;login:

SUMMER 2019 VOL. 44, NO. 2

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

SREcon19 Asia/Pacific
June 12–14, 2019, Singapore
www.usenix.org/srecon19asia

2019 USENIX Annual Technical Conference
July 10–12, 2019, Renton, WA, USA
www.usenix.org/atc19

Co-located with USENIX ATC '19

HotStorage '19: 11th USENIX Workshop on Hot Topics in Storage and File Systems
July 8–9, 2019
www.usenix.org/hotstorage19

HotCloud '19: 11th USENIX Workshop on Hot Topics in Cloud Computing
July 8, 2019
www.usenix.org/hotcloud19

HotEdge '19: 2nd USENIX Workshop on Hot Topics in Edge Computing
July 9, 2019
www.usenix.org/hotedge19

SOUPS 2019: Fifteenth Symposium on Usable Privacy and Security
August 11–13, 2019, Santa Clara, CA, USA
Co-located with USENIX Security '19
www.usenix.org/soups2019

28th USENIX Security Symposium
August 14–16, 2019, Santa Clara, CA, USA
Co-located with SOUPS 2019
www.usenix.org/sec19

Co-located with USENIX Security '19

PEPR '19: 2019 USENIX Conference on Privacy Engineering Practice and Respect
August 12–13, 2019
www.usenix.org/pepr19

WOOT '19: 13th USENIX Workshop on Offensive Technologies
August 12–13, 2019
www.usenix.org/woot19

CSET '19: 12th USENIX Workshop on Cyber Security Experimentation and Test
August 12, 2019
www.usenix.org/cset19

ScAINet '19: 2019 USENIX Security and AI Networking Conference
August 12, 2019
www.usenix.org/scainet19

FOCI '19: 9th USENIX Workshop on Free and Open Communications on the Internet
August 13, 2019
www.usenix.org/foci19

HotSec '19: 2019 USENIX Summit on Hot Topics in Security
August 13, 2019
www.usenix.org/hotsec19

SREcon19 Europe/Middle East/Africa
October 2–4, 2019, Dublin, Ireland
www.usenix.org/srecon19europe

LISA19
October 28–30, 2019, Portland, OR, USA
Submissions due June 18, 2019
www.usenix.org/lisa19

Enigma 2020
January 27–29, 2020, San Francisco, CA, USA
Submissions due August 21, 2019
www.usenix.org/enigma2020

FAST '20: 18th USENIX Conference on File and Storage Technologies
February 24–27, 2020, Santa Clara, CA, USA
Sponsored by USENIX in cooperation with ACM SIGOPS
Co-located with NSDI '20
Submissions due September 26, 2019
www.usenix.org/fast20

NSDI '20: 17th USENIX Symposium on Networked Systems Design and Implementation
February 25–27, 2020, Santa Clara, CA, USA
Sponsored by USENIX in cooperation with ACM SIGCOMM and ACM SIGOPS
Co-located with FAST '20
Spring paper titles and abstracts due September 12, 2019
www.usenix.org/nsdi20

USENIX Open Access Policy
USENIX is the first computing association to offer free and open access to all of our conference proceedings and videos. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Please help us support open access by becoming a USENIX member and asking your colleagues to do the same!

www.usenix.org/membership

www.usenix.org/facebook   twitter.com/usenix   www.usenix.org/linkedin   www.usenix.org/youtube

;login:

SUMMER 2019 VOL. 44, NO. 2

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Ann Heron
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

USENIX ASSOCIATION
2560 Ninth Street, Suite 215
Berkeley, California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738

www.usenix.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published quarterly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

$90 of each member's annual dues is for a subscription to *;login:*. Subscriptions for nonmembers are $90 per year. Periodicals postage paid at Berkeley, CA, and additional mailing offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

EDITORIAL

Musings

RIK FARROW

Rik is the editor of *;login:*.
rik@usenix.org

While musing, I like to wonder what it would be like to live in a world without buggy software. That is, a world very unlike the one we live in. As I write this, Boeing's 737 MAX plane has been grounded, apparently because buggy software and not documenting its possible dangerous effects have killed over 300 people in two separate crashes. Businesses and home users regularly have their data encrypted by criminals demanding ransom. And whole countries are in turmoil via careful manipulation of opinion via social media.

I attend conferences looking for people with interesting and potentially useful ideas. I first met Kostya Serebryany at Enigma 2016, where I tried to get him to write about the work he has been doing in security. He deferred then. Kostya then contacted me in the Fall of 2018 excited about something I find exciting as well: adding security features to hardware. We've published articles from several authors about hardware features to improve security, as well as problems with hardware solutions, such as the ability to extract data from Intel's secure enclave, Meltdown [1].

Kostya most recently has worked on fuzzing, techniques for probing programs for potentially exploitable bugs. In 2015, Peter Gutmann wrote about various fuzzing techniques, something that Kostya has long worked on, and that's related to what he wrote about for this issue [2].

Weaknesses in C/C++

I've long joked that C was a macro-assembly language: a convenience layer for those who needed to write code near to the speed of assembly [3], but with the convenience of variable labels, `for` loops, subroutine call handling, and structures. When I first encountered C, I immediately fell in love with structures, as the concept made some of the things I needed to do so much clearer than calculating offsets in assembler would have been. And, to be honest, I was really bad at calculating offsets. C beat the hell out of writing in Intel assembly (or VAX or Motorola assembler too).

But C and C++ lack certain safety features found in modern languages like Java, Go, Swift, and certainly Rust. In C and C++, you could specify array indices far beyond the end of the array you'd locally allocated, leading to buffer overflows on the stack. You could do this as well in the heap, and you could also do this with pointers into memory. I consider C and C++ to be languages for expert programmers, because they made it so easy to do the wrong thing. I always assumed that the authors of these languages were highly intelligent and expert programmers themselves, and that they had written these languages for their own convenience. In the case of C, that was certainly true, although the authors would be sharing C with other Bell Labs employees and, eventually, professors at various universities.

Bjarne Stroustrup, also at AT&T Bell Labs, came along a bit later, added classes to C, but kept all its wonderful and dangerous flaws. That is, you could create classes and instantiate objects, but you could also overrun arrays, leak memory, and abuse pointers.

Smashing the Stack

The Internet Worm really made people aware of the danger of buffer overruns. The `finger` daemon used the C function `gets()`, which collects a string into an array previously allocated but doesn't check to see whether the length of the array is sufficient. This function still exists in `libc`, and the man page includes the warning, "Never use this function." Makes you sort of wonder why it's still there.

I learned much more about smashing stacks from Elias Levy's famous article about buffer overflows [4]. I recreated the `finger` daemon for class exercises and gave students short C programs they could use to attack the `finger` daemon, whose real purpose was to run the `who` command and return the results over the network. When correctly exploited, the attack would instead run `/bin/sh`.

And this was only part of the problem with C and C++. There were also ways to exploit file structures that contain pointers to functions, or to use a little known option of `format()` to carefully overwrite portions of the stack, allowing exploits that used Return Oriented Programming (ROP). And this is just a partial list.

There are other issues with C/C++ that have to do with pointers. Using `malloc()` returns a pointer to a block of memory, and `free()` releases that block. But it's quite common for programmers to either forget to free memory (a memory leak) or to use a pointer to memory after it had been freed (use-after-free).

During the first time I met Kostya, he showed me dozens of places in the Linux kernel where memory was used after it was freed and was still unpatched upstream. I could tell he was agitated about this.

Today C and C++ are the second and third most popular programming languages (as of April 10, 2019) in the Tiobe Index [5]. Looking at language popularity in another way, I asked Chris Wysopal of Veracode about how many programs in various languages that they analyze each year, and Chris provided me with the diagram in Figure 1. Veracode's numbers, based on the thousands for binary programs analyzed, present a different picture, where C/C++ is less popular.

I found myself wishing that C would just go away, but Kostya assured me that that's not going to be happening, as IoT devices will use slower CPUs and have less memory, and they are going to need compact and fast languages. Damn.

The Lineup

Jasmine Peled, Bendert Zevenbergen, and Nick Feamster have written a column about ethics, regarding something I had never heard of, called mcTLS. You might think that something with *TLS* in its name has to do with encrypting Internet traffic, and you'd be right. However, mcTLS has to do with creating a method so that TLS can be decrypted by middle boxes. If you think this is a bad idea, Peled and her co-authors agree with you, and explain why even the initial researchers should have considered this. Note that the IETF isn't happy about mcTLS either, mainly because including *TLS* in the name violates copyright as well as having the ability to confuse people about their Internet traffic actually being secure.

