0	0.25	0	0	0	0
S20 2020-05-15 10	254	1572	1	3.15	6.74	0	✗	0	0.25	0	0	0	0
S20 2020-10-14 10	257	1595	1	3.11	5.58	0	✗	0	0.25	0	0	0	0
S20 2020-11-23 11	264	1395	1	4.92	9.82	0	✗	0	0.22	0	0	0	0
S20 2021-05-17 11	271	1430	1	4.8	9.58	0	✗	0	0.21	0	0	0	0
S20 2021-10-20 11	274	1447	1	4.74	9.47	0	✗	0	0.21	0	0	0	0
S20 2021-12-23 12	273	1527	0	5.86	11.92	n/a	✗	0	0.2	n/a	5.86	4.32	n/a
S20 2022-04-26 12	276	1536	0	5.8	11.78	n/a	✗	0	0.2	n/a	6.16	4.62	n/a
S20 2022-09-27 12	276	1536	0	5.8	11.78	n/a	✗	0	0.2	n/a	6.16	4.62	n/a
S20 2022-10-24 13	278	1582	0	5.76	12.2	n/a	✗	0	0.19	n/a	20.14	15.49	n/a
S20 2023-02-20 13	279	1592	0	5.73	12.12	n/a	✗	0	0.19	n/a	20.07	15.39	n/a
S20 2023-07-26 13	280	1599	0	5.71	12.45	n/a	✗	0	0.19	n/a	20.0	15.82	n/a

For the S20 firmware images, the security patch level and the Android version are given. Data for the Mi 10 in Table 5 in the appendix.

instructions, but found them too rare for consideration.

Kernel CFI Configuration Clang’s kernel CFI is enabled by the `CONFIG_CFI_CLANG` Kconfig flag [5]. To determine if the kernel of a firmware image enables CFI we use the `extract-ikconfig` script from the Linux repository [70] and double check the decompressed kernel image for CFI-related symbols and strings.

Rust Source Language All analyzed Rust binaries were compiled without CFI protection and hence excluded from the statistics. These files were detected by checking their symbols for Rust-specific functions (e.g., `__rust_alloc`).

OAT Files Some ELF files can contain ahead-of-time compiled DEX code in a custom OAT format [91]. Such files are not CFI-protected and can be detected by a combination of specific symbols, such as `oatdata` and `oatdex`. We exclude them from the statistics.

Library Dependencies Unprotected dependencies massively contribute to the number of available call targets. We collected each analyzed file’s dependencies as indicated in the corresponding ELF structure.

Type Identifiers Passed to `__cfi_slowpath` Extracting type identifiers passed to `__cfi_slowpath` shows which function or class types are actually used for CFI checks. In our analysis, this is done by leveraging Ghidra’s [3] constant propagation analysis. First, the analysis searches calls to `__cfi_slowpath`, which is imported from `bionic` and hence easy to locate. Then, it extracts the first argument that is passed to the function. Since type identifiers are constants, this is well-doable by static analysis.

Type Identifiers and Their Associated Address Ranges Given a binary file, its equivalence classes and their mem-

bers can be constructed by extracting type identifiers and their associated address ranges. The core observation is that this information is encoded in the exported `__cfi_check` function, which is easy to locate. In addition, this function is independent of any global state and uses only arithmetic and control-flow instructions, making it a good fit for symbolic execution [13]. Our implementation uses the Angr framework [99]. It progresses execution states until they either hit a return, indicating a successful check, or a call to `abort()`, in which case they are discarded. Afterward, the collected constraints on the type identifier and the address argument are solved for the successful states, resulting in a mapping from type identifiers to the address ranges of their jump tables or vtables. Finally, it detects whether a target range is for a jump table or a vtable by checking whether a branch instruction is found at the start of the first slot. On a side note, the results of the symbolic execution can also support program analysis. We discuss this further in Section A.1 in the appendix.

5.1.2 Evaluation

Unprotected Binaries We count the number of files with and without CFI protection (as indicated by the existence of the `__cfi_check` export) to determine the general coverage of CFI protection. We divide them into binaries, libraries, and loadable kernel modules and additionally count the shadow call stack usage. We filter out files that are unprotected by design, such as OAT files or binaries that were classified as Rust binaries. Table 2 depicts the numerical results for the CFI coverage of the remaining files. For both binaries and libraries, only the minority of files is CFI-protected, with the

Table 3: Unprotected library dependencies in binaries.

Vendor & Device	Min [%]	Mean [%]	Max [%]
GSI 10	83.78	92.84	100.0
GSI 11	75.24	86.93	100.0
GSI 12	61.76	83.56	100.0
GSI 13	58.97	83.12	100.0
GSI 14	63.41	82.46	100.0
Xiaomi 13	24.14	75.83	100.0
Google Pixel 7	58.97	84.65	100.0
GrapheneOS Pixel 7	58.97	84.65	100.0
Oppo Reno 8 5G	61.76	87.0	100.0
Samsung Galaxy S22	60.98	87.13	100.0
Vivo V25	58.97	88.48	100.0

only exception being the Xiaomi 13 firmware. Concerning loadable kernel modules, build settings seem to be more consistently activating CFI, i.e., all kernel modules are protected, or none are.

Unprotected Dependencies When looking at the few protected binaries, an essential factor is the number of unprotected dependencies they load. Unprotected dependencies have twofold implications: First, CFI-protected control-flow transfers targeting address in them cannot be checked. Hence, every byte in all executable sections in any unprotected dependency is a valid call target. Second, indirect control-flow transfers within the unprotected dependencies are not checked and can be used to reach arbitrary code in the protected parts.

For each of the selected Android firmware images, we compute the recursive dependencies of all protected binaries. Then, we calculate the ratio of unprotected to protected dependencies (selected results in Table 3, see Table 6 for the full results). Inspecting the unprotected libraries shows that system libraries such as `libc` are never protected. Since these libraries are large and offer a variety of gadgets, they pose an attractive target for adversaries [113]. All protected binaries on the analyzed images depend on at least one of them.

Backwards-edge Protection on Android We find that there are two prevalent approaches for protecting return addresses in AArch64-based Android. First, there is the LLVM shadow call stack [6, 109]. Bionic allocates the shadow stack area during process creation, and it is only protected by information hiding through ASLR. Programs not using the shadow call stack will simply overwrite `X18` and ignore the allocated shadow stack area. As seen in Table 2, no binaries and barely any libraries are compiled with the shadow call stack. A probable explanation for its low usage is that a PA-based scheme is used instead, which does not share the weakness against memory disclosure attacks.

Comparably many binaries and libraries use the PA instructions `PACIBSP` and `AUTIBSP` to protect return addresses by tying them to the stack pointer (cf. Section 4.2.1). We also observe some cases with generic PA instructions such as `autia1716` (which authenticates the value in `X17` with `X16` as the context), but they appear only in stack unwinding code.

Development over Time The datasets for the S20 and the

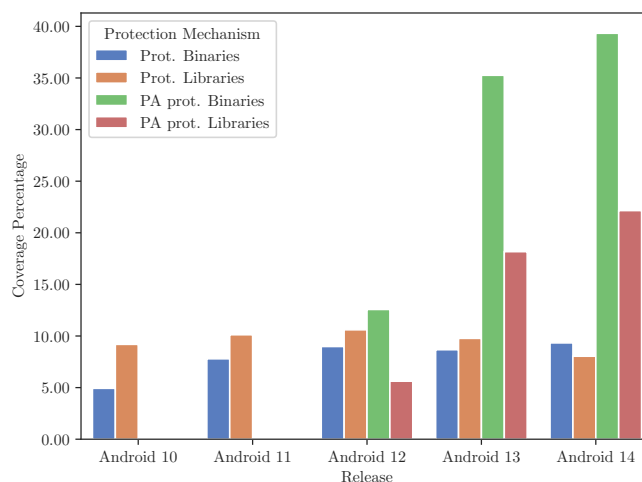


Figure 4: Development of mitigations on the GSI over time.

Mi 10 show that significant changes in CFI coverage occur only with new Android releases (cf. Table 2 and Table 5). Minor fluctuation in CFI coverage within the same Android version happens primarily due to the addition or removal of ELF files. However, some changes in the protection status of existing files, from protected to unprotected and vice versa, also happen in our dataset. Figure 4 shows the development over consecutive Android releases for the GSI. PA-based return address protection has been introduced with Android 12 and has since been extended, although the adoption rate slowed down with Android 14. The LLVM CFI coverage stagnates and, in some cases, even decreases with recent releases. Corresponding figures for the S20 and the Mi firmware can be found in Figure 6 in the appendix.

Memory-safe Code in Rust We find that Rust binaries are rather uncommon, with an average of only 14 files over all Android 13 firmware images. This might be subject to change as Google plans to primarily use Rust for new low-level code in Android [101]. Such a transition period comes with its own issues: Mixed binaries resulting from combined Rust and C / C++ codebases might be more vulnerable to memory corruption attacks because memory-safe parts can be abused to bypass mitigations such as CFI, which are deployed to protect unsafe code [74, 86]. Mixed binaries aside, Rust-based libraries will also contribute to the number of unprotected libraries and hence to the number of unprotected dependencies C / C++ binaries might have. While this issue could be addressed by compiling such libraries with LLVM CFI enabled, cross-language CFI support for Rust is not available yet [32].

Equivalence Class Size Frequencies For an indirect forward-edge control-flow transfer, the equivalence class size expresses the number of available call targets and hence the possible choices to an adversary. Therefore, the distribution of equivalence class sizes is a relevant metric to analyze type-based CFI schemes on a particular platform. Even though it shares the issue that the usefulness of targets is not con-

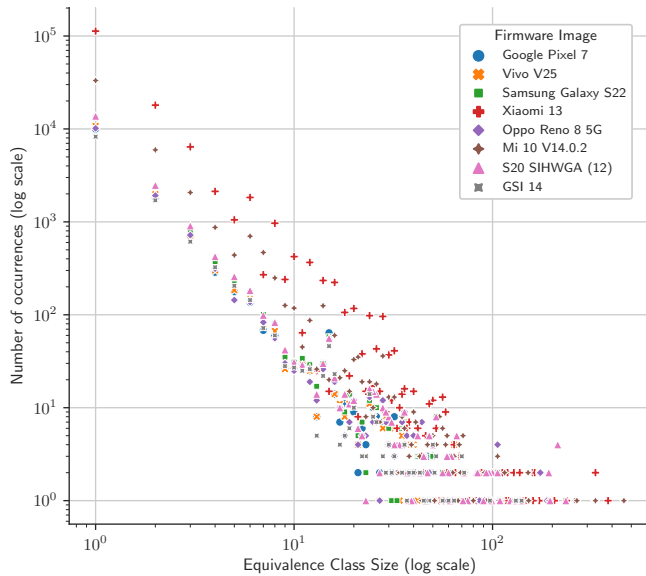


Figure 5: Equivalence class size distribution over all firmware.

sidered, it implicitly only considers full functions instead of arbitrary gadgets. Figure 5 depicts the frequencies with which each equivalence class size appears on the different firmware images. Equivalence classes are counted over all files on the corresponding firmware image as they were extracted from the `__cfi_check` function by our analysis. Equivalence classes consist of either jump tables, as used for checking function pointers, or vtables for checks involving C++ objects. Equivalence classes with the same type identifiers may appear multiple times if used in different files.

The distribution concentrates on single-member classes, with lesser frequencies of larger classes. This characteristic is desirable, as single-member classes imply that an adversary has no choice. We observe that the same outliers are often found across different firmware images due to the fact that they share a common codebase. Concerning the type of the tables these equivalence classes are based on, we found that jump tables are primarily responsible for the largest equivalence classes, with the exception of the Xiaomi 13 firmware.

Reachable Equivalence Classes Not all equivalence classes in a CFI-protected binary and its dependencies are actually reachable, i.e., there is no indirect control-flow transfer targeting them. Calls to `__cfi_slowpath` can be used to determine equivalence classes that are reachable from a binary. This approach is imperfect because `__cfi_slowpath` is also used for other CFI-related checks that are not strictly speaking indirect control-flow transfers. One instance of such a check is the `cfi-nvcall`, which checks non-virtual calls for polymorphic class types by checking the vtable pointer to ensure that the function is called with a compatible object [108]. This can be solved by following the CFG after the corresponding `__cfi_slowpath` call to determine the type of the next branch instruction, keeping only indirect calls. Such

additional analysis steps introduce imprecision and runtime overhead, and we decided to focus on CFI checks independently of their purpose. To get an idea of how a program’s dependencies increase the size of existing equivalence classes, we also included its dependencies for this particular analysis. Because this does not make sense for binaries with no protected dependencies, such files are ignored. Full results are depicted in Table 7 in the appendix. Besides for the Xiaomi firmware, the geometric mean of equivalence class sizes slightly decreases when considering dependencies. While this is contrary to the expectation that equivalence classes grow due to the merging of classes with common type identifiers, it can be explained by the fact that on these firmware images, protected libraries on average, have smaller equivalence classes than binaries. Therefore, the geometric mean decreases when also considering dependencies. The results restricted to reachable equivalence classes change as expected. Overall, reachable equivalence classes are expected to contain more than a single member because else there would be no need for function pointers or virtual functions. Table 7 contains equivalence class sizes below two because a significant portion (cf. Table 3) of dependencies are unprotected, and targets that are located in them cannot be considered.

5.2 Linux Study

The Linux kernel introduced support for Clang’s CFI scheme in release 5.13 [111]. As a more efficient alternative, Linux also supports FineIBT, starting with version 6.2 [71]. FineIBT user-space support is also in the works, but it requires PLT format modifications, thus leading to ABI changes [44]. For backward-edge protection in user space, the Linux kernel 6.6 introduced CET shadow stack support on corresponding platforms [112]. Applications need to signal shadow stack compatibility by setting the `SHSTK` flag in the Executable and Linking Format (ELF) note, and the kernel must be configured with the `X86_USER_SHADOW_STACK` flag [60].

The build configuration of shipped kernels and applications usually depends on the Linux distribution. Because there are many different Linux distributions and their market share is hard to quantify [35], we exemplarily pick the most recent Debian, Ubuntu, Fedora Workstation, and Arch Linux releases as representatives of widespread distributions. As of September 2023, none of them ship a kernel recent enough to support the CET shadow stack, and none enables `CONFIG_CFI_CLANG` by default. On the application side, these distributions set the `-fcf-protection` build flag by default to produce shadow stack and IBT protected binaries, even though kernel support for neither feature is available yet [25, 26, 28, 53]. Clang’s CFI scheme is not specified, probably because it is tied to Clang and unusable with other prevalent compilers such as GCC. On major Linux distributions, fully working mitigation combinations are not deployed yet. For this reason, we refrain from running a binary analysis study on Linux distributions.

Table 4: Windows 11 Insider Preview CFI coverage

File Type	Unprotected	Only WCFG	XFG	EC Size
Exe	2.68%	11.59%	85.73%	1.37 [G.M.]
DLL	2.62%	11.68%	85.70%	1.37 [G.M.]
Sys. DLL	0.91%	2.06%	97.04%	1.38 [G.M.]
Combined	2.63%	11.66 %	85.70%	1.37 [G.M.]

The Sys. DLL column covers all .DLL files located in C:\Windows\System32\ and subdirectories thereof.

5.3 Windows Study

We study the Windows 11 Insider Preview developer build 23440 with respect to WCFG and XFG coverage. WCFG-related metadata in the Portable Executable (PE) header allows us to reliably detect WCFG and XFG usage. The former can be detected by checking the `DllCharacteristics` field [77], while XFG protection is indicated by the `GuardFlags` field of the load configuration. All XFG-instrumented functions are listed in the `GuardCFFunctionTable` in the load configuration and have the corresponding bit set in the flags part of their entry [75]. Hence, we use the following approach: First, we traverse the `GuardCFFunctionTable` and extract all entries with flag-bit `0x08` set. Then, for each entry, we extract the 8 bytes representing the type hash, which precedes the address indicated by its address part. As a result, we obtain the set of XFG-instrumented functions and their type hashes.

WCFG and XFG Coverage First, we measure WCFG and XFG coverage by enumerating all PE files compiled for x64 with either a `.dll` or `.exe` extension. We restrict files to these extensions to exclude files that share the PE format but are irrelevant to our study, such as `.mui` files used for multilingual user interfaces. Additionally, we ignore files without executable sections since they do not require protection. Virtual DLLs are one example of such files [124]. Table 4 gives an overview of the distribution of protection schemes. Contrary to the results from our Android study, the majority of analyzed files on Windows are compiled with XFG instrumentation.

Equivalence Class Sizes The geometric mean equivalence class size is similar to the one we observed on Android (cf. Table 7, second column), even though slightly lower. Besides Windows 11 being a codebase unrelated to Android, a contributing factor to this difference could be that not every protected function in an XFG-instrumented PE file is necessarily XFG protected, as WCFG can be used to protect individual functions. We found that, on average, 95.94% of `GuardCFFunctionTable` entries were marked as XFG protected for files with XFG instrumentation. This means that the on average remaining 4.06% of targets must be considered members of every XFG equivalence class in the corresponding file. The entire distribution of equivalence class sizes is depicted in Figure 7 in the appendix and looks similar to the distribution on Android (cf. Figure 5).

Backwards-edge Protection on Windows WCFG and XFG only protect forward-edge control-flow transfers. Microsoft tested a software-based shadow stack called Return Flow Guard but found it affected by information leakage attacks and an exploitable race condition [14]. Instead, they use hardware-supported schemes on supported platforms: On recent x86-based systems, the shadow stack of Intel’s CET is used to protect backward-edge transfers [69] if the corresponding PE file sets the `CET_COMPAT` extended DLL characteristics bit.

On AArch64-based systems, recent Windows on ARM builds support PA for protecting return addresses [121]. Our analysis of the insider preview dev build 23419 yields a PA file coverage of 92%. To calculate this, we enumerate all `.exe` and `.dll` files for AArch64 and search them for PA-related instructions. Then, we filter out cases of instructions that only appear incidentally in executable sections. The remaining instructions consist of the `PACIBSP` and `AUTIBSP` pair used for signing and authenticating the return address with the stack pointer as context and the `B` key, and the `XPACLRI` instruction for stripping PACs from the link register. Windows 11 on ARM uses the basic PA scheme, in which each return address is tied to the stack pointer value at function entry (refer to Section 4.2.1). Since better designs exist (e.g. [57, 67]), it seems that the current implementation was deemed sufficiently secure, or the complexity or runtime overhead of such solutions was found unacceptable.

Bypassing XFG with Suppressed Functions WCFG supports function suppression, a feature to mark unsafe functions that should never be called indirectly [75]. Such functions are not placed into the WCFG function table and have no bitmap entry to mark them as valid functions. Developers can use this feature by using function modifiers in their code, and Microsoft uses it internally to protect system DLLs. In such DLLs, restricted functions are mostly related to control-flow tasks such as stack unwinding or exception handling.

We found that even though suppressed functions do not appear in the WCFG function table, they still have XFG type hashes. Consequently, they are valid call targets under XFG enforcement, as long as the corresponding call site has the same type hash. For suppressed functions of a sufficiently generic signature (i.e., no custom types appearing only in specific APIs), this implies that they can be reachable by an attacker, especially considering previously described techniques to reach such call sites [38, 41]. We reported this issue to Microsoft and expect that Microsoft will fix this in future releases of Windows and their MSVC compiler by omitting XFG type hashes for suppressed functions.

6 Related Work

Various CFI schemes have been treated in previous research. Thereby lay the focus on either compatibility [123], a combination of precision, security, and performance [19], techniques applicable to resource-restrained embedded and real-

time processing devices [78], the precision of binary-level techniques [114], and the security boundaries of different approaches [66]. [81] introduces a framework for comparing different CFI policies. Since this framework operates on source code, it solves a different task than our analysis. Equivalent studies also exist for hardware-based schemes [33, 65, 105]. Because these surveys mostly compare academic prototypes, they do not address the usage of CFI in practice. In [116], CFI equivalence classes in the Linux kernel are analyzed. Their approach differs from ours, as they extract CFI targets by using an instruction pattern instead of symbolic execution. Consequently, they do not consider CDSO calls.

Several works exist that explore approaches to automated firmware analysis. *Firmalice* [98] detects authentication bypasses in binary programs automatically. It uses symbolic execution to detect such vulnerabilities based on a general and architecture-agnostic model characterizing them. Similarly, [22] employs full-system emulation of Linux-based firmware to identify vulnerabilities by checking accessible web pages, enumerating Simple Network Management Protocol (SNMP) information, and attempting known exploits.

Related to memory safety, the work in [125] runs a large-scale analysis to study the coverage of different mitigations in embedded-device firmware. To detect present mitigations, they use static indicators, including the occurrence of certain strings or symbols, the existence of specific ELF sections, or flags in the program header. However, they do not consider CFI usage. Targeting specifically the Android platform, [51] analyzes Android firmware to investigate its patch level. It builds on multiple static analysis tools to detect missing patches, app attribute misconfigurations, and cryptographic misuse. To perform static analysis tasks targeting pre-installed apps in Android firmware, the *FirmwareDroid* framework [103] was proposed. It has been applied to study advertiser tracker libraries shipped with pre-installed apps. [37] investigates the security of such pre-installed apps, focusing on privilege-escalation vulnerabilities by using a custom static taint analysis.

7 Recommendations for Improving CFI

A comparison of the state of the art CFI research with the schemes found in practice shows a large gap between scientific implementations and their adoption. Researchers identified compatibility as a long-standing issue [123]. When looking at the two instances where production systems and compilers have integrated results from research efforts [44, 110], it becomes evident that corresponding authors had direct ties to the industry. The corresponding financial backing and interest in creating solutions that are applicable to production systems could explain why these authors underwent the effort of submitting patches to LLVM and the Linux kernel.

The CFI schemes observed in practice are primarily coarse-grained. With LLVM's type-based scheme and the intro-

duction of XFG, vendors are moving towards fine-grained schemes, which provide better security. This trend confirms that vendors are interested in moving forward and closing the gap between academia and industry.

Our analysis shows that equivalence classes of the fine-grained schemes tend to be small. Compared to coarse-grained schemes, this indicates a substantial improvement, but outliers exist and contribute to the choices of targets available to adversaries. Existing metrics for measuring CFI protection are insufficient to address this problem, therefore adding to the difficulty in evaluating the benefits of these mitigations.

We found that in the Android ecosystem, popular vendors rolled out CFI support differently over time. This indicates that even if there is a build environment that supports CFI and that is well-maintained and tested, adoption to real-world systems takes time. We strongly encourage vendors to ensure that CFI is applied to all binaries. From the vendor's perspective, system libraries should be shipped with CFI enabled to allow developers to benefit from compiling programs with CFI. Our tools will support vendors in analyzing their systems for potential gaps in CFI support, which might arise due to passing wrong compiler options in subprojects.

8 Conclusion

Our results show that CFI roll-out is not yet a finished process. We found the CFI coverage on Android lacking, especially regarding system-provided shared libraries. In these cases, CFI follows an all-or-nothing principle, meaning that security benefits are basically non-existent without a complete deployment. While the WCFG/XFG coverage we observed on Windows was better, it remains to be seen how long it takes until commercial off-the-shelf software builds are properly shipped with XFG protection once XFG is officially released.

With the increasing adoption of CFI, the hurdle for adversaries grows, who, in the best case, need to develop new exploitation techniques for each vulnerable program, hence raising the required effort and cost of attacks. In addition, specific bugs that would lead to arbitrary code execution without CFI can become unexploitable with CFI protection being applied, requiring adversaries to find stronger primitives. We hope to see CFI fully deployed in the future, along with more effective protection guarantees.

Acknowledgments

We thank the anonymous reviewers and the artifact evaluators for their helpful suggestions. This work has been funded by the German Research Foundation (DFG) in the project CRUST (grant number: 503199853).

Availability

The scripts described in the paper are published here: github.com/seemoo-lab/woot24_cfi_coverage_tools/

References

- [1] Armv8.5-A memory tagging extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf. Whitepaper.
- [2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS '05*, pages 340–353, New York, NY, USA, 2005. Association for Computing Machinery.
- [3] National Security Agency. Ghidra. <https://ghidra-sre.org/>, March 2019.
- [4] Android Open Source Project. AOSP - Control Flow Integrity. <https://source.android.com/docs/security/test/cfi>, 2022.
- [5] Android Open Source Project. AOSP - Kernel Control Flow Integrity. <https://source.android.com/docs/security/test/kcfi>, 2022.
- [6] Android Open Source Project. AOSP - ShadowCallStack. <https://source.android.com/docs/security/test/shadow-call-stack>, 2022.
- [7] Android Open Source Project. AOSP - Homepage. <https://source.android.com/>, 2023.
- [8] Android Open Source Project. AOSP - Arm Memory Tagging Extension. <https://source.android.com/docs/security/test/memory-safety/arm-mte>, 2024.
- [9] AppBrain. Top android phone manufacturers. <https://web.archive.org/web/20230317081510/https://www.appbrain.com/stats/top-manufacturers>, 2023.
- [10] Apple. Apple LLVM fork - pointer authentication documentation. <https://github.com/apple/llvm-project/blob/d43163879bdb9576fff7a5a269d36920eee4ac29/clang/docs/PointerAuthentication.rst>.
- [11] Apple. Apple platform security. https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf, May 2022.
- [12] ARM Holdings. Arm architecture reference manual for a-profile architecture. <https://developer.arm.com/documentation/ddi0487/ha/?lang=en>, February 2022.
- [13] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.
- [14] Joe Bialek. The evolution of CFI attacks and defenses. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf, 2018.
- [15] Andrea Biondo, Mauro Conti, and Daniele Lain. Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. In *NDSS*, San Diego, California, February 2018. Internet Society.
- [16] Mark Brand. First handset with mte on the market. <https://googleprojectzero.blogspot.com/2023/11/first-handset-with-mte-on-market.html>, 2023.
- [17] David Brash. Armv8-a Architecture: 2016 Additions. <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions>, 2016.
- [18] Michael D. Brown and Santosh Pande. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *Proceedings of the 12th USENIX Conference on Cyber Security Experimentation and Test, CSET'19*, page 5, USA, 2019. USENIX Association.
- [19] Nathan Burrow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), April 2017.
- [20] Nathan Burrow, Xinpeng Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, 2019.
- [21] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow bending: On the effectiveness of Control-Flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.
- [22] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.
- [23] Long Cheng, Hans Liljestrand, Md Salman Ahmed, Thomas Nyman, Trent Jaeger, N. Asokan, and Danfeng Yao. Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 114–128, 2019.
- [24] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, CODASPY '16*, pages 38–49, New York, NY, USA, 2016. Association for Computing Machinery.
- [25] Arch Linux Community. makepkg.conf. <https://gitlab.archlinux.org/archlinux/packaging/packages/pacman/-/blob/5fc0f6312b17abf707318c6909275721dab75a54/makepkg.conf>.
- [26] Debian Community. Debian dpkg-buildflags. <https://manpages.debian.org/unstable/dpkg-dev/dpkg-buildflags.1.en.html>.
- [27] GrapheneOS Community. GrapheneOS. <https://grapheneos.org/>.
- [28] Ubuntu Community. Compilerflags - default flags. <https://wiki.ubuntu.com/ToolChain/CompilerFlags>.
- [29] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 952–963, New York, NY, USA, 2015. Association for Computing Machinery.
- [30] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, pages 292–307, 2014.
- [31] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of Coarse-Grained Control-Flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.
- [32] Ramon de C Valle. Tracking issue for LLVM control flow integrity (CFI) support for rust. <https://github.com/rust-lang/rust/issues/89653>, 2023.
- [33] Ruan de Clercq and Ingrid Verbauwhede. A survey of hardware-based control flow integrity (CFI). *CoRR*, abs/1706.07257, 2017.
- [34] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of Path-Sensitive control security. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 131–148, Vancouver, BC, August 2017. USENIX Association.
- [35] DistroWatch. Distrowatch page hit ranking. <https://distrowatch.com/dwres.php?resource=popularity>.
- [36] Victor Duta, Fabian Freyer, Fabio Pagani, Marius Muench, and Cristiano Giuffrida. Let me unwind that for you: Exceptions to backward-edge protection. In *NDSS*, 2023.

- [37] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. FIRMSCOPE: Automatic uncovering of Privilege-Escalation vulnerabilities in Pre-Installed apps in android firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2379–2396. USENIX Association, August 2020.
- [38] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiropoulos-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 901–913, New York, NY, USA, 2015. Association for Computing Machinery.
- [39] Francisco Falcon. How the MSVC compiler generates XFG function prototype hashes. <https://web.archive.org/web/20230518085024/https://blog.quarkslab.com/how-the-msvc-compiler-generates-xfg-function-prototype-hashes.html>, November 2020.
- [40] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054, 2021.
- [41] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC '18*, pages 28–39, New York, NY, USA, 2018. Association for Computing Machinery.
- [42] Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the gadgets. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639, ESSoS 2016*, page 155–172, Berlin, Heidelberg, 2016. Springer-Verlag.
- [43] Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. CFInsight: A Comprehensive Metric for CFI Policies. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.
- [44] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. Fineibit: Fine-grain control-flow enforcement with indirect branch tracking, 2023.
- [45] Alex Gaynor. What science can tell us about C and C++'s security. <https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/>, May 2020.
- [46] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194, 2016.
- [47] E.K. Goktas, A. Oikonomopoulos, Robert Gawlik, Benjamin Kollenda, I. Athanasopoulos, C. Giuffrida, G. Portokalidis, and H.J. Bos. Bypassing Clang's SafeStack for Fun and Profit. In *Black Hat Europe*, November 2016.
- [48] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.
- [49] GRSecurity. Frequently asked questions about rap. https://grsecurity.net/rap_faq, 2023.
- [50] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, 2014.
- [51] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Chenglin Mao, Lingyun Ying, Song Liu, Xiaofeng Liu, Yuanzhi Li, Shanqing Guo, Meining Nie, and Haixin Duan. Can we trust the phone vendors? comprehensive security measurements on the android firmware ecosystem. *IEEE Transactions on Software Engineering*, 49(7):3901–3921, 2023.
- [52] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 1470–1486, New York, NY, USA, 2018. Association for Computing Machinery.
- [53] Red Hat Inc. Using rpm build flags. <https://src.fedoraproject.org/rpms/redhat-rpm-config/blob/f39/f/buildflags.md>.
- [54] Intel. Control flow enforcement technology. https://www.intel.com/content/dam/develop/external/us/en/documents/cat_c17-introduction-intel-cet-844137.pdf, December 2017.
- [55] Intel. New intel vpro platform portfolio. <https://www.intel.com/content/www/us/en/products/docs/processors/core/12th-gen-vpro-deskstop-processors-brief.html>, June 2022.
- [56] Intel. Intel® 64 and ia-32 architectures software developer manuals. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>, 2023.
- [57] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with pactight, 2022.
- [58] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 1868–1882, New York, NY, USA, 2018. Association for Computing Machinery.
- [59] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks. In *NDSS*, 01 2014.
- [60] The kernel development community. Control-flow enforcement technology (cet) shadow stack. <https://www.kernel.org/doc/html/v6.6-rc2/arch/x86/shstk.html>.
- [61] Mustakimur Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 195–211, Santa Clara, CA, August 2019. USENIX Association.
- [62] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 95–110, 2019.
- [63] Hyungseok Kim, Junoh Lee, Soomin Kim, Seungil Jung, and Sang Kil Cha. How'd security benefit reverse engineers? : The implication of intel cet on function identification. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–566, 2022.
- [64] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association.
- [65] Senyang Li, Weike Wang, Wenxin Li, and Dexue Zhang. Hardware-Based Software Control Flow Integrity: Review on the State-of-the-Art Implementation Technology. 11:133255–133280.
- [66] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, New York, NY, USA, 2020. Association for Computing Machinery.
- [67] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. Paystack: an authenticated call stack, 2019.

- [68] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinae Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association.
- [69] Jin Lin. Developer guidance for hardware-enforced stack protection. <https://techcommunity.microsoft.com/t5/windows-kerne-l-internals-blog/developer-guidance-for-hardware-enforced-stack-protection/ba-p/2163340>, 2021.
- [70] Linus Torvalds. The linux kernel: extract-ikconfig. <https://github.com/torvalds/linux/blob/6465e260f48790807eef06b583b38ca9789b6072/scripts/extract-ikconfig>.
- [71] LWN.net. Kernel release status. <https://lwn.net/Articles/924113/>.
- [72] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 941–951, New York, NY, USA, 2015. Association for Computing Machinery.
- [73] Connor McGarr. Exploit development: Between a rock and a (xtended flow) guard place: Examining XFG. <https://web.archive.org/web/20231024102105/https://connormcgarr.github.io/examining-xfg/>, August 2020.
- [74] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-Language Attacks. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.
- [75] Microsoft. Pe metadata. <https://learn.microsoft.com/en-us/windows/win32/secbp/pe-metadata>, June 2021.
- [76] Microsoft. Control flow guard for platform security. <https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard>, 2022.
- [77] Microsoft. Pe format. <https://learn.microsoft.com/en-us/windows/win32/debug/pe-format>, October 2022.
- [78] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. Survey of Control-flow Integrity Techniques for Real-time Embedded Systems. 21(4):41:1–41:32.
- [79] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. Drop the rop fine-grained control-flow integrity for the linux kernel. *Black Hat Asia*, 2017.
- [80] Alan Mujumdar. Armv8.1-m pointer authentication and branch target identification extension. <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension>, 2021.
- [81] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. Analyzing control flow integrity with LLVM-CFI. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19*, pages 584–597, New York, NY, USA, 2019. Association for Computing Machinery.
- [82] Ben Niu and Gang Tan. Modular control-flow integrity. *SIGPLAN Not.*, 49(6):577–587, June 2014.
- [83] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 914–926, New York, NY, USA, 2015. Association for Computing Machinery.
- [84] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 121–138, Austin, TX, August 2016. USENIX Association.
- [85] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851, 2021.
- [86] Michalis Papaevripides and Elias Athanasopoulos. Exploiting mixed binaries. *ACM Trans. Priv. Secur.*, 24(2), January 2021.
- [87] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-grained control-flow integrity through binary hardening. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164, Cham, 2015. Springer International Publishing.
- [88] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224. USENIX Association, June 2003.
- [89] Android Open Source Project. Increase the size of the shadow call stack guard region to 16mb. <https://android-review.googlesource.com/c/platform/bionic/+891622>.
- [90] Android Open Source Project. bionic. <https://android.googlesource.com/platform/bionic>, November 2022.
- [91] Android Open Source Project. Configuring art. <https://source.android.com/docs/core/runtime/configure>, October 2022.
- [92] LLVM Project. Branch target identification code-generation pass. <https://github.com/llvm/llvm-project/commit/4bc81028d48c0ab07e7b429d2a98ed6d15140a23>.
- [93] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.
- [94] Robert Schilling, Pascal Nasahl, and Stefan Mangard. Fipac: Thwarting fault- and software-induced control-flow attacks with arm pointer authentication. In Josep Balasch and Colin O’Flynn, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 100–124, Cham, 2022. Springer International Publishing.
- [95] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, 2015.
- [96] Qualcomm Product Security. Pointer Authentication on ARMv8.3 - Design and Analysis of the New Software Security Instructions. Technical report, Qualcomm Technologies, Inc., 5775 Morehouse Drive, San Diego, CA 92121, U.S.A., January 2017. Available here: <https://www.qualcomm.com/media/documents/files/white-paper-pointer-authentication-on-armv8-3.pdf>.
- [97] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP '19*, New York, NY, USA, 2019. Association for Computing Machinery.
- [98] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Fimalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.
- [99] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.
- [100] StatCounter. Operating system market share worldwide – jan 2020-jan 2023. <https://gs.statcounter.com/os-market-share#monthly-202001-202301>, March 2023.
- [101] Jeffrey Vander Stoep. Memory safe languages in android 13. <https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html>, December 2022.

- [102] Ke Sun, Ya Ou, Yahnui Zhao, Xiaomin Song, and Xiaoning Li. Never let your guard down: Finding unguarded gates to bypass control flow guard with big data. <https://www.youtube.com/watch?v=oD0rKvJcGbs>, March 2017. BlackHat Asia 2017 conference talk.
- [103] Thomas Sutter and Bernhard Tellenbach. FirmwareDroid: Towards Automated Static Analysis of Pre-Installed Android Apps. In *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 12–22, 2023.
- [104] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, USA, 2013. IEEE Computer Society.
- [105] Stefan Tauner and Mario Telesklav. Comparative analysis and enhancement of CFG-based hardware-assisted CFI schemes. *ACM Trans. Embed. Comput. Syst.*, 20(5s), September 2021.
- [106] The Clang Team. Clang Documentation - SafeStack. <https://clang.llvm.org/docs/SafeStack.html>.
- [107] The Clang Team. Clang documentation - control flow integrity. <https://clang.llvm.org/docs/ControlFlowIntegrity.html>, 2023.
- [108] The Clang Team. Control flow integrity design documentation. <https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html>, 2022.
- [109] The Clang Team. Shadowcallstack. <https://clang.llvm.org/docs/ShadowCallStack.html>, 2023.
- [110] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & LLVM. In *23rd USENIX security symposium (USENIX security 14)*, pages 941–955, 2014.
- [111] Sami Tolvanen. Linux kernel - add support for clang cfi. <https://github.com/torvalds/linux/commit/cf68fffb66d6d96209446bfc4a15291dc5a5d41>.
- [112] Linus Torvalds. Merge tag 'x86_shstk_for_6.6-rc1'. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=df57721f9a63e8a1fb9b9b2e70de4aa4c7e0cd2e>.
- [113] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 121–141, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [114] Raturaj K. Vaidya and Prasad A. Kulkarni. Assessing the effectiveness of binary-level cfi techniques. <http://arxiv.org/abs/2401.07148>, 2024.
- [115] Victor van der Veen, Dennis Andriess, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, New York, NY, USA, 2015. Association for Computing Machinery.
- [116] Jonathan Vexler. Characterization of forward-edge control-flow integrity targets in LLVM-compiled linux. https://cs.brown.edu/research/pubs/theses/masters/2020/vexler_jonathan.pdf, 2020.
- [117] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, pages 331–340, New York, NY, USA, 2015. Association for Computing Machinery.
- [118] Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. Object flow integrity. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1909–1924, New York, NY, USA, 2017. Association for Computing Machinery.
- [119] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. SafeHidden: An efficient and secure information hiding technique using re-randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1239–1256, Santa Clara, CA, August 2019. USENIX Association.
- [120] David Weston. Advancing windows security. <https://raw.githubusercontent.com/dwizzle/Presentations/master/Bluehat%20Shanghai%20-%20Advancing%20Windows%20Security.pdf>, 2019. Presented at Bluehat Shanghai 2019.
- [121] David Weston. Mwc 2022: The next microsoft pluton device + pac technology. <https://web.archive.org/web/20221104181427/https://blogs.windows.com/windowsexperience/2022/02/28/mwc-2022-the-next-microsoft-pluton-device-pac-technology/>, February 2022.
- [122] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. Warpattack: Bypassing CFI through compiler-introduced double-fetches, 2023.
- [123] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. Confirm: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1805–1821, USA, August 2019. USENIX Association.
- [124] Pavel Yosifovich, Mark E. Russinovich, Alex Ionescu, and David A. Solomon. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. Microsoft Press, 7 edition, 2017.
- [125] Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zacto, Mauro Conti, Georgios Portokalidis, and Jun Xu. Building Embedded Systems Like It's 1996. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.
- [126] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. ARM Pointer Authentication based Forward-Edge and Backward-Edge Control Flow Integrity for Kernels, 2019.
- [127] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 337–352, USA, 2013. USENIX Association.
- [128] Philipp Zieris and Julian Horsch. A leak-resilient dual stack scheme for backward-edge control-flow integrity. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASI-ACCS '18, pages 369–380, New York, NY, USA, 2018. Association for Computing Machinery.
- [129] Changwei Zou, Yaoqing Gao, and Jingling Xue. Practical Software-Based Shadow Stacks on x86-64. *ACM Transactions on Architecture and Code Optimization*, 19(4):1–26, 2022.
- [130] Changwei Zou, Xudong Wang, Yaoqing Gao, and Jingling Xue. Buddy stacks: Protecting return addresses with efficient thread-local storage and runtime re-randomization. *ACM Trans. Softw. Eng. Methodol.*, 31(2), mar 2022.

A Appendix

This appendix contains a discussion of how type-based CFI schemes can aid binary analysis tasks and a description and Proof of Concept (PoC) of the race condition in LLVM's CDSO CFI shadow mapping. Furthermore, it accommodates measurement data that did not fit into the main parts of the paper: [Figure 6](#) shows how the CFI coverage changed over time with regards to the S20 and Mi 10 firmware images. [Table 5](#) continues [Table 2](#) with measurements for the Mi 10 firmware, [Table 6](#) is an extended version of [Table 3](#) covering all firmware images, [Table 7](#) contains additional measurement data covering geometric means of equivalence class sizes in different categories, and [Table 8](#) specifies the exact versions of the Android images we look at.

A.1 Using CFI to Aid Binary Analysis

Security aside, CFI can also unintentionally help analyze binary programs. For example, it has been shown that the ENDBR instructions used for Intel's IBT feature can be used to improve function boundary detection [63]. We argue that type-based schemes such as LLVM CFI or XFG also aid binary analysis by allowing to infer function addresses and information about their signatures. We propose the following approach:

1. Pre-compute type identifiers for common type signatures and classes and store them for fast look-up.
2. Find and annotate indirectly-callable functions with their corresponding type identifier. For LLVM CFI, the type identifier can be obtained by symbolically executing the `__cfi_check` function (details in [Section 5.1.1](#)). For XFG, this is done by extracting the type identifier that precedes the function.
3. Group annotated functions by their type identifiers.
4. Look up identifiers in the pre-computed data set, and if found, mark the function with the corresponding type. If some function in a set has a known signature, the same type can be applied to all other functions in the same set. The same principle can be applied to manually assigned signatures, which can also be propagated to functions in the same set.

A fundamental limitation of this approach is that only indirectly-callable functions can be analyzed, as only they will have the CFI-related metadata. More specifically, functions that are neither exported nor address-taken will not have jump tables or type hashes, respectively. A second issue is that type identifiers can collide, producing the same identifier for different types. Our approach would then produce wrong function signatures. Even though such collisions are unlikely, they are possible. However, we still think our approach is an

interesting enhancement of typical binary analysis tools and leave a thorough evaluation for future work.

```
void thread_func(uintptr_t target) {
    // Simulate vulnerability to overwrite shadow mapping
    entry
    *reinterpret_cast<uintptr_t*>(target) =
        0xfffffffffffffff;
}

int main() {
    // Simulate leak of allocation address
    uintptr_t alloc = reinterpret_cast<uintptr_t>(
        mmap(nullptr,
            SIZE, PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANON | MAP_NORESERVE, 0,
            0));
    std::cout << "Allocation at: "
        << std::hex << alloc << std::endl;

    // Calculate the address where the mapping entry for
    // target_func is located
    uintptr_t target = (alloc + ((reinterpret_cast<
        uintptr_t>(&target_func) >> kShadowGranularity)
        << 1) - DISTANCE);
    std::cout << "Target at: " << std::hex << target <<
        std::endl
        << "Shadow base at: "
        << std::hex << (alloc - DISTANCE) << std::
            endl;

    // Start a thread to overwrite the target,
    // and trigger shadow mapping update
    std::thread t = std::thread(thread_func, target);
    // The .so file is arbitrary
    void *handle = dlopen("/usr/lib/p7zip/7z.so", 0);
    t.join();

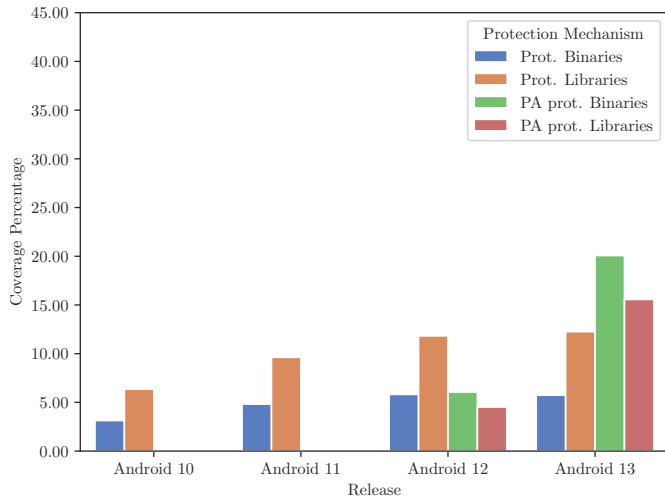
    // Simulate an arbitrary write to redirect the
    // function pointer to the target_function
    int (*func_ptr)(int) = foo;
    func_ptr = reinterpret_cast<int (*)>(&
        target_func);
    func_ptr(5); // This call should fail under LLVM CFI
}
```

Listing 1: Proof of concept for CDSO CFI race condition bypass

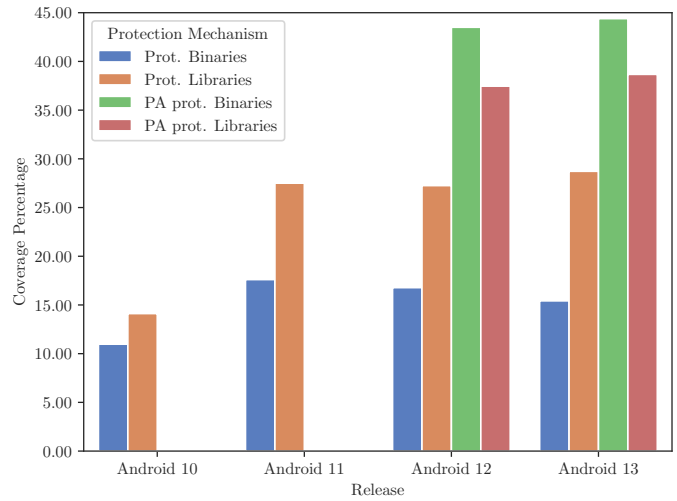
A.2 LLVM CDSO CFI race condition

LLVM's compiler runtime and Android's C standard library bionic [90] handle this by first allocating a new area for the shadow mapping, writing the desired values, and then re-mapping this new area to the previous shadow mapping address. This process opens up a timing window in which the newly allocated mapping is writable.