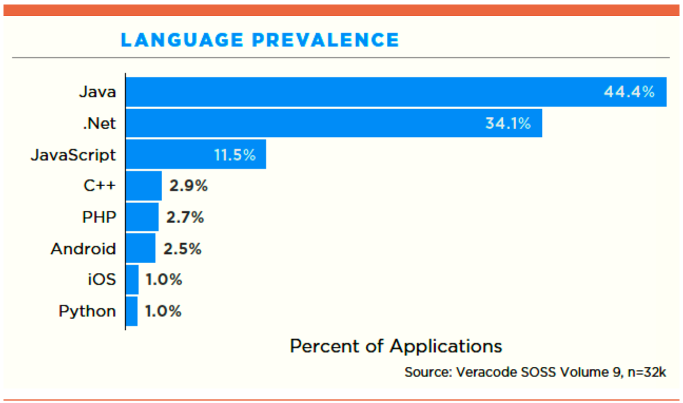


Figure 1: Popularity of programming languages based on programs analyzed for vulnerabilities by Veracode

Kostya Serebryany has written about a security extension in hardware, something I consider a wonderful idea (in case you skipped the earlier part of this column). Sun, now part of Oracle, first came up with the notion of including tags to help prevent a variety of bugs and the successful exploitation of those bugs, and now ARM plans on doing this as well.

I interviewed Mark Loveless, aka Simple Nomad. I've known Mark for many years, and we got together during Enigma '19 to chat and begin this interview. Mark is definitely someone you should call a hacker, unlike Beto O'Rourke, whose membership in the Cult of the Dead Cow predates most of the cDc's hacking activities. Mark has interesting stories to tell.

Anuj Kalia, Michael Kaminsky, and David Andersen have written about eRPC. You might recognize the authors' names from an earlier article about RDMA. This article, like the first one, is based on a paper, this time at NSDI '19. While their paper takes a deeper dive, Kalia et al. explain how this open source RPC library can be faster than those that rely on niche networking technologies.

Daniel Bittman, Peter Alvaro, Darrell Long, and Ethan Miller write about how to avoid bit-flipping in programming data structures. Based on a FAST '19 paper, Bittman et al. explain why bit-flipping may be considered harmful for persistent memories, like Micron's XPoint. But what I particularly like about their work is that it offers a different way of thinking about, and using, traditional data structures like linked-lists and B-trees that is often faster—and involves smaller structures and fewer bit flips.

Vladimir Legeza and Anton Golubtsov tell us how to make logging much more useful. Legeza, now working at Google, and Golubtsov (Amazon) suggest what should be commonsense methods for having standards for your logging messages. Legeza first suggested this idea as an opinion article, but I consider it much more along the line of best practices. I wish I had read such an article 35 years ago!

Laura Nolan considers complexity, taking a different perspective from Dave Mangot's "Boring Tech" article [6] in the Spring 2019 issue. Laura first describes what is meant by software complexity, then how systems complexity differs from the software version. Laura does a great job, and she has volunteered to write columns about SRE issues.

Peter Norton has written about how you can use a tool based on Python to create portable configuration files. The external format is YAML, and the code performs static type checking, helping to prevent errors in configuration.

Mac McEniry decided to cover the use of password managers. Mac has previously written about Hashicorp's Vault (Winter 2017) [7], but this time around he looks at three different Go libraries for secure storage of passwords for use by applications: Keychain (Mac), Windows Credential Manager, and a library called `keyring` that will work on Linux and the other OSes as well.

Dave Josephsen considers just how weird and wonderful it is to be living in the middle of nowhere in Montana. Then Dave gets down to business and begins explaining why he likes Prometheus for monitoring so much and how it's used.

Dan Geer ponders about just how common exploited software bugs might be. Working from various data sources, Dan tells us that the problems with software bugs are much worse than you likely suspect, and even worse than I imagined.

Robert Ferrell suggests that we tone down our expectations for technology. After all, flying cars are still experimental, and even Amazon has decided that having a special button just for ordering laundry detergent might not be the best use of technology.

Mark Lamourine has written three book reviews, covering *Refactoring* (second edition), *Concurrency in Go,* and *Cloud Native Go.* I reviewed David Clark's *Designing an Internet*, and also wrote two short reviews of books for summertime reading: Marcia Bjornerud's *Timefulness* and Max Gladstone's *Empress of Forever.*

In Closing

There are problems with all programming languages. For example, while Rust is much safer by design, you can write Rust code in unsafe mode, disabling its safety features. Java does checks and prohibits array overruns, but the JVM is written in C++, and it has had numerous vulnerabilities over the years.

I also asked Chris Wysopal if he could tell me what proportion of exploitable bugs came from code that processed input, and he answered 75%. If you've been reading *;login:* for the last five years, you will have noticed, and hopefully read, many articles relating to LangSec, for example [8, 9]. LangSec, roughly, is the notion that security could be tremendously improved by paying more attention to input parsing, and Chris's comment about the majority of vulnerabilities coming from input parsing problems supports this.

When I heard about LangSec and learn about efforts to create better support for security in hardware, I imagine that the problem of software insecurity will soon be solved. But I am forgetting several things.

First, most programmers are, by definition, of average skill level. Second, few programmers know much about security, and far fewer have a clue about LangSec. Third, some protocols, like the text (versus binary) version of X.509 certificates, cannot be parsed securely because the design requires a complex parser. And finally, even when ARM or Intel produce security features that will greatly reduce successful exploits, most people won't enable them, either because they don't understand them or because such features cause programs to fail sometimes—an indication of programming flaws they'd prefer to ignore.

References

[1] D. Gruss, D. Hansen, B. Gregg, "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer," *;login:*, vol. 43, no. 4 (Winter 2018): https://www.usenix.org/publications/login/winter2018/gruss.

[2] P. Gutmann, "Fuzzing Code with AFL," *;login:*, vol. 41, no. 2 (Summer 2016) : https://www.usenix.org/publications/login/summer2016/gutmann.

[3] Wikipedia, "Assembly Language: Macros," last modified on March 25, 2019: https://en.wikipedia.org/wiki/Assembly_language#Macros.

[4] E. Levy, "(Aleph One), Smashing the Stack for Fun and Profit," *Phrack,* vol. 7, no. 49: http://phrack.org/issues/49/14.html.

[5] Tiobe Index, April 2019: https://www.tiobe.com/tiobe-index/.

[6] D. Mangot, "Achieving Reliability with Boring Technology," *;login:*, vol. 44, no. 1 (Spring 2019): https://www.usenix.org/publications/login/spring2019/mangot.

[7] C. McEniry, "Go: HashiCorp's Vault," *;login:*, vol. 42, no. 4 (Winter 2017): https://www.usenix.org/publications/login/winter2017/schock.

[8] S. Bratus, M. Patterson, and A. Shubina, "The Bugs We Have to Kill," *;login:*, vol. 40, no. 4 (August 2015): https://www.usenix.org/publications/login/aug15/bratus.

[9] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," *;login:*, vol. 40, no. 1 (February 2015): https://www.usenix.org/publications/login/feb15/bangert.

SECURITY

The Man in the Middlebox

Violations of End-to-End Encryption

JASMINE PELED, BENDERT ZEVENBERGEN, AND NICK FEAMSTER

Jasmine Peled currently works on computer network analysis at the Department of Defense. She recently graduated from Princeton University, where she studied computer science and philosophy. Her work at Princeton focused on how undergraduate computer science courses can better incorporate material about ethics in order to encourage students to consider the ethical and societal implications of the technologies they develop. Jasmine's senior thesis, "Towards a Pedagogy of Principles: Teaching Ethics in Computer Science," received Princeton's Outstanding Senior Thesis Award. jasminepeled21@gmail.com

Ben Zevenbergen is a visiting professional specialist at the Center for Information Technology Policy at Princeton. His work mostly consists of multidisciplinary investigations in the ethical, social, and legal impacts of Internet technologies, and vice versa. At CITP Ben is working on the engineering ethics and political theory impacts of artificial intelligence. Ben is currently finishing a PhD at the Oxford Internet Institute about the research ethics for technical projects that involve unsuspecting Internet users as data subjects. benzevenberger@princeton.edu

We consider the ethical issues of the paper "Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS" [8], which presents a method to extend the Transport Layer Security (TLS) protocol to allow it to support middleboxes. Specifically, to what extent should third parties be able to decrypt traffic between two Internet endpoints for various purposes, ranging from performance to security? This is the first column in a series about ethics that we hope will encourage ongoing discussion and debate in the research community about ethical considerations that may arise in the course of networking, security, and systems research.

Ongoing research in the computer science communities of security, privacy, and networking investigates and develops network applications and appliances that may improve Internet performance and security, often by modifying traffic en route between two Internet endpoints. Middleboxes constitute one such example of this capability; middleboxes are defined as "any intermediary box performing functions apart from normal, standard functions of an IP router on the data path between a source host and destination host" [1]. Middlebox functionality includes transcoding videostreams to different bit rates or detecting attacks, often through inspection of the contents of a packet's payload.

Because some of this functionality can require inspecting the contents of network traffic, these middleboxes may need to break end-to-end encryption, decrypting traffic midstream to facilitate operating on packet contents. mcTLS describes mechanisms for breaking the end-to-end encryption of TLS specifically to enable middleboxes to view and edit data and metadata.

Middleboxes and End-to-End Encryption

The rise of end-to-end encryption is generally heralded as a positive development, as it protects both the integrity and confidentiality of communications between Internet endpoints, thus protecting sensitive transactions and preserving user privacy.

On the other hand, if traffic is encrypted, conventional middleboxes have difficulty performing any operation that depends on seeing packet contents. In response, researchers have grappled with this problem in various ways [6]. One approach involves developing techniques that can still operate on encrypted traffic, including techniques that can perform operations on packet headers alone [5] or limited types of operations on encrypted messages [11]. Yet, certain types of operations that require deep packet inspection may be either inefficient or ineffective when payloads are encrypted; thus, another approach involves developing a "backdoor" of sorts that allows an Internet service provider (ISP) to decrypt encrypted communications in flight.

ISPs have developed an increased interest in deploying middleboxes that perform operations on traffic that is en route between source and destination. For example, ISPs often deploy middleboxes that perform intrusion detection and detect a range of different types of attacks; these middleboxes may also perform certain performance optimizations, such as transcoding a videostream to a lower bit rate or performing other types of optimizations (e.g., WAN acceleration, load balancing). These operations may depend on at least inspecting traffic contents; in some cases, the traffic contents may even be modified.

Nick Feamster is a Professor in the Computer Science Department at Princeton University and the Deputy Director of the Princeton University Center for Information Technology Policy (CITP). He was formerly a Professor at Georgia Tech, and received his MEng and PhD degrees from MIT. He has won many awards for his networking research, at ACM SIGCOMM, IMC, and USENIX NSDI. Nick is also an avid distance runner, having completed nearly 20 marathons and the Comrades ultra-marathon in South Africa.
feamster@cs.princton.edu

Multi-context TLS (mcTLS) is one such technology; it permits ISPs to decrypt secure, end-to-end sessions of TLS Internet traffic by third parties, allowing them to control, read, and write the data in the communications. The authors of the paper [8] outline several technical advantages to mcTLS:

- In-network functions may be more effective at scale, in contrast to relying on endpoint-based functionality alone.
- Middleboxes may be useful for both users and service operators in terms of speed and data storage.
- Middleboxes may help protect personal information by acting as a watchdog over applications that may leak data unwittingly.

mcTLS is based on the premise that, just like end-to-end encryption, middleboxes are a "useful part of the Internet and are here to stay." More generally, the question of whether (and how) middleboxes should have access to encrypted communications is under active discussion in industry standards organizations, such as the Internet Engineering Task Force (IETF) [7].

A natural question concerns whether the increased in-network capabilities that result from breaking end-to-end encryption offer benefits that outweigh the risks of harm to stakeholders. A related question concerns whether the development and deployment of such research should focus on technologies that weaken end-to-end encryption in favor of potentially improved security and performance, versus technologies that can operate on traffic with encrypted payloads, potentially with reduced effectiveness.

The Appropriate Ethical Lens

Ethical analysis can take many forms, which are best understood on a spectrum. On one end of the spectrum is *normative ethics*—as practiced in academic philosophy—which studies reasoning methods such as utilitarianism, deontology, and virtue ethics. *Ethics compliance frameworks* such as research ethics or medical ethics—which consist of more formal procedures for specific professions—are on the other end of the spectrum. In between these two approaches to ethics are several other, more applied types of ethics sub-disciplines, such as information ethics, technology ethics, computer ethics, data ethics, bioethics, animal ethics, among many others. Compliance-ethics frameworks typically consist of "check-box exercises" that may be rooted in law; applied ethics have some generally agreed upon methodologies for reasoning about sectors of society; and normative ethics studies the reasoning methods themselves. For this article, it is relevant to establish whether man-in-the-middle technologies such as mcTLS should be analyzed through the lens of research ethics or through a different approach.

The framework of research ethics is typically an appropriate lens for an academic paper. This framework is commonly applied to a study or experimentation when (1) it presents research in the formal sense, and (2) when the research is conducted with human subjects. In the United States, research in the formal sense is defined in the US Code of Federal Regulations on the Protection of Human Subjects as a "systematic investigation, including research development, testing and evaluation, designed to develop or contribute to generalizable knowledge" [10].

Once it has been established that a given paper constitutes research, the next question is whether the authors conduct research on human subjects. Formal regulations on the protection of human subjects in research apply to persons who conduct research (e.g., the

Common Rule [2]). Although security and networking researchers typically see themselves as conducting research on technical systems, the Internet is more properly understood as a socio-technical system in which humans and technology interact. Humans will often be implicated in data collection.

The mcTLS technology aims to intercept the Internet traffic of humans, though the paper discussed in this column merely proposes a novel functionality but does not actually present data from experimentation on live Internet traffic. Instead, the paper presents the research and development of a new technology. Therefore, the formal framework of research ethics (such as the Common Rule) need not be applied. However, even when formal requirements do not demand research conform to a research ethics checklist, researchers should still assess the broader ethical impact of their work. After all, research that does not constitute "human-subject research" may still affect people, and this series of columns seeks to bring to mind some questions that researchers should be asking themselves.

Research into computers and networked systems have traditionally challenged the principles laid out in existing research ethics procedures, such as the Belmont Report. In response, several computer science communities embraced the Menlo Report [3], which interprets the principles of the Belmont Report [4] and applies them to computer and information security and measurement research specifically. Additional networked systems ethics guidelines were developed through lengthy processes of reflection and iteration in workshops by scholars from many different disciplines [9]. Because the Menlo Report is more applicable to experimentation with human subjects on the Internet, the analysis in this article will lean on the concepts presented in Networked Systems Ethics Guidelines [9].

Technology Ethics Analysis

The Networked Systems Ethics Guidelines suggest that researchers aim to understand a technology within the social context where it operates. This social context includes an analysis of the stakeholders, the aims, benefits, risks of harm, meaning of collected data in context, shifts in power, and an understanding of the affected values. The guidelines then suggest analyzing the impact of the values on stakeholders and the socio-technical environment, the values themselves, and any foreseeable unintended consequences. It is useful to link these analyses to the technical sources of the original design. When the impact of technical alternatives have been considered in minimizing risks of harm, the guidelines suggest managing the residual risks through information governance methods, also known as responsible data stewardship. We will preface each section with a question from the guidelines.

Aims and Benefits

What are the aims and benefits of the project? How will the research benefit society and specific stakeholders?

The technology presented in the mcTLS paper [8] realizes a technology to intercept, analyze, and possibly manipulate Internet traffic that has been encrypted on an end-to-end basis. The proposed tool would replace previous "hacks," which ostensibly decrease security in the existing all-or-nothing security model. The authors state the aims of the mcTLS project concretely as follows: (1) to optimize network resource usage, (2) to improve user experience, and (3) to protect clients and servers from security threats. This tool would only be applied with the consent of all the parties involved in the connection.

Naylor et al. [8] state some further benefits that could be considered as secondary goals. For example, the authors mention that the in-network services may increase competition, innovation, and choice for end-users. Another stated benefit is that the use of middleboxes may reduce energy consumption by all stakeholders on the Internet.

The aims and benefits appear to be presented from the point of view of an ISP or network operator. The interests of end-users on the Internet are scarcely considered. The second-order benefits to society are difficult—if not impossible—to prove or support with evidence, and the paper does not consider some of the unintended social harms that may result from this tool, particularly the fact that breaking end-to-end encryption in this way will give the network operator complete power to read users' Internet traffic.

Privacy

Which definitions or explanations will be used to assess a value? Is the risk of harm high, medium, or low?

The interception and possible processing and dissemination of end-users' Internet traffic data may be considered a violation of their privacy. The concept of privacy is vague and illusive, however, and has thus been difficult to define precisely. Privacy may be best understood as an umbrella term referring to a group of related concepts, issues, and values that protect the individual's private life from intrusions by others. The use of mcTLS on end users' encrypted traffic violates the sub-category of *information privacy*, especially if their data is further processed or disseminated to third parties.

Privacy violations can be harmful in immaterial ways, though they may also reveal information about persons that can lead to physical, financial, reputational, or other types of harm, depending on the actor who receives the information and decides to act upon it. Different types of information have different types of impact on persons when revealed, depending on the context. Given the mediating role of the Internet to support modern life, encrypted Internet traffic intercepted by mcTLS will likely contain a large variety of information types, concerning a large and diverse set of persons.