We successfully exploit this race condition to bypass CFI from within a C++ program, based on an experiment in which the adversary triggers a call to `dlopen` and then immediately starts a thread for writing to the shadow mapping afterward. A PoC for this bypass is shown in the appendix in [Listing 1](#). Such issues indicate that an approach using embedded labels like Window's XFG is a more elegant solution, especially with respect to CDSO checks, but it is incompatible with execute-only memory.



(a) S20 Releases



(b) Mi 10 Releases

Figure 6: CFI coverage development over different Android releases. Bars represent the arithmetic mean over the analysed images within an Android release.

Table 5: CFI coverage of Android firmware images (continuation of Table 2)

Vendor & Version	Files Total [count]			LLVM CFI Protected [%]				Shadow Stack Protected [%]			Pointer Auth. Protected [%]		
	Binaries	Libraries	Kernel Modules	Binaries	Libraries	Kernel Modules	Kernel	Binaries	Libraries	Kernel Modules	Binaries	Libraries	Kernel Modules
Mi 10 2020-03-21 10	384	1987	41	10.94	14.09	0	✗	0	0.15	0	0	0	0
Mi 10 2020-07-15 10	382	1987	42	10.99	14.14	0	✗	0	0.15	0	0	0	0
Mi 10 2020-10-20 10	382	1992	42	10.99	14.11	0	✗	0	0.15	0	0	0	0
Mi 10 2021-01-10 11	383	1542	42	17.75	27.43	0	✗	0	0.19	0	0	0	0
Mi 10 2021-07-07 11	385	1534	42	17.66	27.57	0	✗	0	0.2	0	0	0	0
Mi 10 2022-01-20 11	385	1542	42	17.4	27.5	0	✗	0	0.19	0	0	0	0
Mi 10 2022-04-20 12	396	1681	40	16.67	27.31	0	✗	0	0.12	0	43.43	37.54	0
Mi 10 2023-01-12 12	397	1688	40	16.88	27.19	0	✗	0	0.12	0	43.58	37.38	0
Mi 10 2023-04-03 13	428	1668	40	15.42	28.72	0	✗	0	0.06	0	44.39	38.67	0
Mi 10 2023-05-17 13	428	1668	40	15.42	28.72	0	✗	0	0.06	0	44.39	38.67	0

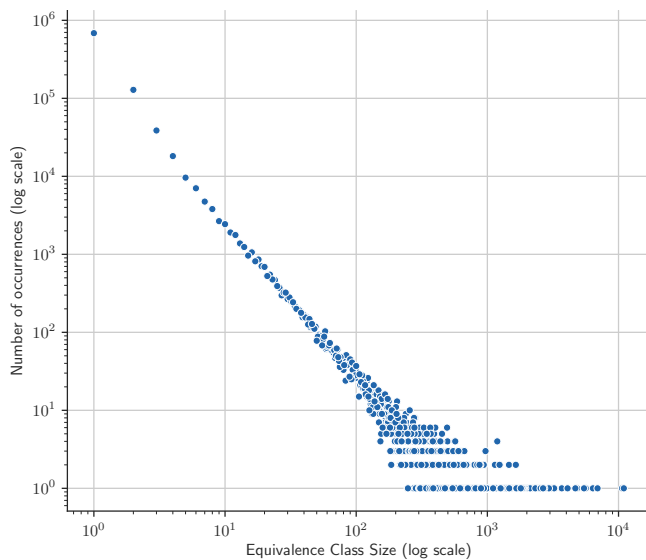


Figure 7: Equivalence class size distribution on the Windows 11 Insider Preview build 23440.

Table 6: Unprotected library dependencies in binaries.

Vendor & Device	Min [%]	Mean [%]	Max [%]
GSI 10	83.78	92.84	100.0
GSI 11	75.24	86.93	100.0
GSI 12	61.76	83.56	100.0
GSI 13	58.97	83.12	100.0
GSI 14	63.41	82.46	100.0
<hr/>			
Xiaomi 13	24.14	75.83	100.0
Google Pixel 7	58.97	84.65	100.0
GrapheneOS Pixel 7	58.97	84.65	100.0
Oppo Reno 8 5G	61.76	87.0	100.0
Samsung Galaxy S22	60.98	87.13	100.0
Vivo V25	58.97	88.48	100.0
<hr/>			
S20 2020-02-19	81.41	89.37	100.0
S20 2020-05-15	81.41	89.37	100.0
S20 2020-10-14	86.36	93.97	100.0
S20 2020-11-23	77.14	88.73	100.0
S20 2021-05-17	77.14	88.73	100.0
S20 2021-10-20	77.14	88.73	100.0
S20 2021-12-23	61.76	87.38	100.0
S20 2022-04-26	61.76	87.48	100.0
S20 2022-09-27	61.76	87.48	100.0
S20 2022-10-24	60.98	87.4	100.0
S20 2023-02-20	60.98	87.42	100.0
S20 2023-07-26	60.98	87.42	100.0
<hr/>			
Mi 10 2020-03-21	77.97	93.81	100.0
Mi 10 2020-07-15	77.97	93.8	100.0
Mi 10 2020-10-20	77.97	93.8	100.0
Mi 10 2021-01-10	53.57	87.03	100.0
Mi 10 2021-07-07	53.57	87.0	100.0
Mi 10 2022-01-20	53.57	86.94	100.0
Mi 10 2022-04-20	49.36	87.14	100.0
Mi 10 2023-01-12	49.36	86.66	100.0
Mi 10 2023-04-03	47.49	86.13	100.0
Mi 10 2023-05-17	47.49	86.13	100.0

Table 7: Geometric mean of equivalence class sizes.

Firmware	All without deps.	All with deps.	Reachable without deps.	Reachable with deps.
GSI 10	1.52251	1.48887	1.61912	2.17204
GSI 11	1.53914	1.47219	1.74867	2.24668
GSI 12	1.5913	1.40786	1.81712	2.35598
GSI 13	1.5868	1.42263	1.91317	2.54419
GSI 14	1.55852	1.49571	1.88483	2.53512
<hr/>				
Xiaomi 13	1.26808	1.31866	1.42438	2.06797
Google P7	1.47118	1.41269	1.74687	1.9628
Reno 8	1.47902	1.40975	1.73381	1.93443
Galaxy S22	1.44893	1.42762	1.7376	1.93009
Vivo V25	1.46451	1.42683	1.76094	1.99112
GrapheneOS	1.46739	1.41381	1.74092	1.95775
<hr/>				
S20 2020-02-19	1.44735	1.43	1.60504	1.76194
S20 2020-05-15	1.44887	1.43365	1.60284	1.76729
S20 2020-10-14	1.44793	1.47095	1.60507	1.71205
S20 2020-11-23	1.46341	1.46212	1.64564	1.81048
S20 2021-05-17	1.46006	1.4641	1.64128	1.81521
S20 2021-10-20	1.46704	1.46407	1.64932	1.82339
S20 2021-12-23	1.47126	1.4107	1.67733	1.8489
S20 2022-04-26	1.47519	1.41253	1.68565	1.85759
S20 2022-09-27	1.47654	1.41281	1.68332	1.84906
S20 2022-10-24	1.48012	1.42268	1.71563	1.89752
S20 2023-02-20	1.47762	1.42254	1.70653	1.8927
S20 2023-07-26	1.47624	1.42209	1.71574	1.89804
<hr/>				
Mi 10 2020-03-21	1.34242	1.3812	1.38852	1.55355
Mi 10 2020-07-15	1.3431	1.37911	1.38709	1.54917
Mi 10 2020-10-20	1.33849	1.37826	1.38092	1.54222
Mi 10 2021-01-10	1.38289	1.41024	1.45177	1.65537
Mi 10 2021-07-07	1.3849	1.41137	1.45473	1.65985
Mi 10 2022-01-20	1.36369	1.40777	1.42972	1.6782
Mi 10 2022-04-20	1.33301	1.37841	1.40426	1.53657
Mi 10 2023-01-12	1.33273	1.3783	1.39999	1.53187
Mi 10 2023-04-03	1.34791	1.37218	1.44178	1.66612
Mi 10 2023-05-17	1.34805	1.37145	1.4436	1.67142

Table 8: Analysed Android Firmware Images

Vendor	Device	Android	Release Date	Firmware Identifier	URL (full URL on-hover)
GSI	n/a	10	October 2019	gsi_gms_arm64-exp-QJR1	dl.google.com
GSI	n/a	11	September 2020	gsi_gms_arm64-exp-RP1A	dl.google.com
GSI	n/a	12	July 2022	gsi_gms_arm64-exp-SQ3A	dl.google.com
GSI	n/a	13	April 2023	gsi_gms_arm64-exp-T3B3	dl.google.com
GSI	n/a	14	August 2023	gsi_gms_arm64-exp-UPB5	dl.google.com
Google	Pixel 7	13	March 2023	panther-t3b2.230316.003-factory-c65097bc	dl.google.com
GrapheneOS	Pixel 7	13	September 2023	2023091800	releases.grapheneos.org
Vivo	V25	13	January 2023	PD2215F_EX_A_13.1.13.5.W30.V000L1	in-sysup-txdl.vivoglobal.com
Samsung	Galaxy S22	13	January 2023	S901BXXU3CWAI_S901BOXM3CWAI_EUX	www.sammobile.com
Xiaomi	Xiaomi 13	13	February 2023	fuxi_eea_global_V14.0.15.0.TMCEUXM	bigota.d.miui.com
Oppo	Reno 8 5G	13	November 2022	CPH2359_MT6893_EX_11_A.18_221121	oppostockrom.com
Samsung	Galaxy S20	10	February 2020	G980FXXU1ATBM	samfw.com
Samsung	Galaxy S20	10	May 2020	G980FXXU2ATE6	samfw.com
Samsung	Galaxy S20	10	October 2020	G980FXXU5BTJ3	samfw.com
Samsung	Galaxy S20	11	November 2020	G980FXXU5CTKG	samfw.com
Samsung	Galaxy S20	11	May 2021	G980FXXS8DUE4	samfw.com
Samsung	Galaxy S20	11	October 2021	G980FXXS8DUJ5	samfw.com
Samsung	Galaxy S20	12	December 2021	G980FXXSCEUL7	samfw.com
Samsung	Galaxy S20	12	April 2022	G980FXXUEFVDB	samfw.com
Samsung	Galaxy S20	12	September 2022	G980FXXSFFVIB	samfw.com
Samsung	Galaxy S20	13	October 2022	G980FXXUFGVJE	samfw.com
Samsung	Galaxy S20	13	February 2023	G980FXXSFHWB1	samfw.com
Samsung	Galaxy S20	13	July 2023	G980FXXSIHWGA	samfw.com
Xiaomi	Mi 10	10	March 2020	V11.0.9.0.QJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	10	July 2020	V11.0.18.0.QJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	10	October 2020	V12.0.6.0.QJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	11	January 2021	V12.2.4.0.RJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	11	July 2021	V12.5.2.0.RJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	11	January 2022	V12.5.8.0.RJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	12	April 2022	V13.0.4.0.SJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	12	January 2023	V13.0.10.0.SJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	13	April 2023	V14.0.1.0.TJBEXM	bigota.d.miui.com
Xiaomi	Mi 10	13	May 2023	V14.0.2.0.TJBEXM	bigota.d.miui.com



Exploiting Android’s Hardened Memory Allocator

Philipp Mao Elias Valentin Boschung Marcel Busch Mathias Payer
EPFL, Lausanne, Switzerland

Abstract

Most memory corruptions occur on the heap. To harden userspace applications and prevent heap-based exploitation, Google has developed Scudo. Since Android 11, Scudo has replaced jemalloc as the default heap implementation for all native code on Android. Scudo mitigates exploitation attempts of common heap vulnerabilities.

We present an in-depth study of the security of Scudo on Android by analyzing Scudo’s internals and systematizing Scudo’s security measures. Based on these insights we construct two new exploitation techniques that ultimately trick Scudo into allocating a chunk at an attacker’s chosen address. These techniques demonstrate — given adequate memory corruption primitives — that an attacker can leverage Scudo to gain arbitrary memory write. To showcase the practicality of our findings, we backport an n-day vulnerability to Android 14 and use it to exploit the Android system server.

Our exploitation techniques can be used to target any application using the Scudo allocator. While one of our techniques is fixed in newer Scudo versions, the second technique will stay applicable as it is based on how Scudo handles larger chunks.

1 Introduction

Most modern critical memory corruption vulnerabilities are heap related [36]. On Android, multiple publicly documented examples demonstrate the feasibility of exploiting a heap-based vulnerability to gain arbitrary code execution [14, 27, 30]. To protect userspace processes against heap vulnerabilities, Google has introduced the hardened Scudo allocator in Android 11 [49].

Since then, Scudo has become the default allocator for native userspace code in the Android Open Source Project. Unless explicitly modified by the vendor, all userspace processes, including apps and higher-privileged system services use Scudo.

Scudo is explicitly designed to increase the cost and complexity of heap-based exploits [3]. To protect itself from at-

tacks, Scudo implements security measures to ensure the integrity of inline heap metadata and to prevent a predictable heap layout.

Exploitation techniques that target the allocator to escalate a heap-bound memory corruption vulnerability into an arbitrary memory write primitive or code execution have a long tradition. The security community has compiled a large compendium of such techniques for allocators like ptmalloc [5, 9, 29, 31, 43] (the glibc allocator) or jemalloc [6, 44] (Android’s previous default allocator). Only by understanding these techniques can an analyst assess the criticality and exploitability of a heap-bound memory corruption vulnerability. However, Scudo has avoided scrutiny. So far, no comprehensive study on techniques targeting Scudo exists.

In this work, we explore the limitations of Scudo’s protections. In particular, we find that Android’s userspace architecture significantly weakens Scudo’s security. All app processes and several system services are forked from a single process (the Zygote process) [21] and end up sharing the same address space layout and allocator state. The allocator state contains the secrets used to protect inline heap metadata and randomize allocation addresses. In a scenario where one Zygote-forked process attacks another such process, the allocator’s secrets are shared between the attacker and the target process. This effectively bypasses any protection relying on the confidentiality of these secrets. Stripped of these security measures Scudo becomes a promising target for exploitation.

We present two exploitation techniques targeting Scudo. In the context of attacking Zygote-forked Android processes, our techniques only require a sufficiently powerful memory corruption primitive, which allows manipulating inline heap metadata, to gain arbitrary memory write. Furthermore, these techniques can be applied to any program utilizing the Scudo allocator. However, in a more generic attack scenario, an additional memory leak primitive is required.

To demonstrate that our findings apply to realistic scenarios, we backport a known vulnerability (CVE-2015-1528 [39]) to Android 14. The vulnerability is a heap under/overflow in Android’s Binder deserialization. We show that in this

scenario the heap underflow can be leveraged by a malicious app to achieve code execution in the system server, using our exploitation techniques.

In summary, we make the following contributions:

- Analysis and systematization of Scudo’s security measures.
- Discovery of two exploitation techniques that target Scudo.
- Exploitation case study utilizing our techniques to achieve arbitrary code execution in the system server on Android 14.
- Discussion of possible mitigations, memory corruption primitives required to leverage our techniques, and Scudo’s impact on Android userspace security.
- Development of a gdb plugin and python library to help in analyzing and exploiting Scudo.

We disclosed our findings to the Scudo maintainers. One of our techniques has been fixed in Android 14. While we proposed a further extension to Scudo, which mitigates our second exploitation technique, it was not merged due to performance concerns. Consequently, Scudo remains susceptible to our second technique. We will open-source our tooling for Scudo along with our exploits.

2 Scudo Security Measures

Scudo is a drop-in replacement for the glibc memory allocator, exposing the same API (e.g., `malloc`, `free`). As a security-hardened allocator, it implements four security measures: (i) isolation, (ii) randomization, (iii) protection, and (iv) separation. In this section, we present these security measures and discuss how they protect from exploiting heap-based vulnerabilities.

To give concrete examples of the security measures impact, we use the example program in Listing 1, which uses the Scudo allocator. The program reads the attacker’s input into the `tmp` buffer (Line 8) in a loop, allocates a `0x18` sized chunk (Line 11), copies data from `tmp` into the chunk (Line 13), and then frees a specific chunk based on the value of `status` (Lines 14 - 16). The attacker controls the values of the `status` and `size` variables. This results in two security vulnerabilities, a heap-based overflow on Line 13 and a double free on Lines 14 and 15.

Isolation. Based on the requested allocation size, chunks are either handled as *primary* or *secondary* chunks. Primary chunks are placed into dedicated heap memory regions, while secondary chunks are allocated separately in their own memory region. To handle primary chunks, Scudo maps multiple

```
1 int main(){
2     char tmp[0x100];
3     void* class_0_secondary_chunk = malloc(0x20000);
4     void* class_1_chunk = malloc(0x8);
5     void* class_2_chunk = malloc(0x18);
6     printf("victim:%p\n", class_2_chunk);
7     while(1){
8         read(0, tmp, 0x100);
9         int status = *(int*)tmp;
10        int size = *(int*)(tmp+sizeof(int));
11        char* chunk = (char*)malloc(0x18);
12        printf("address:%p\n", chunk);
13        memcpy(chunk, tmp+sizeof(int)*2, size);
14        if(status & 0x2){free(chunk);}
15        if(status & 0x4){free(chunk);break;}
16        if(status & 0x8){free(class_2_chunk);}
17    }
18 }
```

Listing 1: A vulnerable example program with a heap buffer overflow and a double free (both values of the `size` and `status` variable are under the attacker’s control). The attacker’s input is read from standard input.

memory regions. Each of these regions is assigned a size range, and chunks sized within the specific range will be allocated from the corresponding region. Zero permission guard pages are used to separate these regions. In Scudo, these size ranges are referred to as *class IDs*. For primary chunks the class ID depends on the size, see Table 5 in the Appendix for a mapping between classes and size ranges. For example, a chunk of size `0x18` is assigned the class ID 2. The class ID-specific regions only hold chunks. Allocator-internal metadata such as lists of freed chunks or the information on memory regions are stored in a separate region protected by guard pages, and libc’s writable section.

In Lines 3-5 of Listing 1, chunks of different sizes are allocated, resulting in a heap memory layout shown in Listing 2. Thus, the heap buffer overflow in Listing 1 can only overwrite memory inside the [Class 2 region] memory region and cannot directly overwrite chunks of other size classes or allocator-internal metadata.

Security Measure *Isolate*:

Chunks are allocated in dedicated isolated memory regions. Furthermore, allocator-internal metadata is stored in separate memory regions from chunks.

Randomization. The chunks inside a region are allocated at random offsets. When a region is first mapped, several addresses where a chunk may be allocated are placed into a so-called *TransferBatch*. The order in which these addresses are returned from the *TransferBatch* is randomized. This randomization is achieved by shuffling the addresses in the *TransferBatch* using a seed stored in the allocator. This randomiza-

```

size      permission
...
0x00001000 --- [Secondary guard]
0x01001000 rw- [Class 0 secondary chunk]
0x00001000 --- [Secondary guard]
...
0xa0006000 --- [Guard and reserve]
0x00040000 rw- [Primary chunk free lists]
0x2ffbf000 --- [Guard and reserve]
0x00040000 rw- [Class 1 region]
0x2ffc0000 --- [Guard and reserve]
0x00040000 rw- [Class 2 region]
0x0ffc0000 --- [Guard and reserve]
...
0x00044000 r-- libc.so
0x00094000 r-x libc.so
0x00004000 r-- libc.so
0x00002000 rw- libc.so
0x00452000 rw- [Allocator metadata]
...

```

Listing 2: An example memory map of Scudo relevant regions. Marked in blue are regions where Scudo stores free lists and other allocator-internal metadata. Marked in orange are regions where chunks are stored. In this example, a single secondary chunk was allocated, and at least one chunk of class IDs 1 and 2 were allocated. Scudo memory regions containing chunks are surrounded by 0 permission guard pages.

tion ensures that addresses of consecutively allocated chunks cannot be predicted, effectively removing the foundation for any heap feng shui attempts. Figure 1 shows the output when running the program in Listing 1 twice for five loop iterations with the same input. As can be seen, the addresses of chunks allocated after one another are not consecutive and also differ between program executions.

Security Measure *Randomize*:
Addresses of consecutive allocations are randomized.

Protection. When allocating a chunk, Scudo places a chunk header at address returned pointer-0x10. The chunk header is shown in Table 1. The relevant fields are the `ClassId`, `State`, and `Checksum`. The `ClassId` stores the chunk’s class ID. The `State` field tracks if the chunk is currently in use or has been freed. To protect this header, Scudo stores a truncated CRC32 checksum of the header fields in the `Checksum` field. The checksum is computed using the chunk’s address, the header, and a 32-bit cookie value, which is randomly generated when the program starts. Listing 3 shows how the checksum is computed. Any time Scudo interacts

```

> ./example < input
victim: 0x7fd4f720f650
address:0x7fd4f720e510
address:0x7fd4f720f750
address:0x7fd4f720f190
address:0x7fd4f720e4d0
address:0x7fd4f720efd0

```

```

> ./example < input
victim: 0x7fd4f7208b50
address:0x7fd4f7208390
address:0x7fd4f7209250
address:0x7fd4f7208990
address:0x7fd4f7209bd0
address:0x7fd4f7209190

```

Figure 1: The output of running the example program in Listing 1 two times to show Scudo randomizing allocation addresses. Note that for this example ASLR was disabled to show the chunk offsets in the same memory region changing between runs.

# bits	Field
8	ClassId
2	State
2	OriginOrWasZeroed
20	SizeOrUnusedBytes
16	Offset
16	Checksum

Table 1: The fields and corresponding sizes in the Scudo chunk header. The `OriginOrWasZeroed` field indicates the origin of the chunk, e.g., `malloc` or `new`. The `SizeOrUnusedBytes` field indicates the exact chunk size. `Offset` is filled with zeros.

with a chunk, it recomputes the checksum and compares it with the `Checksum` field to ensure the integrity of the chunk header. In the example program in Listing 1, if the attacker blindly overwrites the chunk header of the `class_2_chunk` with the heap overflow, Scudo will abort when freeing the chunk (Line 16) as the checksum will not match the header contents. By ensuring that the `State` field has the expected value, i.e., the chunk currently being freed is not already free, Scudo prevents double-free attacks. In Listing 1, if an attacker sends a payload that results in a `status` of `0x6`, this triggers a double free (Lines 14-15). However, Scudo immediately aborts on Line 15 as it detects the double free using the `State` field.

Security Measure *Protect*:
The chunk header, stored inline on the heap, is protected by a checksum.

Separation. Secondary chunks have the same chunk header as primary chunks, with the `ClassId` field set to 0. Additionally, secondary chunks have an extended header beginning at returned pointer - 0x40, see Table 2 for an overview of the fields. This extended *secondary chunk header* stores pointers to a linked list in `next` and `prev` of allocated sec-

```

short checksum(long address, long header, int cookie){
    int intermediate = CRC32(cookie, address);
    intermediate = CRC32(intermediate, header);
    return = (short) (intermediate & (intermediate >> 16)) & 0xffff;
}

```

Listing 3: Pseudocode of how Scudo computes a chunk’s checksum. Address points to the chunk, header is the chunk header without the checksum and cookie is a secret, set when the allocator is initialized.

# bytes	Field	Checksum
0x8	Prev	X
0x8	Next	X
0x8	CommitBase	X
0x8	CommitSize	X
0x8	MapBase	X
0x8	MapSize	X
0x8	Scudo Chunk header	✓

Table 2: The fields and corresponding sizes in the secondary chunk header for 64-bit programs. Only the Scudo chunk header is protected by a checksum.

ondary chunks. It also stores the mapping’s base address and size, with and without the guard pages respectively in MapBase, MapSize, CommitBase, and CommitSize. Importantly, the secondary chunk header is not protected by a checksum. Instead, Scudo relies on the fact that only one chunk is stored in the mapping and that the mapping is surrounded by guard pages to protect the extended header. The chunk header and secondary chunk header are the only instances of Scudo storing metadata inline on the heap. In Listing 1, the chunk class_0_secondary_chunk is allocated on Line 3 and its secondary chunk header is stored at class_0_secondary_chunk+0x40. In Listing 2 this chunk resides in the [Class 0 secondary chunk] memory region.

Security Measure Separate:

Pointers stored inline are placed in separate mappings and protected by guard pages.

3 Threat Model

In our threat model, a malicious attacker-controlled Android app aims to escalate privileges by attacking another app or system service on the same device. The device is running an Android version using Scudo.

The attacker’s goal is to gain code execution in the target process by corrupting the target’s memory. Due to Android’s

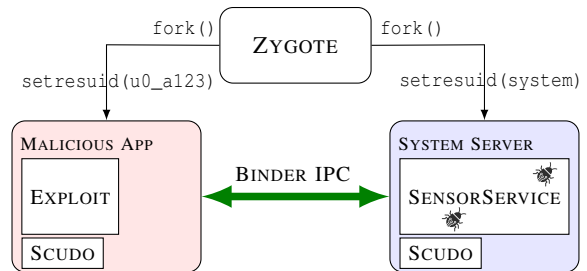


Figure 2: The malicious app is attacking the vulnerable SensorService running in the system server. The app uses Binder IPC to communicate with the SensorService. Both the app and system server are forked from Zygote.

separation of userspace processes, the attacker cannot directly manipulate the target’s memory. Instead, the attacker relies on the target’s exposed functionality to interact with the target’s memory. Furthermore, the target contains memory corruption vulnerabilities triggerable by the attacker. There are many examples of such memory corruption vulnerabilities in apps (CVE-2019-11932 and CVE-2021-24041 [14, 42]), or system services (CVE-2015-1528, Stagefright, CVE-2020-0026, CVE-2019-2136, and CVE-2022-39907 [10, 25, 39–41]). See Figure 2 for a concrete example of our threat model.

The attacker plans to use these vulnerabilities to bypass Scudo’s four security measures and leverage Scudo to achieve an arbitrary memory write for subsequent code execution. In the following sections, we discuss how in the context of our threat model each of Scudo’s security measures is broken.

4 Compromising Protect and Randomize

Both security measures *Randomize* and *Protect* rely on the confidentiality of Scudo metadata and ASLR (Address Space Layout Randomization). Concretely, if an attacker can leak the contents of the TransferBatch or the seed used to shuffle the TransferBatch, the security measure *Randomize* is compromised. With the leaked information, the attacker can pinpoint exactly where Scudo will allocate future chunks. To compromise the security measure *Protect*, the attacker needs to first compromise the security measure *Randomize* or leak the address of the target chunk in another way. Additionally, the attacker also needs to obtain the cookie to calculate the checksum correctly.

In our threat model both security measures *Randomize* and *Protect* are immediately compromised. On Android, all apps as well as several system services are forked from the same Zygote process. This reduces the startup time and reduces memory consumption by sharing RAM pages used for framework code and resources [21] but comes at a devastating cost to security [32]. As a consequence, most Android userspace processes share the same ASLR layout, including Scudo regions. Furthermore, the Zygote process allocates sev-

eral chunks initializing the Scudo allocator i.e., setting the cookie and TransferBatch randomization seed. After forking all of this allocator state is preserved. A malicious app can predict exactly where chunks of other Zygote-forked processes will be allocated by using its allocator as an oracle, which breaks security measure *Randomize*. To break security measure *Protect*, the malicious app can simply read out its Scudo cookie and forge valid checksums for any chunk header.

Security measures *Randomize* and *Protect* are compromised in an attack scenario in which a malicious Android app is attacking another Zygote-forked process.

Going forward, we assume that the attacker has compromised the security measures *Randomize* and *Protect*.

5 Arbitrary Write

The holy grail of heap exploitation is to coerce the allocator into allocating a chunk at an attacker's chosen address. This can lead to code execution for example by allocating a chunk on the stack and writing a ROP (Return Oriented Programming) chain to the chunk. Since the security measures *Randomize* and *Protect* are bypassed, only the measures *Isolate* and *Separate* stand in the way of an arbitrary write. In classical ptmalloc heap exploitation, an arbitrary write is usually achieved by manipulating inline pointers. However, as shown in Section 2 the security measure *Separate* separates inline pointers from the rest of the heap. The only instance of Scudo storing pointers inline is in the secondary chunk header, which is stored in memory regions separated from the rest of the heap.

To go further we assume the attacker to have access to a memory corruption primitive which allows manipulating the header of a victim chunk. This memory corruption primitive is limited to Scudo's primary heap regions. An example of such a primitive is a heap buffer overflow that overwrites the chunk header of a subsequent victim chunk. After bypassing the security measure *Protect*, the attacker can freely set any fields of the overflown chunk header (Table 1) and calculate a valid checksum. When the victim chunk is freed, the checksum verification will succeed and Scudo will parse attacker-controlled metadata. An example of such a primitive is shown in Listing 1. The vulnerable program allows an attacker to overwrite the chunk header of `class_2_chunk` using the heap buffer overflow. Since the attacker has broken the *Randomize* security measure, the attacker can keep allocating chunks in a loop until the overflowing chunk is just below the `class_2_chunk`. Subsequently the `class_2_chunk` can be freed (Line 16) by setting `status` to `0x8`.

Both, manipulating the `State` and `ClassId` fields, are interesting for the attacker. By manipulating the `State` field, a double free can be turned into a UAF (Use After Free). In this

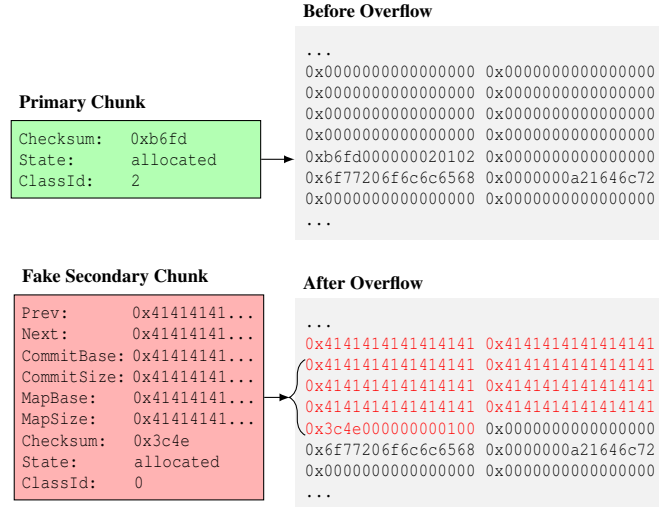


Figure 3: A heap buffer overflow, which overwrites a chunk header and changes a primary chunk's `ClassId` to 0, defeats security measure *Separate*. Note that the figure shows lower addresses at the top, growing downwards.

scenario, the chunk is first freed, the attacker then overwrites the header and changes the `State` field from `freed` back to `allocated`. The chunk is then freed again and ends up in the free list twice, setting up the UAF. However, the UAF is only interesting if attacking the application's data is in scope since the UAF does not give access to any Scudo metadata. As opposed to ptmalloc where a UAF may allow overwriting pointers to other free chunks.

More interesting for the attacker is manipulating the `ClassId` field. By changing the `ClassId` of a primary chunk to 0, the class ID of secondary chunks, the attacker effectively places the secondary chunk header inline on the heap rendering the security measure *Separate* ineffective. In the scenario of an overflow, the secondary chunk header is fully under the attacker's control. Figure 3 illustrates this phenomenon. Before the overflow, the victim chunk is simply a primary chunk with `ClassId` 2. After the overflow, the victim chunk is replaced by a fake secondary chunk with a bogus secondary chunk header. Freeing the overflown victim chunk causes a segfault as Scudo attempts to read the linked list entry at `0x4141414141414141`.

To compromise security measure *Separate* an attacker needs access to a memory corruption primitive which allows the creation of faked secondary chunks.

In the following sections, we present two exploitation techniques, Forged CommitBase and Safe Unlink. Both techniques manipulate this newly created and inlined secondary chunk header in different ways to achieve an arbitrary memory write, thus breaking security measure *Isolate*.

5.1 Forged CommitBase

The Forged CommitBase technique manipulates the CommitBase header field of the inlined secondary header to achieve an arbitrary memory write.

The CommitBase field of the secondary chunk header stores a pointer to the start of the secondary chunk (including the secondary chunk header). After freeing the secondary chunk, the CommitBase is stored in the free list of secondary chunks. When this secondary chunk is used to serve an allocation request, Scudo uses the CommitBase stored in the free list to determine where this chunk is located.

By cleverly setting the CommitBase of the faked secondary header to the desired target address, the attacker can bring Scudo to allocate a secondary chunk at the desired address, breaking security measure *Isolate*.

Figure 4 shows the sequence of events taking place in this exploit and the relevant fields of the faked secondary chunk. At ① the attacker overwrites the primary chunk's header and places the fake secondary chunk on the heap, using a memory corruption primitive (for example an overflow). The CommitBase is set to the target address (0x7fffffff840). As discussed previously, Prev and Next are pointers to entries in a linked list. For the free to succeed, these pointers need to be valid. Fortunately, Scudo checks if the pointers are null. If they are, the unlinking step is skipped. At ② the fake secondary chunk is freed, and the CommitBase address is placed into the secondary chunk free list. At ③ a secondary chunk is requested, which Scudo serves from the secondary free list. Since Scudo uses the address stored in the free list, the newly allocated chunk is located on the stack (at 0x7fffffff840). Note that the attacker is free to choose any CommitBase address.

For this exploit to succeed, at least one secondary chunk needs to be allocated at the time of freeing. Otherwise, the counter of secondary chunks in use is flipped to -1 and Scudo will crash on the next secondary chunk allocation.

Security measure *Isolate* can be bypassed by manipulating the CommitBase field of a secondary chunk header.

In the next section we present an alternative technique, which achieves an arbitrary write by manipulating different secondary chunk header fields.

5.2 Safe Unlink

In contrast to the previous technique, our second technique (Safe Unlink) leverages the unlinking taking place when a secondary chunk is freed to obtain an arbitrary memory write.

In glibc heap exploitation, the "unsafe unlink" attack [45] for newer libc versions exploits a linked list unlink to achieve arbitrary memory write. This technique is almost directly applicable to the unlinking taking place when the secondary

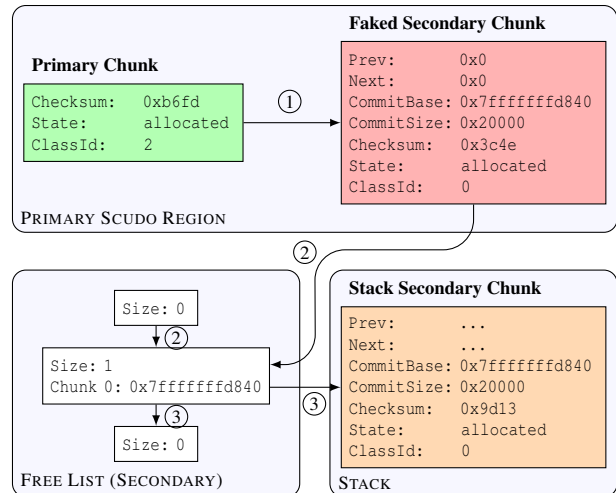


Figure 4: The attacker overwrites a primary chunk's header and modifies the CommitBase. After freeing the chunk, a stack address is placed into the secondary chunk free list. Allocating from the secondary chunk free list then allocates a chunk on the stack. (0x7fffffff840 is a stack address in this example.)

```
struct PerClass {
    short Count;
    short MaxCount;
    void* Chunks[MaxCount];
};
```

Listing 4: The PerClass free list, which stores class-specific free chunks. Count tracks the number of entries in the list. MaxCount is the maximum number of chunks that may be stored in the list. Chunks is an array of pointers, pointing to the address of the free chunks headers.

chunk is freed. Just like newer glibc versions, Scudo diligently checks the integrity of the linked list, see Listing 8 in the Appendix. To leverage this safe unlink, the attacker needs to create a fake linked list. While the glibc exploitation technique relies on an application-specific pointer, for Scudo, we will leverage allocator metadata to fake a linked list with two entries. One entry is the secondary chunk header, the other entry is placed inside the PerClass structure. The PerClass structure is a free list storing pointers to free chunks of a specific class ID. Listing 4 shows the structure of the PerClass free list. It holds the number of chunks, the maximum number of chunks, and a list of pointers to free chunks.

By cleverly forging primary chunks overlapping the fake secondary header and freeing these chunks, the attacker can place pointers to the fake secondary header into the PerClass structure. Figure 5 shows the attacker-created fake linked list before and after unlinking. After unlinking, an address pointing to the free list will be inserted into the free list itself.

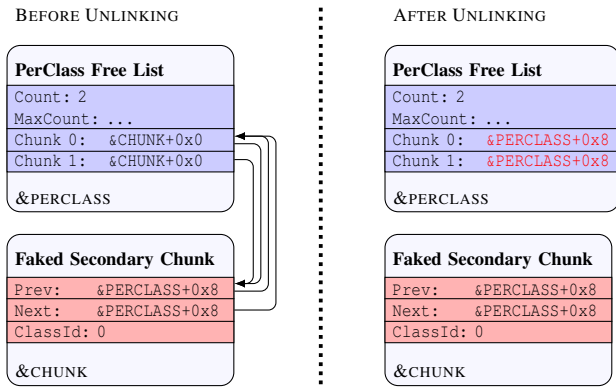


Figure 5: The attacker created linked list with two entries. The linked list is created by freeing fake chunks that overlap the secondary chunk header and adjusting the Next and Prev pointers to point into the PerClass free list. The attacker knows the address of the PerClass free list due to mitigations *Randomize* and *Protect* being broken.

Allocating from this PerClass free list returns a chunk overlapping the free list. The attacker thus gains control over the free list and can control the addresses of future allocations.

In order to create the fake linked list, the attacker needs more powerful memory corruption primitives. Besides being able to forge a secondary chunk, the attacker is also able to trigger two frees at address fake secondary chunk header + 0x10. An example of such a primitive is a controlled free in which the attacker can corrupt a pointer and then have that pointer freed. Furthermore, the attacker can trigger the memory corruption primitive multiple times, i.e., overwriting the fake secondary header three times.

Figure 6 shows the steps needed to set up the fake linked list. At ① the attacker writes a primary chunk header with a chosen ClassId and allocated state to the address where the fake secondary header starts. This fake primary chunk header overlaps with the Next field of the fake secondary header. The fake primary chunk is then freed at ②. Effectively, the address of the fake secondary's CommitBase entry (fake secondary chunk header + 0x10) is passed to free. Consequently, the address of the fake secondary chunk header is placed into the PerClass structure for the chosen ClassId. (Note that Scudo tracks chunks in the PerClass free list by the address of the chunk's header.) The attacker then repeats the previous steps (③ and ④). Now the address of the fake secondary chunk header is twice at consecutive positions in the PerClass free list. Finally at ⑤, the attacker sets up the fake secondary chunk header to complete the fake linked list. Both Next and Prev are modified to point to the first instance of the fake secondary chunk address in the PerClass structure. With this the fake linked list, as seen in Figure 5, has been set up. The attacker knows the location

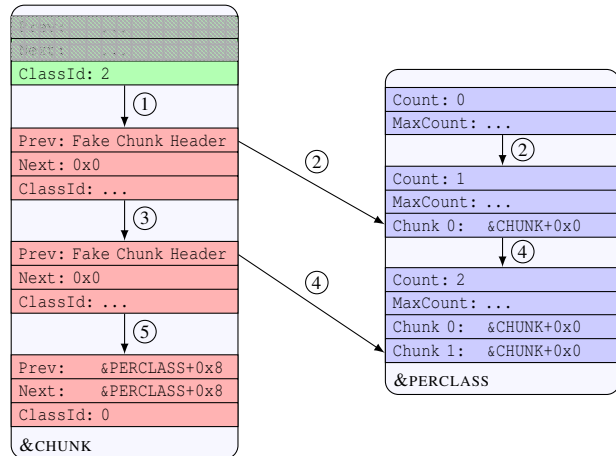


Figure 6: The steps to forge a fake linked list between the secondary chunk header and the PerClass free list. &CHUNK is the address of the fake secondary chunk header. &PERCLASS is the address of the PerClass free list.

of the PerClass structure because of measures *Randomize* and *Protect* being broken. The CommitBase, CommitSize, MapBase, and MapBase fields of the chunk header are not relevant to this exploit. Only the chunk header needs to be overwritten to have ClassId 0.

After the secondary chunk is freed, the attacker can allocate a chunk overlapping the PerClass structure, effectively allowing the attacker to insert any addresses into the free list, breaking the *Isolate* mitigation.

Security measure *Isolate* can be bypassed by manipulating the Prev and Next fields of a secondary chunk header along with cleverly freeing fake chunks into a PerClass free list.

6 Exploitation Case Study

We demonstrate our findings by reintroducing an n-day vulnerability and exploiting the system server on an Android Virtual Device running Android 14 using our techniques.

The Android system server is the first process forked from the Zygote process. It starts all system services, either starting the service in a separate process or starting a new thread running the service inside the system server. The system server is an interesting target for escalating privileges from an app. Firstly, each service running inside the system server is exposed over Binder IPC to normal apps. Binder is the Android-specific IPC mechanism, which facilitates communication between Android apps and Android services. In total, the system server exposes around 42 services [18]. Secondly, the system server runs as the high-privileged system user, just slightly less powerful than root. Third, the system server

restarts after crashing giving the attacker multiple exploitation attempts. Finally, the system server is forked from Zygote, and thus Scudo's security measures *Randomize* and *Protect* are ineffective in our attack scenario as described in [Section 4](#).

To provide the attacker app with a memory corruption primitive to defeat security measures *Isolate* and *Separate*, we backport CVE-2015-1528 [39] to Android 14. CVE-2015-1528 is a heap underflow or overflow in the Binder data deserialization due to missing sanity checks. Listing 5 shows the relevant code and the code changes reintroducing the vulnerability. The `native_handle_create` function allocates the `native_handle` object whose size depends on the `numFds` and `numInts` arguments. Both of these arguments are read from the attacker-controlled Binder data. Since the sanity check on the arguments is removed, an attacker can trigger a heap underflow by setting `numFds` to a negative number, which will cause the first argument of `read` in `readNativeHandle` to point behind the allocated chunk. Likewise, by setting `numInts` to a negative number, a heap overflow is triggered in the loop which reads file descriptors from the Binder data. The change in the loop removes an early exit if reading the file descriptor from the Binder data fails.

At Black Hat USA 2015, Gong [27] used this vulnerability to exploit the system server. In the exploit, Gong coerced `jemalloc` to allocate a chunk on the stack. Almost ten years and one secure allocator later we will show how the same vulnerability remains exploitable in Scudo.