To assess the risk of harm, one must consider the type of attacker who may be interested in the information that mcTLS may expose, the level of technical sophistication they have, what actions could be taken based on the new knowledge, and what the consequences would be for an Internet user. Given the large amount of Internet traffic generated by a variety of end-users that mcTLS could intercept, all types of attackers—from individual hackers to well-resourced government surveillance actors—should be taken into account. Further, mcTLS creates a point of failure for a variety of actors to gain access to Internet traffic through both security vulnerabilities and traditional legal procedures.

Further, mcTLS poses threats to privacy by altering the context in which certain information is processed and handled. Information that may be acceptable for both endpoints of communication to view should not necessarily be shared with third parties. For example, a user may choose to enter Personally Identifiable Information (PII) into a healthcare site in order to receive personalized care, but sharing this information in one context does not constitute approval for their ISP to share it with other companies. This could violate the Health Insurance Portability and Accountability Act (HIPAA), as well as the trust that users place in their ISP to keep communications and data private.

Due to the large variety of users, stakeholders, and their purpose for using the Internet, it is nearly impossible to generalize the risk of harm and define it precisely and meaningfully. This makes it especially challenging to assess the ethical tradeoffs presented by an emerging technology. Further, what may be considered harmless today may become a much larger threat in future. For example, the creation of new data sets may allow identification of Internet users in ways that cannot be foreseen today.

Violations of end-user privacy may be justified to some extent by gaining their consent or when serving the greater good. However, an informed consent notice or other justifications should be based on factual information and informed assessments rather than self-serving arguments of increased efficiency. The complex and international nature of the Internet complicates such an analysis, because risks of privacy harm should first be defined and identified for all affected Internet users in their contexts. This is, of course, a near impossible task.

Autonomy, Consent, and Choice

Do you need to rely on informed consent from participants and stakeholders? Which stakeholders carry the burdens of the study?

The Belmont Report gives guidance regarding the respect for autonomy, balancing the value of autonomy with the interests of others:

"To respect autonomy is to give weight to autonomous persons' considered opinions and choices while refraining from obstructing their actions unless they are clearly detrimental to others" [4].

To achieve the aims and deliver the benefits identified in the paper, the existing security that users currently enjoy due to end-to-end encryption will be violated. Of course, most Internet users may not have a full understanding of the security mechanisms currently in place or even awareness of the existence of end-to-end encryption in the first place. This situation raises the question of whether taking away a good that users enjoy unwittingly as a means to achieve another end—the relative benefit of which is itself debatable—is a valid justification.

Informed consent is widely considered to be a mechanism that operationalizes the concept of autonomy of Internet users. Indeed, the authors state that both endpoints of a connection within which an mcTLS is deployed must consent to its use. However, similarly to the realm of healthcare, a key aspect of informed consent is being informed of reasonable alternatives to the proposed action. In the context of mcTLS, respect for autonomy may be understood as the obligation to fully inform an Internet user of the benefits and risks of harm in their particular context. The rejection of these benefits and risks of harm should not lead to a suspension of their Internet connection but possibly to access an alternative network within which the mcTLS tool is not operational.

Alternatively, an ISP or network operator could choose to base the legitimacy of the increase in power on a more paternalistic approach, whereby they interpret their duty of care to justify the use of mcTLS, along with its benefits and risks of harm. This constitutes a use of power over Internet users that may require some balancing through accountability mechanisms (see the Accountability section, below). For example, the ISP or network operator may choose to publish their considered justification for the use of mcTLS in their network, along with a technical description that allows some auditing of their system, as well as an information governance (or data stewardship) statement to which it can be held accountable by end-users. It is critical, though, that these explanations of benefits and potential harms posed by mcTLS do not simply use technical jargon to scare off the average user from understanding the full implications of middlebox technologies, so that supposed informed consent is, in fact, informed.

Many of these ethical concerns regarding privacy, autonomy, and choice could be resolved through agreements between ISPs and users about whether mcTLS will be implemented and how user data will be used. However, the next two sections present ethical challenges to the deployment of mcTLS which do not have such clear solutions.

Stakeholders and Power Shifts

Are particular stakeholders empowered or disempowered as a result of this project? Which values will the project conceivably impact?

ISPs and network operators will be the actors that implement and have access to mcTLS; these actors ultimately make the decision to implement and deploy such systems. These actors already have significant power over information flows, as the de facto gatekeepers to the Internet with the ability to control, manipulate, and, in some cases, observe data flows between their subscribers and other sites on the Internet. mcTLS further amplifies their power over Internet users, giving them the ability to observe the contents of network communications that might otherwise have been encrypted.

Internet users, on the other hand, will be disempowered over the collection and use of their data. Once a user has given consent to the use of mcTLS on their traffic, it will be difficult to control how their Internet traffic is collected, processed, and further disseminated, which may result in a violation of privacy. An informed consent notice referring to end-to-end encryption and the functionality of mcTLS is unlikely to be meaningful to most Internet users. First, an informed consent notice is unlikely to give the end-user meaningful information regarding the creation of a single-point-of-failure within their Internet traffic and the possible attackers or interested parties that may subsequently gain access to their data. Further, a rejection of the mcTLS tool on their Internet traffic may lead the ISP or network operator to suspend Internet access of the end-users, thereby offering users a choiceless choice (or Hobson's choice) whereby the user is asked to agree with a technically complex violation of their encrypted end-to-end connection. This may constitute a violation of their autonomy.

The mcTLS paper does not differentiate between Internet users in its analysis of benefits and harms. It is important to note that the benefits to some users can result in vastly increasing risks of harm for other users. For example, the use of middleboxes on the Internet traffic of oppressed peoples or whistleblowers in countries where the rule of law is not as effective as the authors' home country should be considered.

Unintended Consequences

Does the project potentially set a precedent for unethical methodologies that could be misused by others in the future?

Although developers of new technologies may not be directly responsible for misuses of their products under the law or under typical "checklist" research ethics restrictions, developers should still take care to mitigate potential unintended negative consequences. It is therefore important that researchers engage actively with the possibility that their methods and technologies may be misused, and design ways to mitigate those identified risks and harms. The most common ways projects influence future malevolent technology uses is through function creep and precedent setting. The following questions can help address the future concerns of creating a technology that enables a so-called "back door" into end-to-end encrypted Internet traffic.

Function creep occurs when functionality of a technology is used for other purposes than for which it was originally intended. Researchers and developers may want to consider for which other—more malevolent—aims the mcTLS technology can be used. It is relevant to consider a wide array of threat actors that would have an interest in using mcTLS for their own aims. When even companies such as Experian and Equifax are unable to keep their data secure, it is important to consider whether users can truly expect ISPs to protect their information and how adding a third party complicates this. How could the developers mitigate these foreseeable malevolent uses through their technical design?

Precedent-setting occurs when other researchers or developers can point at the use of mcTLS's technology or functionality to justify the development and use of new technologies. Technology is typically a double-edged sword that can be used for both good and bad purposes. It is therefore important to interrogate the use of precedents critically. Developers should consider how other future malevolent developers can utilize the existence and use of mcTLS to justify the development and use of technologies that cause more harms. For example, does the interception of end-to-end encrypted traffic by ISPs for efficiency in finding malware justify the interception of encrypted traffic to create profiles of Internet users for law enforcement?

When the risks of harm to stakeholders and potential unintended consequences have been identified, the researchers may pinpoint the technological causes of harms. For example, the main cause of harms is the creation of a back door and concentrated point of access for encrypted Internet traffic. Researchers should consider ways to address these issues and justify why alternative designs (or not acting at all) may be most beneficial.

Accountability

Which measures are taken to allow affected stakeholders to address concerns effectively?

Accountability is the concept that allows actors to be held liable or answerable for their actions. When an actor gains power over other stakeholders from the introduction of a technology, and the new actions may violate particular values, this increase in power should be accompanied by an increase in accountability. Accountability thus serves as a rebalancing mechanism.

Several governance mechanisms exist to allow for the exercise of accountability. For example, data governance policies can include codes of practice for employees and organizations within a sector to limit the extent to which technologies may be (mis) used. Other mechanisms include a statement of data collection policies, data retention periods for collected data, mitigation strategies for unforeseen risks, and limits on the further use or dissemination of collected data. Technical measures include information security strategies, de-identification of collected data, and further encryption of retained data. Meaningful accountability can be achieved when an organization is transparent about these policies and technical choices, as it allows third parties to audit and limit the exercise of power.

Conclusion

The introduction of technology in an environment will inevitably empower some actors over others. This is also true for mcTLS, a tool that breaks the end-to-end encryption of Internet traffic to achieve some beneficial ends, such as increased efficiency in identifying and solving security issues. However, the means by which these ends are achieved may conceivably cause harms to individual Internet users due to the shift in power over Internet traffic. End-users' autonomy and privacy are likely violated, which have further social consequences. The developers may explore options to remedy these violations through technical means. However, not all problems are solvable through technology. Therefore, the actors who employ a technology such as mcTLS should consider rebalancing their newly gained power over Internet users with accountability mechanisms, allowing for transparency (and audibility) of the systems and clear information governance policies to which affected parties can hold the operators to account.

References

[1] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and Issues," 2002: https://tools.ietf.org/html/rfc3234 .

[2] "Federal Policy for the Protection of Human Subjects ('Common Rule')," 1991: https://www.hhs.gov/ohrp/regulations-and-policy/regulations/common-rule/index.html.

[3] D. Dittrich and E. Kenneally, "The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research," US Department of Homeland Security, 2012: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2445102.

[4] National Commission for the Protection of Human Subjects of Biomedical and Behavioral Research, "The Belmont Report: Ethical Principles and Guidelines for the Protection of Human Subjects of Research," Department of Health, Education, and Welfare, 1979: https://www.hhs.gov/ohrp/regulations-and-policy/belmont-report/read-the-belmont-report/index.html.

[5] G. Gu, R. Perdisci, J. Zhang, W. Lee, "BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection," in *Proceedings of the 17th USENIX Security Symposium* (USENIX Security '08), pp. 139–154: http://static.usenix.org/events/sec08/tech/full_papers/gu/gu_html/.

[6] K. Moriarty, "TLS Security and Data Center Monitoring: Searching for a Path Forward," August 2017: https://www.rsa.com/en-us/blog/2017-08/tls-security-and-data-center-monitoring-searching-for-a-path-forward.

[7] K. Moriarty and A. Morton, "Effects of Pervasive Encryption on Operators," 2018: https://tools.ietf.org/html/draft-mm-wg-effect-encrypt-14.

[8] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. R. Rodriguez, and P. Steenkiste, "Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS," in *ACM SIGCOMM Computer Communication Review*, vol. 45 (August 2015), pp. 199–212.

[9] "Networked Systems Ethics—Guidelines," last modified on July 10, 2017: http://networkedsystemsethics.net/index.php?title=Networked_Systems_Ethics_-_Guidelines.

[10] "Code of Federal Regulations, Title 45, Public Welfare, and Part 46, Protection of Human Subjects," Department of Health and Human Services, 2009: https://www.hhs.gov/ohrp/sites/default/files/ohrp/policy/ohrpregulations.pdf.

[11] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep Packet Inspection over Encrypted Traffic," in *ACM SIGCOMM Computer Communication Review*, vol. 45 (August 2015), pp. 213–226.

ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety

KOSTYA SEREBRYANY

Konstantin (Kostya) Serebryany is a Software Engineer at Google. His team develops and deploys dynamic testing tools, such as AddressSanitizer, MemorySanitizer, ThreadSanitizer, and libFuzzer. Prior to joining Google in 2007, Konstantin spent four years at Elbrus/MCST working for Sun compiler lab and then three years at Intel Compiler Lab. Konstantin holds a PhD from Moscow State University of Economics, Statistics, and Informatics and an MS from Moscow State University. kcc@google.com

I discuss memory safety bugs typical to C and C++, current tools and approaches to finding such bugs or mitigating their risk, and a new hardware feature, ARM MTE, that promises to be the biggest improvement since the introduction of page protection.

Memory (Un)safety

More than 30 years after the Internet Worm, we are still talking about memory safety bugs in C and C++ programs. Numerous improvements in the software development process are dwarfed by the exponential increase in the amount of software, its exposed attack surface, and the discovery of new attack techniques.

Memory safety bug is an umbrella term to represent program defects inherent in C and C++ but also present in other languages. The most common classes of bugs are buffer overflows, heap-use-after-free, and stack-use-after-return.

These bugs often make the code vulnerable to exploitation. Malicious actors can leverage memory-unsafe behavior to remotely execute code, leak sensitive information, escalate privileges, or escape VMs. A buffer overflow in OpenSSL, nicknamed Heartbleed, achieved notoriety for its ease of exploitation and high impact. It allowed attackers to steal a server's private memory, including cryptographic information such as keys and passwords, without being detected. But *named* bugs like Heartbleed and Stagefright, a family of remotely exploitable bugs in Android, are just the tip of the iceberg.

Thousands of memory safety bugs are filed as CVEs every year. Roughly two-thirds of all CVEs in the Android platform are memory safety bugs. A similar picture is seen across the industry, affecting browsers, operating systems, and server-side and IoT software [1, 2]. And even these bugs are still the tip of the iceberg. Many more bugs do not get CVEs assigned, and many others remain unknown to software vendors. Some are being silently exploited, others cause hard to detect data corruption, and some lie dormant waiting to strike.

Typical Bugs

Before we dive deeper, let's take a closer look at two of our most beloved insects.

A **heap-buffer-overflow** happens when an object of a certain size is allocated on the heap, and then a pointer to this object is used to access memory outside of the object bounds. Typically, the object is an array of *n* elements, and the code accesses the i-th element where i < 0 or i >= n.

```
int *array = new int[n];  // heap allocation
array[n] = 42;  // buffer overflow
array[-1] = 42; // buffer overflow (underflow)
array[100500] = 42;  // buffer overflow, assuming n <= 100500
```

A **heap-use-after-free** happens when an object is allocated on the heap, and later deallocated, but a pointer to the object is preserved somewhere and is used to access the deallocated memory.

```
Object *obj = new Object;   // heap allocation, or "malloc"
delete obj;                 // heap deallocation, or "free"
obj->member = 0;            // heap-use-after-free, or
                            // access via a dangling pointer
```

In both cases the buggy memory access touches someone else's memory. In the C and C++ standards this is considered undefined behavior. In real life it may cause a loud crash, a silent data corruption, or a convenient back door.

Existing Tools and Practices

We haven't been exactly ignoring the problem for 30 years.

Coding practices and testing tools reduced the likelihood of introducing a memory bug. A test-driven development process together with dynamic testing tools like AddressSanitizer [3] or Valgrind will help avoid many bugs. Fuzzing (and, ideally, fuzz-driven development [4]) will pick up the next layer of bugs. Some memory bugs can be spotted by static analysis.

Software-based code-hardening techniques make it harder for attackers to exploit memory safety bugs that reach production. Stack cookies, non-executable memory, ASLR, control flow integrity (LLVM CFI, Microsoft CFG, Shadow Call Stack), and other techniques help prevent memory safety bugs from diverting program control flow, the end goal of many exploits. Hardened memory allocators, such as Scudo Hardened Allocator or Chrome's Partition Alloc, frustrate exploitation and may make it impossible in some cases.

Hardware-based solutions have begun to appear as well. ARM Pointer Authentication, already available in the most recent Apple hardware, cryptographically authenticates return addresses and discourages attackers from using return-oriented programming (ROP). Intel Control-flow Enforcement Technology is expected to appear soon to solve ROP in a different way, by keeping the return address on a separate stack with special permissions.

All these tools are making our software more stable and secure, but they are not enough. No amount of testing guarantees the absence of bugs, and existing exploit mitigations only prevent some attacks, while almost entirely ignoring others, e.g., data-oriented attacks.

Among the hardware-based solutions two stand out, SPARC ADI and ARM MTE, both implementations of a concept known as **memory tagging** or memory coloring. SPARC ADI has been available in mass-produced hardware since 2016; we covered this feature in an earlier paper [5]. This article focuses on ARM MTE.

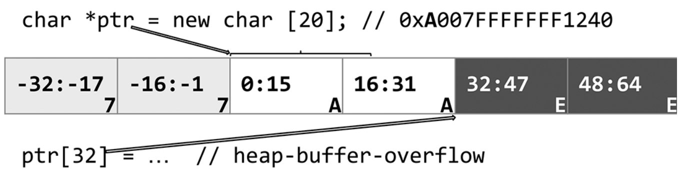

char *ptr = new char [20]; // 0xA007FFFFFFF1240
-32:-17
7
-16:-1
7
0:15
A
16:31
A
32:47
E
48:64
E
ptr[32] = ... // heap-buffer-overflow

Figure 1: Heap-buffer-overflow is detected by MTE because the pointer's address tag 0xA does not match the memory tag 0xE.

ARM MTE

On September 2018 ARM announced the **Memory Tagging Extension**, or MTE [6], a part of the ARM v8.5 architecture. It does not yet exist in real hardware, but everything else about this extension is very promising.