6.1 SensorService

Unlike Gong, who targeted the `WindowsManagerService` in the system server, we will target the `SensorService`. Listing 6 shows the relevant code in the `SensorService`'s `onTransact` function. The `onTransact` function is the service's callback to handle incoming Binder requests. Both the `data` and `code` argument to the function are fully under the attacker's control. By setting the code of the Binder request to `CREATE_SENSOR_DIRECT_CONNECTION`, the attacker can trigger the vulnerable `readNativeHandle` function. After the vulnerable function, the descriptors in the newly created `native_handle` object are tagged with `fdsan`. `Fdsan` is a file descriptor sanitizer implemented to detect use-after-close or double-closes [4]. Since Android 11, `fdsan` aborts the process if an issue is discovered. Concretely, if we pass `numFds` greater than zero, we need to ensure no duplicate integers are present as otherwise `fdsan` will abort in `native_handle_close_with_tag`. The `createSensorDirectConnection` function contains the actual implementation to handle the binder request, we can exit early from this function by setting the `format` variable, read from the Binder data to an invalid value. Finally, the allocated `native_handle` object is freed.

```

---/libcutils/native_handle.c
native_handle_t* native_handle_create(int numFds, int numInts)
{
- if (numFds < 0 || numInts < 0 || numFds > kMaxNativeFds
-   || numInts > kMaxNativeInts){return NULL;}
  native_handle_t* h = malloc(
    sizeof(native_handle_t) + sizeof(int)*(numFds+numInts));
  if (h) {
    h->version = sizeof(native_handle_t);
    h->numFds = numFds;
    h->numInts = numInts;
  }
  return h;
}

---/libs/binder/Parcel.cpp
native_handle* Parcel::readNativeHandle() const
{
  int numFds, numInts; status_t err;
  err = readInt32(&numFds);
  if (err == NO_ERROR) return 0;
  err = readInt32(&numInts);
  if (err == NO_ERROR) return 0;

  native_handle* h = native_handle_create(numFds, numInts);
  //may lead to a buffer overflow if numInts is negative
  for (int i=0; err==NO_ERROR && i<numFds; i++) {
    h->data[i] = dup(readFileDescriptor());
-   if (h->data[i] < 0) {
-     for (int j = 0; j < i; j++) {
-       close(h->data[j]);
-       native_handle_delete(h);
-       return nullptr;}
+   if (h->data[i] < 0) err = BAD_VALUE;
  }
  //may lead to a buffer underflow if numFds is negative
  err = read(h->data + numFds, sizeof(int)*numInts);
  if (err == NO_ERROR) {
    native_handle_close(h);
    native_handle_delete(h);
    h = 0;
  }
  return h;
}

```

Listing 5: The code changes to reintroduce CVE-2015-1528.

In summary, we can get a chunk of any size allocated, trigger a controlled heap underflow or overflow originating from that chunk, and have the chunk freed right afterward. Finally, all of these primitives are accessible via Binder by an unprivileged app.

6.2 Exploitation Over Binder

To gain code execution in the system server context, our malicious app sends two Binder requests to the `SensorService`. The first Binder request leverages the heap underflow to place a stack address into the secondary chunk-free list as described in [Section 5.1](#). The second Binder request allocates this secondary chunk and writes a ROP chain to the stack.

Forging a secondary chunk [Table 3](#) shows the data sent in the first Binder request. The first five fields


```

status_t BnSensorServer::onTransact(uint32_t code,
    const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case CREATE_SENSOR_DIRECT_CONNECTION: {
            CHECK_INTERFACE(ISensorServer, data, reply);
            String16& opPackageName = data.readString16();
            const int deviceId = data.readInt32();
            uint32_t size = data.readUInt32();
            int32_t type = data.readInt32();
            int32_t format = data.readInt32();
            native_handle_t *resource = data.readNativeHandle();
            if (resource == nullptr) {
                return BAD_VALUE;
            }
            native_handle_set_fdsan_tag(resource);
            sp<ISensorEventConnection> ch =
                createSensorDirectConnection(...);
            native_handle_close_with_tag(resource);
            native_handle_delete(resource);
            reply->writeStrongBinder(IInterface::asBinder(ch));
            return NO_ERROR;
        }
        ...
    }
}

```

Listing 6: The relevant parts of the SensorService’s onTransact function [17]. Marked in red is the function call that will trigger the heap memory corruption.

read by the system server from the Binder request are not relevant to our exploit and only serve to make the createSensorDirectConnection function exit early (opPackageName until format). NrFds is set to $-5-9*2$ (-23). -5 moves the underflow start just before the allocated chunk header and $-9*2$ moves the underflow start to the beginning of the faked secondary chunk header. The remaining data is then written to the heap in readNativeHandle (Occurs in the read function, which reads $\text{sizeof(int)} * \text{numInts}$ to the heap). The remaining data contains the secondary chunk header (Prev until MapSize), the chunk header to overwrite the original header (FakeHeader and zeros) and filler data (filler) such that the chunk is allocated from a specific primary chunk class.

Both Next and Prev are set to zero to avoid unlinking. The CommitBase is set to the target stack address and the CommitSize is set to the size of a secondary chunk, we use $0x20000$. To correctly craft the overwritten chunk header (FakeHeader) with ClassId 0, we need both the address of the chunk and the cookie. The cookie can be directly read from the memory of our own app. Although we know exactly at which addresses Scudo will allocate chunks, using the allocator of our own app as an oracle, predicting the native_handle’s allocated address is complicated by the non-determinism of the system server (at least 40 threads each handling Binder requests). We found that allocating a primary chunk of ClassId 32 (the largest class for primary

Type	Value	Name
String	"wow"	opPackageName
int	20	deviceId
int	20	size
int	20	type
int	20	format
int	-23	nrFds
int	0x3f56	numInts
long	0x0	Prev
long	0x0	Next
long	0x7ffdd0b564a8	CommitBase
long	0x20000	CommitSize
long	0x7ffdd0b564a8	MapBase
long	0x20000	MapSize
long	0x507a00000008100	FakeHeader
long	0x0	zeros
char*	0xfd00 * "A"	filler

Table 3: Example of the first Binder request sent to exploit the system server. $0x7ffdd0b564a8$ is the target stack address.

chunks) allowed us to correctly predict the chunk’s address around one out of ten times. Both numInts and filler serve to set the size of the allocated native_handle object.

The CommitBase in our fake secondary chunk header points to the main thread’s stack. The main thread in the system server runs in an infinite loop polling for messages. When writing to the stack, we will overwrite the stored return address of the android:Looper:pollOnce function. Our target stack address is around $0x20000$ (our CommitSize) below where the return address is stored. Note that we do not directly point our chunk at the return address. Doing so would cause ReadNativeHandle to inadvertently write over the stack’s maximum address causing a segfault.

For the first Binder request, we do not need to worry about fdsan because nrFds will be negative and the loop which reads file descriptors from the Binder data in readNativeHandle iterates zero times.

After receiving this Binder request, the BnSensorServer::onTransact function is called. Then, the readNativeHandle function is called, which in turn calls native_handle_create. Our native_handle chunk is allocated, and the heap underflow is triggered, replacing the original primary chunk with our fake secondary chunk. Right afterward, this chunk is freed and the target stack address is placed into the secondary free list. The details of creating the fake secondary chunk and placing the stack address into the secondary chunk free list can be found in Section 5.1.

Writing to the stack Table 4 shows the second Binder request sent to the system server. numInts is set to the size of the ROP chain divided by four. nrFds is set to

Type	Value	Name
String	"wow"	opPackageName
int	20	deviceId
int	20	size
int	20	type
int	20	format
int	0x7fe2	nrFds
int	30	numInts
long[15]	...	ROPChain

Table 4: Example of the second Binder request sent to exploit the system server.

the difference between the allocation size `0x20000` and the size of the ROP chain divided by four. When allocating memory in `native_handle_create` for this `0x20000`-sized `native_handle` object, the chunk is allocated from the secondary free list and placed on the main thread’s stack. In `readNativeHandle`, the ROP chain (`ROPChain`) is written to the stack starting at the stored return address of the `android:Looper:pollOnce` function. After the ROP chain has been written to the stack, there is a race between the main thread returning from the `android:Looper:pollOnce` function and the `SensorService` calling `native_handle_close_with_tag`. If the `SensorService` wins, the process is aborted by `fdsan`, due to duplicate integers being passed as file descriptors to `fdsan`. Instead, if the main thread wins, the first gadget of the ROP chain clobbers the `nrFds` field of the stack-allocated `native_handle`, setting it to a negative number and thus avoiding `fdsan` attempting to close any file descriptors. After clobbering `nrFds`, the ROP chain simply prints to `logcat`. Figure 7 shows the `logcat` output after successful exploitation. The exploit succeeds after around ten attempts.

7 Discussion

In this section, we discuss our presented exploitation techniques focusing on mitigations, generalization, and trade-offs.

Mitigations Independent from our research, the safe unlink exploitation technique as described in Section 5.2 has been fixed in Android 14. The fix changes the `PerClass` free list to store offsets, relative to the primary chunk heap region, instead of pointers. This makes building the fake linked list impossible by freeing chunks. Note that it is still possible to try and construct a safe unlink exploit by targeting application-specific objects to build the linked list. However, we did not find a suitable target inside of Scudo that could be used to build the fake linked list after this fix.

To prevent attackers from creating fake inlined secondary chunks, as described in Section 5, we propose an extension to

Scudo which tracks allocated secondary chunks in an isolated memory region. Any time a secondary chunk is freed, our mitigation would check that a secondary chunk was allocated before at the address to be freed. We opened a pull request on the LLVM repository (where the Scudo source is hosted) to add the proposed mitigation to Scudo [11]. However, the request was not merged due to performance concerns. Without fundamentally changing how secondary chunks are handled or accepting the performance penalty, Scudo will remain vulnerable to these types of attacks.

Generalizing our Exploitation Techniques Our exploitation techniques presented in Section 5 apply to any program using Scudo. Both our techniques require the attacker to know the exact address of the victim chunk, whose class ID will be changed.

In Section 4, we show how the typical threat model on Android renders the security measures *Randomize* and *Protect* ineffective. With *Randomize* broken the attacker knows the addresses and order of allocations for all chunks. However, this may not be enough for a sufficiently complex program to exactly predict the address of the victim chunk. Chunks may be freed and reallocated in a non-deterministic matter. For example for our case study in Section 6, we can only predict the address of our victim chunk one out of ten times. Thus, to apply our techniques to complex targets, either the target program restarts after crashing or the attacker has another way to probe the heap state, such as a program-specific memory leak vulnerability.

In a generic scenario, the attacker needs a powerful memory leak primitive akin to an arbitrary read to break both security measures *Randomize* and *Protect*. To break *Randomize*, the attacker needs to use this primitive to read the seed used to shuffle the `TransferBatch` or directly leak chunk addresses. To break *Protect*, the attacker needs to either read the cookie directly or leak both a chunk’s header and its address. Using this header and address, the attacker can brute force a valid cookie using the code in Listing 3. The brute force attack is viable since the cookie is only 16 bits long. After obtaining a valid cookie, the attacker can forge valid Scudo chunk headers.

Required Memory Corruption Primitives Both our presented exploits (`ForgedCommitBase` and `SafeUnlink`) require a memory corruption primitive that allows forging a fake secondary chunk. The fake secondary chunk can be forged either by overwriting an existing chunk’s header or by passing a controlled pointer to `free`, pointing to a fake secondary chunk.

Both a heap underflow and overflow are examples of primitives that can overwrite an existing chunk’s header. For our case study, we choose a heap underflow primitive. When targeting the system server’s `SensorService`, we are only able to control one chunk. The heap underflow allows us to reliably overwrite only what is needed, i.e., that chunk’s header.

```

...
30963 31252 E SensorService: Ashmem direct channel requires a memory region to be supplied
30963 30981 D CompatibilityChangeReporter: Compat change id reported: 218533173; UID 10142; state: ENABLED
30963 30981 D CompatibilityChangeReporter: Compat change id reported: 262645982; UID 10142; state: DISABLED
30963 31379 W Parcel : Attempt to read object from Parcel 0x7ebd2a8acaa0 at offset 104 that is not in the object list
30963 30963 I H3L10 : FR0m_5y5T3M_53rV3r

```

Listing 7: The logcat output of the system server after successful exploitation, marked in orange is the logcat print triggered by the ROP chain.

Instead, if we used a heap overflow, we would have needed to overwrite chunks used by other Binder threads, increasing the complexity and reducing the reliability of our exploit.

An alternative to overwriting existing chunks is freeing fake attacker-created chunks. For this primitive the attacker needs to be able to control a pointer passed to free, which points to an attacker-controlled memory location. In glibc heap exploitation, the “House of Spirit” [46] leverages this primitive to insert the modified pointer into the free list, assuming the attacker was previously able to write a chunk header at that pointer’s location. The “House of Spirit” is only feasible if the attacker can already write to the start and end of the target memory address. However, in Scudo, the primitive can be used to achieve an arbitrary write primitive. This primitive becomes even more appealing for Scudo exploitation because it is not influenced by random allocations, as the attacker chooses the location of the chunks.

In summary, the ideal memory corruption primitive to exploit Scudo is an attacker-controlled free. For the heap underflow or overflow, the preference heavily depends on how the target application handles the underflow or overflow chunk.

Manipulating Scudo Chunk Header Fields The Scudo chunk header has six fields, see Table 1. In Section 5 we discussed how manipulating header fields is only possible if the Checksum field is set properly and how the State field can be manipulated to induce double frees and use-after-frees. For our exploitation techniques, we overwrite the ClassId field to create a fake secondary chunk. However, for exploitation scenarios where our techniques are not applicable, there exists an alternative way of manipulating the ClassId field to achieve a heap overflow. Instead of changing the ClassId to 0, which transforms the primary chunk to a secondary chunk, an attacker can change the primary chunk’s size by replacing the original class ID with a larger one. Once that chunk is freed, it is placed into the PerClass free list of that larger primary chunk class ID. After said chunk is allocated from the free list, the chunk will overlap other smaller chunks, leading to a heap overflow in the Scudo memory region of the original chunk’s class ID. For the remaining header fields, OriginWasZeroed, SizeOrUnusedBytes, and Offset, we did find a way to manipulate them in a useful manner.

ARM MTE ARM MTE (Memory Tagging Extension) [16] is a hardware security feature. Scudo has added support for MTE early on and in October 2023 the first Android devices supporting MTE running Scudo were released [13].

MTE uses the top bits of pointers to tag memory regions. A new instruction allows assigning a tag to a memory region. After a memory region has been tagged, the region can only be accessed with pointers whose top bits match the assigned tag. Tag mismatches result in a segmentation fault.

Allocators can leverage MTE to detect illegal memory accesses to the heap. By assigning tags to the body of allocated chunks, allocators can probabilistically prevent heap overflows or use-after-frees.

If enabled, Scudo tags the body of primary chunks on allocation and deallocation. The body of secondary chunks is not tagged. The chunk headers are assigned predictable tags (0 and 2 for the chunk headers of primary and secondary chunks respectively, 1 for the secondary chunk header).

With these tagged headers our exploitation techniques are mostly mitigated. Bypassing the *Separate* security measure, as described in Section 5, without crashing due to a tag mismatch is now limited to two specific scenarios. Overwriting the header of an existing primary chunk to create a fake secondary chunk is only possible if there is a chunk just before the overwritten chunk header and that chunk has tag 1, matching the tag assigned to the secondary chunk header. Alternatively, a fake secondary chunk may be forged by creating it on the border between two memory regions tagged with 1 and 2. The secondary chunk header is written to the lower memory region tagged with 1 and the chunk header to the region tagged with 2. Then an arbitrary free, freeing the address just after the forged chunk header passes MTE checks.

Bypassing security measure *Isolate* is only possible with the forged CommitBase technique as the safe unlink technique requires overlaying primary chunk headers (tagged with 0) over the fake secondary chunk header (tagged with 1). The forged CommitBase technique is further hampered by the fact that it needs to point to an address, whose first 0x30 bytes are tagged 1. Otherwise, Scudo crashes as it tries to write the secondary chunk header to the target address with a pointer tagged 1.

In conclusion, ARM MTE almost completely mitigates our exploitation techniques.

Zygote Forking To the best of our knowledge, Android is the only significant production deployment of Scudo. As discussed in [Section 4](#), both security measures *Randomize* and *Protect* are rendered useless for Android userspace processes that are forked from Zygote. Without these security measures, Scudo’s security becomes similar to that of the standard glibc allocator (predictable allocations and inline metadata that may be manipulated), while still incurring a performance overhead (calculating the checksum). This issue affects any Android userspace memory allocator and can only be solved by moving away from Zygote-forked userspace processes. [Table 7](#) in the Appendix shows the userspace processes running on our stock emulator. 33 userspace processes are Zygote-forked (around 30% of all userspace processes), out of which seven processes run as a higher-privileged user. Note that the remaining processes are system apps, which are usually assigned special privileges.

Quarantine An optional Scudo feature is the quarantine which delays freed chunks from being allocated again right away. This can make Use-After-Frees harder to exploit but incurs a heavy performance penalty [\[34\]](#). Related work has shown how the additional complexity and metadata introduced by the quarantine can be exploited for an arbitrary write primitive [\[24, 50\]](#). Since the quarantine is disabled by default and disabled on Android, we omit it from this work.

Scudo deployment Although Scudo is the default allocator in Android’s libc, vendors may choose to utilize jemalloc or implement their allocator. To understand if vendors choose to deploy Scudo we analyzed the firmware of recently released phones. We picked 15 devices, whose firmware is easily available, and analyzed the symbols in the shipped libc binary. [Table 6](#) in the Appendix lists the analyzed devices. Overall out of the 15 devices, 6 devices use Scudo. Of the remaining 9 use jemalloc. From this sample of firmware, it is clear that Scudo is deployed in production but has not replaced jemalloc.

Case Study We backported and exploited the system server on the Android emulator running an x86 image¹. Most Android production devices are ARM-based, however, we designed a data-only exploit. The only part of the exploit that needs to be changed for an ARM device is the ROP chain. Furthermore, the emulator provides high fidelity for the Android userspace [\[22\]](#).

8 Related Work

The security community has recently started showing an interest in Scudo. UnIfuzz [\[50\]](#) gives an overview of Scudo internals and presents two exploits against Scudo quarantine.

¹Android emulator image: system-images;android-34;google_apis;x86_64

Cesare demonstrates how to compute the cookie with z3 after leaking the chunk header and the chunk header’s address [\[15\]](#). More recently multiple blogs have been published detailing the inner workings of Scudo [\[7, 12, 20\]](#). Concurrently to us, Ecob discovered the forged CommitBase exploit and presented his findings at Bsides Canberra [\[24\]](#). In this work, we systematize Scudo’s security mechanisms, put these mechanisms in the context of the Android userspace, present two exploits and demonstrate our findings against a real target.

Besides Scudo, allocators have long been a target. Researchers have demonstrated exploits against the glibc allocator [\[5, 9, 29, 31, 43\]](#) and jemalloc [\[6, 44\]](#). These works serve as an inspiration to us and we hope to extend the community’s compendium of exploitation techniques with our Scudo specific techniques.

HeapHopper [\[23\]](#) and ArchHeap [\[54\]](#) are systems to automatically discover heap exploitation primitives. These systems mainly focus on dmalloc or ptmalloc. ArchHeap included Scudo in its evaluation but failed to discover any exploitation primitives.

Other works have focussed on manipulating the heap’s layout [\[28, 33, 51\]](#). The systems proposed by these works analyze the target program to identify heap manipulation primitives. In our work we focus solely on Scudo and leave identifying the heap manipulation primitives out of scope.

There has been a large body of work on building secure allocators [\[2, 8, 19, 35, 38, 47, 48\]](#) or securing existing allocators [\[1, 26, 37, 52, 53\]](#). Unlike these works, we focus on dissecting the security measures of an existing, widely deployed allocator.

9 Conclusion

We investigated the security of Scudo, Android’s hardened memory allocator. We have found that a large part of Scudo’s security measures are rendered ineffective by Android’s userspace architecture in the context of our attacker model. Given a memory corruption vulnerability, we demonstrate two exploits which manipulate Scudo into allocating a chunk at an attacker’s chosen address.

To demonstrate that our findings are indeed practical, we backported an n-day memory corruption vulnerability to Android 14 and exploited the highly privileged system server from the context of an unprivileged app, achieving a privilege escalation.

In the process of researching Scudo we have developed a gdb plugin, which allows inspecting Scudo chunks and free lists, and a python library to forge Scudo chunk headers. We open-source all these tools along with the code and artifacts of our exploitation case study at <https://github.com/HexHive/scudo-exploitation>.

Acknowledgments

We thank the anonymous reviewers and artifact evaluation committee for their feedback on the paper. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, and DARPA HR001119S0089-AMP-FP-034

References

- [1] Sam Ainsworth and Timothy M Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 578–591. IEEE, 2020.
- [2] Periklis Akritidis et al. Cling: A memory allocator to mitigate dangling pointers. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.
- [3] android developers.googleblog.com. System hardening in android 11. <https://android-developers.googleblog.com/2020/06/system-hardening-in-android-11.html>, 2020. Accessed: January 2024.
- [4] android.googlesource.com. fdsan. <https://android.googlesource.com/platform/bionic/+master/docs/fdsan.md>, 2024. Accessed: January 2024.
- [5] anonymous. Once upon a free()... <http://phrack.org/issues/57/9.html>, 2001. Accessed: January 2024.
- [6] huku argp. Pseudomonarchia jemallocum. <http://www.phrack.org/issues/68/10.html#article>, 2012. Accessed: January 2024.
- [7] Jacob Bech. Advancing cybersecurity: Introduction to the scudo allocator. <https://vectorize.re/blog/internals/introduction-to-scudo/>, 2023. Accessed: January 2024.
- [8] Emery D Berger and Benjamin G Zorn. Diehard: Probabilistic memory safety for unsafe languages. *Acm sigplan notices*, 41(6):158–168, 2006.
- [9] blackngel. Malloc des-maleficarum. <http://phrack.org/issues/66/10.html>, 2009. Accessed: January 2024.
- [10] blog.thalium.re. The fuzzing guide to the galaxy: An attempt with android system services. <https://blog.thalium.re/posts/fuzzing-samsung-system-services/#cve-2022-39907>, 2023. Accessed: March 2024.
- [11] Elias Boschung. Mitigation pull request. <https://github.com/llvm/llvm-project/pull/75295>, 2023. Accessed: January 2024.
- [12] Rodrigo Branco. Scudo hardened allocator — unofficial internals documentation. <https://www.l3harris.com/newsroom/editorial/2023/10/scudo-hardened-allocator-unofficial-internals-documentation>, 2023. Accessed: January 2024.
- [13] Mark Brand. First handset with mte on the market. <https://googleprojectzero.blogspot.com/2023/11/first-handset-with-mte-on-market.html>, 2023. Accessed: March 2024.
- [14] Valerio Brussani. Whatsapp 2.19.216 remote code execution. <http://packetstormsecurity.com/files/154867/Whatsapp-2.19.216-Remote-Code-Execution.html>, 2019. Accessed: January 2024.
- [15] Dr. Silvio Cesare. Breaking secure checksums in the scudo allocator. https://blog.infosectcbr.com.au/2020/04/breaking-secure-checksums-in-scudo_8.html, 2020. Accessed: January 2024.
- [16] community.arm.com. Memory tagging extension: Enhancing memory safety through architecture. <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety>, 2019. Accessed: January 2024.
- [17] cs.android.com. Android sensorservice source. <https://cs.android.com/android/platform/superproject/main/+main:frameworks/native/libs/sensor/ISensorServer.cpp>, 2024. Accessed: January 2024.
- [18] cs.android.com. Android system server source. <https://cs.android.com/android/platform/superproject/main/+main:frameworks/base/services/java/com/android/server/SystemServer.java;l=250?q=systemserver.&sq=&ss=android%2Fplatform%2Fsuperproject%2Fmain>, 2024. Accessed: January 2024.
- [19] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX security symposium (USENIX security 17)*, pages 815–832, 2017.
- [20] Kevin Denis. Behind the shield: Unmasking scudo's defenses. <https://www.synacktiv.com/publications/behind-the-shield-unmasking-scudos-defenses>, 2023. Accessed: January 2024.

- [21] developer.android.com. Overview of memory management. <https://developer.android.com/topic/performance/memory-overview#SharingRAM>, 2024. Accessed: January 2024.
- [22] developer.android.com. Run apps on the android emulator. <https://developer.android.com/studio/run/emulator>, 2024. Accessed: January 2024.
- [23] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, 2018.
- [24] Zac Ecob. Scudo allocator exploitation. In *Bsides Cranberra*, Australia, Sidney, September 2023.
- [25] en.wikipedia.org. Stagefright (bug). [https://en.wikipedia.org/wiki/Stagefright_\(bug\)](https://en.wikipedia.org/wiki/Stagefright_(bug)), 2023. Accessed: March 2024.
- [26] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. PTAAuth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054. USENIX Association, August 2021.
- [27] Guang Gong. Exploiting heap corruption due to integer overflow in android libcutils. <https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege-wp.pdf>, 2015. Accessed: January 2024.
- [28] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, 2018.
- [29] jp. Advanced doug lea’s malloc exploits. <http://phrack.org/issues/61/6.html>, 2003. Accessed: January 2024.
- [30] Mateusz Jurczyk. Mms exploit part 1-5. <https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-1-introduction-to-qmame.html>, 2020. Accessed: January 2024.
- [31] Michel "MaXX" Kaempf. Vudo malloc tricks. <http://phrack.org/issues/57/8.html>, 2001. Accessed: January 2024.
- [32] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439. IEEE, 2014.
- [33] Runhao Li, Bin Zhang, Jiongyi Chen, Wenfeng Lin, Chao Feng, and Chaojing Tang. Towards automatic and precise heap layout manipulation for general-purpose programs. In *NDSS*, 2023.
- [34] llvm.org. Scudo hardened allocator. <https://llvm.org/docs/ScudoHardenedAllocator.html>, 2024. Accessed: January 2024.
- [35] Vitaliy B Lvin, Gene Novark, Emery D Berger, and Benjamin G Zorn. Archipelago: trading address space for reliability and security. *ACM SIGARCH Computer Architecture News*, 36(1):115–124, 2008.
- [36] Matt Miller. Root cause of microsoft rce cves by patch year. <https://twitter.com/epakskape/status/984481101937651713/photo/1>, 2018. Accessed: February 2024.
- [37] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.
- [38] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584, 2010.
- [39] nvd.nist.gov. Cve-2015-1528. <https://nvd.nist.gov/vuln/detail/CVE-2015-1528>, 2015. Accessed: January 2024.
- [40] nvd.nist.gov. Cve-2019-2136. <https://nvd.nist.gov/vuln/detail/CVE-2019-2136>, 2019. Accessed: March 2024.
- [41] nvd.nist.gov. Cve-2020-0026. <https://nvd.nist.gov/vuln/detail/CVE-2020-0026>, 2020. Accessed: March 2024.
- [42] nvd.nist.gov. Cve-2021-24041. <https://nvd.nist.gov/vuln/detail/CVE-2021-24041>, 2021. Accessed: March 2024.
- [43] Phantasmal Phantasmagoria. The malloc maleficarum. <https://seclists.org/bugtraq/2005/Oct/118>, 2005. Accessed: January 2024.
- [44] Shmarya Rubenstein. A tale of two mallocs: On android libc allocators. In *INFILTRATE 2018*, United States, Miami Beach, April 2018.
- [45] how2heap shellphish. unsafe_unlink. https://github.com/shellphish/how2heap/blob/master/glibc_2.38/unsafe_unlink.c, 2020. Accessed: January 2024.

- [46] how2heap shellphish. house_of_spirit. https://github.com/shellphish/how2heap/blob/master/glibc_2.31/house_of_spirit.c, 2022. Accessed: January 2024.
- [47] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403, 2017.
- [48] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A tunable secure allocator. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 117–133, 2018.
- [49] source.android.com. Scudo. <https://source.android.com/docs/security/test/scudo>, 2024. Accessed: January 2024.
- [50] unlfuzz.github.io. Scudo internals, attacking scudo’s quarantine. <https://unlfuzz.github.io/index.html>, 2020. Accessed: January 2024.
- [51] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. Maze: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1647–1664, 2021.
- [52] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, Jung-Won Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing use-after-free attacks with fast forward allocation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2453–2470, 2021.
- [53] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. Pumm: Preventing use-after-free using execution unit partitioning. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 823–840, 2023.
- [54] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1111–1128, 2020.

ClassId	Size Start	Size End
1	0x0	0x10
2	0x11	0x20
3	0x21	0x30
4	0x31	0x40
5	0x41	0x50
6	0x51	0x60
7	0x61	0x80
8	0x81	0xa0
9	0xa1	0xb0
10	0xb1	0xd0
11	0xd1	0x110
12	0x111	0x150
13	0x151	0x1b0
14	0x1b1	0x240
15	0x241	0x310
16	0x311	0x440
17	0x441	0x660
18	0x661	0x820
19	0x821	0xa00
20	0xa01	0xc20
21	0xc21	0x1000
22	0x1001	0x1200
23	0x1201	0x1bc0
24	0x1bc1	0x2200
25	0x2201	0x2d80
26	0x2d81	0x3780
27	0x3781	0x4000
28	0x4001	0x4800
29	0x4801	0x5a00
30	0x5a01	0x7300
31	0x7301	0x8200
32	0x8201	0x10000
0	0x10001	...

Table 5: Chunk sizes and the corresponding class ID on Android 14.

Appendix

```

void remove(T *X) {
    T *Prev = X->Prev;
    T *Next = X->Next;
    if (Prev) {
        CHECK_EQ(Prev->Next, X);
        Prev->Next = Next;
    }
    if (Next) {
        CHECK_EQ(Next->Prev, X);
        Next->Prev = Prev;
    }
    Size = Size - 1;
}

```

Listing 8: Excerpt from the Scudo source code, which unlinks the secondary chunk from the linked list of allocated secondary chunks. Scudo checks the integrity of the linked list with the `CHECK_EQ` macro. The `CHECK_EQ` macro aborts if the arguments are not equal.

Device	Date	Allocator
Samsung S24	2/2024	Scudo
Samsung S23 Ultra	1/2024	jemalloc
Samsung M14 5G	12/2023	jemalloc
Samsung A34 5G	10/2023	Scudo
Samsung Galaxy Z Fold 5	2/2024	jemalloc
Google Pixel 8	2/2024	Scudo
Google Pixel Fold	2/2024	Scudo
Xiaomi Redmi Note 13 5G	2/2024	jemalloc
Xiaomi Redmi 12 5G	12/2023	jemalloc
Xiaomi Redmi 13C 5G	1/2024	jemalloc
Xiaomi Redmi Note 12 4G	11/2023	jemalloc
Vivo y35	1/2024	Scudo
Vivo y73	1/2024	Scudo
Oppo A96 5G	6/2023	jemalloc
Oppo Reno 8 Pro	1/2024	jemalloc

Table 6: The analyzed firmware to understand if vendors deploy Scudo. The Date column denotes the firmware’s release date. Out of the 15 devices, 6 use the Scudo allocator. The remaining 9 use jemalloc.

User	Name
system	system_server
u0_a160	com.android.systemui
webview_zygote	webview_zygote
network_stack	com.android.networkstack.process
bluetooth	com.google.android.bluetooth
secure_element	com.android.se
radio	com.android.phone
u0_a172	com.google.android.ext.services
u0_a158	com.google.android.apps.nexuslauncher
u0_a169	com.google.android.permissioncontroller
u0_a129	com.google.android.gms.persistent
u0_a142	com.google.android.inputmethod.latin
u0_a129	com.google.android.gms
u0_a127	com.google.android.as
u0_a131	com.google.android.googlequicksearchbox:interactor
u0_a130	com.google.android.apps.messaging:rcs
system	com.android.emulator.multidisplay
u0_a131	com.google.android.googlequicksearchbox:search
u0_a130	com.google.android.apps.messaging
u0_a129	com.google.android.gms.unstable
u0_a185	com.google.android.providers.media.module
u0_a129	com.google.process.gservices
u0_a92	android.process.media
u0_a154	com.google.android.gm
u0_a184	com.google.android.rkpdapp
u0_a175	com.google.android.adservices.api
u0_a129	com.google.process.gapps
u0_a81	android.process.acore
u0_a144	com.google.android.apps.photos
u0_a173	com.google.android.devicelockcontroller
u0_a124	com.google.android.settings.intelligence
u0_a151	com.google.android.contacts
u0_a162	com.google.android.apps.wallpaper

Table 7: The 33 userspace processes which are forked from Zygote on the Android 14 emulator (system-images;android-34;google_apis;x86_64). Overall there are 107 userspace processes out of which 33 (30%) are Zygote-forked. 7 (6%) Zygote-forked processes are running as higher-privileged users. Note that the remaining processes are mostly privileged system apps. Compromising such an app from a normal app still escalates the attacker's privileges. Any additional, user-installed app will also be Zygote-forked.

Breaking Espressif’s ESP32 V3: Program Counter Control with Computed Values using Fault Injection

Jeroen Delvaux¹, Cristofaro Mune², Mario Romero¹, Niek Timmers² *

¹ {Jeroen.Delvaux, Mario.Romero}@tii.ae, Technology Innovation Institute, Abu Dhabi, UAE

² {cristofaro, niek}@raelize.com, Raelize, Rotterdam, The Netherlands

Abstract

Espressif introduced the ESP32 V3, a low-cost System-on-Chip (SoC) with wireless connectivity, as a response to earlier hardware revisions that were susceptible to Fault Injection (FI) attacks. Despite its FI countermeasures, we are the first to bypass all security features of the ESP32 V3 with an FI attack, including Secure Boot and Flash Encryption. First, we alter encrypted flash contents to set the 32-bit outcome of a Cyclic Redundancy Check (CRC) on the bootloader signature to an arbitrary value, which we then load into the Program Counter (PC) register of the Central Processing Unit (CPU) using a single Electromagnetic (EM) glitch. This allows us to jump to *Download Mode* in Read-Only Memory (ROM), which provides arbitrary code execution and access to unencrypted flash contents. As far as we know, this is the first successful FI attack, bypassing both Secure Boot and Flash Encryption with a single glitch, on a target with FI countermeasures. As the vulnerabilities are in hardware, they cannot be fixed, and a new hardware revision would be required. In response to our findings, Espressif issued a Security Advisory, AR2023-005, and requested a Common Vulnerabilities and Exposures (CVE) identifier, CVE-2023-35818.

1 Introduction

Espressif’s ESP32 is a low-end System-on-Chip (SoC) with Wi-Fi and Bluetooth connectivity, which sparked commercial use in millions of embedded devices. Notable security features such as Secure Boot and Flash Encryption are supported. As shown in Fig. 1, the Secure Boot implements a *chain of trust* where code stored in internal Read-Only Memory (ROM) authenticates bootloader code stored in external Flash. The latter, in turn, authenticates application code stored in Flash. A chain of trust is needed as the ROM is made by Espressif and the flash contents are made by customers of Espressif. Note

*The four authors contributed equally and are ordered alphabetically by last name.

that Flash is a Multi-Time Programmable (MTP) Non-Volatile Memory (NVM).

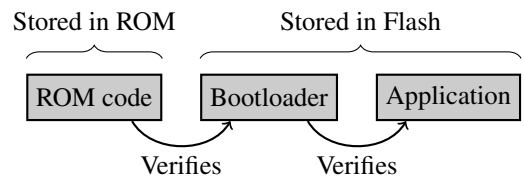


Figure 1: Chain of trust in a Secure Boot.

Vulnerabilities in the ROM code are particularly worrisome because (i) they compromise the entire chain, and (ii) they cannot be fixed by a software patch. The same holds for vulnerabilities that are purely in hardware. In this work, we attain this worst-case scenario of breaking the chain at the root. We leverage several weaknesses in the design of the ROM code through ElectroMagnetic Fault Injection (EMFI), which corrupts the executed instructions. Before listing our precise contributions, we situate our work into a brief history of FI attacks on the ESP32.

1.1 History of FI Attacks on ESP32

The first version of the chip, the ESP32-V1, was released in 2016, and its CPU implements the Xtensa Instruction Set Architecture (ISA) [32]. The following FI attacks were reported:

- In 2019, Riscure and LimitedResults [22] independently disclosed the first FI attack on the ESP32-V1: the digest verification of Secure Boot was skipped through a precisely timed voltage glitch (CVE-2019-15894) [12]. If Flash Encryption is disabled, this allows executing a modified bootloader.
- Still in 2019, LimitedResults [22] reported a second FI attack using supply-voltage glitching (CVE-2019-17391): bits stored in electronic fuses (eFuses), which is One-Time Programmable (OTP) NVM configured by Espressif’s customers, are corrupted while being transferred

to shadow registers. By corrupting read-protection bits stored in eFuses, keys that are also stored in eFuses, can be read out. In 2020, Raelize reproduced this attack using EMFI instead of voltage glitching [27].

- In 2020, Raelize reported an FI attack to bypass Secure Boot with Flash Encryption enabled, leveraging a peculiarity of the ROM to leave the Universal Asynchronous Receiver-Transmitter (UART) bootloader permanently enabled (CVE-2020-13629) [28]. For their attack, they leveraged retained data in the internal SRAM across warm resets, in order to control the PC register of the CPU.

In response to the above FI attacks, Espressif hardened the security design of the ESP32 and released ESP32 Chip Revision v3.0 in 2020 [7]. At the time of writing this paper, this is the latest revision. For the sake of brevity, we refer to this revision as ESP32 V3. Compared to the ESP32 V1, four significant changes are made:

- Secure Boot transitioned from symmetric-key cryptography, *i.e.*, the Advanced Encryption Standard (AES), to public-key cryptography, *i.e.*, Rivest–Shamir–Adleman (RSA) signatures. The ESP32 V3 only stores the public key; the private key is stored externally.
- While analyzing the ROM code, which was publicly released by Espressif as an ELF file [9], we identified the insertion of numerous redundancies, *e.g.*, eFuse bits are read out multiple times. Such redundancies are often used as FI countermeasures [2, 24, 35].
- The UART bootloader can now be disabled using a dedicated eFuse bit.
- Enabling Flash Encryption is encouraged as part of the newly introduced *Release Mode*. Stated otherwise, the security of a chip with Flash Encryption disabled is considered suboptimal.

To the best of our knowledge, Espressif has not made any statements about potential hardware countermeasures. Despite the above FI countermeasures, several FI attacks were reported on the ESP32 V3:

- In 2022, Ledger’s Donjon [1] reported the first FI attack on the ESP32 V3, targeting a hardware accelerator of the Advanced Encryption Standard (AES) used for decrypting the Flash contents [14]. Through Body Biasing Injection (BBI), a fault analysis recovered the AES key. The same result was also achieved with a pure Side-Channel Attack (SCA): power-consumption traces were found to be correlated with Hamming distances between consecutive AES states. However, the authors were unsuccessful in retrieving the AES key with EM-FI, likely

because of redundancies in the ROM code. More precisely, corrupting multiple OTP transfers with multiple EM pulses was found to be infeasible.

- In 2023, we were the second to report an FI attack on the ESP32 V3, albeit the first to succeed with EM-FI. The benefit is that EM-FI is less invasive than BBI, *i.e.*, the latter technique requires opening the plastic chip packaging so that a microprobe can reach the backside of the die [26]. The prime reason for our attack to succeed is that only a single EM pulse is required, *i.e.*, the complexity of jointly optimizing the glitch parameters of multiple pulses is avoided. Instead of AES and OTP transfers, we target ROM code running on the CPU, shortly before the RSA signature of the Flash contents is verified. This article describes this attack in more detail.

Several new releases of the ESP32 use RISC-V as ISA instead of Xtensa. In 2023, Courdesse [3] combined SCA and FI to achieve arbitrary code execution on two of these releases: the ESP32-C3 and the ESP32-C6 [15]. First, a power analysis recovered the AES key that encrypts the first 128-byte block of the Flash, which allows to insert arbitrary code into this block. Next, a voltage glitch bypasses Secure Boot such that the inserted code is executed. More precisely, the glitch causes a stack buffer to overflow, thereby overwriting a function return address with a pointer to the code.

1.2 Contributions

We present a novel FI attack against the ESP32 V3, which chains multiple vulnerabilities and uses a single EMFI glitch to access the decrypted flash contents. Our attack works on the most secure configuration and bypasses all countermeasures. Using commercially available tooling, our attack can be reproduced in minutes once effective glitch parameters such as timing and location are found.

By modifying the encrypted flash contents, we force the ROM’s Cyclic Redundancy Check (CRC32) outcome to an arbitrary 32-bit value, which is then loaded into the CPU’s Program Counter (PC) using an EM glitch. This way, we redirect the code execution to the ROM’s Download Mode, which provides access to the decrypted flash contents. We are the first to load a computed value into the PC register of a CPU using a glitch. Moreover, as far as we know, this is the first example of a successful bypass of both Secure Boot and Flash Encryption using a single glitch, on a target with FI countermeasures.

1.3 Disclosure Timeline

The attack described in this paper was responsibly disclosed:

- A technical report specifying the attack was sent to Espressif on April 7, 2023.

- Espressif requested a Common Vulnerabilities and Exposures (CVE) identifier, which was created as CVE-2023-35818 on June 17, 2023.
- Espressif published Security Advisory AR2023-005 on its website on July 11, 2023 [16].
- Espressif transferred a bug bounty of USD 2229 on September 25, 2023.

1.4 Structure

The remainder of this paper is structured as follows. Section 2 provides preliminaries on the ESP32 V3. Section 3 provides the theory of our attack. Section 4 provides practical experiments. Section 5 concludes this work.

2 Preliminaries on Espressif’s ESP32 V3

2.1 System Overview

As is shown in Fig. 2, the ESP32 V3 chip communicates with an external MTP NVM in the form of a Serial Peripheral Interface (SPI) Flash chip. This Flash chip stores the bootloader and the application, which can be signed and/or encrypted. The symmetric encryption key is stored in OTP NVM in the form of fuses. The public key for verifying signatures is stored in Flash, and to protect its integrity, a hash digest of the public key is stored in OTP NVM.

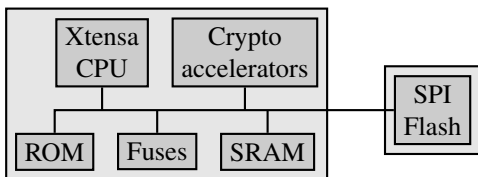


Figure 2: Relevant components of the ESP32 V3.

2.2 Xtensa Instruction Set Architecture

The CPU implements the Xtensa ISA [32]. Instructions are encoded in a 24-bit format, or if it concerns a common use case, in a so-called narrow (n) 16-bit format that can freely be intermixed with the 24-bit format. For example, the 24-bit *move* instruction `movi`, which sets a register to a 12-bit constant, has a 16-bit alternative `movi.n`, which sets a register to a 7-bit constant.

The ISA features 64 general-purpose registers of 32 bits each. However, only 16 registers are visible at any given time through a rotating window, and are labeled `a0` to `a15`. As illustrated in Fig. 3, the window moves back and forth with each function return and function call respectively. For any given subroutine, the return address is stored in register `a0`, the stack pointer is stored in register `a1`, and the input/output

operands are stored in registers `a2` to `a7`. Hence, a caller that causes the window to shift with 8 registers, as is the case for the `call8` instruction, passes operands in registers `a10` to `a15` to physically match the subroutine.

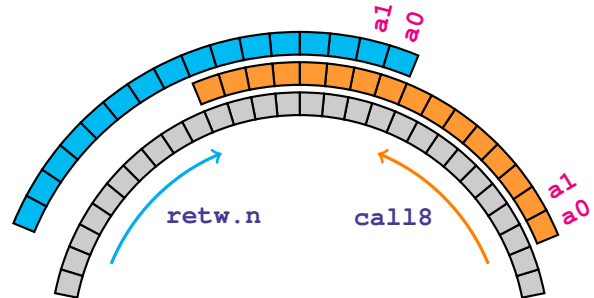


Figure 3: Xtensa rotating window, where the caller and the subprogram are colored orange and cyan respectively.

Given that the shift in window can be either 4, 8, or 12 registers, the two most significant bits of `a0` encode the shift, whereas the 29 least significant bits determine the return address.

2.3 Secure Boot V2

Once an ESP32-based product is fully developed and ready for commercial release, Espressif recommends configuring the chip in Release Mode. Consequentially, Secure Boot and Flash Encryption are both enabled. As illustrated in Fig. 4, the order of operations for constructing the Flash contents is sign-then-encrypt, not encrypt-then-sign. We make abstraction of the application in Fig. 1, given that a forgery of the bootloader inherently compromises the application. Below, the cryptographic algorithms for Secure Boot and Flash Encryption are specified.

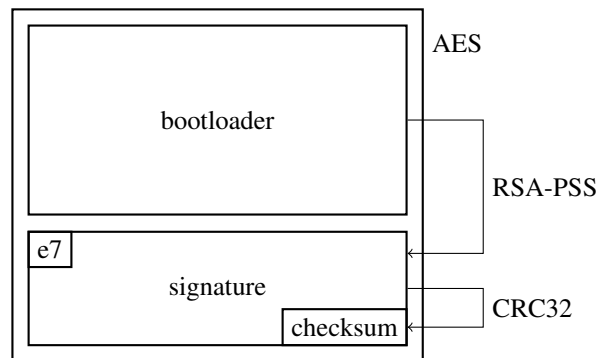


Figure 4: Signed and encrypted Flash data.

2.3.1 Secure Boot: Digital Signatures

Signatures are based on the RSA public-key algorithm, whilst adopting recommendations from Public-Key Cryptography Standards (PKCS) version 2.2, which is published as Request for Comments (RFC) 8017 [21]. More precisely, RSA-3072 is used, in which the public modulus and the signature are each 3072 bits, or 384 bytes. Instead of signing the bootloader image itself, the image is first fed into a Secure Hash Algorithm (SHA) and its digest is signed instead. More precisely, SHA-256 is used, which has a digest of 256 bits, or 32 bytes. The digest is encoded by the Probabilistic Signature Scheme (PSS).

As detailed in Table 1, the produced signature block contains 1216 bytes, starting with the magic byte `0xe7` and ending with a 32-bit checksum. The magic byte is aligned with a 4 KB boundary, *i.e.*, its physical address is an integer multiple of `0x4000`. The public key of RSA is part of the signature block and consists of a modulus, an exponent, and pre-calculated constants that accelerate verification. A SHA-256 digest of the public key is burned into eFuses. The CRC is computed over the 1196 preceding bytes.

Table 1: Signature block format [6].

Offset (bytes)	Size (bytes)	Description
0	1	Magic byte, <code>0xe7</code>
1	1	Version number, <code>0x02</code>
2	2	Zero padding, <code>0x0000</code>
4	32	SHA-256 digest of image
36	384	RSA public modulus
420	4	RSA public exponent
424	384	Pre-calculated constant
808	4	Pre-calculated constant
812	384	Signature
1196	4	CRC32
1200	16	Zero padding, <code>0x00 . . . 00</code>

As can be seen from the publicly released ROM code [9], verification at boot time consists of five consecutive checks. If any of them fails, an error message is printed via UART, and the device is reset. The first check compares the first byte of the signature block to the magic byte `0xe7`. The second check compares the recomputed CRC-32 checksum to its stored counterpart. The third check compares the recomputed SHA-256 digest of the public key to its counterpart stored in fuses. The fourth check compares the recomputed SHA-256 digest of the bootloader image to its stored counterpart. The fifth and last check is the verification of the RSA signature.

2.3.2 Flash Encryption

Flash encryption [5] relies on AES-256. The 256-bit key is stored in eFuses. Espressif adopted a custom *mode of oper-*

ation which is fully parallelizable, *i.e.*, consecutive 128-bit blocks can be encrypted independently of one another, and the same holds for the decryption. Note that this entails random access.

The key for each 128-bit block is derived by XORing the master key stored in eFuses with the physical address of the 256-bit block. Hence, each derived key encrypts two adjacent blocks. For performance reasons: Flash encryption, which is an infrequent operation, uses AES decryption, whereas Flash decryption, which happens on every boot, uses AES encryption.

3 Theory of the Attack

3.1 PC Control Through FI

Originally, PC control through FI was performed in absence of Flash encryption, which is an easier setting than ours. We first describe the original technique, and then introduce our workaround for tackling Flash encryption.

3.1.1 Without Flash Encryption

In 2016, Timmers *et al.* [34] described an FI attack that sets the PC register to a controlled value on CPUs implementing ARM's AArch32 execution state. These controlled values originate from a source that is under control of an attacker, *e.g.*, unencrypted flash. In ARM's AArch32 execution state, the PC register can be used as a destination register for many instructions, which was found to be ideal. Only a single load instruction needs to be corrupted by a glitch in order to load a controlled value directly into the PC register. An effective approach [33] is described below:

- Overwrite the original bootloader in flash with a code payload and sled of pointers. These pointers point to the destination address in executable memory at which the code payload is copied to.
- As a result, when the device is powered, the ROM code will copy the code payload and the pointers to the same destination as the original bootloader. Then, assuming Secure Boot is enabled, the signature check would fail and the target is reset.
- For the attack, a glitch is injected after the code payload is copied, but while the pointers are being copied. This glitch modifies the destination operand of a load instruction such that a controlled value is loaded into the PC register. This effectively executes the code payload well-before the signature is verified.

The above approach is also possible on CPU architectures where the PC register is not directly addressable, including ARM's AArch64 and Xtensa. For these type of architectures,

the PC register can only be controlled indirectly, e.g., by corrupting the operand of a branch, jump or return instruction.

3.1.2 With Flash Encryption

On modern SoC where Flash contents are encrypted, the technique in Section 3.1.1 might still work, on the condition that the CPU operates directly on ciphertext. Then, a controlled value can be loaded into the PC register simply by overwriting the ciphertext.

However, on the ESP32, the flash contents are decrypted on-the-fly by a hardware implementation of AES. This process is done completely transparent to the CPU, which never operates on the encrypted contents, only on the decrypted contents. Therefore, any modification in the external flash will end up in the context of the CPU as *gibberish*. In theory, a brute-force attack on the 32-bit address space might still be possible, *i.e.*, the ciphertext is randomly manipulated until the pointer of interest is found. In practice though, the time needed for performing this search is likely excessive, given that devices typically take a few milliseconds to boot.

Therefore, we decided to find another method for slipping in one or more controlled values, which we intend to load into the PC register using a glitch. On the ESP32 V1, Raelize [28] leveraged the UART bootloader, which could not be disabled. However, Espressif patched this vulnerability on the ESP32 V3. We decided to slip in a controlled 32-bit value into the context of the CPU by tampering with the CRC operation that is performed over the signature block. Setting the PC register of the CPU to the result of this CRC32 operation using a glitch is the main novelty of this paper. Gratchoff [19] previously described this as a potential approach, however, to the best of our knowledge, this has never been performed in practice.

3.2 Modifying the Flash

We demonstrate our attack on a bootloader which prints “Hello, World!”. There is no application, as shown in Fig. 1, because being able to execute a modified bootloader compromises the application by default. The boot log observed on the UART is given in Fig. 5. Additional line breaks have been inserted to accommodate the two-column format of this paper.

The ROM code reports explicitly that secure boot is enabled and the secure boot verification succeeded. Even though not specifically reported, flash encryption is enabled as well. Any change to the bootloader or its signature block, both of which are stored encrypted in flash, causes an error message to be displayed in the boot log. If the signature block is modified, the checksum verification fails, and the error message in Fig. 6 is printed. The key observation is that that the checksum verification is done in the plaintext domain.

```
ets Jul 29 2019 12:21:46
rst:0x1 (POWERON_RESET),boot:0x13
(SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,
hd_drv:0x00,wp_drv:0x00
mode:2, clock div:2
secure boot v2 enabled
secure boot verification succeeded
load:0x3fff0020 len:0xc8c
load:0x40078000 len:0x2020
load:0x40080400 len:0xeac
entry 0x40080640
I (41) boot: ESP-IDF v5.0.1-397-g3050ea656f 2nd
stage bootloader
I (41) boot: compile time 16:51:07
I (41) boot: chip revision: v3.0
I (45) boot.esp32: SPI Speed      : 40MHz
I (50) boot.esp32: SPI Mode      : DIO
I (54) boot.esp32: SPI Flash Size : 2MB
I (59) boot: Enabling RNG early entropy source...
Hello, World!
```

Figure 5: UART for a bootloader that prints “Hello, World!”.

```
...
secure boot v2 enabled
Sig block 0 invalid: Stored CRC 0xbaaeaf78
calculated 0xdeadbeef
secure boot verification failed
```

Figure 6: UART boot log where the CRC fails.

We refrain from corrupting the first 16-byte block of the signature as this includes a byte at offset 0 which is used as a magic value. Whenever this value is not **0xe7**, the signature block is not considered a signature block and the checksum verification is not performed. The error message that is printed when the magic value is modified is shown in Fig. 7.

```
...
secure boot v2 enabled
No signature block magic byte found at signature
sector (found 0xc3 not 0xe7). Image not V2 signed?
secure boot verification failed
```

Figure 7: UART boot log where the magic byte is corrupted.

By performing manipulations of the ciphertext, we can solve a system of linear equations and set the recomputed checksum to any 32-bit value of choice. In fact, the hexspeak value **0xdeadbeef** in Fig. 6 is no coincidence, and serves to demonstrate this ability. For our attack, we modify this hexspeak value into a pointer, *i.e.*, a memory address, which we then load into PC using a glitch.

3.3 Solving Equations

We now specify how the system of linear equations is constructed. As this section is mathematical, unlike the rest of this paper, a notation system is introduced. Variables and constants are denoted by characters from the Latin and Greek alphabets respectively. Orthogonal to this convention: scalars are denoted by regular lowercase characters, binary vectors are denoted by bold lowercase characters, and binary matrices are denoted by bold uppercase characters. All vectors are column vectors.

We leverage that CRC-32 is an affine function, as formalized in Eq. (1), where \oplus denotes XORing and where constant $\boldsymbol{\gamma} \in \{0, 1\}^{32}$ only depends on the size of the input \mathbf{x} . For inputs $\mathbf{x} \in \{0, 1\}^{9568}$, which corresponds to the first 1196 bytes of the signature block, it holds that $\boldsymbol{\gamma} = \mathbf{0x6b691bc6}$. Given that $\boldsymbol{\gamma}$, eventually, cancels out, its value is inconsequential.

$$\text{CRC-32}(\mathbf{x}_1 \oplus \mathbf{x}_2) = \text{CRC-32}(\mathbf{x}_1) \oplus \text{CRC-32}(\mathbf{x}_2) \oplus \boldsymbol{\gamma}. \quad (1)$$

Input $\mathbf{x} \in \{0, 1\}^{9568}$ spans 75 AES plaintext blocks $\mathbf{p} \in \{0, 1\}^{128}$, as formalized in Eq. (2) where \parallel is the concatenation operator. The first block, \mathbf{p}_0 , contains the magic byte $\mathbf{0xe7}$. The last block, \mathbf{p}_{74} , shares only 96 bits with \mathbf{x} , and the 32 excluded bits comprise the stored checksum $\mathbf{s}_{\text{stored}} = \text{CRC-32}(\mathbf{x})$.

$$\mathbf{x} \triangleq \mathbf{p}_0 \parallel \mathbf{p}_1 \parallel \dots \parallel \mathbf{p}_{73} \parallel (\mathbf{p}_{74} \bmod 2^{96}). \quad (2)$$

The external Flash contains the corresponding ciphertexts $\mathbf{c}_0, \mathbf{c}_1, \dots, \mathbf{c}_{74}$, which we can alter. The first block, \mathbf{c}_0 , is unaltered. Otherwise, the magic byte is not found with probability 255/256, and the UART boot log is uninformative, as shown in Fig. 7.

Instead, we consecutively alter blocks \mathbf{c}_1 to \mathbf{c}_{32} . In the first iteration, we overwrite \mathbf{c}_1 with a value \mathbf{c}_1^* that is selected uniformly at random from $\{0, 1\}^{128}$. Consequentially, the corresponding plaintext block \mathbf{p}_1 changes to an unknown value $\mathbf{p}_1^* \triangleq \mathbf{p}_1 \oplus \mathbf{e}_1$. By booting the ESP32 with this modification, and parsing the UART log, we obtain the checksum difference $\mathbf{d}_1 \triangleq \mathbf{s}_{\text{calculated},1} \oplus \mathbf{s}_{\text{stored}}$. From linearity in Eq. (1), it follows that the difference \mathbf{d}_1 only depends on plaintext error \mathbf{e}_1 , as specified in Eq. (3). Nevertheless, $\mathbf{e}_1 \in \{0, 1\}^{128}$ cannot be recovered from $\mathbf{d}_1 \in \{0, 1\}^{32}$ due to the 96-bit difference in length, *i.e.*, there are many \mathbf{e}_1 's that result in the same \mathbf{d}_1 . This is fine: \mathbf{e}_1 does not need to be recovered, and we merely store the pair $(\mathbf{c}_1^*, \mathbf{d}_1)$ for further use.

$$\mathbf{d}_1 = \text{CRC-32}(0_{128} \parallel \mathbf{e}_1 \parallel 0_{9312}) \oplus \boldsymbol{\gamma}. \quad (3)$$

Now, the same principle is repeated to obtain pairs $(\mathbf{c}_2^*, \mathbf{d}_2)$ until $(\mathbf{c}_{32}^*, \mathbf{d}_{32})$, as formalized in Eq. (4). Again, recovery of \mathbf{e}_2 until \mathbf{e}_{32} is unnecessary.

$$\begin{aligned} \mathbf{d}_2 &= \text{CRC-32}(0_{256} \parallel \mathbf{e}_2 \parallel 0_{9184}) \oplus \boldsymbol{\gamma}. \\ &\vdots \\ \mathbf{d}_{32} &= \text{CRC-32}(0_{4096} \parallel \mathbf{e}_{32} \parallel 0_{5344}) \oplus \boldsymbol{\gamma}. \end{aligned} \quad (4)$$

Instead, we linearly combine the known differences \mathbf{d}_1 until \mathbf{d}_{32} into a desired difference $\mathbf{d} \triangleq \mathbf{s}_{\text{pointer}} \oplus \mathbf{s}_{\text{stored}}$, where $\mathbf{s}_{\text{pointer}}$ is the memory address we want to jump to. This is achieved by solving the system of linear equations in Eq. (5) for $\mathbf{z} \in \{0, 1\}^{32}$. Note the absence of constant $\boldsymbol{\gamma}$. Each bit z_i of \mathbf{z} , where $i \in [1, 32]$, determines whether or not the corresponding ciphertext block should be corrupted: if $z_i = 0$, the original ciphertext is \mathbf{c}_i remains in place, otherwise, the random value \mathbf{c}_i^* is used.

$$\mathbf{D}\mathbf{z} = \mathbf{d}, \quad \text{where } \mathbf{D} = (\mathbf{d}_1 \quad \mathbf{d}_2 \quad \dots \quad \mathbf{d}_{32}). \quad (5)$$

One problem remains though: the matrix $\mathbf{D} \in \{0, 1\}^{32 \times 32}$ is not necessarily invertible. Under the assumption that \mathbf{D} is selected uniformly at random from $\{0, 1\}^{32 \times 32}$, which is a reasonable abstraction, the probability that \mathbf{D} is invertible given in Eq. (6). The proof is straightforward and imagines that columns are added one-by-one [25]: if the previous $i - 1$ columns are linearly independent, the addition of column i causes linear dependence with probability 2^{i-33} . For example, the first and last columns cause linear dependence with probability $1/2^{32}$ and $1/2$ respectively. Logarithms help with numerical evaluation, and result in a probability of around 28%.

$$\begin{aligned} \Pr(\text{rank}(\mathbf{D}) = 32) &= \prod_{i=1}^{32} (1 - 2^{-i}) \\ &= \exp\left(\sum_{i=1}^{32} (\log(2^i - 1) - \log(2^i))\right) \approx 28\%. \end{aligned} \quad (6)$$

To ensure that \mathbf{D} is invertible, we check whether its rank increases for each column \mathbf{d}_i that is added, as formalized in Algorithm 1. If the rank does not increase, a new corrupted ciphertext \mathbf{c}_i^* is selected uniformly at random. As can be seen from the invertibility proof [25], it are usually the last few columns that require a retry. Observe that there is no need to retake measurements if we would want to build images for more than one pointer of interest.

Alternatives to Algorithm 1 could be devised. For example, instead of gathering pairs $(\mathbf{c}_i^*, \mathbf{d}_i)$ for 32 AES blocks, pairs could be gathered for, say, 40, blocks. From these 40 blocks, 32 blocks that result in an invertible \mathbf{D} are then retained.

Although solving a system of equations is the canonical approach, it is only possible because the signature block happens to be long. Originally, we disclosed an alternative method to Espressif that would also have worked for small signature blocks, at the minor inconvenience of a 32-bit brute-force

Algorithm 1: Measurement for CRC insertion

Input: Original bootloader, $\mathbf{b} \in \{0, 1\}^*$ **Input:** Index of magic byte, $m \in \mathbb{N}$ **Input:** Pointer of interest, $\mathbf{s}_{\text{pointer}} \in \{0, 1\}^{32}$ **Output:** Modified bootloader, $\mathbf{b}^* \in \{0, 1\}^*$

```
1  $\mathbf{C} \leftarrow \mathbf{0}_{32 \times 128}$ 
2  $\mathbf{D} \leftarrow \mathbf{0}_{32 \times 32}$ 
3 for  $i \leftarrow 1$  to 32 do
4    $\mathbf{b}^* \leftarrow \mathbf{b}$ 
5   do
6      $\mathbf{c}_i \leftarrow \{0, 1\}^{128}$ 
7      $\mathbf{b}^*[m+i128 : m+i128+127] \leftarrow \mathbf{c}_i$ 
8     Program  $\mathbf{b}^*$ 
9     Fetch  $\mathbf{s}_{\text{calculated}}$  and  $\mathbf{s}_{\text{stored}}$  from UART
10     $\mathbf{D}[:, i] \leftarrow \mathbf{s}_{\text{calculated}} \oplus \mathbf{s}_{\text{stored}}$ 
11    while  $\text{rank}(\mathbf{D}) \neq i$ 
12     $\mathbf{C}[:, i] \leftarrow \mathbf{c}_i$ 
13  $\mathbf{b}^* \leftarrow \mathbf{b}$ 
14  $\mathbf{z} \leftarrow \mathbf{D}^{-1}(\mathbf{s}_{\text{pointer}} \oplus \mathbf{s}_{\text{stored}})$ 
15 for  $i \leftarrow 1$  to 32 do
16   if  $\mathbf{z}[i]$  then
17      $\mathbf{b}^*[m+i128 : m+i128+127] \leftarrow \mathbf{C}[:, i]$ 
```

search. In this method, we perturb $\eta \geq 4$ blocks of the signature, and for each block, we select $\lambda \geq 2$ ciphertexts \mathbf{c}^* uniformly at random from $\{0, 1\}^{128}$, where $\lambda^\eta > 2^{32}$. We store pairs $(\mathbf{c}_{i,j}^*, \mathbf{d}_{i,j})$, where $i \in [1, \eta]$ and $j \in [1, \lambda]$, and where $\mathbf{d}_{i,j}$ is given in Eq. (7).

$$\mathbf{d}_{i,j} = \text{CRC-32}(0_{128:i} \parallel \mathbf{e}_{i,j} \parallel 0_{9440-128:i}) \oplus \gamma. \quad (7)$$

Next, the goal is to find indices $j_1, j_2, \dots, j_\eta \in [1, \lambda]$ such that $\mathbf{d}_{1,j_1} \oplus \mathbf{d}_{2,j_2} \oplus \dots \oplus \mathbf{d}_{\eta,j_\eta} = \mathbf{d}$. This search took less than one hour on a laptop. The corresponding ciphertexts $\mathbf{c}_{1,j_1}^*, \mathbf{c}_{2,j_2}^*, \dots, \mathbf{c}_{\eta,j_\eta}^*$ are applied.

3.4 Attack Surface for FI

The result of the checksum operation, *i.e.*, the pointer of interest, propagates through several CPU registers before the chip, eventually, resets. This propagation path can be followed with relative ease, given that Espressif published the ROM code in ELF format [9]. If the ROM code would not have been published, the code would have to be extracted from the device through either delayring [17, 18] or an exploit [4, 31]. The ELF file is loaded in Ghidra, which is reverse-engineering software that decompiles assembly instructions into C, among other features. Our analysis reveals that the proverbial *attack surface* for FI comprises three subroutines.

The first subroutine is `crc32_le`, which computes the checksum, and is shown in Fig. 8. The XOR operation at

address `0x4005d019` writes the computed checksum into register `a2`. If this instruction could be corrupted such that the destination register `a2` changes to the return address `a0`, as formalized in Corruption 1, the pointer of interest would be loaded into the PC register of the CPU. Given that `a2` and `a0` are encoded as four-bit fields `0x2` and `0x0` respectively, this would only require a single bit flip.

```
crc32_le()
0x4005cfec entry a1, 0x20
0x4005cfef movi.n a8, 0xff
0x4005cff1 xor a2, a8, a2
0x4005cff4 l32r a9, 0x4005cfe8
0x4005cff7 movi.n a8, 0x0
0x4005cff9 j 0x4005d014
0x4005cffc add.n a10, a3, a8
0x4005cffe l8ui a10, a10, 0x0
0x4005d001 addi.n a8, a8, 0x1
0x4005d003 xor a10, a10, a2
0x4005d006 extui a10, a10, 0x0, 0x8
0x4005d009 addx4 a10, a10, a9
0x4005d00c l32i.n a10, a10, 0x0
0x4005d00e srli a2, a2, 0x8
0x4005d011 xor a2, a10, a2
0x4005d014 bne a8, a4, 0x4005cffc
0x4005d017 movi.n a3, 0xff
0x4005d019 xor a2, a3, a2
0x4005d01c retw.n
```

Figure 8: ROM code of `crc32_le`.

Corruption 1. At address `0x4005d019`, the instruction `xor a2, a3, a2` with encoding `0x302320` is corrupted into `xor a0, a3, a2` with encoding `0x300320`.

The second potential target for FI is subroutine `ets_secure_boot_verify_signature`, which is the caller of `crc32_le`, and the relevant part is shown in Fig. 9. Due to the shifting window, the pointer is returned in register `a10` after the `call8` instruction at address `0x4006547e`, and is copied to register `a13` at address `0x40065485`. If the contents of `a10` differ from the stored checksum in register `a12`, a branch is taken at address `0x40065488` to resume normal operation. Otherwise, the subroutine `ets_printf` is called at address `0x40065491` to print the CRC error message. The unconditional jump at address `0x40065565` results in a reset.

Again, taking PC control by overwriting the return address `a0` is plausible. Most notably, the move instruction at address `0x40065485` could be corrupted such that the destination register changes from `a13` to `a0`, as formalized in Corruption 2. Although this entails three bit flips, the probability of

```

0x40065474 movi    a12, 0x4ac
0x40065477 movi.n  a10, 0x0
0x40065479 mov.n   a11, a6
0x4006547b movi    a2, 0x4ac
0x4006547e call8   crc32_le
0x40065481 add.n   a2, a6, a2
0x40065483 l32i.n  a12, a2, 0x0
0x40065485 mov     a13, a10
0x40065488 beq    a10, a12, 0x40065498
0x4006548b l32r   a10, 0x40065428
0x4006548e mov     a11, a7
0x40065491 call8   ets_printf
0x40065494 j      0x40065565

```

Figure 9: ROM-code fragment of `ets_secure_boot_verify_signature`.

occurrence could be significant depending on the unknown *fault model*: all flips are of the type $1 \rightarrow 0$, and they all occur within a single 4-bit field. Different fields are processed by different circuits, so there is no reason why setting an entire field to zero would be unrealistic.

Corruption 2. At address `0x40065485`, the instruction `mov a13, a10` with encoding `0x20daa0` is corrupted into `mov a0, a10` with encoding `0x200aa0`.

Alternatively, it might be possible to corrupt the opcode of the move instruction and turn it into an unconditional register jump `jx`, as formalized in Corruption 3. Although this entails four bit flips, it equates to setting two out of six fields to zero.

Corruption 3. At address `0x40065485`, the instruction `mov a13, a10` with encoding `0x20daa0` is corrupted into `jx a10` with encoding `0x000aa0`.

The third and last subroutine for potential FI is `ets_printf`, which prints a formatted string similar to its C counterpart `printf`. The pointer of interest is passed as an argument through register `a13`. The ROM code is not analyzed here due to its length.

3.5 Pointers of Interest

As listed in Table 2, we jump to two ROM functions. The first function, `ets_fatal_exception_handler`, prepares a formatted string and calls `ets_printf`, as shown in Fig. 10. The relative simplicity of a print enables us to efficiently tune EM-FI glitch parameters later-on: the delay, the power, and the XY coordinates. Furthermore, because the value of five registers is printed, useful insights about the injected fault can potentially be gained.

Once suitable glitch parameter values are found, we change the Flash image of our target device and jump to *Download*

Table 2: Pointers of interest in the ROM code.

Address	Function
<code>0x40006864</code>	<code>ets_fatal_exception_handler</code>
<code>0x80006864</code>	
<code>0x40008ceb</code>	<code>UartDwnLdProc</code>
<code>0x80008ceb</code>	
<code>0x40006864</code>	<code>l32r a10, 0x3ff9e820</code>
<code>0x40006867</code>	<code>mov.n a11, a6</code>
<code>0x40006869</code>	<code>mov.n a12, a5</code>
<code>0x4000686b</code>	<code>mov.n a13, a4</code>
<code>0x4000686d</code>	<code>mov.n a14, a3</code>
<code>0x4000686f</code>	<code>mov.n a15, a2</code>
<code>0x40006871</code>	<code>call8 ets_printf</code>

Figure 10: ROM-code fragment of `ets_fatal_exception_handler`.

Mode instead, *i.e.*, the ROM function `UartDwnLdProc`. The latter jump is more restrictive than `ets_fatal_exception_handler` because three input parameters should have proper values.

Given that the windowing mechanism of the Xtensa ISA has a crucial role in this matter, we experiment with addresses of the form `0x4XXXXXXXX` and `0x8XXXXXXXX`. The return instruction `retw.n` uses the two most significant bits of `a0` are to determine the shift in window, whereas the 29 least significant bits determine the next PC.

3.6 Simulating Faults with GDB

Before building the FI setup and performing the attack in practice, we simulated the desired faults with Espressif’s GNU Debugger (GDB) to confirm their effect. For this purpose, we prepared a bootloader where the signature block is corrupted, and the recomputed checksum is, consequentially, incorrect. Upon flashing this bootloader, Corruption 1 and Corruption 2 are simulated as shown in Fig. 11a and Fig. 11b respectively. In both simulations, we set a *hardware breakpoint* at the targeted instruction and overwrite register `a0` with the desired pointer during the break.

To determine whether or not `ets_fatal_exception_handler` is reached, we merely need to observe the UART output, and check whether or not the string is printed. For Download Mode, there is no welcome message, but we can check whether or not an additional hardware breakpoint deep within this mode is reached. Our conclusion is that pointers of the form `0x8XXXXXXXX` result in a successful jump, whereas pointers of the form `0x4XXXXXXXX` do not.

For Corruption 3, we performed similar GDB experiments.

```

hbreak *0x4005d01c    hbreak *0x40065485
continue             continue
set $a0 = 0x80006864 set $a0 = 0x80006864
continue             continue

```

(a) Corruption 1

(b) Corruption 2

Figure 11: Simulation of (a) Corruption 1 in `crc32_le` and (b) Corruption 2 in `ets_secure_boot_verify_signature` with GDB.

However, the conclusion is different: pointers of the forms `0x4XXXXXXXX` and `0x8XXXXXXXX` both result in a successful jump.

4 Practical Experiments

4.1 Target Preparation

We target an ESP32-DevKitC V4 [8] with an ESP32-WROOM-32E module [13], which is a small-sized and commercially available development board produced by Espressif. To enable EM-FI, we removed the metal shield that covers both the ESP32 V3 chip and the SPI flash chip with a KADA 852D+ hot air gun. No-clean flux is applied to facilitate this process.

Upon confirming that the board survived the hot air, we manually enable the security features of the ESP32 V3 by burning eFuses. Although Espressif provides a partially automated process, the manual approach is more convenient for developing an attack: the security features can be enabled one by one, instead of altogether automatically. Figure 12a shows the eFuses for enabling Secure Boot. The SHA-256 digest of the RSA public key is obtained from the file `rsa.pem`. Figure 12b shows the eFuses for enabling Flash encryption. The 256-bit AES key is contained in the binary file `aes.bin`. Figure 12c shows the eFuses for enabling Release Mode.

Burning the eFuses for enabling Flash Encryption, as shown in Fig. 12b, is postponed as long as possible. Although our attack works equally well with and without Flash Encryption, this allows us to gradually develop the attack and compare timing in the two cases.

Likewise, burning the eFuse for disabling Download Mode, as shown in Fig. 12d, is postponed as long as possible. Although this security feature does not preclude our attack, in which we enter Download Mode by directly jumping to address `0x40008ceb`, one cannot easily program the external Flash anymore after the eFuse is burned. Recall from Algorithm 1 that at least 33 manipulated Flash images need to be programmed to set the recomputed checksum to an arbitrary pointer. Starting from a valid signed and encrypted image, where the bootloader prints “Hello, World!”, we created four images that correspond to the four interesting jump locations in Table 2. Only after tuning the glitch parameters, we burn

```

$ espefuse.py burn_key_digest rsa.pem
$ espefuse.py burn_efuse ABS_DONE_1 1

```

(a) Secure Boot.

```

$ espefuse.py burn_key flash_encryption aes.bin
$ espefuse.py burn_efuse FLASH_CRYPT_CNT 1
$ espefuse.py burn_efuse FLASH_CRYPT_CONFIG 15

```

(b) Flash Encryption.

```

$ espefuse.py burn_efuse DISABLE_DL_ENCRYPT 1
$ espefuse.py burn_efuse DISABLE_DL_DECRYPT 1
$ espefuse.py burn_efuse DISABLE_DL_CACHE 1
$ espefuse.py write_protect_efuse FLASH_CRYPT_CNT

```

(c) Release Mode.

```

$ espefuse.py burn_efuse UART_DOWNLOAD_DIS 1

```

(d) Download Mode.

Figure 12: Burning eFuses for (a) enabling Secure Boot, (b) enabling Flash Encryption, (c) enabling Release Mode, and (d) disabling Download Mode.

the eFuse. Because the Flash is external, programming in principle remains possible, at the minor inconvenience of soldering an SPI programmer to the chip.

4.2 EM-FI Setup

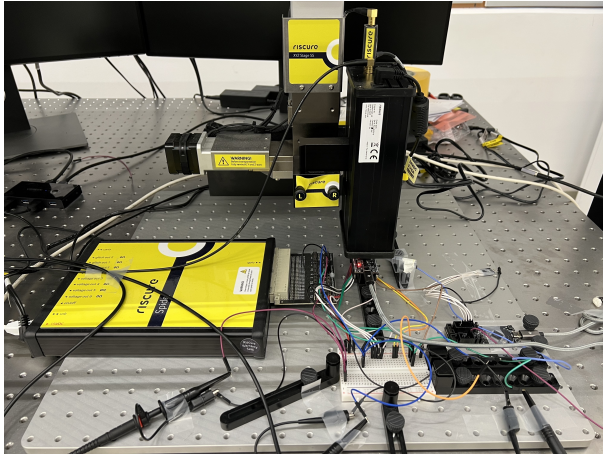
We used Riscure’s EM-FI setup [29]. The motorized XYZ stage is shown in Fig. 13a. We used the large red Classic probe tip, which has a diameter of 4 mm. The targeted ESP32 V3 board is stabilized with double-sided tape. A desktop computer communicates with the board via the Micro-USB connector; a Windows COM port provides the serial interface. As is shown in Fig. 13b, an electric wire is attached to the *chip-enable* pin from the SPI Flash chip, thereby providing a timing reference (*i.e.* trigger) for the EM glitches.

4.3 Tuning Glitch Parameters

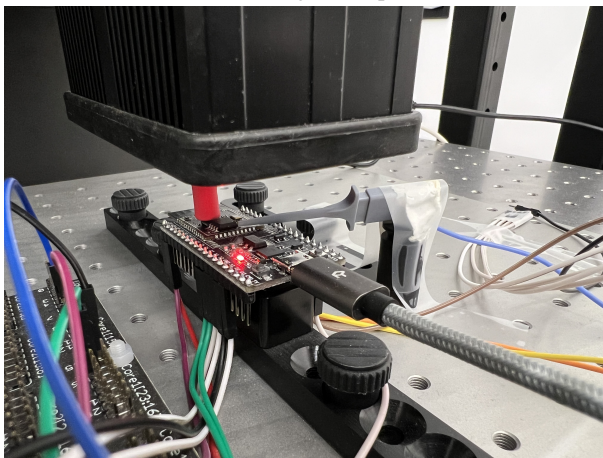
4.3.1 Coarse Timing: Execution Trace

The most crucial glitch parameter to be tuned is the timing. As in prior work, we use the so-called *chip-enable* signal of the Flash chip as a timing reference. As shown in Fig. 14, the *chip-enable* signal is observed to consist of five relatively large blocks where data is copied from Flash to SRAM. We used a Teledyne LeCroy WavePro 804HD oscilloscope to take these measurements.

Next, we determine when `crc32_le` is executed with respect to these five copy blocks. For this purpose, we use Espressif’s GDB to trace the program execution. We created a



(a) XYZ stage and Spider



(b) Target and glitch amplifier

Figure 13: Riscure EM FI setup.

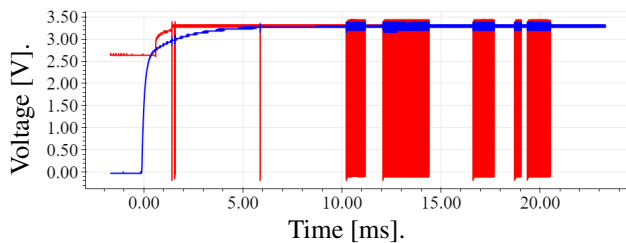


Figure 14: Blocks of data copied from Flash to SRAM. The reset signal is colored blue; the chip-enable signal is colored red.

Python script that starts from a hardware breakpoint in `main`, and then pauses at each instruction with `stepi` until the program ends. At each pause, we log the function name, the value of the program counter, and the value of registers `a0` to `a15`. This whole process takes less than two hours. As shown in Fig. 15, `crc32_le` is executed shortly after block #5.

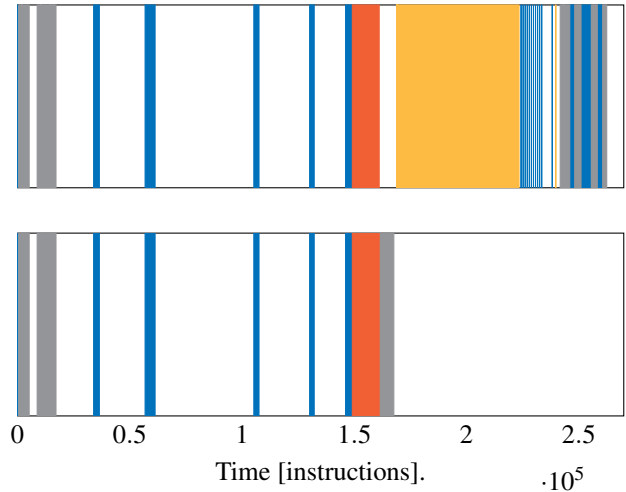
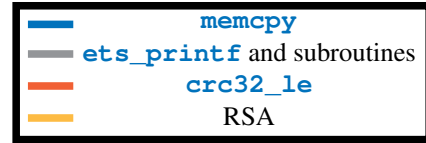


Figure 15: Execution trace when the CRC is correct (top) and wrong (bottom).

Remark that execution traces are only one possible method to obtain course timing information. An unexplored alternative is SCA, *e.g.*, by taking power-consumption measurements. In this approach, power traces are collected for the two classes, where the recomputed checksum is correct and wrong for the first and second class respectively. Initially, the two classes should be quasi indistinguishable, and shortly after `crc32_le`, the classes should diverge drastically.

4.3.2 Refined Timing: FI as Virtual Oscilloscope

Next, we refine the timing by using EM-FI as a virtual oscilloscope [20, 23, 30]. We inject glitches in a large time interval while fetching the CRC error string from UART, both with and without Flash Encryption. Because only tiny differences could be observed, we only show results obtained with Flash Encryption enabled in Fig. 16 and all subsequent scatter plots.

For Fig. 16 and all subsequent plots, we adopt the color legend from Table 3. Green dots represent the baseline, *i.e.*, the fault has no observable effect in the UART output, and the device eventually resets because the stored and recomputed checksums are different. Yellow dots represent fault-induced crashes, *i.e.*, the fault and not the checksum difference causes the target device to reset. Cyan dots indicate that the CRC error string is deformed, *e.g.*, characters are missing or corrupted. Orange dots indicate that the CRC error string is well-formed, but the recomputed checksum is corrupted. Purple dots indicate that the stored checksum is corrupted instead. Pink dots indicate that the recomputed and stored checksums

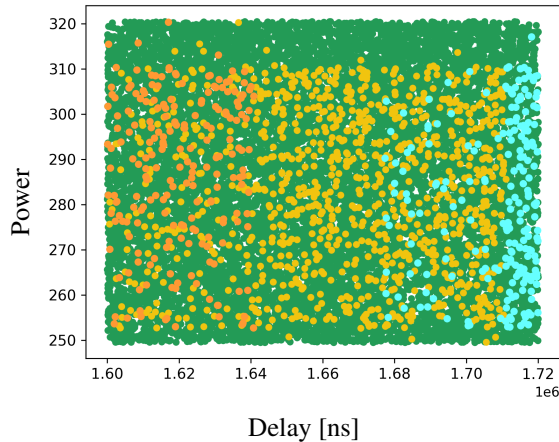


Figure 16: FI as an oscilloscope.

are both corrupted. Red dots indicate successful PC control.

Table 3: Color legend for the scatter plots in Figs. 16 to 19.

green	Nominal response.
yellow	No response, <i>i.e.</i> , a crash.
cyan	Deformed CRC error string.
orange	Altered recomputed checksum.
purple	Altered stored checksum.
pink	Altered recomputed and stored checksums.
red	Successful jump.

For improved visibility, three measures are taken for all scatter plots. Firstly, small random errors are added to the shown pair of variables, which is also a common practice in Riscure’s own visualization software, Spotlight. Otherwise, most dots would coincide. Secondly, dots are drawn in the order of the legend in Table 3. Otherwise, a small number of red dots could be obscured by large numbers of green and yellow dots, for example. Thirdly, dots with different colors might be drawn with different diameters.

In Fig. 16, two regions are of particular interest. The region with the orange dots corresponds to `crc32_le`. The region with cyan dots is part of `ets_printf`. From the ROM code execution trace analysis, we know that `ets_printf` starts well before the cyan dots appear. The function `ets_secure_boot_verify_signature` is likely executed in the region between the orange and the cyan dots. This is not visible because of the very small number of instructions the function is composed of.

4.3.3 XY-Coordinates and Power

The XYZ-stage is used to scan the surface of the ESP32 chip. Although the surface is approximately square, the EM-FI probe is partially blocked by the neighboring Flash chip and is free to move in a rectangular area of roughly 5 mm × 2 mm.

Because the Flash chip has only eight pins, displacement though soldering is possible, but is unnecessary for the attack to succeed. Within the rectangular area, the probe moves in a 30-by-30 grid.

Fig. 17 shows the result of our surface scan. For clarity, only the green, yellow, and red dots are shown. Red dots represent successes, *i.e.*, the string in `ets_fatal_exception_handler` is successfully printed. For this particular scan, we used the `0x8XXXXXXXX` address. The key takeaway of Fig. 17 is that the probe can be placed inside at a relatively large fraction of the chip’s surface in order to inject a successful glitch. Stated otherwise, finding the proverbial *needle in a haystack* primarily applies to time, not space. This is unsurprising because the CPU is relatively large and, arguably, the centerpiece of the chip.

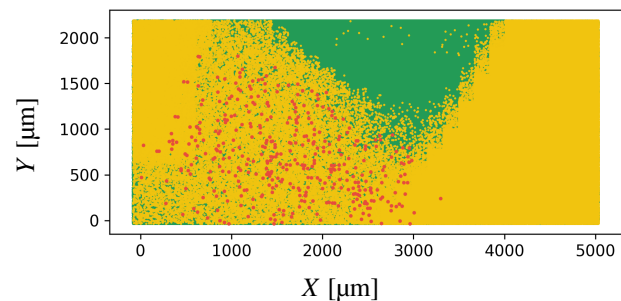


Figure 17: Scan of the chip surface.

Figure 18 shows a similar scatter plot, now pairing the glitch delay and the glitch power. The power randomly varies between 20% and 100% of the physical maximum; 500 is merely a scaling factor configured in software. The key takeaway of the plot is that two different instruction corruptions result in the desired jump.

Our setup performs around 3.4 attempts per second. The red dots can be reproduced with success rates of around 2%, upon fixing the position (X, Y) , the delay, and the power.

4.4 Root Cause Analysis

As for virtually all FI attacks described in the literature, there is no absolute certainty about the exact instruction corruption that caused the attack to succeed. Nevertheless, clues can be obtained.

The easiest available source of clues is the UART log. Recall that `ets_fatal_exception_handler` prints registers `a2` to `a6`. By matching the printed values to the GDB execution trace, we conclude that `a2` to `a6` from `ets_secure_boot_verify_signature` are printed. The addition `add.n a2, a6, a2` at address `0x40065481` is confirmed to take place, *i.e.*, the instruction corruptions happen from `0x40065483` onwards. Another clue obtained from UART is that for the second cluster of red dots in Fig. 18, the CRC error string is printed, whereas for the first cluster,

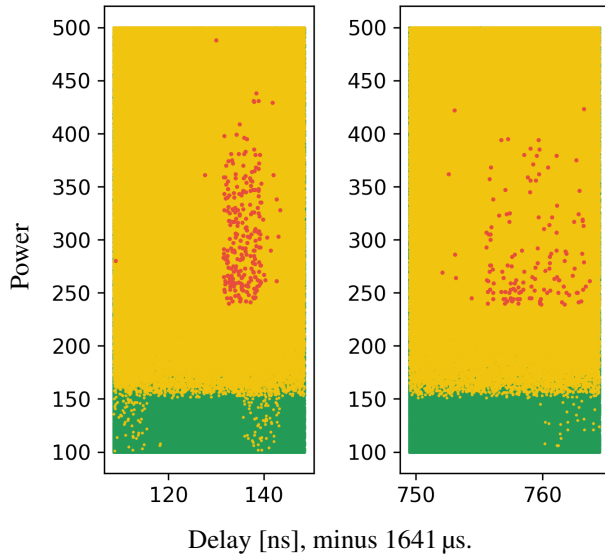


Figure 18: Delay versus power.

this print is missing. Based on the above observations, we set forth a hypothesis for each cluster:

1. For the first cluster, we corrupt an instruction in `ets_secure_boot_verify_signature` between addresses `0x40065483` and `0x40065491`. This corruption causes a jump to `ets_fatal_exception_handler` with immediate effect, and without shifting the register window. The above behavior is consistent with `jx` in Corruption 3, but inconsistent with overwriting the return address `a0` in Corruption 2.
2. For the second cluster, we corrupt an instruction in the beginning of `ets_printf`. This corruption causes a jump to `ets_fatal_exception_handler` with a delayed effect, and rotates back the window with eight registers. This behavior is consistent with overwriting the return address `a0`.

A second source of clues is the aforementioned notion of using FI as an oscilloscope. Figure 19 covers a narrow time interval around the two clusters of red dots. Purple dots indicate that the stored checksum is wrong, whereas the recomputed checksum is correct. Pink dots indicate that both checksums are wrong. We see a stripe pattern with a period of around 25 ns. This corresponds to a frequency of 40 MHz, which is also the frequency of the external crystal oscillator.

The first cluster of reds dots is located within a purple region, which is consistent with Corruption 3. Note that the stored checksum is loaded right before.

4.5 Jumping to Download Mode

After having tuned our parameters, we prepare the `0x80008ceb` image for Download Mode and burn the eFuse in Fig. 12d. If successful, we can leverage this mode to read and write memory, and execute arbitrary code.

To verify that we are successful in getting into *Download Mode*, we use UART to send the packet below, which is a command for reading memory. As defined in Espressif’s Serial Line Internet Protocol (SLIP) [11], each packet begins and ends with byte `0xc0`. The second byte is `0x00` and indicates that the packet is a request. The third byte is `0x0a` and indicates the nature of the request: reading data from a memory address. Byte 4 and 5, with value `0x0400`, indicate that four bytes of data are attached to this packet, *i.e.*, the memory address. Bytes 6 to 9, with value `0x00000000`, are unused. Bytes 10 to 13 encode the memory address `0x3f401000` in *little endian*. This virtual address is mapped to physical address `0x1000` of the external Flash, where the firmware file header is written [10], starting with a magic byte `0xe9`.

```
c0000a04000000000000010403fc0.
```

The ESP32 responds with the packet below. Unlike before, the second byte is `0x01` and indicates that the packet is a response. The third byte is still `0x0a`, repeating the nature of the request. Again, byte 4 and 5, with value `0x0400`, indicate that four bytes of data are attached. Byte 6 to 9, with value `e9030210`, are decrypted Flash contents. Figure 20 displays the Flash contents before and after encryption, which confirms the match.

```
c0010a0400e903021000000000c0.
```

The success rate for jumping to Download Mode is the same as for jumping to `ets_fatal_exception_handler`: roughly 2%. Because an attacker only needs to succeed once, further optimizing this success rate is unnecessary.

5 Conclusion

Our work demonstrates that the ESP32 V3, even though it is specifically hardened against FI attacks, is still vulnerable. Using a single EM glitch, we were able to bypass the SoC’s most significant security features, *i.e.*, *Secure Boot V2*, *Flash Encryption*, the disabling of *Download Mode* by burning fuses, and the enabling of *Release Mode* by burning fuses. We have no reasons to believe that a skilled and resourceful attacker would be unable to perform this attack on a commercial product that incorporates an ESP32 V3 chip.