The extension introduces a notion of two types of tags: address tags and memory tags.

An **address tag** is a 4-bit value stored at the top of every pointer in the process. MTE utilizes *top-byte-ignore*, an existing AArch64 feature that instructs the hardware to ignore the topmost byte of addresses, allowing this byte to be used as user-controlled metadata. Therefore MTE is applicable only to 64-bit software.

A **memory tag** is a 4-bit value associated with every aligned 16-byte region of application memory (*memory granule*). The way memory tags are stored is a hardware implementation detail. Logically, every 16 bytes of memory now contain an extra 4 bits of metadata in addition to 128 bits of data.

Every time a heap region is allocated, the software chooses a random 4-bit tag and marks both the address *and* all the newly allocated memory granules with this tag. The load and store instructions verify that the address tag matches the memory tag, causing a hardware exception on tag mismatch. MTE introduces new instructions to manipulate the tags.

Let's look at the example in Figure 1. When the user code requests 20 bytes of heap to be allocated, operator `new()` rounds up the size to the 16-byte boundary (i.e., to 32), allocates a 32-byte chunk of memory (i.e., two 16-byte memory granules), chooses a random 4-bit tag (in this case, 0xA), puts this tag into the top-byte of the address, and updates the tags for the two newly allocated memory granules (the white-colored regions in the diagram). The adjacent memory regions have different memory tags (light gray granules have the tag 0x7, dark gray granules have the tag 0xE), so when the code tries to access memory at offset 32 from the pointer, MTE raises an exception because the tag of the pointer does not match the tag of the memory granule being accessed.

Figure 2 demonstrates an example of how heap-use-after-free is detected. On deallocation, operator `delete()` changes the tag of all three deallocated granules of memory from 0xD to 0x4,

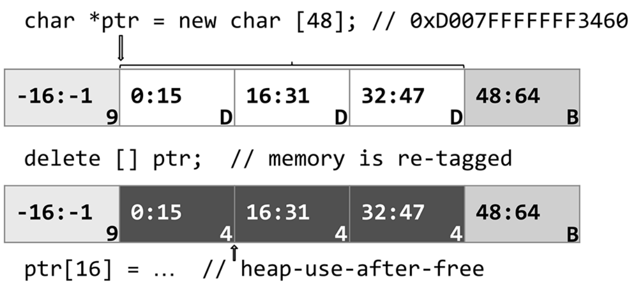

char *ptr = new char [48]; // 0xD007FFFFFFF3460
-16:-1 9
0:15 D
16:31 D
32:47 D
48:64 B
delete [] ptr; // memory is re-tagged
-16:-1 9
0:15 4
16:31 4
32:47 4
48:64 B
ptr[16] = … // heap-use-after-free

Figure 2: Heap-use-after-free is detected by MTE because the pointer's address tag 0xD does not match the memory tag 0x4.

so that any access to this memory via an old (*dangling*) pointer causes an exception because the pointer still has the old tag 0xD. The adjacent memory regions (tagged with 0x9 and 0xB) are not affected by retagging of this region.

You may have noticed that bug detection with MTE is probabilistic. Indeed, there are only 16 possible values of a 4-bit tag. One random tag will be different from another random tag with a probability of 15/16 or ~93%. It is up to the software to decide whether to increase this probability with other tricks. For example, in order to detect contiguous buffer overflows with perfect accuracy, the allocator may enforce that tags for adjacent chunks are never equal.

With MTE, the heap memory is tagged inside `malloc()` and `free()`, and the tag checking is performed by the hardware. It means that recompilation will not be required for detecting heap-related bugs. MTE can also identify stack-use-after-return and buffer overflows on the stack or in global variables, but it will require recompilation with extra compiler options.

Comparison with AddressSanitizer

AddressSanitizer is a widely used tool for detecting memory safety issues. It uses compiler instrumentation to observe all loads and stores. Its specialized malloc "poisons" *red zones* around heap objects to detect buffer overflows and keeps freed memory in *quarantine* to detect use-after-free. The red zones and the quarantine are the major causes of AddressSanitizer's high memory overhead.

MTE is conceptually similar to AddressSanitizer: both detect bugs at runtime, both require special functionality in malloc and free, and both require some amount of compiler support.

However, the use of address tags makes MTE sufficiently different: it does not require red zones or quarantine to detect bugs. This allows MTE to consume less memory. Moreover, MTE performs checking in hardware, thus eliminating the overhead of compiler instrumentation for every load and store.

Compared to AddressSanitizer, MTE brings the following benefits:

- MTE checking can be turned on and off at runtime.
- CPU overhead is expected to be very small, hopefully a small single-digit percentage, while AddressSanitizer typically has 2x–3x slowdown.
- MTE can find heap-related bugs without recompilation.
- Due to the small overhead, the same binary can be used for testing and for production.
- MTE's memory overhead is 3%–5%, compared to 2x–3x for AddressSanitizer.
- Memory accesses that happen far from the object bounds or long after the object lifetime are more likely to be spotted by MTE than AddressSanitizer, which makes MTE a better exploit mitigation.

The only downside of MTE is that it may fail to detect buffer overflows that happen within the 16-byte granule:

```
char *array = new char [13];  // allocates one 16-byte granule
array[14] = 0;  // access within the same 16-byte granule
```

Various software strategies are possible to improve bug detection for such cases with additional cost or complexity.

Uses of MTE

We envision several different usage modes for MTE.

First, MTE is going to be a much nicer version of AddressSanitizer for testing and fuzzing. It will find more bugs at a fraction of the cost. In many cases it will allow testing using the same binary as shipped to production.

Second, MTE could be used as a mechanism for testing in production (e.g., crowdsourced bug detection), always-on or enabled randomly. For client software, such as web browsers, it means that when a bug happens on a user device it will be detected, and, with user consent, an actionable bug report will be sent to the vendor. For server-side software it means that even the rarest bugs will be detected immediately once they get triggered.

Finally, MTE can be seen as a strong security mitigation. It is true that it prevents exploitation with less than 100% probability, but the probability is still very high, and the first failed exploitation attempt will warn the user and the software vendor. We believe that memory tagging will detect the most common classes of memory safety bugs in the wild, helping vendors identify and fix them and discouraging malicious actors from exploiting them.

Other clever ways to use MTE will likely be discovered. MTE may allow building debuggers with infinite hardware watchpoints, efficient race detectors, or faster garbage collectors.

HWASAN

The full potential of memory tagging will only be available with future hardware, several years from now. But you can reap some of the benefits now, like significantly reduced memory consumption, by using a software implementation of memory tagging: HWASAN (hardware-assisted AddressSanitizer) [7]. HWASAN is similar in spirit to AddressSanitizer, but its smaller memory footprint makes it a better choice on memory-restricted devices, such as mobile phones. Today, the tool only supports 64-bit ARM CPUs, since it requires the top-byte-ignore feature and a small modification in the kernel to allow passing tagged addresses to system calls.

Compatibility

MTE and HWASAN offer a high level of compatibility with existing code bases. We build the Android platform and the Chromium browser with HWASAN with few source code changes.

However, we have observed several cases of incompatibility. In one such case, pointers to a particular type had application-specific metadata stored in the top 16 address bits. In another case, a pointer was cast to double and then back, losing the lower address bits. In one more case, the code computed difference between the addresses of local variables from different stack frames as a way to measure recursion depth. All these cases were easy to fix.

Related Work

With this article I hope to increase the awareness of the concept of memory tagging, as well as ARM's fantastic Memory Tagging Extension, so that other CPU vendors adopt it sooner rather than later. Unlike most other existing hardware security extensions, ARM MTE directly addresses the memory safety bugs, that is, the root cause of many vulnerabilities, not just how attackers happen to exploit their consequences today. Beyond its effectiveness as a mitigation, MTE also serves as an effective bug detection tool that can be deployed in the wild. But even MTE is not a panacea for all classes of memory safety bugs.

Intra-Object-Buffer-Overflow

There are other classes of C/C++ bugs waiting to be dealt with. One such bug class is called *intra-object-buffer-overflow*.

```
struct S {
  int array[5];
  int another_field;
};

int GetInt(int *p, size_t idx) {
  return p[idx];
}
int Foo(S *s) {
  return GetInt(s->array, 5);
}
```

Here, by accessing an array out of bounds we end up reading another field in the same struct. In this case, AddressSanitizer, HWASAN, or MTE will not find the bug because the access happens within the same heap- (or stack-) allocated object. The Undefined Behavior Sanitizer (UBSan) can detect some simper cases, but not the more complex ones like this one because the function `GetInt()` that accesses the memory has lost the static bound information available in `Foo()`. There were multiple attempts to solve this problem (including at least one hardware extension, Intel MPX), but none were practical enough to be widely used.