Moreover, we believe to have demonstrated an FI technique that is versatile enough to be applied to various architectures, which includes vendors other than Espressif. Our approach

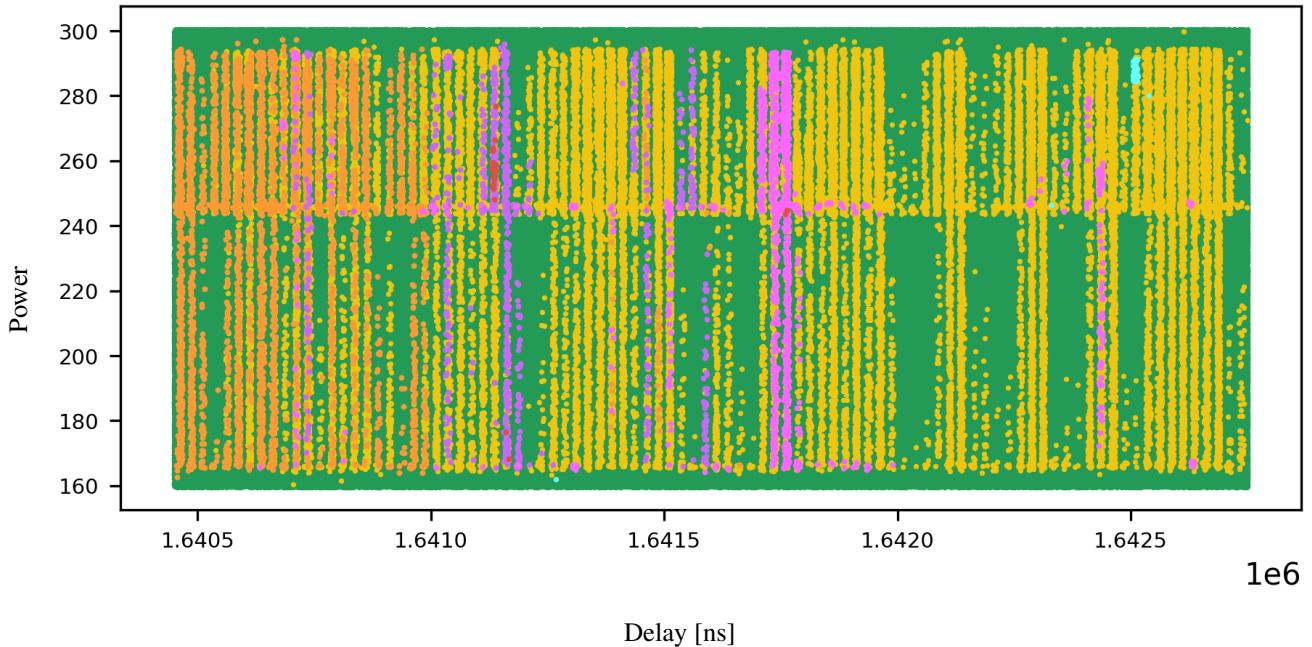


Figure 19: FI as an oscilloscope, revisited.

```

00000000 E9 03 02 10 3C 06 08 40 EE 00 00 00 00 00 03 00 ....<...@.....
00000010 00 FF FF 00 00 00 00 01 20 00 FF 3F C4 0C 00 00 .....?....
00000020 FF FF FF FF 28 50 04 00 FF AC 00 00 01 00 00 00 ....(P.....
00000030 00 F0 F5 3F 00 00 00 00 04 00 00 00 05 00 00 00 .....?.....
00000040 06 00 00 00 07 00 00 00 41 73 73 65 72 74 20 66 .....Assert f
00000050 61 69 6C 65 64 20 69 6E 20 25 73 2C 20 25 73 3A ailed in %s, %s:

```

(a) Unencrypted.

```

00000000 BB C3 FC 39 C1 52 A1 1B 05 D8 E9 FF A2 4E D3 64 ...9.R.....N.d
00000010 7C 55 95 FC DC 5C AA BB AC 81 38 A1 0F 99 62 42 |U...\.8...b
00000020 98 D1 9C 13 66 1C 49 D1 E4 C4 42 6F D9 76 24 55 ....f.I...Bo.v$U
00000030 DD 4A C4 ED FB 01 05 18 29 02 4A 7A F4 01 4E 52 .J.....)Jz..NR
00000040 C1 2C B9 02 77 6F DE 4B 72 24 1A DB 2D A9 1D 3E .,.wo.Kr$.-..>
00000050 39 E1 0D BB A3 6F B A B1 DA E5 02 A0 27 76 00 64 9....o.....'v.d

```

(b) Encrypted.

Figure 20: Hexadecimal dump of a Flash image (a) before encryption and (b) after encryption.

marks the first successful demonstration of loading an arbitrary value into the PC register of a CPU without being able to directly control the value. Modifying ciphertext in order to load the result of a computation on the plaintext into the PC using a single glitch represents a previously unseen level of complexity for such attacks.

The vulnerabilities we exploited on the ESP32 V3 require a new hardware revision as they cannot be mitigated by a software patch. If such a revision would be made, the attack could be mitigated by simply not printing the checksum values on the serial interface. However, given that variations on our FI technique are not limited to the checksum operation, the printing of any information on the serial interface

should be carefully assessed. Either way, Espressif indicated that the attack presented in this article does not apply to the ESP32-S2, ESP32-C3, ESP32-S3, and future chips. We did not investigate what is different for those chips that would yield our attack inapplicable.

Acknowledgments

We thank Espressif for establishing a smooth vulnerability-disclosure process.

References

- [1] Karim M. Abdellatif, Olivier Hériveaux, and Adrian Thillard. Unlimited results: Breaking firmware encryption of ESP32-V3. *Cryptology ePrint Archive*, Paper 2023/090, January 2023. <https://eprint.iacr.org/2023/090>.
- [2] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer’s apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.
- [3] Kévin Courdesses (Courk). Fault injection attacks against the ESP32-C3 and ESP32-C6, January 2024. <https://courk.cc/esp32-c3-c6-fault-injection#esp32-c3-c6-fault-injection> [Accessed: Feb 9, 2024].

- [4] derrek, nedwill, and naehwert. Nintendo hacking 2016 – game over, December 2016. 33rd Chaos Communication Congress (33C3), https://media.ccc.de/v/33c3-8344-nintendo_hacking_2016 [Accessed: Mar 11, 2024].
- [5] Espressif. ESP-IDF programming guide – flash encryption. <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html> [Accessed: Mar 12, 2024].
- [6] Espressif. ESP-IDF programming guide – secure boot v2. <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/secure-boot-v2.html> [Accessed: Mar 12, 2024].
- [7] Espressif. ESP32 chip revision v3.0 – user guide. https://www.espressif.com/sites/default/files/documentation/esp32_chip_revision_v3_0_user_guide_en.pdf [Accessed: Mar 12, 2024].
- [8] Espressif. ESP32-DevKitC V4 getting started guide. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/hw-reference/esp32/get-started-devkitc.html> [Accessed: Mar 12, 2024].
- [9] Espressif. ESP32 series ROM ELF files. <https://github.com/espressif/esp-rom-elfs/releases> [Accessed: Mar 12, 2024].
- [10] Espressif. Firmware image format. <https://docs.espressif.com/projects/esptool/en/latest/esp32/advanced-topics/firmware-image-format.html> [Accessed: Mar 12, 2024].
- [11] Espressif. Serial protocol. <https://docs.espressif.com/projects/esptool/en/latest/esp32/advanced-topics/serial-protocol.html> [Accessed: Mar 12, 2024].
- [12] Espressif. Espressif security advisory concerning fault injection and secure boot (cve-2019-15894), 2019. https://www.espressif.com/en/news/Espressif_Security_Advisory_Concerning_Fault_Injection_and_Secure_Boot [Accessed: Mar 12, 2024].
- [13] Espressif. ESP32-WROOM-32E – ESP32-WROOM-32UE – Datasheet v1.6, 2023. https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf [Accessed: Mar 12, 2024].
- [14] Espressif. Security advisory concerning breaking the hardware AES core and firmware encryption of ESP32 chip revision v3.0. Technical report, 2023. https://www.espressif.com/sites/default/files/advisory_downloads/AR2022-003%20Security%20Advisory%20Concerning%20Breaking%20the%20Hardware%20AES%20Core%20and%20Firmware%20Encryption%20of%20ESP32%20Chip%20Revision%20v3.0%20-%20v2.0%20EN.pdf [Accessed: Mar 12, 2024].
- [15] Espressif. Security advisory concerning bypassing secure boot and flash encryption using CPA and FI attack on ESP32-C3 and ESP32-C6. Technical report, 2023. https://www.espressif.com/sites/default/files/advisory_downloads/AR2023-007%20Security%20Advisory%20Concerning%20Bypassing%20Secure%20Boot%20and%20Flash%20Encryption%20using%20CPA%20and%20FI%20attack%20on%20ESP32-C3%20and%20ESP32-C6%20EN.pdf [Accessed: Mar 12, 2024].
- [16] Espressif. Security advisory concerning bypassing secure boot and flash encryption using EMFI. Technical report, 2023. https://www.espressif.com/sites/default/files/advisory_downloads/AR2023-005%20Security%20Advisory%20Concerning%20Bypassing%20Secure%20Boot%20and%20Flash%20Encryption%20Using%20EMFI%20EN.pdf [Accessed: Sep 12, 2023].
- [17] Travis Goodspeed. GameBoy ROM tutorial, March 2023. <https://github.com/travisgoodspeed/gbrom-tutorial> [Accessed: Mar 12, 2024].
- [18] Travis Goodspeed. Thread on X by @travisgoodspeed, November 2023. <https://threadreaderapp.com/thread/1728420233050747287.html> [Accessed: Mar 12, 2024].
- [19] James Gratchoff. Proving the wild jungle jump. Technical report, University of Amsterdam, July 2015. https://www.os3.nl/_media/2014-2015/courses/rp2/p48_report.pdf.
- [20] Tim Hummel. Exploring effects of electromagnetic fault injection on a 32-bit high speed embedded device microprocessor. Master Thesis, University of Twente, July 2014.
- [21] Internet Engineering Task Force (IETF). RFC 8017 – PKCS #1: RSA cryptography specifications version 2.2. <https://datatracker.ietf.org/doc/html/rfc8017> [Accessed: Mar 12, 2024].
- [22] LimitedResults. Fatal fury on ESP32: Time to release hardware exploits. BlackHat Europe 2019, December 2019. <https://www.blackhat.com/eu-19/briefings/schedule/#fatal-fury-on-esp-time-to-release-hardware-exploits-17336>.

- [23] Nourdin Aït El Mehdi. Analyzing the resilience of modern smartphones against fault injection attacks. Master Thesis, Delft University of Technology, June 2019.
- [24] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4:145–156, 2014.
- [25] The On-Line Encyclopedia of Integer Sequences. A048651, July 2007. <https://oeis.org/A048651> [Accessed: Mar 12, 2024].
- [26] Colin O’Flynn. Low-cost body biasing injection (BBI) attacks on WLCSP devices. In Pierre-Yvan Liardet and Nele Mentens, editors, *19th Conference on Smart Card Research and Advanced Applications (CARDIS 2020)*, volume 12609 of *Lecture Notes in Computer Science*, pages 166–180. Springer, November 2020. <https://eprint.iacr.org/2020/1228.pdf>.
- [27] Raelize. Breaking SoC security by glitching OTP data transfers, 2020. <https://hardwear.io/usa-2022/speakers/cristofaro-mune.php>.
- [28] Raelize. Espressif ESP32: Bypassing encrypted secure boot (CVE-2020-13629), September 2020. <https://raelize.com/blog/espressif-esp32-bypassing-encrypted-secure-boot-cve-2020-13629/>.
- [29] Riscure. EM-FI transient probe. <https://www.riscure.com/products/em-fi-transient-probe/> [Accessed: Mar 12, 2024].
- [30] Albert Spruyt, Alyssa Milburn, and Łukasz Chmielewski. Fault injection as an oscilloscope: Fault correlation analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):192–216, Dec. 2020.
- [31] Michael Steil. 17 mistakes Microsoft made in the Xbox security system. In *22nd Chaos Communication Congress*, 2005.
- [32] Tensilica, Inc. Xtensa instruction set architecture (ISA) – Reference manual, April 2010. <https://0x04.net/~mwk/doc/xtensa.pdf> [Accessed: Mar 12, 2024].
- [33] Niek Timmers and Cristofaro Mune. Using fault injection to turn data transfers into arbitrary execution, 2019. <https://powerofcommunity.net/poc2019/Niek.pdf>.
- [34] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using fault injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2016)*, pages 25–35. IEEE Computer Society, August 2016.
- [35] Marc Witteman and Martijn Oostdijk. Secure application programming in the presence of side channel attacks. In *RSA conference*, volume 2008, 2008.



Basilisk: Remote Code Execution by Laser Excitation of P–N Junctions Without Insider Assistance

Joe Loughry
Netoir.com
joe@netoir.com

Kasper Rasmussen
University of Oxford
kasper.rasmussen@cs.ox.ac.uk

Abstract

Inadvertent photosensitivity of P–N junctions has been known for a long time, but most of the attacks that have been demonstrated are covert channels, requiring an adversarial presence on the device. We show not only how it is possible for an external attacker to bias a P–N junction with a low power laser, without any kind of insider assistance, but also how this kind of attack can be used to perform logic level attacks on the target device and thus interfere with the device’s operation. The technique requires precision but is feasible in practice with off the shelf hardware, as long as the attacker has a line of sight to the target. It can result in attacks that include crashing a computer, change memory contents, alter the instruction stream of a running program, alter messages on a shared communication bus, insert new messages, or prevent communication. Most of these attacks have never been demonstrated before without insider assistance. We demonstrate that under the right circumstances the attack can lead to arbitrary code execution on the target device. We show a working proof of concept including remote code execution, and quantitative measurements leading to testable predictions. Mitigation of this vulnerability is challenging and countermeasures will in most cases require hardware changes.

1 Introduction

Semiconductors are the substrate on which most modern electronics are built. Semiconductors can be “doped”, *i.e.*, contaminated, with elements that makes it either positively charged or negatively charged, with the interface between positively and negatively charged sections called a P–N junction. These junctions are the basic building blocks of all transistors and diodes (and by extension logic gates, IC and LEDs) and are thus present in basically every device.

In some cases these components are exposed to the outside world, either by design in the case of LEDs, or accidentally in case any part of a circuit board can be seen through vent holes or other openings in a device. Exposed P–N junctions that

are electrically connected to a shared communication bus are vulnerable to being optically “pumped” by a modulated laser beam. The effect is to reverse the LED, turning it into a current source, or to bridge the diode, turning it into a conductor, thereby affecting the circuit that the component is connected to.

The effect depends on the type of component and the surrounding electric field. In photovoltaic mode, an LED in forward bias will be reversed if sufficient optical power is pumped into it at the right wavelength; this causes the LED to generate a photocurrent that runs backwards through the connected circuits of the computer, driving the connected circuit *high*; in photoconductive mode, *e.g.*, an electrostatic discharge (ESD) protection diode in reverse bias, pumped by an infrared laser, conducts current in the opposite direction, grounding the bus and driving it *low*. The effect is transient, and leaves no evidence behind.

We explored the feasibility of conducting such attacks in practice and the extent to which this effect can be used by an attacker to affect a victim system. We found it is in fact possible to affect P–N junctions using an external light source, in practical conditions. Further more it is possible to achieve a range of different effects depending on what the exposed LED or diode is connected to.

We conducted a number of experiments to determine the parameters for a successful attack. These include the angle at which the laser hits the diode, the power level needed and the modulation of the laser. All experiments are done with low cost devices that are easily available to anyone, *e.g.*, a laser module from a Blu-ray player.

We present two working proofs of concept. First an attack on a live I²C bus running between commercially available devices, and second, an attack on the CPU–memory bus on a pedagogically minimized CPU. The first shows that we can inject or alter messages on the I²C bus that will be accepted as legitimate by other devices on the bus. The second gives the attacker arbitrary execution privilege (with some interesting constraints) on the CPU.

Mitigation of the vulnerability is not straightforward. In

most cases countermeasures will require hardware changes and we discuss when and how this can be done in practice.

2 Photosensitivity of P–N junctions

Semiconductor P–N junctions are photosensitive because photons generate electron–hole pairs when they hit a semiconductor under the right conditions, specifically the depletion layer that forms in the presence of an electric field between the p-type and n-type doped regions. Electron–hole pairs form when a photon with the right amount of energy is absorbed by a semiconductor atom [84, p. 80]. In zero-bias mode, the photocurrent generated is proportional to irradiance on the P–N junction [86, Chapter 6, p. 238].

The result of this conversion depends on the electric field, *i.e.*, how the junction is biased. If the P–N junction, say in an LED, is zero biased and illuminated by a suitable wavelength, it goes into what is called photovoltaic mode. Electrons are swept towards the anode, and holes are swept towards the cathode. This makes the cathode positive with respect to the anode, sending an electric current through any circuit attached to the LED but in the opposite direction from the way that usually makes the LED light up. The same thing happens if the LED is forward biased, *i.e.*, lit.

If the P–N junction is reverse biased, as in an ESD protection diode on a shared bus, it goes instead into photoconductive mode when illuminated by the laser. Here, electrons are swept by the electric field towards the cathode, immediately recombining with holes there, lowering the resistance to electric current and making the P–N junction into a conductor. Because the depletion layer is widened by the reverse bias voltage, making a larger volume where electron–hole pairs may be created by absorption of photons, photoconductive mode is more sensitive than photovoltaic mode.

The semiconductor used for the “P” and “N” parts of the junction influences the best laser frequency to use to excite it. The same 980 nm infrared photons that work well on silicon ESD protection diodes are too low in energy (too long a wavelength) to work effectively on, say, gallium arsenide (GaAs) doped LEDs. However a typical 520 nm green solid-state laser works well in that case.

3 Related Work

Unwanted photosensitivity of electronic components has been a known issue for a long time. In 1952, the first production IBM 701 mainframe computer failed at its unveiling when the flashbulbs of news photographers disrupted the Williams tube memory of the computer [10, 17, 35, 75]. Semiconductor memory chips, if not protected from visible or ultraviolet light, are similarly sensitive [51, 82, 89].

In 2015 a flip-chip voltage regulator on the Raspberry Pi 2 single-board computer (SBC) was found—once again when it

was being photographed for a press release before the product introduction—to crash the computer whenever exposed to xenon strobe camera flashes [28, 87]. Raspberry Pi 3 was later found to suffer a similar problem with a much larger chip-scale package integrated circuit (IC) on the back side, a Broadcom BCM43438 Wi-Fi and Bluetooth chipset (U19) this time [70]. In both cases, the root cause was found to be failure to specify an opaque backside laminate (BSL) on the chips [61].

Glass-encapsulated small signal diodes have long been known to pick up noise from overhead fluorescent lighting fixtures [13]. Photosensitive LEDs have been used before for communication [23, 81] or light detection [14, 52–55] or power delivery [1, 26, 49, 62, 63].

Other researchers have proposed ways to make a covert channel out of this effect. All such efforts require an insider with the ability to run a listening process on the target device [41]. This is not an unreasonable assumption, as STUXNET surely proved [19, 40], but what separates those efforts from this paper is that Basilisk does not require participation or cooperation by an insider.¹

Most of the remaining published research related to compromising optical emanations (optical TEMPEST) concerns information flow out of the computer system; only a few papers in the literature address information flow inwards [24, 39, 59, 74]. Perhaps closer are laser injection attacks on optical fiber components of a quantum key distribution system, but in that case the real target was an optical receiver actively listening for a signal [30, 73]. A good survey of signal injection attack vectors is [31].

Sugawara *et al.* demonstrated coupling between a relatively high power laser and microelectromechanical systems (MEMS), *e.g.*, microphones and accelerometers, a clear parallel to our work because it similarly depends on the physical principle of energy transfer from the laser to the target system in order to effect coupling to a subsystem that was not listening for optical signals directly [83]. Rampazzi *et al.* (2020) found evidence of both photomechanical and photovoltaic effects at work [71, §4.3] and this work was further extended [22] by Cyr *et al.* (2024) but MEMS attacks are always targeting sensors, with the intention of falsifying measurements; our work is targeting any device that has an exposed P–N junction and can give direct access to the internal state of the machine.

Basilisk is not precisely the same thing as optical fault injection; that work is done on decapsulated chips, not on conventionally exposed P–N junctions [3, 5, 25, 33, 36, 37, 64–67, 78–80]. Laser fault injection is a technique primarily used in chip design and manufacturing for reliability testing. Focused radio frequency, x-ray, subatomic particle, or

¹Randal (2023) makes the useful distinction amongst (1) covert channels, where both sender and receiver are malicious, (2) side channels, where only the receiver of information is malicious, and (3) fault injection, where only the sender is malicious [72, §2.1].

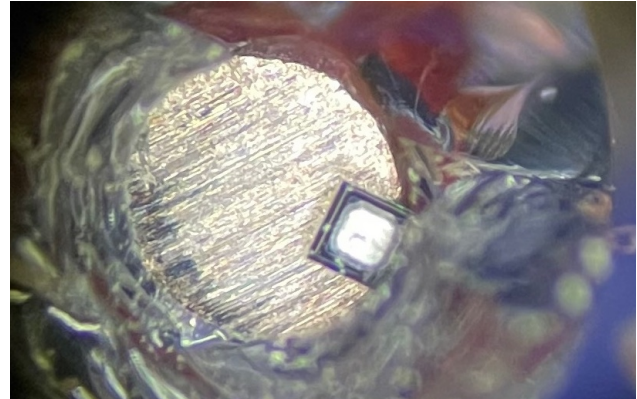
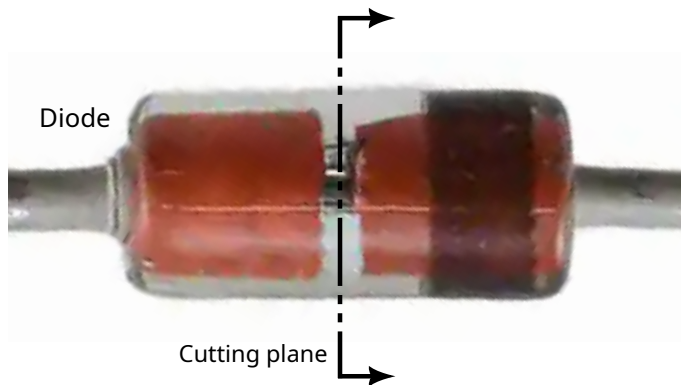


Figure 1: Example of an ESD protection diode. The actual silicon diode is the small square shape visible in the gap between the electrodes. It is often located off-center in the gap, decreasing the angle from which it can be hit.

laser radiation can be used to induce permanent or temporary changes in electronic circuit elements, leading to error states resulting in failures of the system [4, 90]. Light-induced voltage (or current) by means of a scanning laser is a test and characterization method used by semiconductor manufacturers to optimize process changes. It may be used in margin testing to assess reliability. Environmental conditions such as temperature, clock speed, or power supply voltage may be varied to induce faults; the latter is the basis for glitching attacks [29, 32].

Single event upset (SEU) testing (*e.g.*, cosmic rays) is important for space vehicles and devices designed to operate in the vicinity of a nuclear reactor. Electromagnetic interference (EMI) testing for electromagnetic compatibility (EMC) is a type of fault injection, and Basilisk may be considered intentional electromagnetic interference (IEMI).

Laser or photoflash fault injection at the basic component level is usually done on a decapped chip or bare die, under a microscope, allowing for precise placement and small spot size. As an attack vector, optical fault injection is normally used for key extraction—from TV set top boxes, electricity meters, smart cards, and payment terminals, in the course of hardware security module (HSM) testing, KG-type military link encryptors, or trusted platform module (TPM) chips [38, 48]. Bar-EI *et al.* (2004) contains a comprehensive list of active protections in hardware—duplication of circuits, multiple redundancy with or without time shifting, and error-correction codes—and software such as are routinely used in spacecraft to guard against SEU events [5].

In contrast to all these localized events, we aim to show that ranged attacks are practicable on networking or programmable logic controller (PLC) equipment used in factory automation and control. Instead of extracting information, such as cryptographic keys, our capability is to take over control of the system.

4 Basilisk

To demonstrate the various effects of P–N junction excitation we design an injection attack framework called Basilisk. This attack framework allows an external attacker to compromise an air-gapped system, without the need for internal collaboration.

Fundamentally the low-level effect of a Basilisk attack is to pull an internal wire in the target device high or low, depending on the type of diode that it is connected to and the surrounding circuit.

This can be used to disrupt a number of higher level tasks, including chip-to-chip communication, assert error- or interrupt conditions or to send or alter commands on a communication bus. For example, as we demonstrate in detail in Section 7, it can be used to corrupt or modify instructions fetched from memory, change the data sent to a display or communication module, or simply crash a device and make it unresponsive.

For Basilisk to be effective an attacker must be able to target a P–N junction directly. In this explanation we will use ESD protection diodes as examples although the same applies to LEDs or any other P–N junction encased in a transparent material. Figure 1 shows a magnified view of a diode cut across the center to expose the silicon chip. Observe that the silicon chip is sandwiched in a gap between cylindrical metal electrodes, and the chip is often off-center in the gap, making it easier or harder to target with a laser beam.

4.1 System model

The system model for Basilisk is depicted in Figure 2. It is quite broadly applicable and only has a few requirements.

First of all, in order to conduct a Basilisk attack the target system must have an exposed P–N junction. This can come in many forms, *e.g.*, in the form of an indicator LED designed to be visible from the outside; or in the form of an ESD

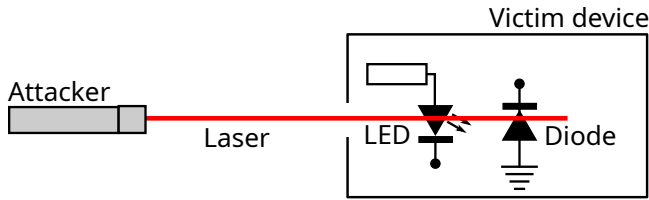


Figure 2: System and adversary model. The victim system has an exposed diode (or LED) that can be targeted by an adversary with a laser beam.

protection component (often a diode) mounted somewhere on a printed circuit board visible through an opening in the case.

The exposed diode must be connected to a useful target. The adversary is able to pull the wire connected to the diode either high or low depending on the type of diode and the surrounding circuit, but typically not both. The diode must be connected to a circuit that if pulled high (or low) will have an effect on the rest of the system. This is very often the case in practical systems and we will show a number of practical examples of such systems throughout the rest of the paper.

Finally, for some of the attacks to be effective the target device must be turned on and running code. This is not always strictly required, for example one can imagine an example where a soft power switch can be activated with a Basilisk attack or a reset line pulled low to reset a halted CPU, but in most practical cases we will assume that the target system is running.

The vulnerability exists whenever exposed PN junctions (like LEDs or ESD protection components) are connected directly or indirectly to electronic circuits carrying sensitive information.

4.2 Adversary model

The adversary must have line of sight to the target diode. This is not trivial in practice, but neither is it very difficult. In Section 5 we define a metric called active area that measures how precisely an attacker must aim to be effective. We show that it is indeed feasible to do even without specialist equipment.

For certain attacks it is necessary to be able to modulate the laser, *i.e.*, turn it off an on, say, in order to create valid packets. We assume the attacker is able to do this as fast as is needed for the attack. This is a fairly easy requirement in practice, as the modulation need not be any faster than the communication protocol under attack, *i.e.*, under 5 MHz for I²C.

Although not always needed, we grant the adversary full knowledge of the timing of any messages that are transmitted across a targeted bus. This is somewhat of an over approximation, but it models the case where this information is available through other side channels. If this information is not available, attacks that require specific timing become probabilistic.

Basilisk works by pulling the wire connected to the diode

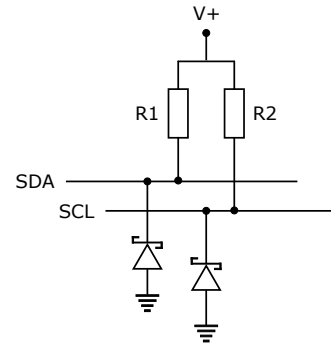


Figure 3: ESD protection diodes on the I²C bus—when illuminated—are able to pull the bus down against the effect of pull-up resistors R1 and R2.

either low or high, but not both. For this reason the attacker can only change a binary 1 to 0, (or 0 to 1) but not both ways. This is not as much of a limitation as it first appears, because shared communication buses have pull-up (or pull down) resistors making the default bus state high (or low) as in Figure 3. The attacker can send arbitrary messages in that case by, say, pulling the bus low when needed and letting it go back up to high by just turning off the laser. Nevertheless it is a limitation that can come into play in some circumstances. We describe one example of this in Section 7.1.

Note that this technique does not permit the attacker to receive information from the circuit under attack, only send messages. If bidirectional communication is needed, another side channel must be used to read information. Such side channels are in fact often available, *e.g.*, when attacking a display as we demonstrate in Section 7.2, but the attacks we describe do not need to read from the device.

5 Diode Attack Surface

In this section we demonstrate the specific conditions under which it is possible for an external attacker to use a laser to execute a Basilisk attack. We introduce the measurement setup and then use it to drive ESD protection diodes or LEDs into photoconductive- or photovoltaic mode respectively, thus controlling the signal level of the connected circuits.

5.1 CMOS Logic Circuits

Before we continue we need to clarify the target voltage for our experiments. In CMOS, 3.3 V circuits signals are supposed to be either above $V_{IH} = 2.0$ V to indicate a logic high condition or below $V_{IL} = 0.8$ V to indicate logic low. The gap between V_{IL} and V_{IH} is not to be used to avoid ambiguity and to provide a buffer—called the noise margin—against small fluctuations in the electrical signal changing the logic state.

However CMOS logic is a binary system and the “undefined” state cannot actually be represented in hardware, so

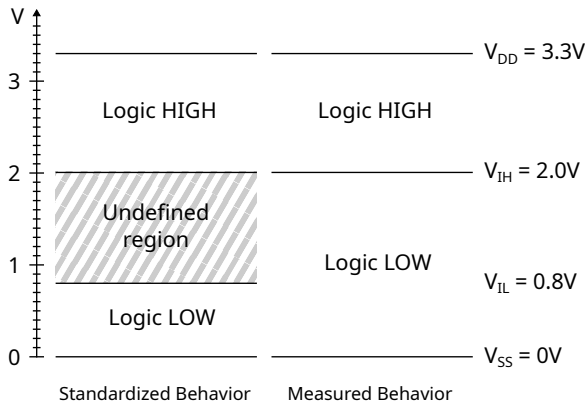


Figure 4: CMOS logic levels for 3.3 V circuits. Signals above 2 V are logic high, and below 0.8 V are logic low. In all our experiments we found that devices will default to a logic low condition in the undefined region, so although it ought never to be used, a signal below 2 V is sufficient for an attack.

in practice; in our experiments, anything below 2 V is interpreted as a logic low condition.² This difference is illustrated in Figure 4. This slightly higher value for a logic low condition is helpful for our attack since the lower the attacker wants to drive the signal, the more power is needed. Given the observed behavior we can use 2 V as the threshold for a successful attack.

5.2 Attack Measurements

To make sure our measurements and results are applicable to a real world system, we make all our measurements on an experimental setup consisting of two devices (bus controller and target) communicating over an I²C-bus. The bus has ESD protection diodes and external pull-up resistors to allow us to experiment with different values. We can change the resistor values and bus voltage independently; this mimics the I²C specification, which allows a wide variety of values to be used [60, 68]. A schematic of the test setup can be seen in Figure 5 and a photo in Figure 6.

The measurement setup consists of a pair of linear actuators at 90° to each other to allow for a systematic raster-scan of the diode under attack. The actuators are driven by stepper motors which enable precise repeatable measurements to be taken. Each raster scan of the diode moves the laser beam across a 1 mm² area of one of the ESD protection diodes on the I²C bus, stopping every 20 μm to measure the voltage on the bus.

Basilisk attacks are ideally suited for a shared communication bus because such buses use open collector (or open drain) drivers. A device wishing to transmit will drive the bus low to send a binary 0 or simply release the bus and allow the

²Our determination of this value is supported by Lancaster (1974, 1977) [42–44].

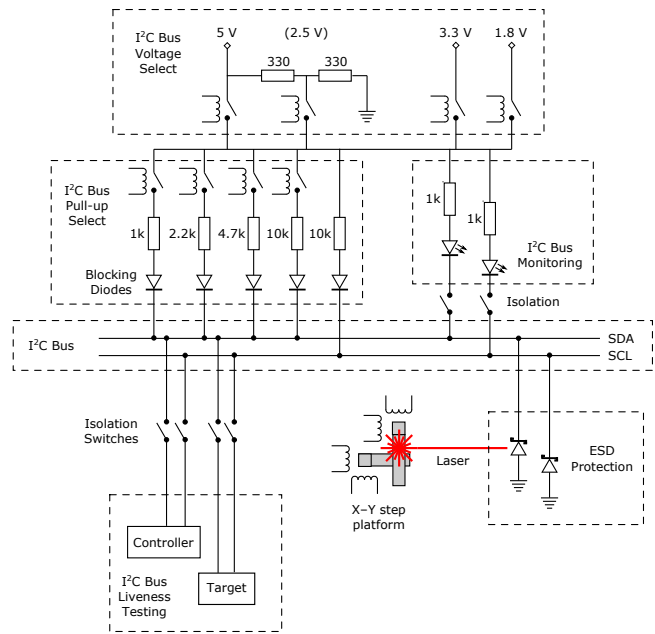


Figure 5: Schematic of the experimental apparatus.

pull-up resistor to return the bus to its logic high condition (binary 1) between zero bits or between transmissions.

Typical values for I²C bus pull-up resistors are 2.2 kΩ or 4.7 kΩ; lower values pull the bus up more strongly, allowing faster communication; conversely, a higher value resistor like 10 kΩ is a weaker pull-up.

The ESD protection diodes tested are a common type of glass-encapsulated DO-35 size small signal diode, a type 1N34A equivalent silicon Schottky diode chosen for its fast recovery speed. The diode is connected in reverse bias with its anode at ground potential. Reverse bias makes the diode non-conductive under normal circumstances, so it doesn't affect the operation of the bus. If the voltage on the bus ever exceeds the reverse breakdown voltage of the diode, *e.g.*, during a power surge, the diode becomes conductive and shunts the power surge to ground, protecting the bus and the connected devices [47, 85].

We test several different lasers from near infrared (IR) devices at 780 nm, to a longer wavelength of 808 nm and 980 nm in order to identify the type best suited for a particular diode type. The lasers all have a fixed power (intensity) rating between 3–5 mW.

We use our measurement setup to trace a raster pattern with the laser, back and forth over the diode, to identify the best place to direct the laser during an attack. This is illustrated in Figure 7. The lasers are focused to the smallest achievable spot size at a working distance of 32–35 mm. This is not critical for the attack to work but it gives our measurements a higher resolution, allowing us to map out diode vulnerabilities in more detail. Note that even when focused, the beam from a semiconductor laser is slightly elliptical, so we repeated

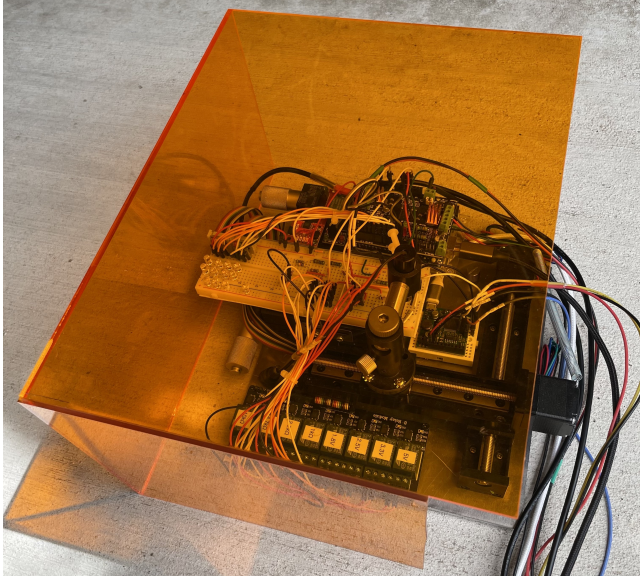


Figure 6: Experimental apparatus. Stepper motor linear actuators raster scan the laser, its elliptical beam axis marked, across the glass envelope of an ESD protection diode, obscured here by the laser mount. The color of the radiation shielding is wavelength-dependent, here for wavelengths shorter than 550 nm.

all the experiments with the long axis of the elliptical beam oriented parallel, perpendicular, and diagonally to the long axis of the gap between the electrodes in the ESD protection diode under test.

One run of the experiment consists of setting the bus voltage and pull-up resistor values, then scanning the laser spot over the glass body of the diode as shown in Figure 7. Voltage measurements are taken at fifty evenly spaced points along each scan line, for a total of 2500 measurements per run. Each run is repeated sixteen times for different combinations of bus voltage and pull-up resistor values, then the laser is rotated 45 degrees to shift the elliptical beam axis. We have experimented extensively with different wavelengths as well, however for most diode types we have one laser that is clearly the most efficient, so we only present the data for that set of experiments. See Section 8.2 for more details on laser wavelengths.

Voltage measurements are made with a 10-bit analogue-to-digital converter (ADC) channel 0 of an Arduino Uno single-board computer, against a 5 V reference. The ADC was allowed to settle for 500 ms every time the bus voltage was changed, and several voltage measurements were taken at every point, then averaged. The resolution of the ADC is 4.88 mV. All power supply voltages were verified with a Fluke 107 multimeter to be within 0.1 V of spec before the start of each run.

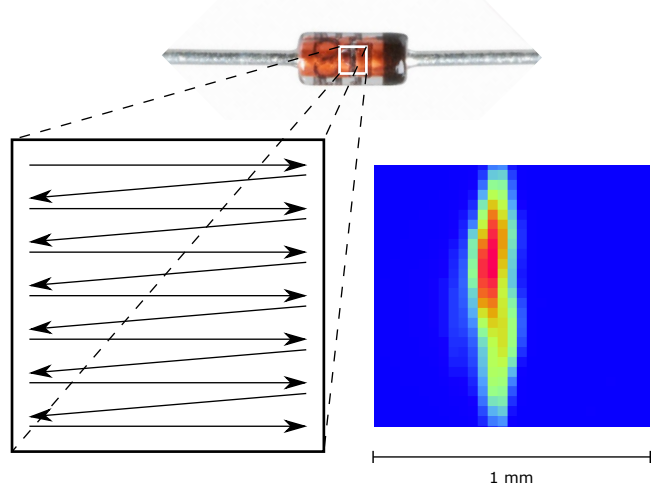


Figure 7: The raster pattern used to scan the diodes and LEDs to obtain repeatable results. Each run consists of 2,500 individual measurements.

6 Experimental Results

After a comprehensive series of tests on both ESD diodes and LEDs we have detailed results that show both options as a viable entry point for a Basilisk attack. In the following we present our findings for the two diode types.

In both cases the results of the experiments are a series of voltage measurements across the measurement area. This area is 1 mm^2 for the ESD diodes and 25 mm^2 for the larger LEDs.

6.1 ESD Diodes

A representative result is shown in Figure 8. The size and shape of the gap between the electrodes can be clearly seen. There is some indication of the size and location of the silicon chip. The black isovolt curve indicates the $V_{IH} = 2.0 \text{ V}$ level where the laser has forced the bus voltage below the logic threshold. Any hit from the laser inside the 2 V contour line will be seen as a logic low signal on the I²C bus. We call this region the *active area* of the diode and it serves as a good metric for how easy it is to execute the attack for a specific set of parameters. If the active area is small for a chosen set of parameters, it means that those parameters make it more difficult to drive the signal low. Conversely if the area is large it means that it is comparatively easier.

To demonstrate how an attacker can best influence an exposed diode, and what circuit types are more vulnerable, we look at each of the parameters individually. These include beam axis rotation, bus voltage, and pull-up strength.

Beam rotation. Figure 9 depicts example results for three different beam rotations. The active areas (region inside the isovolt curve) look slightly different for the three rotations but

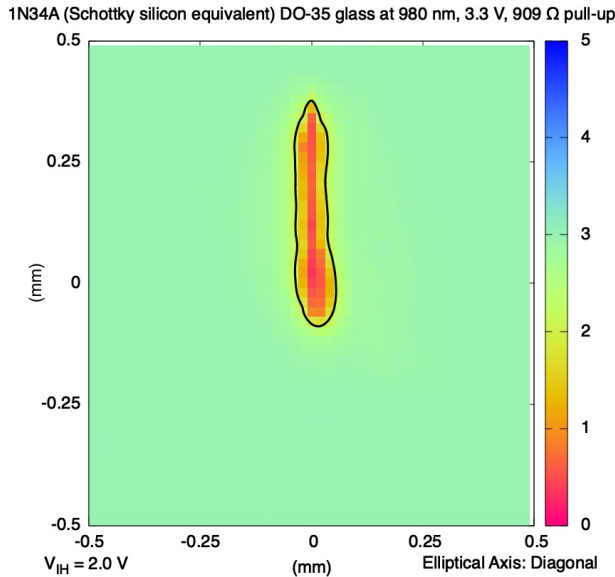


Figure 8: Voltage measurements from a 980 nm laser. The size and shape of the gap between the electrodes can clearly be seen. The black isovolt curve delineates the *active region* where the laser was able to force the bus voltage below the logic threshold V_{IH} , thereby imposing a binary zero on the bus. Simply releasing the bus, by turning off the laser, is sufficient to send a binary one.

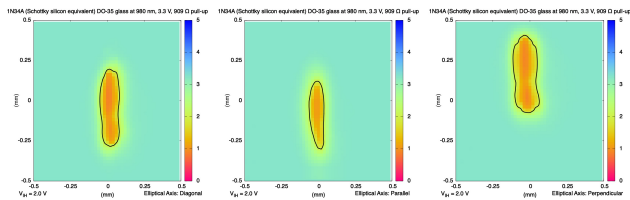


Figure 9: Effect of beam elliptical axis rotation: diagonal 45° (left), parallel 0° (middle), and perpendicular 90° (right).

it seems hard to draw a firm conclusion. To help with that we plot the average size of each of the active areas over 10 runs in Figure 10. Here it can be seen that we have a larger active area if the beam is perpendicular, *i.e.*, rotated 90° . While it is tempting to conclude that perpendicular beams are better, it is likely an artifact of the geometry of a specific diode, as the internal placement of the silicon die is believed to be random. However what we can learn from this is that rotation matters, and in a practical scenario rotating the beam might yield a bit of extra efficiency if the diode can only be targeted with an off center beam, or if the power of the laser is limited.

Bus voltage and pull-up resistor strength. Figure 11 shows the effect of bus voltage and pull-up resistor strength. Voltage varies from top to bottom and pull-up resistor strength varies from left to right. Observe how the active area becomes larger

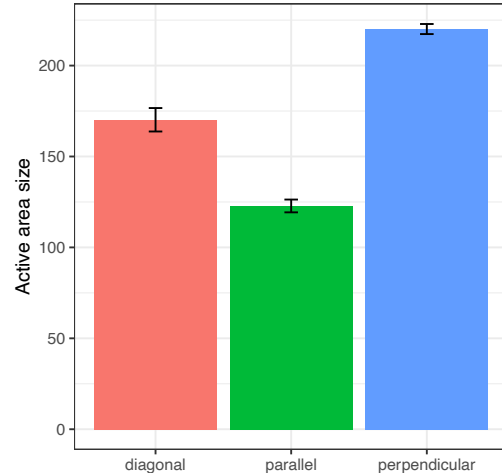


Figure 10: Average size of the active area (over 10 runs) for each of the three elliptical axis beam rotations. The error bars indicate standard deviation.

at lower bus voltages and with weaker pull-up resistors. This makes intuitive sense since with a lower bus voltage there is less of a voltage difference between a logic high and logic low signal, and thus it takes less to pull the signal down to a low state. Similarly for weaker pull-up resistors there is less current flow available to counteract the attempt to pull the signal level down to low.

For example, while at 5 V with a strong pull-up, attacking the I²C bus is difficult, it is easy to attack a bus at 3.3 V with any reasonable pull-up resistor value.

The isovolt contour lines support the prediction that the attack will get easier in future as system voltages fall from 5 V TTL through 3.3 V CMOS to 2.5 V and 1.8 V LVC MOS [2, introduction]. In general, we can now predict whether a given combination of component type, bus voltage, pull-up resistor value, and laser wavelength is reversible.

6.2 Light Emitting Diodes (LEDs)

Doing the same set of experiments again for LEDs serve two purposes. First we want to show that Basilisk attacks are possible on LEDs as well, which is important since LEDs are often more available as targets. Second we want to investigate how different colored LEDs behave. The behavior will be different since the semiconductor is doped with different elements in order to produce different colored light. Furthermore, some LEDs are encased in a colored resin, which could influence the effectiveness of our attack laser.

An LED is a significantly bigger target than an ESD diode. This makes aiming the beam and beam rotation less important so we will not reproduce that part of the experiment here. We instead focus on the most important aspects that we found

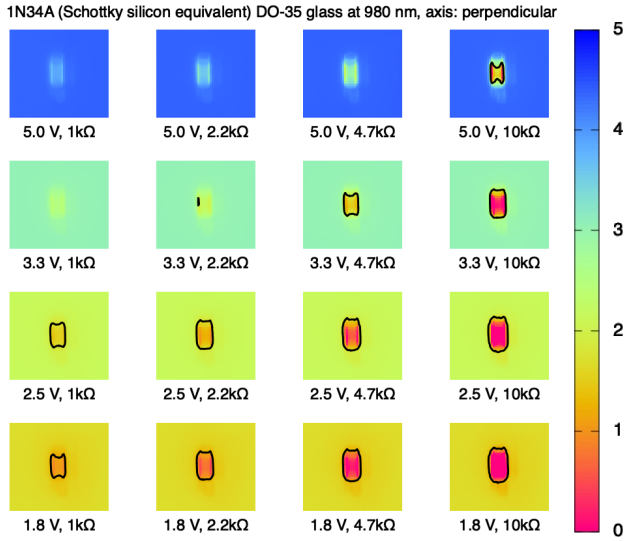


Figure 11: Voltage measurements from varying I^2C bus voltage and pull-up resistor strength as independent variables. All attacks are done with an 980 nm laser. The black isovolt lines are 2.0 V for 5 and 3.3 V logic, 1.7 V for 2.5 V logic, and 1.17 V for 1.8 V logic.

will determine the resulting bus voltage, namely the color of the LED that is being attacked, the bus voltage of the victim system, and the pull-up resistor strength.

LED color. We tested the Basilisk attacks on six LEDs (four different colors) with the same 405 nm (violet/blue) laser to get a measure for how color impacts the efficiency of the attack. The results can be seen in Figure 12.

We see that pink LEDs exhibit the weakest response which is likely due to a phosphor layer absorbing a lot of the shorter wavelength before it gets to the chip. The blue and green LEDs both respond fine to the attack, and it is clear that those LEDs are vulnerable enough to be used in practice. The most vulnerable LED color is white. We have not attempted to uncover the specific physical reason for these differences, just note that with the exception of the pink LED, all had fairly large active regions, making them easy to hit.

Bus voltage and pull-up resistor strength. Another thing to note from Figure 12 is that the voltage in the active region, *i.e.*, inside the isovolt curve, is not as low as it is for ESD protection diodes. While LEDs are large and easy to hit, they do not drive the voltage as far down with the same laser power, *i.e.*, they are less efficient. While an ESD protection diode easily drives the voltage down to zero if hit correctly, LEDs usually bottom out at 1–1.5 V. As mentioned in Section 4, this signal level is considered undefined for CMOS circuits but in practice it is enough to trigger a logic low condition, which means it is good enough for our purpose.

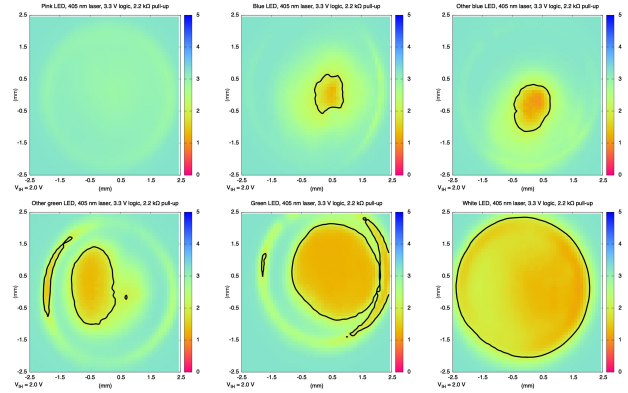


Figure 12: Voltage measurements of different colored LEDs. pink (top left), blue (top middle, top right), green (bottom left, bottom middle), and white (bottom right). All attacks are done with a 405 nm laser.

7 Case Studies

In the previous section we demonstrated that an adversary can change the logic level of an I^2C bus with a laser. But that falls short of demonstrating remote code execution on a live CPU, which requires the attacker to have precise control of timing.

We next show that the attack vector can be used to perform a meaningful attack. We do this through two case studies. In the first one we attack a small computer with a simple instruction set to demonstrate how we can effect arbitrary code execution. The second case study demonstrates that Basilisk attacks can be performed against off the shelf hardware.

The two case studies also allow us to demonstrate both types of Basilisk attack: photovoltaic and photoconductive. In the first case study the exposed diode is an LED connected to the memory bus so we can use photovoltaic attacks to drive the bus wires to their logic high state. In the second case we attack the ESD protection diodes on an I^2C bus so we use a photoconductive attack to drive the bus low.

7.1 Changing the running instruction stream

To demonstrate the practicality of remote code execution, at the same time keeping the complexity low so all details are visible, we constructed a minimal 4-bit computer called M5. It has status lights on the memory bus—a feature sadly lacking in most computers designed since the 1970s—and an instruction set architecture (ISA) small enough to make Figure 17 feasible. We wanted to be able to show the reachability analysis between instructions without the overwhelming complexity of a modern ISA like ARMv7 or RISC-V. The point is to show that because of an interesting constraint on the attacker—who may only be able to change a binary 0 to 1—but not the other way around—certain opcodes are reachable from certain other opcodes, but not always the one you want.

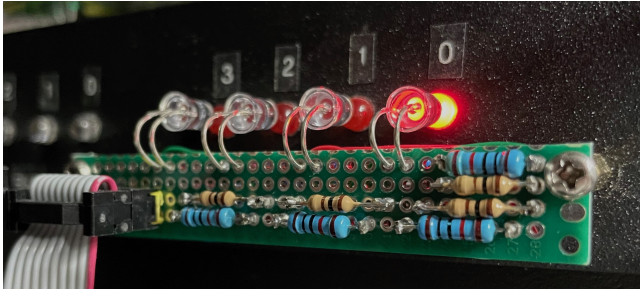


Figure 13: The attacker watching the accumulator to get a phase lock on the CPU at a cycle time of 250 μ s.

M5, shown in Figures 15 (a) and 16, is a minimalist CPU intended not so much to show the practicality of the attack against real hardware but—quite the opposite—to highlight certain unique difficulties of the attack, beyond obvious ones like aiming and focusing.

The accumulator register is visible on the front panel (Figure 13)—visibility is key to establishing a phase lock on the CPU. It has a simple instruction set (Table 1) to make feasible the reachability analysis in Figure 17, and will halt if it decodes an illegal opcode. Cycle time is 250 μ s but this was limited by the speed of the laser drivers, not the FPGA.

Both CPU and memory are implemented in the FPGA but the bus between them was routed externally to be accessible for probing outside the FPGA’s internal interconnection fabric, as shown in the schematic of Figure 15 (a). The FPGA used is a Lattice Semiconductor iCE40-HX8K breakout board, configured in Verilog with the open source Project iCEstorm. All experiments in this section are done with a 405 nm, 5 mW laser.³

It is important to note this is not an FPGA vulnerability; the attack happens outside the FPGA’s internal interconnection fabric, on external I/O pins, which are all standard CMOS.

M5 runs the microcode for each instruction on a sixteen-step cycle. Figure 14 shows a typical instruction, opcode mnemonic STA, which stores the value currently in the accumulator register to a specified memory location.

The attacker needs to know a lot about the CPU and the program that is currently running to perform the attack successfully. Firstly, the attacker needs to establish a phase lock on the internal state of the CPU, and from it to accurately measure the cycle time, because the entire attack is predicated on cycle counting.

Here, phase locking is accomplished by watching the accumulator display for changes, because the display always changes at a known microcode cycle. From the direction and magnitude of the change, the attacker can deduce what

³With the laser focused at infinity, any portion of the beam lying within a 5 mm diameter acceptance cone at the target will automatically be captured and index matched to the P–N junction, because optical systems—in this case, the lens of the LED—are time-reversible.

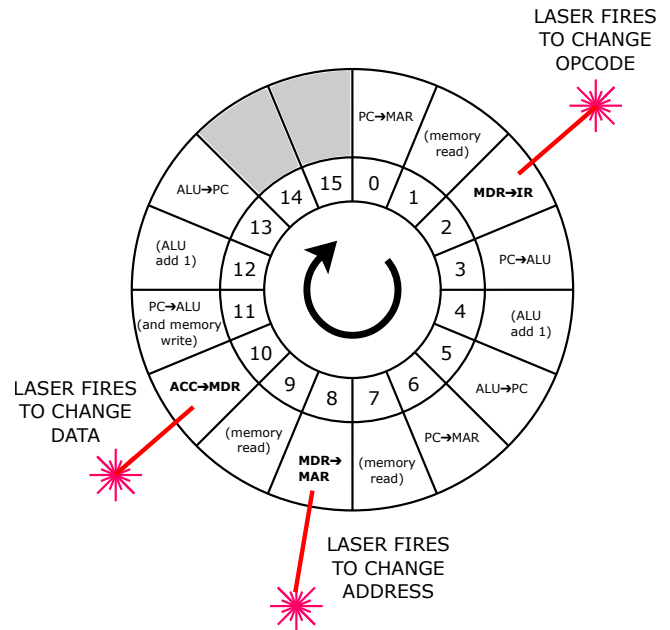


Figure 14: Timing diagram of the microcode showing where the laser fires during fetch and execution of a single instruction (here, STA for “store accumulator”).

instruction was running (for example, INC or DEC if the accumulator value changed by one, or STA if the value changed by more than one). From the time between changes, the attacker can calculate the cycle time by dividing by the number of instructions executed between changes and looking up the cycle time of each instruction, which may be different.

All this could be accomplished a different way simply by watching the bus LEDs. We do it by means of the accumulator simply to illustrate the general principle that the attacker is not necessarily attacking the same LED as the one being watched.

After the attacker has established a phase lock and measured the cycle time, the attack proceeds by counting cycles into a predicted part of the fetch–execute cycle and firing the lasers at the instant when the desired value is known to be on the bus (Figure 14). Typically, the laser fires more than once during a particular instruction; for example, once to change the opcode, again four and a half cycles later to change the memory address, and five cycles after that to change the data being written to memory.

The result of the attack can be seen on the right side of Figure 15. Memory map (b) shows what the memory contents of M5 looked like before the attack; (c) shows what it looks like after. The attacker fired the lasers a total of twenty times in six seconds to force the normal program in locations 0–7 to write a new program in high memory at locations 9–14, and then forced a branch to it.

We did not collect any data on the probability of the attack being successful, because we found it to be reliable under the conditions we set up. The lasers are bolted in position, aimed

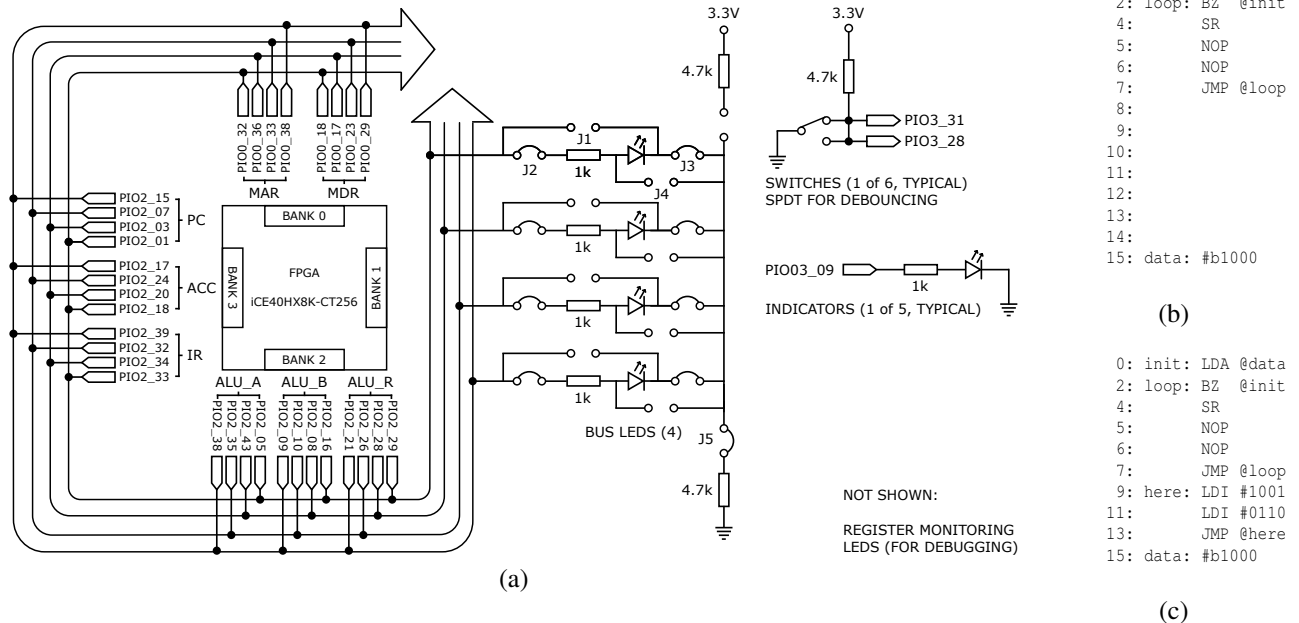


Figure 15: (a) Schematic of the M5 computer, built around a Lattice iCE40HX8K-CT256 FPGA. Jumpers J1–J4 form an H-bridge to allow flexibly reorienting the polarity of bus LEDs for experimentation. The bus is pulled down by jumper J5. (b) Memory map before the attack. (c) Memory map after the successful attack.

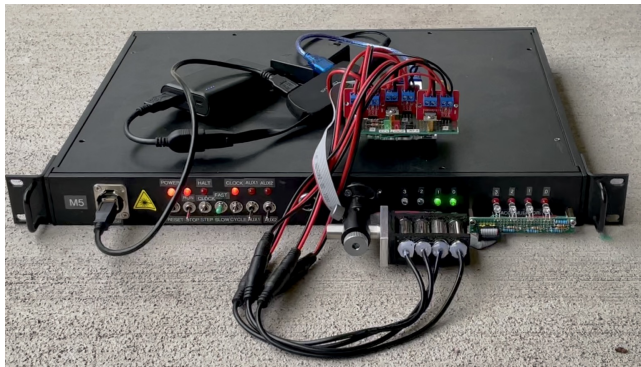


Figure 16: The M5 computer. Four lasers controlled by an Arduino with MOSFET laser drivers, are mounted aimed at the bus LEDs. The four LEDs on the right are the accumulator (A) register and the LEDs and switches on the left are used to interact with the computer.

at the bus LEDs from a range of 2 cm, because it removes an independent variable (aiming error) from the experiment.

The lasers used in this experiment were 405 nm near-UV diode laser modules of unknown power rating. The lasers were extracted from “cat toys” sold on Amazon.com at <https://www.amazon.com/gp/product/B09Y4D7NFB/>. In operation, they draw approximately 150 mA each from a 3.3 V supply, so their optical power must be < 500 mW and is probably considerably less, as they get warm in continuous

operation. *These are absolutely not eye-safe and should never have been sold as cat toys.*

For safety when using 405 nm lasers, we recommend an enclosure made from #2422 transparent orange polycarbonate sheet 3 mm thick, as shown in Figure 6 when that apparatus was running at 405 nm.

The lasers are modulated by switching their power supply on and off with a MOSFET. Two important considerations apply to these lasers; firstly, they need 3.3 V and will burn out quickly at 5 V, but the MOSFETs won’t switch a load less than their gate (control) voltage. So to make the MOSFETs work and avoid burning out the lasers, always switch 5 V through the MOSFET, and drop it down to 3.3 V with an LM317 voltage regulator between the MOSFET and the laser.

These lasers were chosen for use because they exhibit quick response when modulated in this way, typically < 100 μs turn-on and turn-off latency. Many other laser modules from other sources, when measured, had a turn-on latency of more than 4000 μs, limiting modulation to < 0.25 kHz.

Note that the bus is pulled down by the 10 kΩ resistors. There is no particular reason why it needs to be that way; it’s only to make clear that LEDs under laser illumination drive their signals in reverse.

Using the M5 computer we can now demonstrate a few different adversarial capabilities, the first and easiest being to crash the computer using a Basilisk attack, and the second being arbitrary code execution (with some minor constraints).

To crash the computer, the attacker directs an unmodulated

Mnemonic	Opcode	Instruction
NOP	0000	no operation
LDA	0001	load accumulator (addr)
INC	0010	increment accumulator
DEC	0011	decrement accumulator
STA	0100	store accumulator (addr)
BZ	0101	branch if $A \neq 0$ (addr)
JMP	0110	unconditional branch (addr)
SR	0111	shift right accumulator
LDI	1001	load immediate

Table 1: M5 instruction set. It consists of 9 instructions including NOP. This is intentionally simple but Turing complete.

laser to any of the bus LEDs for a few seconds. This has any of the following effects: (1) changing a valid opcode to an invalid one; (2) changing the value of a memory address; (3) changing the contents of a memory read or write operation; or (4) changing the control flow of the program.

To achieve arbitrary code execution, the attacker needs to know something about the internal state of the CPU in order to synchronize precisely. Watching the accumulator display (Figure 13) is sufficient to obtain a phase lock on the CPU’s internal state because the display changes at a known cycle offset within the STA instruction.

Even with knowledge of the running program and the CPU timing, the attack is not trivial. By targeting some or all of the bus LEDs, the attacker can change instructions, or data, or addresses after they are fetched from memory but before they are executed by the CPU. However the attacker can only set bits, *i.e.*, change a 0 to a 1, not the other way around.⁴ This means that for every instruction there is a certain set of different instructions the attacker can reach. This is illustrated in Figure 17. The level of effort is not dissimilar to finding “gadgets” in return-oriented programming (ROP) or selecting opcodes from the printable character subset of the Intel X86 instruction set architecture [12, 15, 16, 57, 58, 76]. Another analogy would be to looking a few moves ahead in a chess game.

The address of a load or store operation can be easily redirected to high memory addresses simply by setting the most significant bit. The attacker can use many loops through the program, setting a few bits here and there, gradually building up the desired program in high memory. Once done, the attacker can simply redirect a branch to the new code.

In case the code in high memory cannot be created exactly how the attacker wants, maybe because some bit combinations were inaccessible, the constructed code can run fixups on itself the first time it runs. This enables the attacker to use the available instructions to write the desired program. After that,

⁴This is in photovoltaic mode; in photoconductive mode, we might only be able to *reset* a bit.

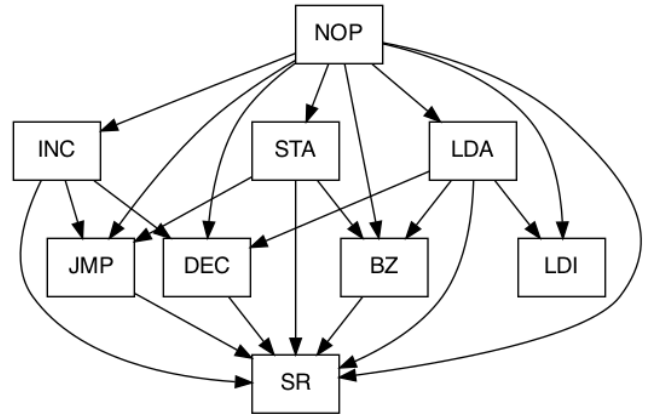


Figure 17: Allowable transitions in the instructions set of the computer defined in Table 1 if the attacker can only set bits, but not reset them.

the fixed-up code runs, and the attacker has full control.

The attacker must be very careful not to crash the running program. If the CPU ever halts, the attack is blocked.

7.2 Attacking an I²C bus

We have demonstrated how a Basilisk attack can lead to arbitrary code execution on a toy computer. In this section we prove the viability by attacking an I²C bus between commercially available devices at 100 kbits⁻¹.

The I²C bus is a serial, synchronously clocked communication bus that is widely used and has a tolerant specification we can abuse. It underlies the System Management Bus (SMBus) and PMBus, as well as being incorporated in many other real-world interfaces including HDMI, PCIe, DVI, and VESA [18]. A timing diagram of the I²C bus can be seen in Figure 19. It consist of two wires: synchronous data (SDA) and synchronous clock (SCL). It is less expensive to attack than a parallel bus (*e.g.*, as used on M5), because the attacker needs to aim and synchronise at most two lasers. Timing is dictated by the sending device and can be arbitrarily slow. This further reduces the difficulty for an attacker since there is no need for synchronization with an existing clock.

I²C is a shared bus with pull-up resistors making the default state of the bus logic high. It uses open collector drivers to pull the bus low when needed so that two devices trying to send at the same time will not cause hardware damage to each other. This makes it ideal as a target for Basilisk attacks.

The I²C bus in our setup (see Figure 18) is provided with ESD protection devices (1N34A Schottky equivalent) on each of the two lines providing an entry point for a Basilisk attack. Figure 20 shows this attack working in practice—on LEDs, in this case. The figure shows an oscilloscope trace of the two I²C wires with the attacker driving both wires to execute the attack. Note how, in the upper trace, the receiver acknowl-

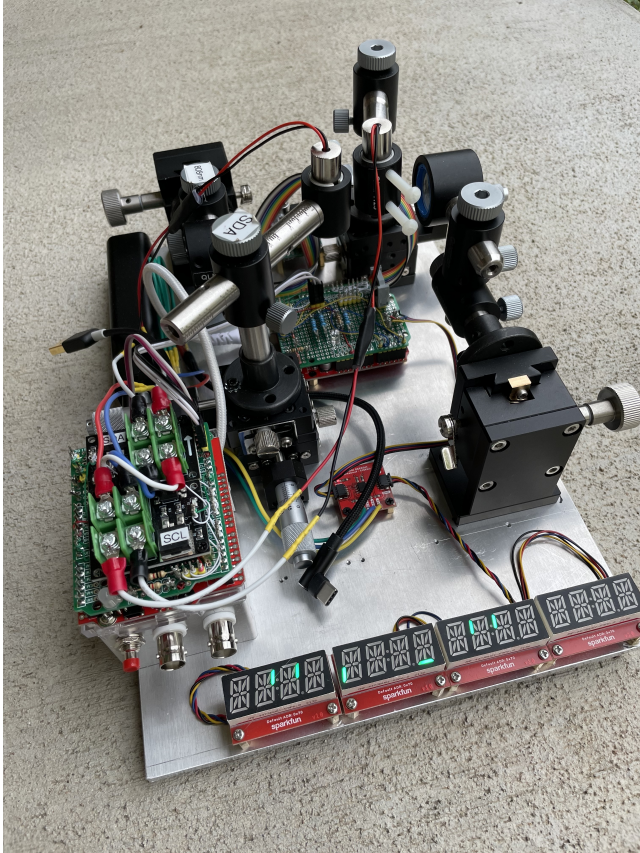


Figure 18: Proof of concept Basilisk attack on I²C bus, 780nm lasers irradiating 1N34A glass-enclosed ESD protection diodes. Representative devices on the bus include quad alphanumeric displays and nonvolatile memory.

edges each byte sent via the lasers by pulling the bus low. We can be sure that this is indeed the receiver pulling the bus low because it is pulled down all the way to 0 V, which the attacker cannot achieve. The LED can only pull the bus down to about 1.2 V (from $V_{DD} = 3.3$ V) but that is enough.

We identify three attacks: denial of service, message manipulation, and message insertion. The first and easiest attack is denial of service where the attacker simply sends a constant beam of light. Even if the attacker only controls one of the two bus lines, this attack effectively prevents any other communication from taking place.

The second attack is message manipulation where an attacker alters messages sent by other nodes, subject to analogous restrictions as in the M5 example—the attacker can only reset bits, not set them. This attack requires knowledge of the messages going across the bus and precise control over the timing of the attack. The sender of the message is able to detect the change, and will interpret it as bus contention, backing off automatically, but the intended receiver does receive the altered message.

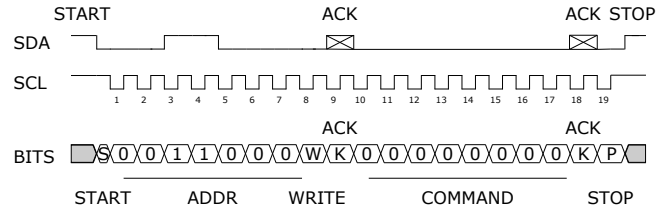


Figure 19: I²C bus timing diagram. The I²C protocol uses two wires, one for synchronous data (SDA) and one for a synchronous clock (SCL). Both of these are driven by the sender, except for the acknowledgement which is driven by the receiver.

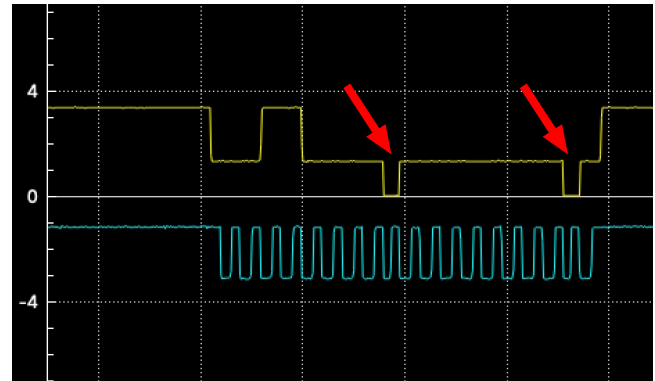


Figure 20: Oscilloscope trace showing successful communication with an I²C device via laser illumination of status LEDs. Upper trace (yellow) is SDA; lower trace (blue) is SCL. The attacker is only able to pull the bus down to about 1.2 V, not all the way to ground. We know communication was successful because of the two negative-going acknowledgement pulses in the SDA trace, generated by the receiving device—one of the alphanumeric display modules, in this case—which is able to pull the I²C bus all the way to ground.

The third and final attack is to transmit messages on an empty bus. It assumes that the attacker knows when the bus is empty, but beyond that there is no further need for precise synchronization with the existing devices.

8 Discussion

Introducing energy to the system—along with information—violates assumptions made by the hardware designer.

8.1 Laser power

In most cases accuracy and precision of aiming can be at least partially substituted with more power.

Our experiments used reclaimed laser modules from old CD, DVD, and Blu-ray players. This is in part to show that Basilisk attacks can be done with very inexpensive hardware,

but also that is what we could easily source. The downside of this is that we do not have specifications for all the lasers, specifically we do not have calibrated intensity measurements.

The lasers we have are sufficient to execute the attacks, but being able to precisely control the optical power and to focus the beam to a smaller spot size (*e.g.*, 10 μm) would allow for a more effective delivery of power to the P–N junction.

We observed standard laser safety protocols including use of warning signs, shielding, beam blocks, and protective glasses [6–8, 46, 88]. Hazards related to frequency-doubled 532 nm green laser pointers are well-known [11, 20].

There is no particular reason to think the light need be coherent, or even monochromatic. This is supported by the fact that xenon flash tubes are as effective as lasers.

8.2 Laser Wavelength

The lasers we have allowed us to test a variety of shorter wavelengths, 650 nm (red), 532 nm (green), and 405 nm (blue/violet) in addition to the infrared 780 nm, 808 nm, and 980 nm lasers we used. Shorter wavelengths tend to be more effective for LEDs and longer wavelengths are better for silicon.

8.3 Countermeasures

There are two main classes of countermeasures, active and passive. Active countermeasures rely on first detecting that an attack is taking place and then taking corrective action; attacks might be detected optically [9] or electrically [56]. Passive countermeasures minimize the attack surface, either by eliminating vulnerable components or shielding them from influence. Opaque chip packages can mitigate the attack, but this is no option for indicators that must remain visible.

Minimize the number of exposed LEDs or other photosensitive components on a device. Avoid connecting exposed P–N junctions directly to circuits carrying sensitive information.⁵ Existing electromagnetic interference (EMI) reduction techniques may be effective against photovoltaic mode, but are unlikely to be effective in photoconductive mode.

It is interesting to note that the most effective wavelength for silicon ESD protection diodes (not LEDs) is in the infrared part of the spectrum, which makes the attacks stealthy.

8.4 Other Targets

There are a few areas where it is surprisingly common to have LEDs directly connected to sensitive circuits. We found that CAN bus devices commonly have them [34, 50]. Attacking a differential signaling bus like CAN is more challenging, as it will require the attacker to exploit both the photovoltaic and

⁵Indicators are not often found connected directly to a shared bus, because they load the circuit, slowing communication. LEDs are sometimes buffered by a transistor or op-amp driver, which has the advantage of a brighter indicator, but with the drawback of a slight overhead in cost and complexity.

photoconductive methods on electrically adjacent components at the same time for it to work.

In certain circumstances it is possible that a Basilisk attack can damage the victim device. LEDs driven into photovoltaic mode tend to pull the circuit on their cathode sides more negative. This can, if the circuit on the cathode side of the LED is a low positive voltage, result in the circuit going below ground which could be damaging for sensitive electronics.

8.5 Commercially Available Hardware

We have been able to demonstrate the effect only on one piece of commercially available hardware: a 5 mm RGB color-changing LED often found in light-up toys [77].

The device was found to be disrupted by 405 nm, 520 nm, 650 nm, 780 nm, 808 nm, and 980 nm lasers, and the color changing sequence can be reliably reset to red at a distance of 25 cm by a xenon camera flash; this is consistent with the emission spectrum of xenon, which is rich in near-IR.

The effect was first reported in a comment on the article about the Raspberry Pi 2 glitch mentioned earlier [21, 28]. The chip inside is believed to be a CDT3447 or similar [69].

9 Conclusion

We present an attack framework we call Basilisk, after the mythical animal that could kill with a single glance [27, 45].

While the photosensitivity of semiconductor diodes is a known phenomenon, we demonstrate the practical requirements for an external attacker to use the effect as an attack vector. We show that Basilisk attacks are feasible in practice, both against ESD protection devices and LEDs, and can be performed as long as the attacker has line of sight access.

Our results go beyond a feasibility study. We show two concrete attacks that have serious consequences. Depending on an attacker’s knowledge of the victim and equipment complexity, it is possible to achieve a number of effects, from denial of service to arbitrary code execution on an air-gapped computer system. Our results lead to testable predictions about the vulnerability of any shared bus that uses open collector (or open drain) tristate drivers.

Minimization is the most effective countermeasure, followed by buffering LEDs—which might be enough to block photovoltaic mode attacks only—but mitigation remains a challenge, especially for indicators that need to be exposed.

Availability

Verilog source code for FPGA implementation of the M5 CPU, Arduino source code for the attacker’s equipment and the experimental apparatus, raw data, and scripts for data reduction and plotting are available on GitHub at https://github.com/jloughry/basilisk_artifacts.

References

- [1] Rhett Allain. You can power a calculator with some LEDs, 11 March 2019. <https://www.wired.com/story/you-can-power-a-calculator-with-some-leds/>.
- [2] Analog Devices. Low voltage logic interfacing. Tutorial MT-098, Analog Devices, Inc., 2009. <https://www.analog.com/media/en/training-seminars/tutorials/mt-098.pdf>.
- [3] Ross Anderson and Markus Kuhn. Tamper resistance—a cautionary note. In *Second USENIX Workshop on Electronic Commerce*, pages 1–11, Oakland, California, 18–21 November 1996. <https://www.usenix.org/conference/2nd-usenix-workshop-electronic-commerce/tamper-resistance-cautionary-note>.
- [4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January–March 2004. Available from https://www.nasa.gov/pdf/636745main_day_3-algirdas_avizienis.pdf.
- [5] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer’s Apprentice guide to fault attacks. IACR e-print 2004/100, International Association for Cryptologic Research, 2004. <https://eprint.iacr.org/2004/100>.
- [6] Ken Barat. Eye safety in the laser lab: Using the humble beam block shows infinite wisdom. *Photonics Spectra*, August 2007.
- [7] Ken Barat. Laser safety and the optical table. *Photonics Spectra*, October 2007.
- [8] Kenneth L. Barat. In laser safety, little mistakes can have big consequences. *Photonics Spectra*, March 2005.
- [9] Don Barber, Vikram Kanth, Zachary White, and John McEachen. Spatial frequency detection of optical signals embedded in the environment. In *2022 32nd International Telecommunication Networks and Applications Conference (ITNAC)*, 2022.
- [10] Charles J. Bashe, Lyle R. Johnson, Emerson W. Pugh, and John H. Palmer. *IBM’s Early Computers*. MIT Press, 1985.
- [11] ‘Brainiac75’. The issue with green laser pointers. <https://www.youtube.com/watch?v=iR1Ku5dnbH8>, 5 August 2018.
- [12] Jerome Bruandet. Anatomy of the EICAR antivirus test file. Ninja Technologies Network, 26 August 2021. <https://blog.nintechnet.com/anatomy-of-the-eicar-antivirus-test-file/>.
- [13] James Bryant. Glass diodes may see the light—and hum. *Analog Dialog*, 45, May 2009.
- [14] James Bryant. LEDs are photodiodes too. Planet Analog, 5 August 2014.
- [15] Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. Return-oriented programming: Exploitation without code injection. In *Black Hat US 2008*, 2008.
- [16] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of CCS 2008*, pages 27–28, Alexandria, Virginia, October 27–31, 2008.
- [17] Christopher P. Burton. Replicating the Manchester Baby: motives, methods, and messages from the past. *IEEE Annals of the History of Computing*, 27(3):44–60, July–September 2005.
- [18] Zitai Chen and David Oswald. PMFault: Faulting and bricking server CPUs through management interfaces. arXiv preprint arXiv:2301.05538 [cs.CR], Cornell University, 13 January 2023.
- [19] Sean Collins and Stephen McCombie. Stuxnet: the emergence of a new cyber weapon and its implications. *Journal of Policing, Intelligence and Counter Terrorism*, 7(1):80–91, 2012.
- [20] Annette Colton. Study of laser pointer safety. Study, Lucid Optical Services, 2010.
- [21] ‘Conundrum1885’. RE. Re. Xenon DEATH FLASH!!!! *The Register*, 14 March 2015. https://forums.theregister.com/forum/all/2015/02/08/raspberry_pi_2_camera_flash_glitch/#c_2465254.
- [22] Benjamin Cyr, Vedant Sumaria, Yan Long, Srinivas Tadi-gadapa, Takeshi Sugawara, and Kevin Fu. How lasers exploit photoacoustic and photoelectric phenomena to inject signals into MEMS microphones. Preprint, Research Square, 11 April 2024.
- [23] Paul Dietz, William Yerazunis, and Darren Leigh. Very low-cost sensing and communication using bidirectional LEDs. Technical Report TR2003-35, Mitsubishi Electronics Research Laboratories (MERL), 201 Broadway, Cambridge, Massachusetts 02139, USA, July 2003.
- [24] Jeroen Domburg. Optical mouse cam. <http://spritesmods.com/?art=mouseeye>, 2006.

- [25] Mathieu Dumont, Pierre-Alain Moellic, Raphael Viera, Jean-Max Dutertre, and Rémi Bernhard. An overview of laser injection against embedded neural network models. arXiv preprint arXiv:2105.01403 [cs.CR], Cornell University, 4 May 2021.
- [26] Arno Erzberger. Der LED fehlt der doppelpeil. *Elektronik*, 21 June 2016. <https://www.elektroniknet.de/power/energy-harvesting/der-led-fehlt-der-doppelpeil.131470.html>.
- [27] Oliver Evans. Selections from the bestiary of Leonardo Da Vinci. *Journal of American Folklore*, 64(254):393–396, October–December 1951.
- [28] Kelly Fiveash. ‘Camera-shy’ Raspberry Pi 2 suffers strange ‘XENON DEATH FLASH’ glitch. *The Register*, 8 February 2015.
- [29] Curtis Franklin. Glitching: The hardware attack that can disrupt secure software. *Dark Reading*, 18 October 2019. <https://www.darkreading.com/edge-articles/glitching-the-hardware-attack-that-can-disrupt-secure-software>.
- [30] Juan Carlos Garcia-Escartin, Shihan Sajeed, and Vadim Makarov. Attacking quantum key distribution by light injection via ventilation openings. *PLoS One*, 15(8):e0236630, August 2020.
- [31] Ilias Giechaskiel and Kasper Rasmussen. Taxonomy and challenges of out-of-band signal injection attacks and defenses. *IEEE Communications Surveys & Tutorials*, 22(1), First Quarter 2020.
- [32] Bret Giller. Implementing practical electrical glitching attacks. In *Black Hat Europe 2015*, Amsterdam, 10–13 November 2015. <https://www.blackhat.com/docs/eu-15/materials/eu-15-Giller-Implementing-Electrical-Glitching-Attacks.pdf>.
- [33] D. H. Habing. The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits. *IEEE Transactions on Nuclear Science*, 12(5):91–100, 1965.
- [34] Microchip Technology Inc. *CAN Bus Analyzer User’s Guide*, 2020.
- [35] Karla Jennings. *The Devouring Fungus: Tales of the Computer Age*. W. W. Norton and Company, Inc., 1990.
- [36] Stefanos Koffas and Praveen Kumar Vadnala. On the effect of clock frequency on voltage and electromagnetic fault injection. arXiv preprint arXiv:2310.13389 [cs.CR], Cornell University, 20 October 2023. Published in AIHWS workshop held for Applied Cryptography and Network Security Conference (ACNS 2022); DOI: https://doi.org/10.1007/978-3-031-16815-4_8.
- [37] Thilo Krachenfels, Heiko Lohrke, Jean-Pierre Seifert, Enrico Dietz, Sven Frohmann, and Heinz-Wilhelm Hübers. Evaluation of low-cost thermal laser stimulation for data extraction and key readout. *Journal of Hardware and Systems Security*, 4(1):24–33, 2020.
- [38] Thilo Krachenfels, Heiko Lohrke, Jean-Pierre Seifert, Enrico Dietz, Sven Frohmann, and Heinz-Wilhelm Hübers. Evaluation of low-cost thermal laser stimulation for data extraction and key readout. arXiv preprint arXiv:2006.06290 [cs.CR], Cornell University, 11 June 2020.
- [39] Ireneusz Kubiak. *Specjalne fonty komputerowe w bezpieczeństwie elektromagnetycznym cyfrowych standardów graficznych: TEMPEST optyczny*. Wojskowa Akademia Techniczna, Warsaw, 2020.
- [40] David Kushner. The real story of Stuxnet. *IEEE Spectrum*, 50(3):48–53, March 2013.
- [41] Butler W. Lampson. A note on the confinement problem. *Comm. ACM*, 16(10):613–615, October 1973.
- [42] Don Lancaster. Build Digiviewer II. *Popular Electronics*, 6(3):63–69, September 1974.
- [43] Don Lancaster. *TTL Cookbook*. SAMS, Indianapolis, Indiana, 1974.
- [44] Don Lancaster. *CMOS Cookbook*. SAMS, Indianapolis, Indiana, 1977.
- [45] David Langford. comp.basilisk FAQ. *Nature*, 402:465, 2 December 1999.
- [46] Laser Institute of America. Seven-year update drives national laser safety standard forward. *Photonics Spectra*, June 2007.
- [47] Littelfuse.com. Application hints for transient voltage suppression diode circuits. Application Note AND8230/D, Littelfuse, Inc., 16 September 2016.
- [48] Heiko Lohrke, Shahin Tajik, Thilo Krachenfels, Christian Boit, and Jean-Pierre Seifert. Key extraction using thermal laser stimulation: A case study on Xilinx Ultrascale FPGAs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):573–595, 2018.
- [49] Marcel Meli and Niklas Roth. LEDs und photodiode als energy harvester. *Elektronik*, 3 June 2016. <https://www.elektroniknet.de/power/energy-harvesting/leds-und-photodioden-als-energy-harvester.130892.html>.

- [50] Microchip Technology Inc. *MCP2515 CAN Bus Monitor Demo Board User's Guide*, 2014.
- [51] Micron Technology. *IS32 Optic Ram data sheet*. Boise, Idaho, USA, May 1984.
- [52] Forrest M Mims. Using LED's as light detectors. *Popular Electronics*, 11(5):86–88, May 1977.
- [53] Forrest M Mims. *Light Emitting Diodes*. Howard W. Sams & Co., Inc., Indianapolis, Indiana, USA, 1973.
- [54] Forrest M Mims. Bidirectional optoisolator puts two LEDs nose to nose. *Electronics*, 52(10):127, 10th May 1979.
- [55] Forrest M Mims. How to use LEDs to detect light. *Make*, 36:136–138, 2014.
- [56] Saleh Khalaj Monfared, Kyle Mitard, Andrew Cannon, Domenic Forte, and Shahin Tajik. LaserEscape: Detecting and mitigating optical probing attacks. arXiv preprint arXiv:2405.03632 [cs.CR], Cornell University, 6 May 2024.
- [57] Tom Murphy. C with ABC! In *11th Special Interest Group on Harry Q. Bovik (SIGBOVIK)*, Pittsburgh, Pennsylvania, 31 March 2017. Association for Computing Heresy.
- [58] Tom Murphy. Compiling C to printable x86, to make an executable research paper, 31 March 2017. https://www.youtube.com/watch?v=LA_DrBwkiJA.
- [59] Ben Nassi, Adi Shamir, and Yuval Elovici. Oops!...I think I scanned a malware. arXiv preprint, 03 2017.
- [60] NXP Semiconductors. *UM10204: I2C-bus specification and user manual, Rev. 7.0*, 1 October 2021.
- [61] ON Semiconductor. Wafer-level chip-scale package (WLCSP) at ON Semiconductor. Application Note 5075/D, Fairchild Semiconductor, October 2018.
- [62] Masashi Ono, Parthiban Santhanam, Wei Li Li, Bo Zhao, and Shanhui Fan. Experimental demonstration of energy harvesting from the sky using the negative illumination effect of a semiconductor photodiode. *Applied Physics Letters*, 114(16):161102, 2019. <https://aip.scitation.org/doi/10.1063/1.5089783>.
- [63] Sung-Yun Park, Kyuseok Lee, and Hyunsoo Song. Simultaneous imaging and energy harvesting in CMOS image sensor pixels. *IEEE Electron Device Letters*, 39(4):532–535, April 2018.
- [64] Dmytro Petryk, Zoya Dyka, Jens Katzer, and Peter Langendoerfer. Metal fillers as potential low cost countermeasure against optical fault injection attacks. arXiv preprint arXiv:2103.12436 [cs.CR], Cornell University, 17 January 2022.
- [65] Dmytro Petryk, Zoya Dyka, and Peter Langendörfer. Sensitivity of standard library cells to optical fault injection attacks in IHP 250 nm technology. In *9th Mediterranean Conference on Embedded Computing (MECO)*, Budva, Montenegro, 8–11 June 2020.
- [66] Dmytro Petryk, Zoya Dyka, Eduardo Perez, Mamathamba Kalishettyhalli Mahadevaiaha, Ievgen Kabin, Christian Wenger, and Peter Langendörfer. Evaluation of the sensitivity of RRAM cells to optical fault injection attacks. In *23rd Euromicro Conference on Digital System Design (DSD)*, Kranj, Slovenia, 26–28 August 2020.
- [67] Dmytro Petryk, Zoya Dyka, Roland Sorge, Jan Schaeffner, and Peter Langendoerfer. Optical fault injection attacks against radiation-hard registers. arXiv preprint arXiv:2106.07271 [cs.CR], Cornell University, 18 January 2022.
- [68] Phillips Semiconductor. *The I2C-Bus and How to Use It (including specifications)*, April 1995.
- [69] Qipeng Semiconductor Co., Ltd. *CDT3447 3 LED Fade-in and fade-out output control IC*, 2008. <http://www.bowin-ic.com/hk/IC/LED%20flasher%20IC/CDT3447.pdf>.
- [70] Paul Rako. EEVblog #901 - Raspberry Pi 3 photoflash problem. YouTube, 16 July 2016.
- [71] Sara Rampazzi, Benjamin Cyr, and Daniel Genkin. Light commands: Hacking voice assistants with lasers. In *Black Hat Europe 2020*, Virtual, 7–10 December 2020. <https://www.blackhat.com/eu-20/briefings/schedule/index.html#light-commands-hacking-voice-assistants-with-lasers-21731>.
- [72] Allison Randal. This is how you lose the transient execution war. arXiv preprint arXiv:2309.03376 [cs.CR], Cornell University, 6 September 2023.
- [73] Khushboo Rani, Hansika Weerasena, Stephen A. Butler, Subodha Charles, and Prabhat Mishra. Modeling and exploration of gain competition attacks in optical network-on-chip architectures. arXiv preprint arXiv:2303.01550 [cs.CR], Cornell University, 3 March 2023.
- [74] Melanie R. Rieback, Bruno Crispo, and Andrew S. Tanenbaum. Is your cat infected with a computer virus? In *Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM '06)*, pages 169–179, Pisa, Italy, 13–17 March 2006. IEEE Computer Society.