A potential solution would combine dynamic bounds checking, static analysis (proving that either the code is correct or that dynamic checks are effective), and the banning of certain language constructs (like passing sub-objects without their bound information to unknown functions). For modern C++ code, perhaps the best solution is to replace arrays inside structs or classes with std::array and rely on the runtime for bounds checking.

Type-Confusion

Another bug class not directly addressed by MTE is *type-confusion*.

```
struct Image {
  int pixels[100];
};

struct Secret {
  int sensitive_data[200];
};

Secret *secret = new Secret;
...
DrawOnScreen((Image*) secret);
```

This code performs a cast between incompatible types; the following memory accesses in `DrawOnScreen()` will mistakenly access sensitive data without violating object bounds or lifetimes.

A potential solution is to use a stricter subset of C++ that disallows some invalid casts statically (via compile-time errors) and some other invalid casts dynamically (using a mechanism such as implemented in LLVM CFI).

Uninitialized Memory

A side effect of MTE is that whenever a memory allocation is tagged, it can also be initialized at no extra cost. The new ARM instructions can store memory tags and initialize the memory itself at the same time. Therefore, enabling MTE for an application's heap and stack will mitigate most vulnerabilities from another class, *uses of uninitialized memory*.

However, we do not have to wait for MTE to eradicate this class of bugs. For example, Clang/LLVM 9.0 will have an option [8] to automatically initialize all stack variables.

Safer Languages

No discussion of memory safety in C and C++ can ignore the existence of "safe languages." Java, Go, Swift, and Rust, among others, are indeed much *safer*, and in many cases they are a better choice for developing new software.

But none of them are really *safe*. Go and Swift have data races, Java's huge runtime is itself written in C++, and only Rust comes close to being safe, at a cost of a (subjectively) steeper learning curve.

All of these languages, of course, have the "unsafe" escape hatch. Whenever the unsafe section is used, it turns the language into C, but just slightly worse, because fewer tools, practices, and habits are available for that language to avoid memory safety bugs. Here, again, Rust is probably the best with its support for AddressSanitizer and fuzzing. MTE will be useful for Rust and any other memory-safe language with "unsafe" code.

Besides, the billions of lines of C and C++ code are not going away any time soon.

GWP-ASan

GWP-ASan [9] is another bug detection tool that finds heap-use-after-free and heap-buffer-overflows. It relies on protected guard pages, the old trick used in the Electric Fence Malloc and similar tools. But there is a twist: guarded allocations are sampled. This means that the overhead, and the bug detection probability, can be scaled to be arbitrarily small. The small probability of bug detection can be improved by deploying the tool at large scale in production. We are beginning to detect bugs this way in the Google Chrome browser and other software.

GWP-ASan is not a replacement for AddressSanitizer or HWASAN since it handles a smaller subset of bugs and has very low detection probability, but it finds bugs that evade testing and only manifest in production. In the most performance-critical applications, where even 1% overhead is prohibitively expensive, we will be able to use MTE to implement sampled bug detection similar to GWP-ASan, but with a much lower cost and hence higher sampling and detection rate.

Conclusion

Once available in hardware, the ARM Memory Tagging Extension will reduce C and C++ memory unsafety from disastrous to tolerable. Hopefully, other hardware vendors will implement their variants of memory tagging. Before that happens, don't forget to test your software with all available testing tools (e.g., AddressSanitizer or HWASAN) and fuzzers (e.g., libFuzzer), and harden your binaries in production.

Acknowledgments

I want to thank my colleagues Vlad Tsyrklevich, Dmitry Vyukov, Alexander Potapenko, and Evgeniy Stepanov for helping me prepare this article.

References

[1] K. Serebryany, "Hardware Memory Tagging to Make C/C++ Memory Safe(r)," iSecCon'18: https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/MTE-iSecCon-2018.pdf.

[2] M. Miller, "Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape," BlueHat 2019: https://www.youtube.com/watch?v=PjbGojjnBZQ.

[3] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," 2012 USENIX Advanced Technical Conference (USENIX ATC '12): https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf.

[4] K. Serebryany, "OSS-Fuzz—Google's Continuous Fuzzing Service for Open Source Software," 26th USENIX Security Symposium (USENIX Security '17): https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

[5] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, D. Vyukov, "Memory Tagging and How It Improves C/C++ Memory Safety": https://arxiv.org/pdf/1802.09517.pdf.

[6] Arm A-Profile Architecture Developments 2018: Armv8.5-A: https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a.

[7] HWASAN documentation: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[8] J. F. Bastien, "Automatic Variable Initialization": https://reviews.llvm.org/D54604.

[9] GWP-ASan for Chromium documentation: https://chromium.googlesource.com/chromium/src/+/lkgr/docs/gwp_asan.md.

Interview with Mark Loveless

RIK FARROW

Mark Loveless—aka Simple Nomad—is a security researcher, hacker, and explorer. He has worked in startups, large corporations, hardware and software vendors, and even a government think tank. He has spoken at numerous security and hacker conferences worldwide on security and privacy topics, including Black Hat, DEFCON, ShmooCon, RSA, AusCERT, among others. He has been quoted in television, online, and print media outlets as a security expert, including CNN, *Washington Post, New York Times,* and many others. He also knows They are out to get him. ml@markloveless.net

Rik is the editor of *;login:*. rik@usenix.org

I first met Mark Loveless online, which is appropriate. We were both part of a discussion group of journalists and hackers, although I didn't fit neatly into either group. Over time, I learned some things about Mark, mainly that he was best known for Novell Netware security tricks and hacks.

We had met in person a couple of times in the past, but then I learned that Mark was speaking at Enigma 2019 [1], and we planned to get together a couple of times for dinner. I learned a lot more about Mark, some of which we can reveal here.

Rik Farrow: Do some people still call you by your hacker name?

Mark Loveless: Most people who have gray hair call me Simple Nomad.

RF: When you started your career, were you interested in security?

ML: When I started, the security elements of it were more of a hobby. I liked those elements of it, but I never in a million years expected that's where I'd end up. I kinda fell into it.

My dad brought home an Apple II, because he was a big computer nerd who worked on mainframes, and that's where I started. First cassette tapes, then floppy disks after a while. My dad had formed a warez group with his friends at work. They would buy some software, and my job was to crack the copy protection.

RF: You were a teenager?

ML: Yeah. This was the last of the '70s to the early '80s, and I got really good at it. You could call up these companies and get developers on the phone. I talked to both "Steves" at Apple, but I was more excited when I called the Infocom people who did the game Zork. They were, to me, rock stars. I even talked to Bill Gates once, after I had gone back and forth with their support. Gates said, "You're an idiot, why are you doing it that way?"

RF: LOL!

ML: "I'm a kid, I don't know anything," I said. It's not like they were teaching us anything. In college, I got mainframe assembly and used punch cards. That's where I grew up. I got better at it over time.

RF: Where did you start working?

ML: American Airlines, a job at the help desk. I didn't have the on-paper experience for anything else. I had dropped out of college to be a famous rock star and needed something that would pay the bills. So I just went through the whole technological upheaval. My first experience from the Novell Netware was there.

I worked on the Sabre software, used by travel agencies for reservations, and what that included was a Novell server and some nodes. A lot of the nodes were diskless and booted off the network. There was a gateway machine, with a floppy drive, so it could talk over X.25.

My dad got a modem, and he had an X.25 account. So not long after that, I could use his account. I slowly began to know about how all of Sabre was working. I moved from Oklahoma to Texas because I got a job consulting, working with travel agencies. And that's where this all really took off. I got a lot of exposure to a lot of technology at that point.

I soon moved over to the railroad, Burlington Northern at that time, which later merged with Santa Fe Railroad.

RF: What a lot of people probably don't know is that railroad right-of-ways is where a lot of our communication infrastructure is buried.

ML: Right. This goes back to Civil War times. Wherever they are putting in rail, railroads are putting in telegraph lines. If someone else wants to put up telegraph lines that cross their right-of-way, the railroad can say that you have to hook up to our stuff if you want to cross the train tracks. As a result, the railroad got very smart: we'll just run extra cable along all our tracks, and they would lease this cable to communications companies. At one point, Sprint would advertise their *pin drop* network, that was so quiet: 50% of that ran on Burlington Northern networks. Next, people moved to fiber.

Even though "railroad" seems archaic, you picture coal-fired locomotives...the railroads have a lot of infrastructure. They had the largest IBM mainframe outside of the US government.

RF: You were working with Novell at AA?

ML: Yes, but it was also big time at the railroad. I had been on BBS and hacking forums, and it seemed that everyone was specialized. I noticed there wasn't much on Novell Netware, and I decided to focus there. I had access to some huge servers and huge installations, and there were test systems I could play with too.