- [75] D. E. Rosenheim. Installation of the first production 701. *Annals of the History of Computing*, 5(2):146–147, April–June 1983.
- [76] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, Alexandria, Virginia, October 28–31, 2007.
- [77] Shenzhen Xuancai Electronc Co., Ltd. *Color Changing LED*. Shen Zhen, China, 2011. <http://cdn.sparkfun.com/datasheets/Components/LED/changingLED.pdf>.
- [78] Sergei Skorobogatov. Fault attacks on secure chips: from glitch to flash. In *Design and Security of Cryptographic Algorithms and Devices (ECRYPT II)*, Albena, Bulgaria, 29 May–3rd June 2011.
- [79] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Redwood Shores, California, USA, 13–15 August 2002.
- [80] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, pages 2–12, Redwood Shores, California, 13–15 August 2002.
- [81] R. Stojanović and Dejan Karadaglić. Single LED takes on both light-emitting and detecting duties. *Electronic Design*, 55(16):53–54, 18th July 2007.
- [82] Charles Stross. “Nothing like this will be built again”. *Charlie’s Diary*, 2010. <https://www.antipope.org/charlie/blog-static/rants/nothing-like-this-will-be-buil.html>.
- [83] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: Laser-Based audio injection attacks on Voice-Controllable systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2631–2648. USENIX Association, August 2020.
- [84] S. M. Sze and Kwok K. Ng. *Physics of Semiconductor Devices*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.
- [85] Toshiba. Basics of ESD protection (TVS) diodes. Application note, Toshiba Electronic Devices & Storage Corporation, 27 May 2022.
- [86] Endel Uiga. *Optoelectronics*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
- [87] Liz Upton. Xenon death flash: a free physics lesson. *Raspberry Pi blog*, 9 February 2015.
- [88] John Wallace. Many laser-safety eyewear products do not meet specs for shielding light from ultrafast lasers. *Laser Focus World*, 29th November 2017. <http://www.laserfocusworld.com/articles/2017/11/many-laser-safety-eyewear-products-do-not-meet-specs-for-shielding-light-from-ultrafast-lasers.html>.
- [89] D. G. Whitehead, I. Mitchell, and P. V. Mellor. A low-resolution vision sensor. *J. Phys. E: Sci. Instrum.*, 17:653–656, 1984.
- [90] J. F. Ziegler, H. W. Curtis, H. P. Muhlfield, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O’Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, T. D. Sullivan, and C. W. Wahaus. IBM experiments in soft fails in computer electronics (1978–1994). *IBM Journal of Research and Development*, 40(1):3–18, January 1996.

SOK: 3D Printer Firmware Attacks on Fused Filament Fabrication

Muhammad Haris Rais*
Department of Computer Science
Virginia State University
mrais@vsu.edu

Muhammad Ahsan
Department of Computer Science
Virginia Commonwealth University
ahsanm5@vcu.edu

Irfan Ahmed
Department of Computer Science
Virginia Commonwealth University
iahmed3@vcu.edu

Abstract

The globalized nature of modern supply chains facilitates hostile actors to install malicious firmware in 3D printers. A worm similar to Stuxnet could stealthily infiltrate a printer farm used for military drones, resulting in the production of batches with a variety of defects. While cybersecurity researchers have extensively delved into the designing and slicing stages of the printing process and explored physical side channels for offensive and defensive research, the domain of firmware attacks remains significantly underexplored. This study proposes a classification tree for firmware attacks, focusing on the attack goals. We further propose nine distinct firmware attacks within these categories to demonstrate and understand the impact of compromised firmware on a standard fused filament fabrication printer. The study evaluates these attacks through relevant destructive and non-destructive tests, including assessing the tensile strength of the printed parts and conducting air quality tests at the printing premises. The study further investigates the viability of forty-eight attacks, including nine that we propose, across the 3D printing stages: the design stage (involving CAD file manipulation), the slicing stage (involving G-code file manipulation), and the printing stage (involving firmware manipulation). Drawing on our understanding of the 3D printing attack surface, we introduce an Attack Feasibility Index (AFI) to assess the feasibility of attacks at different printing stages. This systematization and examination advances the comprehension of potential 3D printing attacks and urges researchers to delve into cybersecurity strategies focused on counteracting feasible attacks at specific printing stages.

1 Introduction

The popularity of additive manufacturing (AM) is on the rise [1], with critical industrial sectors such as aerospace [2], automotive, and healthcare [3] utilizing 3D-printed functional

parts. Consequently, malicious actors now have greater incentives to attack AM setups and sabotage the printed parts. Concurrently, the current industry trend towards fully connected and converged IT and industrial networks [4] potentially extends the reach of cyber attackers to manufacturing units. Over the past few years, the research community has been actively engaged in both offensive and defensive aspects of AM security. AM is a cyber-physical system (CPS) and as any other CPS technology (SCADA, IOT, etc.) subjected to potential security breaches [5–11].

The existing offensive research focuses on either stealing intellectual property (IP) information through side channels [12, 13] or inducing defects in the printed part [14]. Being fundamentally different from its predecessor technologies, AM or 3D printing (3DP) offers unique attack opportunities for an adversary to sabotage the physical properties of the printed parts. These attacks degrade the object’s mechanical strength without modifying the dimensions, weight, center of mass, and other measurable attributes [15]. Although the researchers acknowledge the possibility of firmware attacks [16, 17], most sophisticated attacks are demonstrated at the design and the slicing stages. Moreover, no taxonomy of firmware attacks exists in AM security literature.

This study aims to systematize the knowledge of firmware attacks on 3D printers by developing a classification tree focusing on specific attack objectives. Starting with the fundamental goals of surveillance, denial of service, and integrity breach, the tree extends to include technically attributable sub-goals related to the components of the printing process, the printing premises, the printed parts, and the target assembly for which the parts are being printed.

To demonstrate the utility of the classification tree, we showcase nine novel firmware attacks emerging from diverse nodes within the tree, encompassing surveillance, denial of printing service, object integrity, printer damage, and printing premises-related attacks. For instance, *print your own grave* attack prints a tool and uses it to physically damage the printer’s components, such as the printing bed. Another attack, named *incurable*, deceives the user by mimicking com-

*Rais completed this work while he was a PhD student at Virginia Commonwealth University

mon printing faults, leading to prolonged and ineffective troubleshooting of the printing environment. In a sabotage attack on printing premises, the adversary contaminates the printing facility’s environment, leading to potential health hazards. The study demonstrates the impact of these attacks on a standard FFF-based 3D printer running Marlin, the most widely used open-source firmware for 3D printers [18].

The difficulty level of implementing an attack may vary depending on the attack goal. For example, denying printing services by blocking printing instructions is a straightforward task for malicious firmware. In contrast, managing the computation and space complexity involved in scaling an object at the firmware level is challenging. Assessing the complexity of attacks is crucial for understanding the risks involved and prioritizing defensive measures. However, our search revealed no existing research on the complexity analysis of additive manufacturing (AM) attacks.

To fill this gap, we conducted an in-depth analysis of 48 attacks, including those we proposed, to evaluate their implementation complexity at various stages of the printing process. To summarize our findings, we introduce the Attack Feasibility Index (AFI), which represents the feasibility or difficulty level of implementing a specific category of attacks at a particular stage of the printing process.

The findings from the AFI indicate that not all attacks are feasible at any stage of the printing process. For instance, sabotaging the printing premises proves infeasible when targeting the design stage of the process chain. Consequently, cybersecurity solutions optimized for the design stage may not prioritize detecting such attacks.

Contributions. This study offers three main contributions:

1. An attack-goal-focused classification tree for the firmware attacks on 3D printers.
2. Implementation and evaluation of nine novel firmware attacks on Marlin-based FFF 3D printer.
3. An analysis of the feasibility of 48 AM attacks at various stages of the AM process chain, providing insights into their implementation complexity at each stage.

2 Background and Related Work

2.1 Fused Filament Fabrication - FFF

Additive manufacturing, also known as 3D printing (3DP), is a manufacturing technique that constructs objects by adding material layer by layer. This approach fundamentally differs from traditional manufacturing techniques, such as subtractive manufacturing and forging. 3DP offers numerous advantages over previous methods, such as printing complex geometries in a single part, reduced wastage, customized instead of bulk production, and a rapid design-to-production cycle. The ASTM

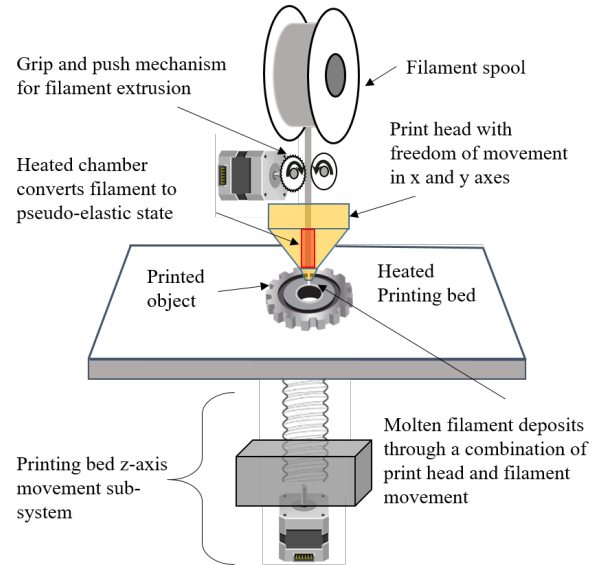


Figure 1: Fused filament fabrication printer

International standards organization defines seven AM methods, including material extrusion, powder bed fusion, and vat photopolymerization, among others [19]. Material extrusion, one of the most commonly used additive manufacturing (AM) techniques, predominantly employs the Fused Filament Fabrication (FFF) process, which is the focus of this study.

Figure 1 provides an overview of a typical FFF printer, which creates three-dimensional (3D) objects by extruding molten filament onto a heated printing bed. For first-layer adhesion and to prevent object warping, the printing bed is maintained at a temperature close to the glass transition temperature of the filament. The printing process starts with the solid filament from a material spool fed into the print head by a stepper motor. The print head features a heated chamber that melts the filament into a molten, piezo-elastic state.

The print head utilizes the molten filament extruded from the nozzle orifice to draw a single-layer geometry, resembling a 2D plotter printer with a finite thickness, usually only a fraction of a millimeter. The filament’s molten state allows it to pass through the small nozzle orifice, facilitating bonding (fusion) with the previously extruded material to create a solid geometry. Once a layer is complete, the printing bed moves down to create space between the nozzle and the object for the next layer. This layer-by-layer process continues until the desired object is fully printed.

2.2 Related Work

This section summarizes current research endeavors focused on firmware attacks targeting material-extrusion AM systems and their classification.

Attacks Classification. Several research studies have proposed taxonomies for cyber-physical system (CPS) attacks

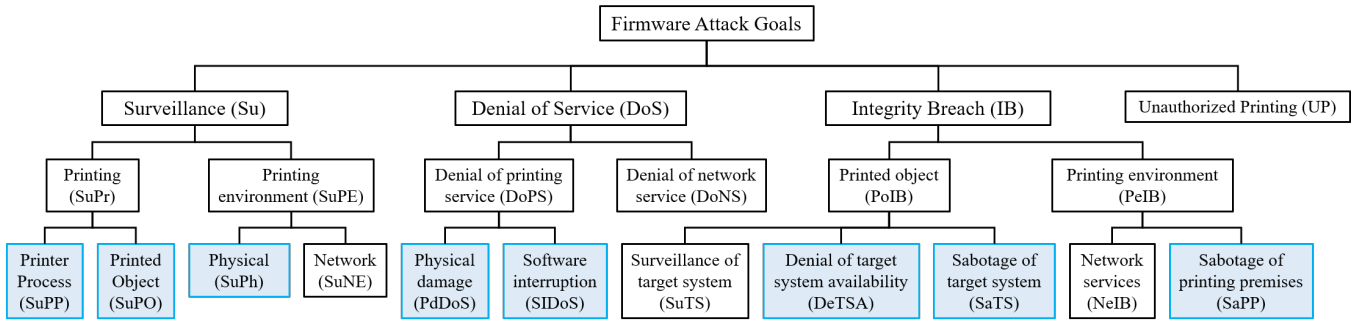


Figure 2: Classification of firmware attacks (categories in blue are covered through proposed attacks)

in AM systems. Yampolskiy et al. [20] developed an attack taxonomy focusing on semantically identical manipulations introduced by compromised elements. Their taxonomy includes a subset of targeted properties known as ‘attack targets’ but does not delve into the attacker’s goals or consider denial of service attacks. In another study [21], they characterize attacks based on manipulated properties.

Pan et al. [22] proposed a taxonomy that comprises vulnerability, attack vector, attack target, and attack impact; however, it is not focused on attack goals. Mahesh et al. [23] presented a four-level attack taxonomy for AM systems, starting with attack goals, methods, targets, and countermeasures. However, their taxonomy does not cover an in-depth attack categorization. Moreover, they see service denial and IP theft as methods rather than attack goals. Wu et al. [24] developed a taxonomy for AM attacks that includes two parallel streams of cyber and physical attacks. It only enumerates a few attack outcomes under cyber and physical attack consequences. Gupta [16] presented multiple supply chain models for an AM process and highlighted the types of attacks associated with those models. They further discussed the risks associated with the studied attacks. The paper summarily mentions the firmware attack vector without delving into the details.

Our study takes a distinctive approach by constructing a multi-tier attack categorization tree based on attack goals within additive manufacturing (AM) systems. This focus allows us to elucidate the strategic intentions behind various attack methodologies and enables the development of targeted countermeasures tailored to thwart specific attack goals.

Firmware Attacks on 3D Printers. Researchers have demonstrated sabotage attacks on 3D printers at pre-firmware stages, such as during the design and slicing stages. Additionally, surveillance attacks through side channels have been extensively investigated [25–27]. Nevertheless, exploration into firmware attacks remains relatively limited. Xiao [28] demonstrated firmware attack feasibility on 3D printers. He showcased a thermal manipulation attack by modifying the open-source RepRap firmware through a USB-based serial connection. Moore et al. [29] studied the impact of malicious firmware on print quality by manipulating extruder feed rate

or printing alternate geometries. However, their study did not comprehensively analyze the attacks achievable through firmware compromise.

Pearce et al. [30] presented "FLAW3D", a bootloader trojan capable of attacking AVR-based Marlin-supported 3D printers. They demonstrated two low-footprint attacks that could reduce the strength of printed parts. The authors mentioned that only simple manipulation could be feasible due to memory constraints in the bootloader space. Do et al. [31] extracted data from a network-connected printer by exploiting the authentication process vulnerability.

3 Classification of Firmware Attacks Goals

Motivation. Our motivation for developing the attack classification tree for firmware attacks (Figure 2) stems from the observation that current classifications focus predominantly on the attack actions. Attackers often employ a consistent set of malicious actions at varying intensities to achieve different goals. For instance, low-magnitude thermodynamic manipulation may degrade the object’s properties sufficiently for it to fail during operation after installation in the target system. Conversely, high-intensity thermodynamic variations might result in the production of an utterly misshapen object that never gets installed in the target system. In cyber-physical systems, a detection solution might effectively identify actions performed at a higher intensity while failing to recognize those same actions at lower intensities. Consequently, we can classify these detection solutions as effective against one attack category but ineffective against another.

Methodology. The methodology involves an iterative division of the attack goals space. Initializing with the top-tier categories encompass surveillance, denial of service, and integrity breach. We also introduce a specific category referred to as ‘unauthorized printing,’ which pertains to the printing of illegal objects without the process owner’s approval. As we move along the tree, the attack goals and the subsequent firmware interactions get more specific. To maintain brevity and comprehensiveness, we limited our exploration to the categories associated with the printed object, the printing process,

and the printing environment without delving into further hierarchy. For example, actions resulting in physical damage to any part of the printer, such as targeting the printing bed or nozzle, are encompassed under a single attack goal, denoted as ‘PdDoS.’ The subsequent subsections provide a concise overview of the nodes in the categorization tree.

3.1 Surveillance

Surveillance attacks do not modify/sabotage the printing process itself, rather they aim at stealing the printing facility or the printing process information.

3.1.1 Printing Surveillance (SuPr)

Printing process surveillance can be further categorized into Surveillance of the Printing Process (SuPP) and Surveillance of the Printed Object (SuPO). The IP information is highly valuable, as its disclosure could result in significant financial losses for businesses. For example, competitors might be interested in gaining insights into the types of prototypes being printed in the research lab. Surveillance can also assist in accomplishing more adversarial goals, such as planning future attacks. In such cases, information pertaining to the network [32], control software and the printer are used to fingerprint the system or design an attack specific to the print geometry.

3.1.2 Printing Environment (SuPE)

3D printers can effectively act as spying devices to gather the premises or the environment data. Instead of surveillance of the printing process, the sensing data could be used to illicitly gather information about the facility itself. The information could be from physical sensing systems (SuPh), e.g., cameras, temperature gauges, or network traffic (SuNT), which provide insights into the connected devices over the network. An attacker can use various ways to exfiltrate the information, e.g., by hiding artifacts in the printed object [33].

3.2 Denial of Service (DoS)

A malicious firmware can pursue numerous intriguing Denial of Service (DoS) goals by exploiting the digital or physical domains of the printing process. These goals are segmented into two primary categories: Denial of Printing Service (DoPS) and Denial of Network Services (DoNS).

3.2.1 Denial of Printing Service (DoPS)

DoPS is accomplished by instigating physical or software disruptions in the printing process. These disruptions may involve causing physical damage to the printer or the printed object, or they can be carried out through software-based attacks aimed at halting or interrupting the printing operation.

a) Physical Damage (PdDoS). PdDoS can be achieved by either physical damage to the printer (**PdTP**) or by causing physical damage to the geometry (**PdTG**). In PdTG, the adversary prints obviously defective or entirely different parts to ensure rejection during post-printing inspection. ‘Print your own grave’ and ‘Incurable’ attacks proposed and demonstrated in Section 5 illustrate these attack categories, respectively. ‘Print your own grave’ prints a tool and uses printing heuristics to damage the printer physically. Conversely, ‘Incurable’ injects observable printing problems in the printed parts to misguide the user into fruitless troubleshooting efforts. These problems include stringing, poor bridging, z-wobble, warping, etc.

b) Software Interruptions (SIDoS). A malicious firmware can initiate a DoS attack without physically damaging the printer or the printed part. For instance, setting the command buffer length to zero, triggering an indefinite sleep mode, or circumventing the core printing instructions within the firmware’s main loop can effectively lead to a denial of printing service attack.

3.2.2 Denial of Network Services (DoNS)

In this category, malicious firmware conducts traditional Denial of Service (DoS) attacks on networked devices. These attacks encompass network or application-level flooding attacks or exploiting vulnerabilities to crash victim processes.

3.3 Integrity Breach (IB)

This category encompasses unconventional attack goals that offer high dividends to the adversary. It is subdivided into two categories based on whether the attack compromises the integrity of the printed object or the printing environment.

3.3.1 Printed Object (PoIB)

Printed object integrity breach attacks introduce subtle defects that may find their way through the quality inspection of the target system. The extent of damage depends on the target system instead of the printer. For instance, hidden defects in a 3D-printed car wheel can lead to serious road accidents. This category is subdivided into three types:

a) Surveillance of Target System (SuTS). SuTS attacks aim to collect the target system information using the printed object. While no attacks in this category have been demonstrated thus far, the literature suggests the feasibility of incorporating some form of spying capability into a 3D-printed object. For example, printing an RFID tag [34] or watermarking [35] could potentially disclose the location of the printed part.

b) Denial of Target System Availability (DeTSA). DeTSA attacks are designed to achieve a denial of service at the target assembly. For example, scaling and axial misalignment attacks, demonstrated in Section 5, are intended to introduce scaling or alignment errors in specific parts, making them

infeasible to assemble them into the target system, thereby resulting in target system unavailability.

c) Sabotage of Target System (SaTS). While service denial is an additional consequence, SaTS attacks are intended to inflict damage on the target system rather than solely causing a denial of service. Researchers have investigated these attacks in pre-firmware stages [14, 36]. The feasibility of SaTS firmware attacks is demonstrated in Section 5.

3.3.2 Printing Environment (PeIB)

Given that the printing environment encompasses digital and physical domains, PeIB is subdivided into two categories.

a) Network Services (NeIB). In NeTB attacks, malicious firmware acts as a rogue network element, launching integrity attacks on the computing devices accessible over available networks.

b) Sabotage of Printing Premises (SaPP). These attacks physically damage the printing premises, encompassing both the facility and personnel. Through malicious firmware, an attacker can circumvent high-temperature safety controls and exploit filament flammability characteristics to cause a fire hazard [37], or raise the volatile organic compounds (VOC) count in the air to potentially increase the risk of respiratory, cardiovascular, and other disorders [38, 39].

3.4 Unauthorized Printing (UP)

Attackers may use malicious firmware to carry out unauthorized printing, such as producing counterfeit goods and manufacturing illegal arms. While inputting static G-code files for these purposes may appear straightforward, the memory limitations present a practical obstacle to launching such attacks using malicious firmware.

4 Threat Model

In today's industrial landscape, if malicious firmware infiltrates an FFF printer, it can easily conceal itself from typical printer control software. Operating covertly and evading detection, malicious firmware can enable attackers to execute their objectives for prolonged periods with minimal risk of discovery. An adversary can install malicious firmware through several methods. For instance, researchers have successfully exploited vulnerabilities in printer control software to gain unauthorized access [40]. Once the control software process is compromised, attackers can exploit the printer's standard upgrade routine to install malicious firmware. Additionally, the supply chain offers another avenue for installing such firmware. Brief physical access to the printer during its supply chain journey or while in operation is sufficient to compromise the firmware. This phase follows established tactics observed in previous firmware attacks, as documented

in various studies [28, 30, 41–44]. In the subsequent stage, malicious firmware exploits potential vulnerabilities to achieve adversarial goals, as detailed in Section 5.

5 Proposed Firmware Attacks

This section presents nine new attacks chosen from the attack categorization tree nodes colored in blue in Figure 2. While the tree encompasses categories related to network surveillance and integrity breaches, we exclusively targeted those emphasizing the specific aspects of the printing process. These attacks are novel at the firmware level, with three previously demonstrated through the manipulation of design files or G-code files. We provide insight into each attack's motivation, the corresponding path within the attack categorization tree, the challenges encountered, the methodology employed, and the outcomes. These attacks are executed on the open-source Marlin firmware, widely utilized in commonly available printers within industrial settings. Specifically, we demonstrated these attacks on the Ultimaker2+ 3D printer.

5.1 Object Geometry (OG) Stealing

Attack motivation. Stealing the design of a competitor's new prototype offers a significant advantage in time, resources, and market positioning. While prior research has explored IP theft attacks by reverse-engineering the emissions in the physical domain during the printing process [26], our approach distinguishes itself by performing it at the firmware level.

Attack categorization. Surveillance → SuPr → SuPO.

Outcome. Stolen geometry of the printed part.

Challenge. Marlin firmware running on an embedded system has limited storage, making it infeasible to save the large G-code files that represent printed objects.

Method. In this attack, malicious firmware records the potentially useful instructions by developing a small engine that efficiently identifies and captures the sketch of the printed object using three approximations: (1) ignoring the complete infill structure, (2) truncating the sub-millimeter part of x,y coordinates, and (3) activating once per mm of z-axis movement. To address the challenge of identifying the outer shell of the printed object, a circular buffer with sufficient length to accommodate the object's vertices is introduced. The shell is printed at the start or end of each layer, and a shell identification algorithm is employed on the ring buffer at the layer-change event.

One byte is adequate for representing each axis position data in binary format for the case study printer, which has printing-bed dimensions of less than $255 \times 255 \text{ mm}^2$. The engine captures the approximate shape of an object using just 256 bytes and stores it in the EEPROM. As our approach focuses on finding vertices, it is independent of the object's

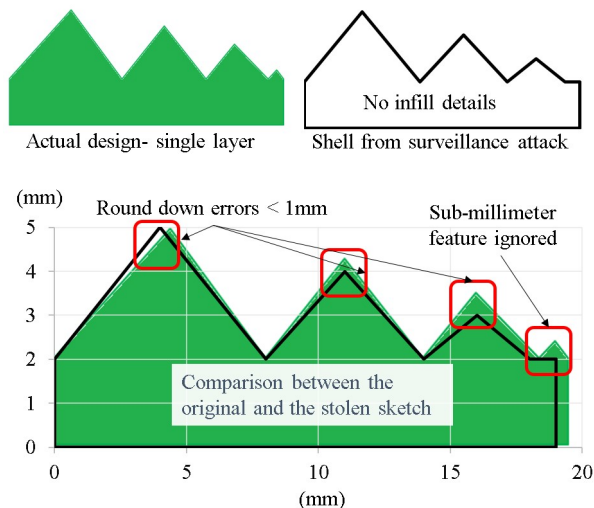


Figure 3: Geometry outline exfiltrated via surveillance attack

size. When an attacker inserts an SD card into the printer, the firmware verifies it and downloads the stolen information within 5 seconds. A variant of this attack can also collect printer hardware configuration information and environment data, such as the ambient temperature, using physical sensors.

Evaluation results. Figure 3 illustrates the results of this attack, showcasing an original design (in green), the stolen outline sketch, and an overlaid image highlighting any approximation errors. Despite the omission of sub-millimeter features, the captured sketch provides valuable information to the adversary regarding the object’s shape and size, utilizing only 256 bytes compared to the original 32 KB. The attack code disregards infill and solely captures vertices, increasing spatial efficiency with the object’s size. For example, a scaled-up version of this object (measuring 10 cm x 2.5 cm x 1 mm) occupies 270 KB of space, while the attacked file still maintains a size of 256 bytes. Furthermore, the percent approximation errors in vertex locations decrease as the object size increases.

5.2 Print Your Own Grave (PYOG) Attack

Attack motivation. Physical damage to the printer is an effective way to cause denial of printing service (DoPS). In addition to service disruption, the attack entails financial losses incurred from replacing the damaged components.

Attack categorization. DoS → DoPS → PdDoS → PdIP.

Outcome. A shattered printing bed glass sheet.

Challenge. The printing glass is secured through retaining clips over the solid metal sheet. The nozzle is the only other part that comes in contact with the glass sheet. Hitting the nozzle with the bed at maximum speed doesn’t provide enough impact to cause damage to the glass.

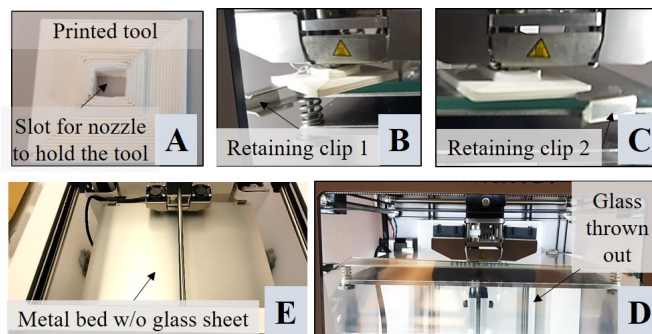


Figure 4: Glass-breaking attack stages

Method. PYOG attack exploits the printing function to damage the printer. The presented version of the attack specifically aims at breaking the printing bed glass sheet by throwing it out of the printer. Exploiting the nonexistence of a hardware protection layer between the printing bed and the nozzle, we initially attempted to break the glass by overriding the firmware checks and hitting the bed against the nozzle. The approach, however, does not provide enough impulsive force to break the glass that resides securely over the metal bed. The malicious firmware addresses this challenge by adopting a more sophisticated strategy. It begins by printing a destruction tool, holding it using the nozzle, allowing it to cool down, and then intelligently scans the edges of the printing bed to compromise the glass sheet retaining clips. Finally, the malicious code pushes the glass from the rear edge to throw it out of the printer.

The attack can be triggered by a specific instruction or by an inactivity period. The attack covers two additional categories during execution. The first category is ‘software interruption,’ achieved by introducing a planned pause and not accepting any printing commands during that time to allow the destruction tool to cool down enough to be detached from the printing bed. The second category is ‘unauthorized printing,’ which is achieved by printing the destruction tool.

Evaluation results. Figure 4 presents a pictorial view of the attack sequence from A to E. The attack utilizes only 20 lines of code to print the tool, normally requiring over 27,000 G-code instructions and more than 500 KB of space. The entire attack code fits well within the available flash memory by only increasing it from 130 KB to 134 KB.

5.3 Incurable: Printing Faults Impersonation

Attack motivation. Troubleshooting cyber-physical systems, particularly 3D printers, is a laborious and time-consuming task. The rectification and optimization of system configurations necessitate extensive verification through actual printing operations. This motivates us to introduce attacks that mimic known and obvious faulty behaviors, misleading users into common printing problems and wasting valuable time and

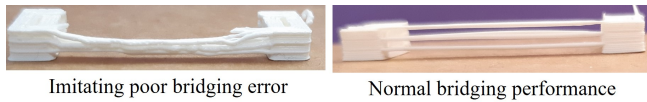


Figure 5: Bridging error imitation attack

effort in futile troubleshooting.

Attack categorization. DoS → DoPS → PdDoS → PDtG.

Outcome. False impression of the poor bridging problem.

Challenge. Real-time bridge identification in print geometry.

Method. This attack exhibits a poor bridging problem, which tests a printer’s ability to extrude filament between two raised points without sagging. An extrusion instruction from $A_{x,y}$ to $B_{x,y}$ in i^{th} layer will belong to a bridge if there is no extrusion between $A_{x,y}$ and $B_{x,y}$ in $(i-1)^{th}$ layer. To identify a bridge, the attacker must maintain spatial information of the current and previous layers. The attacker cannot analyze and map detailed printing instructions on a compute-constraint system to ensure uninterrupted printing. Hence, the attacker uses a coarse representation of a $100 \times 100 \text{ mm}^2$ targeted zone by only a 5×5 elements array (named layer-map), where each element represents a square of $20 \times 20 \text{ mm}^2$. The bridging performance is typically evaluated over 20 mm and beyond [45]. When a move instruction is received, the layer map is updated, and once a layer is completely printed, it is saved to identify any bridges in the next layer. For each move instruction, the attacker checks if there is any extrusion at the corresponding location in the previous layer. The move instruction is categorized as part of a bridge if there is no extrusion. To create poor bridging performance, the attacker modifies and uses permutation of multiple parameters, including slowing down the cooling fan, increasing the extrusion amount, and reducing printing speed.

Evaluation results. We evaluated the attack by printing a shape with three bridges across 25 mm apart pillars, with each bridge added five layers above the previous one. As shown in Figure 5, the attack successfully imitated poor bridging performance. The sag visible on the 25 mm gap between the pillars might mislead users into attributing the poor bridging issue to inefficient printing settings.

5.4 Object Feature (OF) Scaling

Attack motivation. 3D printing is increasingly used to manufacture critical components for larger assemblies, like turbine blades [46]. If a sub-component of a replacement part is slightly scaled up or down during printing, it will not fit in the assembly, resulting in a delay in service of the target system.

Attack categorization. IB → PoIB → DeTSA.

Outcome. The attack slightly modifies the dimensions to deny fitment of the printed part.

Challenge. This attack goal can be easily accomplished at the

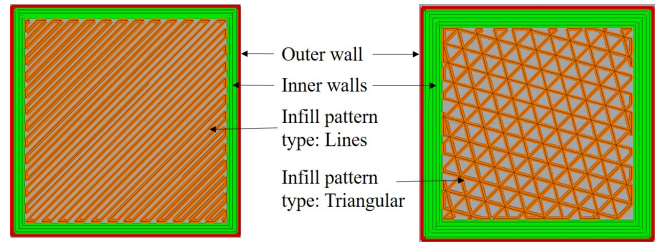


Figure 6: Internal layers composition

designing or the slicing stage using the ‘scaling’ switch in the software. On the contrary, scaling each printing instruction at the firmware level leads to the scaling of tiny segments connecting consecutive extrudates, which exposes the attack (see Figure 7). Scaling requires additional instructions, and since the firmware receives printing instructions in a temporal sequence, it cannot plan for scaling while printing. Consequently, achieving perfect real-time scaling at the firmware level is not feasible.

Method. We exploit the printing format used by slicer software to execute a scaling attack through firmware. A single layer comprises the outer wall structure, the infill pattern for intermediate layers, and skin for the outer layer. Figure 6 shows two infill patterns encapsulated by varying numbers of walls. The outer walls mark the object’s edges and create a directed cycle where the destination coordinates for a move instruction are repeated after ‘k’ instructions (where ‘k’ represents the number of edges in the object). The wall structure is printed adjacent to the layer change event.

Due to memory constraints, tracking the destination coordinates of all move instructions is not feasible. To overcome this problem, the attacker creates a circular buffer containing one more entry than the maximum number of edges in the anticipated polygon. The firmware searches for a directed cycle to identify a geometrical feature and builds an extra wall around it. Under generic printing settings, wall thickness is proportional to the nozzle diameter, which implies a 0.8 mm to 1.2 mm difference in dimensions across the two opposite walls for 0.4 mm and 0.6 mm nozzles. The attacker uses the change-of-layer instruction to manage the limited computation power to trigger the polygon identification routine. Once the polygon is identified, the attacker selects appropriate coordinates outside the object and prints a new one by adjusting the sequence of the coordinates in the identified cycle.

Evaluation results. A rectangular prism with dual sizes was printed to assess the attack’s impact. Figure 7 shows a visual comparison between original and attacked samples. While no discernible alterations are evident in the infill structure, the attacked sample exhibits additional walls. We measured the distance between opposite edges at five distinct locations to analyze the dimensional changes. The results indicate an average increase of $0.96 \text{ mm} \pm 0.25 \text{ mm}$ for each dimension.

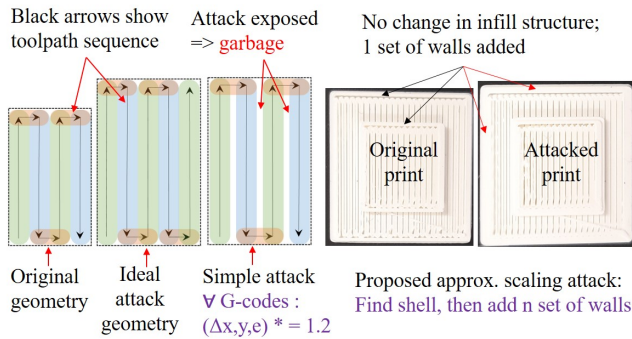


Figure 7: 2-dimensional object feature scaling attack

5.5 Axial Misalignment

Attack motivation. The motivation behind this attack is similar to the scaling attack. However, instead of altering the object’s dimensions (which are relatively easier to measure), this attack deliberately misaligns a coupling feature over the printing axis to prevent the part from fitting correctly in the target assembly.

Attack categorization. IB → PoIB → DeTSA.

Outcome. The outcome is an axial misalignment of the coupling slot to deny fitment.

Challenge. In addition to the challenges mentioned in the scaling attack, identifying a feature that will ultimately become a coupling candidate, such as a slot, stud, etc., is also challenging.

Method. To overcome this challenge, the G-code execution pipeline is delayed by k_{max} printing instructions to ensure that the attack circular buffer as described in Section 5.4 is filled before the printing starts. The malicious firmware searches for a directed cycle within a layer. The temporal distance of the identified cycle from the layer-change event and the length of the constituent lines distinguish between the directed cycles representing a fitment feature and the outer wall structure. Once a change-of-layer event occurs, the x or y coordinates of

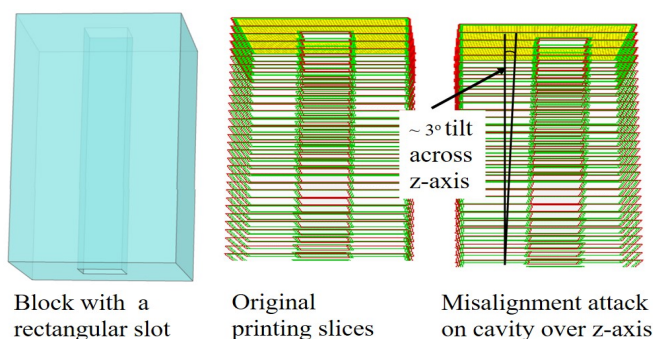


Figure 8: Geometric feature (coupling slot) misalignment attack

all vertices of the directed cycle are modified proportionately to the z-axis value to achieve a continuous drift in the feature. This attack targets objects with precise fitment requirements, like driving shafts, assemblies, nuts, and bolts.

Evaluation results. We implemented this attack on a rectangular female square-fitting slot that couples with a male driving shaft. As presented in Figure 8, the attack introduces a 3° axial shift, leading to coupling issues with the male shaft. Unlike Section 5.4, this attack achieves the goal without increasing the number of printing instructions. A variant of this attack only relocates a single vertex of a coupling feature.

5.6 Internal Cavity Attack

Attack motivation. Reducing an object’s strength through hidden cavities is a well-known concern during the design [14] and slicing stages [17]. However, achieving a similar result through malicious firmware remains uncharted territory. The innovation of this approach lies in its execution via firmware. Unlike the digital outputs of the design and slicing stages, the firmware’s output during the printing stage is a physical object and is not amenable to standard digital integrity checks. Therefore, the cavity attacks at the firmware stage pose greater risks than those at earlier stages, highlighting the need to investigate firmware-based cavity attacks.

Attack categorization. IB → PoIB → SaTS.

Outcome. Inducing an internal cavity in the printed part.

Challenge. A critical consideration for the success of SaTS attacks is stealthiness. If the attack is exposed, it becomes a DoPS attack. It implies that the cavity should only exist within the internal layers. Deciding on the location and number of layers to induce cavities is an additional challenge.

Assumption. This attack is predicated on the assumption that the target object exhibits symmetry along the z-axis. This assumption is valid for ASTM tensile and flexure test models and generally holds for most real-world objects, at least for specific segments of the layer structure.

Method. The malicious firmware initially determines the number of G-code instructions in a layer. In the second step, firmware utilizes the G-code instruction ‘M73’ to identify the candidate internal layers for the attack. In the absence of an ‘M73’ response, a backup heuristic rule can be used to estimate the number of layers in the object using the standard span-to-thickness ratio of 16:1 recommended by ASTM International Standard [47]. To ensure the cavity remains concealed from the sides within each layer, the attacker splits the instruction into three parts and only stops the filament motor for the central part. The attack ceases once the internal layers are complete, and the printer resumes producing the unaltered top layers.

Evaluation results. To evaluate the attack, we printed two ASTM-compliant tensile bars. Figure 9 illustrates the cavity



Figure 9: Cavity attack specimen during and after print

Sample type	Peak load (N)		Peak stress (N)	
	Avg of 6 samples	Std. dev	Avg of 6 samples	Std. dev
Original	498.44	39.65	15.38	1.17
Attacked	419.49	24.54	12.67	0.75
Difference	78.96		2.72	
% Reduction	15.84		17.66	

Table 1: Tensile test results for filament density attacks

in the left image after pausing the printing process. The top and bottom layers finally cover the cavity.

5.7 Object Density Variation Attack

Attack motivation. While the cavity in the attack presented in Section 5.6 gets obfuscated in the final object, it is visible during the printing. A more stealthy way to achieve SaTS attack is to reduce the part’s density at a critical location. Like cavity, the density variation attack has also been studied at the slicing stage [17]. Hence, the novelty is in its implementation through malicious firmware. Although researchers have achieved generic density variation by attacking the printer [30], our attack is localized with a reduced footprint and improved results.

Attack categorization. IB → PoIB → SaTS.

Outcome. Reduced object density at a targeted location.

Assumptions and Challenges. This attack has the same challenges and the set of assumptions detailed in Section 5.6.

Method. The attack preparation steps are the same as those described in Section 5.6, with two changes. Firstly, the zone of interest is increased from one target infill line to a group of lines. Secondly, the retract instructions required for a clean cavity are not included. Instead, the attack manipulates the extruder and filament speed ratio.

Evaluation results. We printed six ASTM-compliant tensile bars using the original and attacked firmware and observed no visual or dimensional differences between the two sets of prints. Tensile tests were subsequently conducted using the MTS Insight 30 machine, with the results presented in Table 1. The attacked samples show a 15.84% and 17.66% reduction in the peak tensile load and stress values, respectively.

5.8 Filament Erosion Attack

Attack motivation. FFF printers estimate the filament quantity using the steps of the stepper motor and the configured filament diameter value. If the filament is partly eroded, the printer will extrude less filament in that region. This motivates

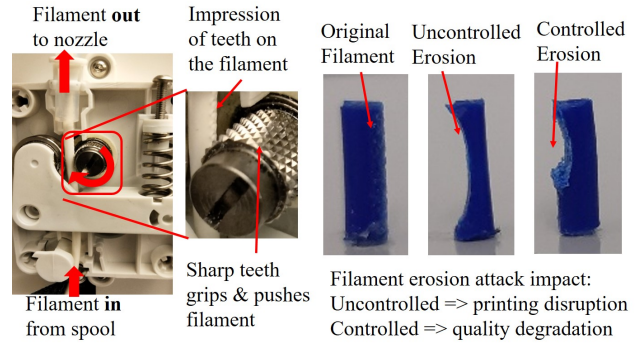


Figure 10: Filament erosion attack

us to present a new SaTS attack that erodes the filament to reduce the part density.

Attack categorization. IB → PoIB → SaTS.

Outcome. Reduced density of the printed parts.

Challenge. While filament erosion may occasionally occur during normal printing operations due to specific routine faults, such as a clogged nozzle, deliberate induction of this phenomenon is not supported by any instructions or functions. If erosion goes beyond a certain point, the printer may not push the filament further, resulting in a denial of service. The challenge lies in creating an erosion function that achieves maximum erosion while maintaining the printer’s ability to continue normal printing operations.

Method. While the desired outcome is similar to that in Section 5.7, this attack employs an indirect method. In most FFF printers, the extruder motor’s shaft teeth grip the filament with the support of a free-rotating roller (see Figure 10). The teeth push the filament axially towards the heated nozzle as the motor rotates. In this attack, a portion of the filament is eroded as it passes through the feeding chamber, reducing the filament quantity at the point of attack. When the defected (eroded) filament portion passes through the nozzle, it creates low-density zones in the printed object. A carefully planned filament erosion attack can thus lower the material density at a critical region, reducing the strength of the printed part. Figure 10 illustrates an example of a filament erosion attack.

The attack uses two methods to erode the filament. The first method involves compelling cold extrusion through the nozzle. Due to the filament diameter being larger than the nozzle orifice, the force exerted by the teeth on the solid filament is insufficient for extrusion through the nozzle. Instead of advancing it, the rotating wheel’s teeth merely chip the filament off. The attack bypasses the firmware test routine for the minimum temperature required for extrusion. The second method uses a burst of high-jerk oscillatory movements to break the grooves formed by the gear pressure, resulting in filament erosion. The second method causes less erosion but still achieves the defective printing goal.

Evaluation results. We evaluated the effectiveness of the

erosion attack using two attack instances. With PLA filament of 2.85 ± 0.1 mm diameter, we observed that a cold extrusion motion beyond 1 sec reduced weight from 0.077g to 0.049g, representing a 36% reduction but also interrupting regular operation. Consequently, such an attack could only cause a denial of service. If the attack lasts up to 0.5 secs, the material reduction is up to 20% while regular operation continues, ensuring the required stealthiness to achieve a SaTS attack. The attack employing high-jerk oscillatory move instructions requires additional time to induce material reduction. Conversely, the second method does not necessitate a waiting period for filament cooling, thus offering greater operational flexibility. The second attack resulted in a 15% reduction, with the equivalent length of filament weighing 0.065g.

5.9 Printing Facility Air-quality

Attack motivation. Given their cyber-physical nature, 3D printers not only facilitate innovative manufacturing processes but also have the potential to negatively impact the physical environment. This reality motivates our investigation into contamination attacks targeting the printer’s surroundings.

Attack categorization. IB → PeIB → SaPP.

Outcome. Poor air quality at the printing facility.

Method. This attack compromises the air quality in a printing facility by increasing the emission of microparticles and volatile organic compounds (VOCs) through two malicious actions. Initially, it searches for idle state to initiate cold scrubbing bursts at high speeds, which chip fine particles from the filament. Subsequently, the attack manipulates the printer by turning on the nozzle heater while disabling the temperature feedback control circuit. This action leads to the unregulated emission of fumes, VOCs, and microparticles. Due to the elevated temperatures, low-density fluid drips from the nozzle, depositing suspicious droplets on the printing bed. To evade detection, the attacker retracts the filament after leaving only a minimal amount in the chamber and then raises the temperature, which also serves to shorten the attack duration. These actions threaten the environmental health and safety conditions within the facility. Furthermore, the attack may remain undetected with odorless filaments, resulting in prolonged exposure and potential health consequences for the workers.

Evaluation results. To evaluate the attack’s impact, we conducted experiments to measure the particles and VOC count before and after the attack. Specifically, measurements were taken 5 minutes before the attack and 5 minutes, 30 minutes, and 1 hour after each attack instance. The experiment was repeated five times. We conducted the attacks in a well-ventilated environment with minimum human interaction. Furthermore, no personnel were present in the lab facility to avoid potentially hazardous circumstances.

The findings from these experiments are depicted in Figure 11. The data reveal a significant increase in VOCs from 6

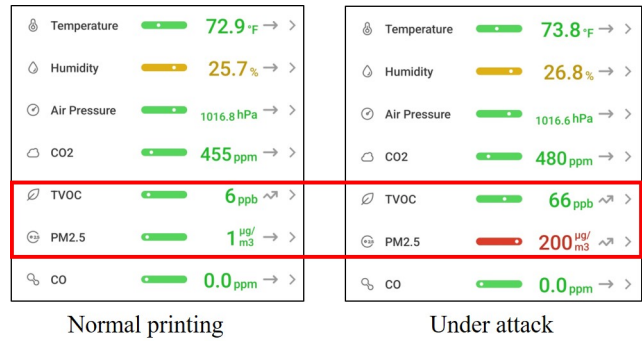


Figure 11: Air quality stats for facility contamination attack

parts per billion (ppb) to 66 ppb, and particulate matter of 2.5 micron or smaller diameter ($PM_{2.5}$) values surged from $1 \mu\text{g}/\text{m}^3$ to $200 \mu\text{g}/\text{m}^3$, exceeding the safe limit of $10 \mu\text{g}/\text{m}^3$ [48]. Due to high ventilation, the recorded values at 30 minutes and one hour after the attack were within normal ranges.

6 Attacks Feasibility/Complexity Analysis

6.1 Analysis Methodology

Motivation. The attacks described in Section 5 vary significantly in the workload requirements. Depending on the process’s stage, the feasibility of initiating an attack can range from trivial to infeasible. For example, object scaling is straightforward at the design stage but becomes complex at the firmware stage. Conversely, thermodynamic attacks are easier to execute through malicious firmware but are unfeasible at the design stage. If an attack is not viable at a particular stage, there is no benefit in implementing defensive measures against it. This prompts us to undertake a comprehensive feasibility analysis of the attack goals (Figure 2) throughout all stages of the printing process chain.

Methodology. We begin our analysis by identifying various independent stages in the printing process that attackers could target. We then develop a comprehensive set of feasibility criteria to assign feasibility scores to each of the 48 existing and proposed attacks presented in Table 2. We analyze the proposed attacks using the data presented in Section 5 and draw upon relevant results and findings from the literature on similar attacks.

Printing process stages. We analyze the printing process to delineate the independent stages vulnerable to attacks, as illustrated in Figure 12. We treat stages 1a and 1b as a single stage because an attacker who captures the 3D model file (1b) can execute the same attacks by compromising the design software (1a). In contrast, the slicer software (2a), printing profile (2b), and G-code file (2c) each offer distinct capabilities to an attacker, hence considered as distinct stages.

Feasibility and complexity criteria. This study employs two factors to evaluate the feasibility of achieving attack goals.

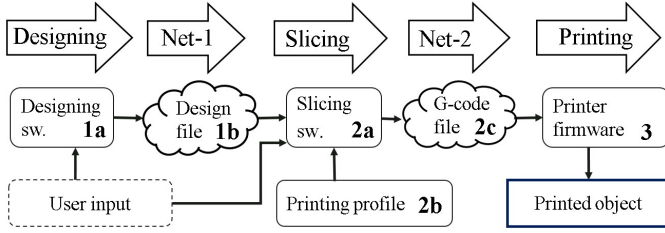


Figure 12: Stages in AM process highlighting cyber artifacts compromisable through a cyberattack

The first factor, f_1 , assesses the ability to ascertain whether an attack conforms to its intended objective at a specific stage. The second factor, f_2 , considers the availability of necessary methods to execute attack actions.

For example, at the design stage, the absence of tools to modify the thermal profile in the design file renders dynamic-thermal attacks [17] unfeasible, resulting in an f_2 score of zero. In contrast, a simple command can alter the thermal profile at the G-code stage, typically warranting the highest f_2 score. However, the firmware’s limited temporal perspective complicates the precise placement of the attack, resulting in a low f_2 score for these attacks at this stage.

At any particular stage, an attack is considered as:

- an infeasible attack if no execution mechanism or means of confirming compliance with the attack criteria exists.
- a high difficulty attack (●) if both the factors are not readily available and require additional effort to estimate or calculate them.
- a medium-difficulty attack (◐) if one but not both the factors are readily available
- a low-difficulty attack (○) if both the factors are readily available

The feasibility score of the n^{th} attack at the m^{th} stage, $FS_{n,m}$, is defined in Eq. 1 as the product of $f_{1,n,m}$ and $f_{2,n,m}$, where f_1 and f_2 are assigned the values of 0, 1, and 2 for ‘not available,’ ‘not readily available,’ and ‘readily available’ respectively.

$$FS_{n,m} = \begin{cases} \text{Infeasible} & \text{if } f_{1,n,m} \times f_{2,n,m} = 0 \\ \text{High difficulty} & \text{if } f_{1,n,m} \times f_{2,n,m} = 1 \\ \text{Medium difficulty} & \text{if } f_{1,n,m} \times f_{2,n,m} = 2 \\ \text{Low difficulty} & \text{if } f_{1,n,m} \times f_{2,n,m} = 4 \end{cases} \quad (1)$$

6.2 Attack Analysis

Table 2 outlines the attack actions, literature reference, their type in the attack categorization tree, and the feasibility status in light of the above-mentioned criteria. Due to space constraints, we collectively discuss them under the following subsections. Where, $A_1 - A_{48}$ in the discussion refers to the attack serial number in the table. A short description of each attack is provided in the Table 4 (Appendix A).

Designing stage. The designing stage focuses on the geometry of the desired object. The fitment attacks for DeTSA ($A_{25} - A_{27}$), anisotropy attacks (A_{45}), and geometric feature insertion or removal (A_{42}) for SaTS are the simplest and most accurate at the designing stage. Surveillance attacks on the printed object (A_4) and the connected network (A_6) are also feasible if the attacker has access to the designing software process.

Slicing and control software. With an STL file and printing profile as the input and a G-code file as the output, the stage offers a vast spectrum of attack opportunities. It outperforms all other stages in achieving SaTS goals ($A_{29} - A_{47}$). However, the stage is less effective for DeTSA and PDtG attack goals.

Printing profile. The printing profile comprises parameters used by the slicer software to attain a set of printing instructions. It can easily launch attacks related to global parameter settings ($A_{17}, A_{34}, A_{35}, A_{37}, A_{38}$).

G-code file through Net-2. The chronological structure of a G-code file suits the introduction of localized defects to achieve most of the DeTSA ($A_{25} - A_{27}$), SaTS ($A_{29} - A_{47}$), and PDtG ($A_{14} - A_{23}$) attacks. Infill pattern and density attacks (A_{34}, A_{35}), however, are challenging to execute.

Firmware. For incurring damage to the printer and facility ($A_9 - A_{13}, A_{48}$), and most of the other PDtG ($A_{14} - A_{23}$), the firmware stage leads all other stages. However, firmware is the second-best stage to launch the DeTSA and SaTS attacks, following the slicer. One reason is the difficulty in achieving the required stealthiness and accuracy due to its limited temporal view.

6.3 Attack Feasibility Index - AFI

To assess the feasibility of attack categories at different stages of the printing process, we introduce the term ‘Attack Goal Feasibility Index’ (AFI), ranging from 0 to 1. An AFI value of 0 indicates that an attack goal is not feasible at a particular stage. In contrast, an AFI value of 1 indicates low difficulty as defined in Section 6.1. The index incorporates the cumulative effect of all attacks in a particular category and is calculated as follows:

$$AFI_{g,s} = \frac{1}{n \times F_{max}} \sum_{i=1}^n FS_{i,s} \quad (2)$$

where, $AFI_{g,s}$ is the AFI value for the attack category g at stage s . n is the total number of attacks in the category g , F_{max} is the numeric value ‘4’ assigned to the ‘low-difficulty’ level, and $FS_{i,s}$ represents the feasibility score of i^{th} attack at stage s .

Table 3 displays the Attack Feasibility Index (AFI) for the examined attacks, broken down by the stages (Figure 12). Only the slicing software (2a) and the firmware (3) stage demonstrate non-zero AFI values across all attack goals. The normalized AFI value for SaTS attacks is 0.89 for Stage 2a

Sr. No.	Attack Name	Ref.	Attack goal category	Designing (1a/1b)	Slicing control software (2a)	Printing profile (2b)	Net-2 (G-code file) (2c)	Firmware (3)
1	Printer info	[49]	SuPP	-	●	-	●	○
2	Design SW info	[50]		○	●	-	-	-
3	Slicer/Control info	[50]		●	○	-	●	●
4	OG info	[25, 26], P* (5.1)	SuPO	○	○	-	○	●
5	Print profile info	[27]		-	○	○	○	●
6	Network device	[32]	SuNT	○	○	-	○	○
7	Process artefacts	[33]		●	●	-	●	●
8	Facility info	P	SuPh	-	-	-	-	●
9	PYOG	P* (5.2)	PDtP	-	●	-	●	●
10	Breaking limits	P		-	-	-	-	○
11	Nz impair	P		-	-	-	-	●
12	Extruder fracture	[28]		-	-	-	-	●
13	Nz burning	P		-	-	-	-	●
14	OG scaling	P* (5.4)	PDtG	○	○	○	○	○
15	OG thermal	[28]		-	○	○	○	○
16	Incurable	P* (5.3)		-	●	●	●	●
17	Warping	[17]		-	○	○	○	○
18	FK thermal	[28]		-	○	-	●	○
19	Trajectory unsync	[51], [28]		-	●	●	●	●
20	PS profile	P		-	-	-	-	○
21	PS unsync	P		-	-	-	-	○
22	FK reduction	[30]		-	○	-	○	○
23	Clogging	[28]		-	●	●	●	●
24	MAC/ARP corruption	[50]	SIDoS	-	○	-	-	○
25	Vertex relocation	[52]	DeTSA	○	●	-	●	●
26	OF scaling	P* (5.4)		○	-	-	-	-
27	Fitment	P* (5.5)		○	-	-	●	●
28	NT manipulation	[49]	NeIB	○	○	-	○	●
29	IF line spacing	[15]	SaTS	-	○	-	○	●
30	IF vertex spacing	[15]		-	○	-	○	●
31	FS manipulation	[17], [40]		-	○	-	○	○
32	FK cavity	[17], P* (5.6)		-	○	-	○	●
33	FK density	[17], P* (5.7)		-	○	-	○	●
34	IF pattern	[53]		-	○	○	●	-
35	IF density	[53]		-	○	○	●	-
36	PS cavity	[17]		-	●	-	●	●
37	IF exclusion	[40], [30]		-	○	○	○	○
38	% IF	[54]		-	○	○	●	●
39	GF change	[29], [51]		-	○	-	○	●
40	Dyn.Thermal	[17], [54]		-	○	-	○	●
41	PS local	[54]		-	○	-	○	○
42	OF insert/remove	[14], [55]		○	-	-	-	-
43	LT local	[54]		-	○	-	○	○
44	GC sequence	[54], [56]		-	○	-	○	○
45	Anisotropy	[55]		○	○	-	●	-
46	GC manipulation	[54], [56]		-	○	-	○	○
47	Erosion	P* (5.8)	-	●	-	●	●	
48	PF air quality	P* (5.9)	SaPP	-	●	-	-	○

- : Not feasible ● : High difficulty ○ : Low difficulty P/P* : Proposed/Proposed and Evaluated

Table 2: Categorization of existing and proposed attacks with their feasibility and difficulty at various stages

Attack Goal Category	Normalized attack feasibility index for process stages				
	1.	2a.	2b.	2c.	3.
SuPr	0.5	0.8	0.2	0.5	0.45
SuPE	0.25	0.25	0	0.25	0.5
PDtP	0	0.04	0	0.04	0.57
PDtG	0.5	1	1	1	1
SIDoS	0	0.88	0	0.88	1
DoNS	1	1	0	1	1
DeTSA	1	0.25	0	0.25	0.25
SaTS	0.25	0.89	0.25	0.61	0.56
NeIB	1	1	0	1	0.5
SaPP	0	0.38	0	0	1
UP	0.25	1	0	1	0.25
Cumulative AFI per process stage	0.43	0.68	0.13	0.59	0.64

1 : Designing 2a : Slicing software 2b : Printing profile
2c : G-code file 3: Firmware

Table 3: Stage-wise feasibility summary for attack goals

and 0.56 for Stage 3 (firmware). The normalized AFI value for DoPS is 0.43 for Stage 2a and 0.79 for Stage 3. The Cumulative AFI indicates almost equal feasibility at the slicing (2a) and firmware (3) stages, with the print profile being the least feasible stage (2b) for launching an attack.

7 Firmware Attack Countermeasures

To fortify 3D printing systems against firmware attacks, a robust, integrated approach encompassing prevention, detection, and mitigation strategies is essential. Primary vectors for firmware attacks include compromises in the supply chain, unauthorized physical access during the operational phase, and intrusions into the printer network and the trusted printer control software. Supply chain security is a comprehensive field that involves safeguards to protect against breaches of trust at any stage. Some of these measures include using encryption and blockchain technology to ensure data integrity and prevent counterfeiting [57]. To overcome the computational constraints of 3D printers, lightweight cryptographic solutions such as Elliptic Curve Cryptography (ECC) and the Elliptic Curve Digital Signature Algorithm (ECDSA) provide suitable encryption and authentication services [58]. To thwart Man-in-the-Middle (MiTM) attacks, secure communication protocols, like TLS, should be implemented to encrypt data exchanges involving the printer. The printer control software, endowed with high-level privileges like firmware installation, must be fortified with robust authentication and authorization measures. To prevent booting from or upgrading to malicious firmware, industry-standard techniques such as cryptographic signing of firmware files, secure boot mechanisms, and the integration of hardware-based security modules like TPM (Trusted Platform Module) or HSM (Hardware Security Module) should be implemented.

A reliable firmware acquisition and subsequent static analysis can help identify malicious code. A hardware-based firmware acquisition method utilizing debugging ports, such as JTAG, effectively bypasses many upper-level deception tactics employed by attackers to evade detection [59]. As a scalable alternative to static analysis, a future direction could involve examining running firmware through cyber-physical fuzzing. The solution would monitor the printer’s state in response to smartly generated application-layer probes (G-codes) in a closed loop to promptly expose any malicious behavior within the firmware.

Should an attacker circumvent these preventative measures, additional safeguards can prevent malicious firmware from fulfilling its intended goals. A signature-based anomaly detection solution would be beneficial for detecting malicious firmware behavior. A more comprehensive approach involves a cyber-physical anomaly detection system that analyzes both physical-operational data such as acoustic, electric current, and magnetic fields [53, 60, 61] and digital domain data such as network traffic and application logs. This system can utilize heuristics or machine learning techniques to identify attack signatures and behavioral anomalies. Another forward-looking strategy involves integrating quality control measures into the cybersecurity loop. Since physical processes are inherently imperfect, resulting in low-magnitude deviations, it is crucial to differentiate between benign and harmful deviations. To this end, implementing feasible versions of standard quality control processes, such as real-time micro CT scanning, could enhance the anomaly detection capabilities [62].

These multi-layered strategies will significantly enhance the defense of 3D printing setups against the continually evolving landscape of cyberattacks.

8 Conclusion

This study presents a novel approach to understanding and classifying firmware attacks in additive manufacturing. We propose a firmware attack classification tree focused on attack goals rather than attack actions. Additionally, nine attacks on Marlin firmware are demonstrated on the Ultimaker2+ 3D printer. Through a series of destructive and non-destructive tests, including tensile strength and air-quality testing, we confirm the effectiveness of these attacks. To analyze the attacks, we introduce an Attack Feasibility Index (AFI), representing a feasibility score for an attack at a specific stage of the printing process. An analysis of 48 attacks, including existing and proposed ones, confirms that all attack goals could not be achieved by attacking any single stage of the printing process. We observe that firmware is not the optimal stage to launch attacks aimed at sabotaging the printed part. This study will inspire further research into additive manufacturing attacks and guide cybersecurity researchers in developing defense solutions tailored to specific stages of the printing process and their corresponding feasible attacks.

References

- [1] KBV Research, “Share & industry trends analysis report by type, by technology, by sales channel, by end-use, by regional outlook and forecast, 2021–2027,” *ReportLinker: Lyon, France*, 2022.
- [2] GE Aviation. (2018) New manufacturing milestone: 30,000 additive fuel nozzles. Online Available: <https://www.ge.com/additive/stories/new-manufacturing-milestone-30000-additive-fuel-nozzles>.
- [3] N. Gupta, C. Weber, and S. Newsome, “Additive manufacturing: status and opportunities,” *Science and Technology Policy Institute, Washington*, 2012.
- [4] S. R. Chhetri, S. Faezi, N. Rashid, and M. A. Al Faruque, “Manufacturing supply chain and product lifecycle security in the era of industry 4.0,” *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 51–68, 2018.
- [5] A. Muhammad, B. Afzal, B. Imran, A. Tanwir, A. H. Akbar, and G. Shah, “onem2m architecture based secure mqtt binding in mbed os,” in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2019, pp. 48–56.
- [6] A. Ayub, N. Zubair, H. Yoo, W. Jo, and I. Ahmed, “Gadgets of gadgets in industrial control systems: Return oriented programming attacks on plcs,” in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2023, pp. 215–226.
- [7] M. Ahsan and M. Ali, “Lsstk: Lightweight solution to preventing stack from buffer overflow vulnerability,” in *2023 17th International Conference on Open Source Systems and Technologies (ICOSST)*, 2023, pp. 1–7.
- [8] S. A. Qasim, A. Ayub, J. Johnson, and I. Ahmed, “Attacking the iec 61131 logic engine in programmable logic controllers,” in *Critical Infrastructure Protection XV: 15th IFIP WG 11.10 International Conference, IC-CIP 2021, Virtual Event, March 15–16, 2021, Revised Selected Papers 15*. Springer, 2022, pp. 73–95.
- [9] B. Imran, M. Ahsan, A. H. Akbar, and G. A. Shah, “D4gw: Dtls for gateway multiplexed application to secure mqtt(sn)-based pub/sub architecture,” *Internet of Things*, vol. 26, p. 101172, 2024.
- [10] B. Imran, B. Afzal, A. H. Akbar, M. Ahsan, and G. A. Shah, “Misa: Minimalist implementation of onem2m security architecture for constrained iot devices,” in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.
- [11] A. Ayub, W. Jo, S. A. Qasim, and I. Ahmed, “How are industrial control systems insecure by design? a deeper insight into real-world programmable logic controllers,” *IEEE Security & Privacy*, vol. 21, no. 4, pp. 10–19, 2023.
- [12] J. Gatlin, S. Belikovetsky, Y. Elovici, A. Skjellum, J. Lubell, P. Witherell, and M. Yampolskiy, “Encryption is futile: Reconstructing 3d-printed models using the power side-channel,” ser. RAID ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 135–147.
- [13] M. A. Al Faruque, S. R. Chhetri, A. Canedo, and J. Wan, “Acoustic side-channel attacks on additive manufacturing systems,” in *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPs)*, 2016, pp. 1–10.
- [14] S. Belikovetsky, M. Yampolskiy, J. Toh, J. Gatlin, and Y. Elovici, “dr0wned–cyber-physical attack with additive manufacturing,” in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017.
- [15] M. H. Rais, M. Ahsan, V. Sharma, R. Barua, R. Prins, and I. Ahmed, “Low-magnitude infill structure manipulation attacks on fff-based 3d printers,” in *Critical Infrastructure Protection XVI*. Springer, 2022, pp. 205–232.
- [16] N. Gupta, A. Tiwari, S. T. Bukkapatnam, and R. Karri, “Additive manufacturing cyber-physical system: Supply chain cybersecurity and risks,” *IEEE Access*, vol. 8, pp. 47 322–47 333, 2020.
- [17] M. H. Rais, Y. Li, and I. Ahmed, “Dynamic-thermal and localized filament-kinetic attacks on fused filament fabrication based 3d printing process,” *Additive Manufacturing*, p. 102200, 2021.
- [18] B. Jovanović, I. Gadjanski, J. Burazer, L. Nikolić, N. Babić, and M. Lečić, “R&d in a fab lab: Examples of paste extrusion method,” in *Proceedings of 5th International Conference on Advanced Manufacturing Engineering and Technologies*, V. Majstorovic and Z. Jakovljevic, Eds. Cham: Springer International Publishing, 2017, pp. 461–467.
- [19] ISO/ASTM 52900:2021, “Additive manufacturing — General principles — Fundamentals and vocabulary,” *ASTM International, West Conshohocken, PA*, 2021. [Online]. Available: <https://www.iso.org/standard/74514.html>
- [20] M. Yampolskiy, W. E. King, J. Gatlin, S. Belikovetsky, A. Brown, A. Skjellum, and Y. Elovici, “Security of additive manufacturing: Attack taxonomy and survey,” *Additive Manufacturing*, vol. 21, pp. 431–457, 2018.

- [21] M. Yampolskiy, A. Skjellum, M. Kretzschmar, R. A. Overfelt, K. R. Sloan, and A. Yasinsac, "Using 3d printers as weapons," *International Journal of Critical Infrastructure Protection*, vol. 14, pp. 58–71, 2016.
- [22] Y. Pan, J. White, D. Schmidt, A. Elhabashy, L. Sturm, J. Camelio, and C. Williams, "Taxonomies for reasoning about cyber-physical attacks in iot-based manufacturing systems," *International Journal of Interactive Multimedia and Artificial Intelligence*, 2017.
- [23] P. Mahesh, A. Tiwari, C. Jin, P. R. Kumar, A. N. Reddy, S. T. Bukkapatnam, N. Gupta, and R. Karri, "A survey of cybersecurity of digital manufacturing," *Proceedings of the IEEE*, vol. 109, no. 4, pp. 495–516, 2020.
- [24] M. Wu and Y. B. Moon, "Taxonomy of cross-domain attacks on cybermanufacturing system," *Procedia Computer Science*, vol. 114, pp. 367–374, 2017.
- [25] C. Song, F. Lin, Z. Ba, K. Ren, C. Zhou, and W. Xu, "My smartphone knows what you print: Exploring smartphone-based side-channel attacks against 3d printers," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 895–907.
- [26] M. A. Al Faruque, S. R. Chhetri, A. Canedo, and J. Wan, "Acoustic side-channel attacks on additive manufacturing systems," in *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, 2016, pp. 1–10.
- [27] Q. Do, B. Martini, and K.-K. R. Choo, "A data exfiltration and remote exploitation attack on consumer 3d printers," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 10, pp. 2174–2186, 2016.
- [28] X. Z. Hang, "Three demos of attacking arduino and rebrap 3d printers, code to Keynote at XCon2013 (2013)," <https://github.com/secmobi/attack-arduino-and-rebrap>, 2016.
- [29] S. B. Moore, W. B. Glisson, and M. Yampolskiy, "Implications of malicious 3d printer firmware," in *Proceedings of Hawaii Int. Conf. Syst. Sci., 2017*, 2017, pp. 1–10.
- [30] H. Pearce, K. Yanamandra, N. Gupta, and R. Karri, "Flaw3d: A trojan-based cyber attack on the physical outcomes of additive manufacturing," *IEEE/ASME Transactions on Mechatronics*, vol. 27, no. 6, pp. 5361–5370, 2022.
- [31] Q. Do, B. Martini, and K. R. Choo, "A data exfiltration and remote exploitation attack on consumer 3d printers," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 10, pp. 2174–2186, 2016.
- [32] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *20th Annual NDSS symposium*, 2013.
- [33] M. Yampolskiy, L. Graves, J. Gatlin, A. Skjellum, and M. Yung, "What did you add to my additive manufacturing data?: Steganographic attacks on 3d printing files," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 266–281.
- [34] R. Colella, F. P. Chietera, and L. Catarinucci, "Analysis of fdm and dlp 3d-printing technologies to prototype electromagnetic devices for rfid applications," *Sensors*, vol. 21, no. 3, 2021.
- [35] Y. Gao, W. Wang, Y. Jin, C. Zhou, W. Xu, and Z. Jin, "Thermotag: A hidden id of 3d printers for fingerprinting and watermarking," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2805–2820, 2021.
- [36] M. H. Rais, M. Ahsan, and I. Ahmed, "Fromepp: Digital forensic readiness framework for material extrusion based 3d printing process," *Forensic Science International: Digital Investigation*, vol. 44, p. 301510, 2023.
- [37] K. e. a. Babu, "Fire behavior of 3d-printed polymeric composites," in *Journal of Materials Engineering and Performance*. Springer, 2021, pp. 30:4745–4755.
- [38] Q. Zhang, M. Pardo, Y. Rudich, I. Kaplan-Ashiri, J. P. S. Wong, A. Y. Davis, M. S. Black, and R. J. Weber, "Chemical composition and toxicity of particles emitted from a consumer-level 3d printer using various materials," *Environmental Science & Technology*, vol. 53, no. 20, pp. 12 054–12 061, 2019, PMID: 31513393.
- [39] S. Pirela, J. Martin, D. Bello, and P. Demokritou, "Nanoparticle exposures from nano-enabled toner-based printing equipment and human health: state of science and future research needs," *Critical reviews in toxicology*, vol. 47, pp. 1–27, 05 2017.
- [40] E. Kurkowski, A. V. Stockum, J. Dawson, C. Taylor, T. Schulz, and S. Sheno, "Manipulation of g-code tool-path files in 3d printers: Attacks and mitigations," in *Critical Infrastructure Protection XVI*. Springer, 2022, pp. 205–232.
- [41] T. Hudson, X. Kovah, and C. Kallenberg, "Thunderstrike 2: Sith strike," *Black Hat USA Briefings*, 2015.
- [42] C. Kallenberg and R. Wojtczuk, "Speed racer: Exploiting an intel flash protection race condition," *Bromium Labs (January 2015)*, 2015.

- [43] D. Ibdah, N. Lachtar, A. A. Elkhail, A. Bacha, and H. Malik, "Dark firmware: a systematic approach to exploring application security risks in the presence of untrusted firmware," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 413–426.
- [44] L. Garcia, F. Brasser, M. Cintuglu, A. R. Sadeghi, O. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics! attacking plcs with physical model aware rootkit," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. NDSS, 2017. Reston, VA, USA: Internet Society, 2017.
- [45] T. H. Gabriel Boyd. 3D printing bridging: 6 tips for perfect bridges. <https://all3dp.com/2/bridging-3d-printing-tips-tricks-for-perfect-bridges/>. Last Updated: Nov 12, 2022.
- [46] A. Sinha, B. Swain, A. Behera, P. Mallick, S. K. Samal, H. M. Vishwanatha, and A. Behera, "A review on the processing of aero-turbine blade using 3d print techniques," *Journal of Manufacturing and Materials Processing*, vol. 6, no. 1, 2022.
- [47] ASTM, "Standard test methods for flexural properties of unreinforced and reinforced plastics and electrical insulating materials," Modified on July 24, 2017.
- [48] M. Gaskill. (Jan 6, 2023) EPA Proposes to Strengthen Air Quality Standards to Protect the Public from Harmful Effects of Soot. <https://www.epa.gov/newsreleases/epa-proposes-strengthen-air-quality-standards-protect-public-harmful-effects-soot>.
- [49] S. Alyxandra Van, K. Elizabeth, P. Tiffany, T. Curtis, D. Joel, R. Mason, and S. Sujeet, "Attack-defense modeling of material extrusion additive manufacturing systems," in *Critical Infrastructure Protection XVI*. Springer, 2022, pp. 121–153.
- [50] S. Y. Nam, D. Kim, and J. Kim, "Enhanced arp: preventing arp poisoning-based man-in-the-middle attacks," *IEEE communications letters*, vol. 14, no. 2, pp. 187–189, 2010.
- [51] S. Moore, P. Armstrong, T. McDonald, and M. Yampolskiy, "Vulnerability analysis of desktop 3d printer software," in *2016 Resilience Week (RWS)*. IEEE, 2016, pp. 46–51.
- [52] Y. Gao, B. Li, W. Wang, W. Xu, C. Zhou, and Z. Jin, "Watching and safeguarding your 3d printer: Online process monitoring against cyber-physical attacks," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 2, no. 3, Sep. 2018.
- [53] C. Bayens, T. Le, L. Garcia, R. Beyah, M. Javanmard, and S. Zonouz, "See no evil, hear no evil, feel no evil, print no evil- malicious fill patterns detection in additive manufacturing," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1181–1198.
- [54] M. H. Rais, Y. Li, and I. Ahmed, "Spatiotemporal g-code modeling for secure fdm-based 3d printing," in *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems*, 2021, pp. 177–186.
- [55] S. E. Zeltmann, N. Gupta, N. G. Tsoutsos, M. Maniatakos, J. Rajendran, and R. Karri, "Manufacturing and security challenges in 3d printing," *JOM*, vol. 68, no. 7, pp. 1872–1881, Jul 2016.
- [56] S. Belikovetsky, Y. A. Solewicz, M. Yampolskiy, J. Toh, and Y. Elovici, "Digital audio signature for 3d printing integrity," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 5, pp. 1127–1141, 2019.
- [57] S. Zeadally and J. Bou abdo, "Blockchain: Trends and future opportunities," *Internet Technology Letters*, vol. 2, 09 2019.
- [58] B. Hammi, S. Zeadally, and J. Nebhen, "Security threats, countermeasures, and challenges of digital supply chains," *ACM Comput. Surv.*, vol. 55, no. 14s, jul 2023. [Online]. Available: <https://doi.org/10.1145/3588999>
- [59] M. H. Rais, R. A. Awad, J. Lopez, and I. Ahmed, "Jtag-based plc memory acquisition framework for industrial control systems," *Forensic Science International: Digital Investigation*, vol. 37, p. 301196, 2021.
- [60] J. Gatlin, S. Belikovetsky, S. B. Moore, Y. Solewicz, Y. Elovici, and M. Yampolskiy, "Detecting sabotage attacks in additive manufacturing using actuator power signatures," *IEEE Access*, vol. 7, pp. 133 421–133 432, 2019.
- [61] S. R. Chhetri, A. Canedo, and M. A. Al Faruque, "Kcad: kinetic cyber-attack detection method for cyber-physical additive manufacturing systems," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8.
- [62] M. Ahsan, M. H. Rais, and I. Ahmed, "Sok: Side channel monitoring for additive manufacturing - bridging cybersecurity and quality assurance communities," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023, pp. 1160–1178.

A Appendix: Attacks Description

Table 4 gives a generic description of all the analyzed attacks.

Sr. #	Attack Name	Attack Action
1	Printer info	Exfiltrating printer information e.g. manufacturer, model, firmware version, etc.
2	Design SW info	Exfiltrating CAD software information e.g. name, version, etc.
3	Slicer/Control info	Exfiltrating slicing printer control software information e.g. name, version, etc.
4	OG info	Stealing printed object geometry information (IP Theft)
5	Print profile info	Extracting printing profile (thermal, infill pattern, density, etc.) to facilitate
6	Network device	Using compromised printer to extract networked devices information
7	Process artefacts	Using AM process artefacts as information carrier
8	Facility info	Printing facility information e.g. stealing environment temperature, cameras, etc.
9	PYOG	Print your own grave: Break the printing glass
10	Breaking limits	Making printer go beyond limits to cause damage to the end-stops/limit switches
11	Nz impair	Hitting the nozzle to the print bed to physically damage the nozzle orifice
12	Extruder fracture	Hitting the extruder assembly against the printer walls to physically fracture it
13	Nz burning	Heating the nozzle for a longer period with the cooling fan turned off
14	OG scaling	Scaling up or down the print object outer geometry
15	OG thermal	Deforming the object geometry through thermodynamic manipulation by reducing the fan speed or changing its state.
16	Incurable	Impersonating low quality bridging defect (Section 5.3)
17	Warping	Adding warping defects to the print geometry by changing thermal parameters
18	FK thermal	Lowering nozzle temperature resulting in cold filament extrusion
19	Trajectory unsync	Unsynchronized nozzle trajectory for x,y,e axes
20	PS profile	Manipulating the trapezoidal speed profile to cause excessive nozzle jerks
21	PS unsync	Unsynchronized x,y extruder speed during printing
22	FK reduction	Decreasing feed-rate to cause material underflow
23	Clogging	Partially clog the printer nozzle resulting in material underflow
24	MAC/ARP corruption	Denying printer access by manipulating mac table or through ARP poisoning
25	Vertex relocation	Relocating one or more selected vertices
26	OF scaling	Scaling up/down print object specific feature
27	Fitment	Axial misalignment of the print feature to cause fitment issues for intended assembly
28	NT manipulation	Traffic manipulation to breach network integrity
29	IF line spacing	Changing infill-lines spacing to reduce build part strength
30	IF vertex spacing	Changing infill vertices spacing to reduce build part strength
31	FS manipulation	Filament-state manipulation to evade nozzle kinetic detectors
32	FK cavity	Cavity through filament-kinetics w/o modifying toolpath
33	FK density	Localized density variation by filament status/speed change
34	IF pattern	Changing the Infill pattern for example from honeycomb to linear etc.
35	IF density	Changing the Infill density (1% or more)
36	PS cavity	Modifying printing speed for localized zones
37	IF exclusion	Excluding the infill pattern in the print geometry
38	% IF	Changing % fill for the infill pattern e.g. from 50% to 25% and making it more sparse
39	GF change	Replacing the printing instructions (G-code) file
40	Dyn. thermal / bonding	Manipulating interlayer bonding by changing thermal properties at localized zones
41	PS local	Manipulating the printing speed at localized regions
42	OF insert/remove	Adding or removing a geometric feature in the print geometry
43	LT local	Localized changes to layer thickness by manipulating z-profile
44	GC sequence	Localized modification in the toolpath sequence e.g., following a different printing path
45	Anisotropy	Changing print direction to vary anisotropic properties of the print object
46	GC manipulation	Insertion, removal, or modification of the printing instructions
47	Erosion	Causing filament erosion based density attack
48	PF air quality	Microparticles and VOC flooding to degrade air quality of the printing facility

Table 4: Generic description of the studied attacks

B Appendix: Algorithms for Firmware Attacks

Algorithm 1 Printed Object Surveillance Attack

```
1: Output: Object sketch file theft
2: Phase-1: Sketch Compilation
3: On restarts: *eepromAtkend → spyFile
4: if G-code == G0 or G1 then
5:   if not spyFile then
6:     if L.Change() && Zdst ≥ (Eno * Za + 1) then
7:       Shell ← Find_Shell()
8:       *eepromloc ← L.Header; loc++
9:       *eepromloc ← Zdst; loc++
10:      for P in Shell do
11:        *eepromloc ← Px; loc++
12:        *eepromloc ← Py; loc++
13:      end for
14:    else
15:      if Zdst == Zcurrent then
16:        Queue ← Queue ∪ Px,y
17:      else
18:        if printingDone() then
19:          Queue.reset()
20:          spyFile = 1
21:          *eepromAtkend-1 ← (loc - loco)
22:          *eepromAtkend ← 0x01
23:          ResetQueue, loc
24:        end if
25:      end if
26:    end if
27:  end if
28: end if
29: Continue_execution
30: Phase-2: File transfer
31: if SDinserted && SDstateChange then
32:   if spyFile then
33:     if SDauthenticate() then
34:       SD.openFile("spidy.txt", 'w')
35:       for i = 0 to *eepromAtkend-1 do
36:         SD.write(*eeprom(loco+i))
37:       end for
38:       SD.closeFile()
39:       spyFile = 0
40:       *eepromAtkend-1 ← 0x00
41:       *eepromAtkend ← 0x00
42:     end if
43:   end if
44: end if
```

Algorithm 2 Print Your Own Grave: Break the Glass

```
1: Output: Breaking the printing glass
2: Trigger: An unused G-code G98
3: Preheat the printing bed and nozzle
4: for layer = 1 to n do      ▷ n is the desired number of layers
5:   x ← 112.5 + osc × 0.1
6:   y ← 112.5 + osc × 0.1
7:   osc ← -osc
8:   if layer > 8 then
9:     line-count ← small-square ▷ Destruction tool feature 1
10:  else
11:    line-count ← big-square ▷ Destruction tool feature 2
12:  end if
13:  for line = 1 to m do      ▷ m is the number of lines
14:    if line < 4 then
15:      speed ← slow
16:    else if layer > 4 then
17:      speed ← moderate
18:    else
19:      speed ← fast
20:    end if
21:    x ← x + dir × lenx
22:    y ← y + dir × leny
23:    e ← e + dir × lene
24:    Move to (x, y, e)
25:    lenx, leny ← lenx, leny + 0.8
26:    lene ← 0.058 × lenx
27:  end for
28: end for
29: while nozzle and printing bed cool down do
30:   wait!
31: end while
32: Grip(printed-tool): Nozzle jams in cavity and holds the tool
33: Unlock(retaining-clips)
34: Manipulate y, z position variables
35: Move nozzle tip beyond and below glass sheet
36: Guide the glass out through the walls and dispose
```

Algorithm 3 Attack to Simulate Bridging Errors Over X-Axis

```
1: Output: Poor bridging performance
2: Context: Attack resides within Move instruction code region
3: G-code instruction: Move from  $A$  to  $B$ 
4: if  $B_z < \text{Layer}_{\text{width}}$  then
5:   Initialize layer-number
6: else if  $B_z > A_z$  then
7:   Increment layer-number
8:   Copy  $\text{LMAP}_{\text{current}}$  to  $\text{LMAP}_{\text{prev}}$  # Layer Map
9: else if  $\Delta e > 0 \wedge \Delta x \neq 0$  then
10:  direction  $\leftarrow (B_x > A_x) ? +1 : -1$ 
11:  xvar  $\leftarrow \text{round}(A_x, 20 \text{ mm})$ 
12:  yvar  $\leftarrow \text{round}(A_y, 20 \text{ mm})$ 
13:  while xvar  $< B_x$  do
14:    if xvar within Attack-Zone then
15:       $(i, j) \leftarrow \text{LMAP}_{\text{ref}}$  index for  $(xvar, yvar)$ 
16:      if  $\text{LMAP}_{\text{prev}}[i, j] == 0$  then
17:        attack-the-command  $\leftarrow \text{true}$ 
18:         $\text{LMAP}_{\text{current}}[i, j] = 1$ 
19:      end if
20:      xvar  $\leftarrow xvar + 20 \text{ mm}$ 
21:    end if
22:  end while
23:  if attack-the-command then
24:    Modify extruder settings:
25:     $T \leftarrow T + 5^\circ\text{C}$   $\triangleright$  Increase temperature by  $5^\circ\text{C}$ 
26:     $F \leftarrow 0.5 \times F$   $\triangleright$  Reduce feedrate to 50%
27:     $S \leftarrow 0.5 \times S$   $\triangleright$  Reduce fan speed to 50%
28:     $L \leftarrow 1.25 \times L$   $\triangleright$  Increase extrusion length by 25%
29:    Execute the move command
30:    Revert modifications to  $T, F, S, L$ 
31:    attack-the-command  $\leftarrow \text{false}$ 
32:  end if
33: end if
```

Algorithm 4 Object Feature Scaling Attack

```
1: Output: Enlarged geometry over x and y axes
2: Initialize new object
3: G-code instruction rx: Move from  $A$  to  $B$ 
4: if  $B_z > A_z$  then
5:   while Queue-size  $\geq 3$  do
6:     Update Tail position in queue
7:     initialize Polygon-found  $\leftarrow \text{false}$ 
8:     while Head not reached do
9:       Traverse the queue
10:      if Tail coordinates found then
11:        Polygon-found  $\leftarrow \text{true}$ 
12:        break
13:      end if
14:    end while
15:    if Polygon-found then
16:      break
17:    end if
18:    Decrement Queue-size
19:  end while
20:  if Polygon-found then
21:     $P_{\text{tail}'}$   $\leftarrow$  Find position outside polygon adjacent to  $P_{\text{tail}}$ 
22:    Move to  $P_{\text{tail}'}$ 
23:    for  $P_i \in \text{Tail to Head}$  do
24:       $P'_i \leftarrow$  Find corresponding position for  $P_i$ 
25:       $P_{i_e} \leftarrow P_i$ 
26:      Move to  $P'_i$ 
27:    end for
28:    Adjust  $P_{\text{Tail}_e}$  for extra filament used
29:    Attack accomplished for the current layer
30:    Reset Queue for the next layer
31:  end if
32: else if  $B_z = A_z$  then
33:   Add  $B$  at Tail; Update Head and Tail positions
34: else
35:   Reset Queue
36: end if
```

Algorithm 5 Misalignment Attack

```
1: Output: Misaligning an axial slot by  $\theta^\circ$  to cause fitment error
2: Initialize new object
3: if Polygon-found then  $\triangleright$  logic defined in Algorithm 4
4:  temporalCountingStart = True  $\triangleright$  increase temporalDiff
5:  if (layer-change) then
6:    if temporalDiff  $> \text{minGap}$  then  $\triangleright$  Internal feature found
7:      for eachG-code  $\in$  polygon do
8:        Either  $x \leftarrow x + (\text{layerHeight} / \tan(90-\theta))$ 
9:        Or  $y \leftarrow y + (\text{layerHeight} / \tan(90-\theta))$ 
10:       Or  $x, y \leftarrow x, y + (\text{layerHeight} / \tan(90-\theta))$ 
11:      end for
12:    else  $\triangleright$  No internal feature found
13:      misalignCompleteObject OR skipAttack
14:      execute_buffered_G-codes
15:    end if
16:    Reset Queue for the next layer
17:  end if
18: end if
```

Algorithm 6 Internal Cavity Attack

```
1: Output: A cavity inside the object
2: Procedure:
3: if layerCount == 0 then ▷ To find total commands in a layer
4:   cmdPerLayer ← cmdPerLayer + 1
5: end if
6: if  $Z_{new} > Z_{old} + \text{minLayerWidth}$  then
7:   layerChange ← true
8:   targetCmdNo ←  $\frac{\text{cmdPerLayer}}{2} - 2$ 
9: end if
10: if  $30 \leq M73.\text{value} \leq 70$  then ▷ Printing internal layers
11:   attackStatus ← true
12:   layerChange ← false
13: end if
14: if attackStatus == True then
15:   currentCmd ← currentCmd + 1
16:   if currentCmd = targetCmdNo ± 2 then
17:      $C_1, C_2, C_3 \leftarrow \text{split\_symmetric}(\text{currentCmd})$ 
18:      $C_2 \leftarrow \text{mute\_extruder\_part}(C_2)$ 
19:     skip G-code(currentCmd)
20:     execute G-code( $C_1$ )
21:     retract filament(4 mm)
22:     execute G-code( $C_2$ )
23:     advance filament(4 mm)
24:     execute G-code( $C_3$ )
25:   end if
26: end if
```

Algorithm 7 Object Density Variation Attack

```
1: Output: Low-density zones in the internal layers
2: Procedure:
3: if layerCount == 0 then ▷ To find total commands in a layer
4:   cmdPerLayer ← cmdPerLayer + 1
5: end if
6: if  $Z_{new} > Z_{old} + \text{minLayerWidth}$  then
7:   layerChange ← true
8:   targetCmdNo ←  $\frac{\text{cmdPerLayer}}{2} - 2$ 
9: end if
10: if  $30 \leq M73.\text{value} \leq 70$  then ▷ Printing internal layers
11:   attackStatus ← true
12:   layerChange ← false
13: end if
14: if attackStatus then
15:   currentCmd ← currentCmd + 1
16:   if currentCmd = targetCmdNo ± 2 then
17:     modifiedCmd = mute_extruder_part(currentCmd)hg/
18:     skip G-code(currentCmd)
19:     execute modifiedCmd
20:   end if
21: end if
```

Algorithm 8 Filament Erosion Attack

```
1: Output: Reduced filament quantity
2: Procedure:
3: Method-1
4: if printingInitiates == True then ▷ Suitable at the start of
   printing
5:   if G-code == heatNozzle( $T_n$ ) then ▷ preheating command
6:     bufferG-code()
7:     setExtrudeMinTemp( $0^\circ\text{C}$ ) ▷ To bypass the safety check
8:     moveExtruder(30) ▷
9:     executeHeatNozzle( $T_n$ )
10:    restoreExtruderMinTemperature
11:    printingInitiates = False
12:   end if
13: end if
14: Method-2
15: if printingStatus == True then ▷ It is repeated throughout
   printing
16:   estimateAttackZone ▷ Based on object & bowden tube
17:   if attackZone == True then ▷ x times in a layer
18:     eliminateMaxChecks(speed, acceleration, jerk)
19:     retreatFilament(5) ▷ To avoid any spilling
20:     for  $i \in \text{oscCount}$  do ▷ high-jerk moves, 20-50
21:       peakJerkMoves(±4) ▷ to-and-fro at peak settings
22:     end for
23:     advanceFilament(5)
24:   end if
25: end if
```

Algorithm 9 Printing Facility Air Quality Attack

```
1: Output: Degraded air quality of the printing facility
2: Procedure:
3: if idleDuration > inactivity_threshold then ▷ Establish idle
   status by the absence of temperature and movement G-codes
4:   if nozzleTemperature < 150 then ▷ suitable for cold
   extrusion
5:     initiateColdExtrudeBursts() ▷ using Algorithm 8
6:     preheatNozzle(180) ▷ using M109 G-code
7:     retractFilament(4) ▷ to avoid drops on the bed
8:     disableHeaterFeedback()
9:     switchOnHeater() ▷ may achieve  $T_n$  up to  $350^\circ\text{C}$ 
10:    hold_and_wait() ▷ 1-5 mins
11:    switchOfHeater()
12:    enableHeaterFeedback()
13:   end if
14:   resetIdleDuration()
15: end if
```