RF: That's great.

ML: You could do really fun stuff. That was my introduction to running UNIX I had legitimate access to.

RF: This is in the '90s?

ML: Yes. The railroad had 35,000 employees plus union workers and no security department. It was me and my boss who became the security department.

RF: Many companies don't want security, as they prefer to "keep things simple."

ML: The weirdest thing we ever ran into was a department's mainframe program where the passwords were just four characters and we wanted to increase the length to eight characters. They came back and said, "We have union workers, we know what their typing rate is, and they are doing data entry. We don't want to waste keystrokes. We also have your help desk statistics, and we know how long it takes to do a password reset. Based upon what we think the number of password resets will be, moving to an eight character password will double the help desk workload." They had also calculated the amount of time it would take to type in the *extra* four keys based on the average of their users' typing ability.

We ended up doing a compromise. "How about six characters and we'll buy your department Post-It notes for a year?"

RF: LOL!

ML: Done! We had to spend part of our budget on Post-It notes for inter-office bribery. We printed up Post-It notes with the number of the help desk and "Do not write your password on this" printed on them.

That's what security was like back then. Portions of upper management would wonder, why do you even need passwords? Everyone knows our rates as we are required to publish them because of the DOT.

RF: So they weren't worried about someone coming in and changing all their rates?

ML: Exactly! But obviously, the thing that really cemented the security department in the company was when we had a virus outbreak and it affected hundreds of computers. We manned a hotline and came up with a battle plan. We would have a war room, we would fix this. We gained a lot of street cred, so to speak, from that.

I was still doing research on the side, the Novell Netware Hack-FAQ, and all that stuff. NMRC.org is still technically up and running.

And I was finding security bugs in software we were evaluating and using internally at the railroad. I reported a bug to a division of Bindview Corporation.

RF: What was Bindview?

ML: Bindview wrote management software computer systems. They also had an Internet security scanner, and I found something in there. So I wrote them that I was going to disclose the weakness. They patched the bug, and they ended up offering me a job.

I was thrilled because the railroad was going to have me work 12-hour shifts because of Y2K. By then the security department had grown, so it would me and one other person working round-the-clock, 12-hour shifts, for two weeks straight. Of course, nothing happened, but I left at the beginning of December.

At that point, I had been Simple Nomad on the Internet. The railroad was familiar with that Simple Nomad guy, and they could care less. They said just keep it separate, because I was helping secure their systems. I didn't report anything I knew about that would endanger the railroad. Or I would make certain that things were patched before I'd go public with it. They were very cool about it.

RF: Could you talk about responsible disclosure? You worked a lot with Novell.

ML: Novell had a reputation with hackers. Back in the day, they did this thing with a couple of Russian hackers who reported bugs. Novell hired the hackers, had them sign nondisclosure agreements, then fired them. The NDA were lifetime NDAs, which meant that Novell didn't have to patch what the hackers had found. They weren't US citizens, so what the Russians did instead was to pass the information to me, because I had a website.

So I was very wary about contacting Novell. And I was keeping my identities separate.

One time I contacted Novell by email, telling them that I've got this flaw, and I wanted to talk with them on the telephone about it. They say okay and gave me an 800 number. Being the paranoid hacker type, I knew that even though I could suppress the ANI, the automatic numeric identifier, what is now called caller-ID, in local exchanges or normal long distance, they could get the calling number because the receiving end of the 800 number was paying the bill. I decided that I was not calling on that 800 number.

Novell was using a PBX system, called Meridian Mail, and I had sort of a zero-day for that system. I could dial in, go through a sequence of numbers and steps, and I would get a dial tone, so I could dial out. I would use a computer to handle the sequence, and then I could call long distance for free from that PBX. I used that trick with Novell. I called up the PBX, dropped out to a dial tone, looked up the number of the security person using the Meridian Mail system, then called him up. I asked him for the extension for the conference call, and to him it appeared I was calling from an internal number. "Where are you?" he asked, and I answered, "I'm in Texas." I got the number for the conference call.

I was paranoid that Novell was going to do something.

RF: Because of what Novell had done to the Russian hackers, this seems likely.

ML: I reacted to that. Sometimes Novell employees wouldn't leak bugs but would say "Look at this."

RF: They would point you in the right direction.

ML: Correct. I had problems with other companies too. Microsoft at the time was weird about stuff when you reported it to them. I tried reporting something to ISS, who had a security scanner, and they got really weird about it. We just backed away from that. We were doing this on the side as a hobby, and they wanted us to present our disclosure policy signed with our PGP key. And that just seemed too weird.

RF: You worked for MITRE, the defense contractor that publishes the CVEs for bugs. What was that like?

ML: Weird. I did work on standards like CVE and CWE mainly, and dabbled in a few other standards. But a lot of what I did also involved working for the security department responsible for answering to attacks against MITRE's systems. That group only dealt with APT attacks, and that was some eye-opening stuff. I can't go into much detail, but I can expand on some general concepts. Most of them were Chinese APT groups; we would refer to them as campaigns. We tracked dozens and dozens of individual little things from phishing email subject lines, various IP addresses, recipient lists, compilers used to compile backdoors, and on and on, and patterns emerged. We actually didn't really care *who* was attacking us per se; we mainly wanted to know if we'd seen them before so we could anticipate their next move.

Granted, there were all kinds of clues that most of the attacks were Chinese sponsored, but I had tons of friends in the security community saying, "APT is made up, people can fake their IP address, it's not the Chinese, and APT isn't real." I'd have to bite my tongue since most of the proof that it was, for example, Chinese sponsored was from classified briefs and whatnot. I mean I had a security clearance.

I think the one I hated the most was the argument that these attackers didn't live up to the "A" part of APT. They weren't "advanced." I'd hear from friends that "they don't use zero-day all the time, so they aren't advanced. I'd be using wicked cool zero-day." Oh no, you would not. My background was in hacking—I was breaking into systems in my youth—and you *never* wasted a zero-day on a target unless you really wanted in there and all of the low-level stuff didn't work. It was like these people who were "playing offense" by doing penetration testing really thought they were actually hacking.

Hackers, and these APT actors as well, did not have Statements of Work to not attack production systems or to limit themselves to a two-week engagement. No, hackers and APT actors would cheat, commit felonies, take months and months to get in, hit production systems, lie, intimidate, steal, whatever. When you reverse engineered an APT backdoor and found your internal DNS servers' IP addresses hard-coded into the exe, you knew you were dealing with someone advanced. They'd been in before, they knew your internal network layout, and so what if they didn't use a zero-day to get in. Advanced meant they played serious and played to win at any cost. At times I wanted to punch some of my friends in the mouth. "Not advanced" my ass. Now of course it is all out in public, and everyone accepts APT as real.

I remember my first classified meeting and how disappointed I was. I mean everything said in that meeting I'd already known well before working there. I think the only thing I didn't know was the fancy names of everything. Goofy code names. I was like shit, so no reverse engineering UFOs or something?

Speaking of which, I had no idea how I got a security clearance. I'd been under investigation by more than one government agency for hacking, and they still approved my DOD-sponsored security clearance. I know this in part because a few years earlier the NSA tried to recruit me, and I stated, "I can't work for you, I have a file, I've been investigated," and they were like, "Well sure, we've read it, and yes, you'll never work for law enforcement, but you can work for us, we're the good guys, we're the NSA." They actually said that, "We're the good guys." Whatever, I turned them down.

Truthfully, the most interesting work I did at MITRE was all classified, and I can't talk about it, but it was some really cool stuff. However, it was nothing to do with UFOs unfortunately.

RF: Also, you worked for Duo Security. What did you do for them?

ML: Yeah. After MITRE I went to Duo Security. The idea was Dug's (Dug Song, the CEO) that we form a Duo research group and do security research like the old days—make it fun and entertaining. Get content out and speak at conferences, do press interviews, all that. We'd be smart security people doing cool stuff who happened to work at Duo Security.

In essence we were doing a form of marketing. The Marketing Department loved us; well, we were probably an ass pain to work with, but once we got to know each other we had a blast. I loved working there, the product was cool, and if you were at a conference wearing a Duo shirt people would run up and tell you they loved you and would want selfies with you.

Originally the plan was to IPO, and they even were hiring C-levels with that in mind, but then the Cisco offer came in and that ended that. Many of us were heartbroken, since Cisco is a huge corporation with a radically old-style corporate infrastructure. Sexy, cloud-based startup to a division of an old school corporation. Sure they were trying to go a lot more modern, hence the buyout, but it still hurt. By then the focus was on making Duo look both attractive and useful to Cisco's bottom line, so my Duo job rapidly went away. I had the opportunity to work in one of several Cisco departments, but instead I left. I wanted to work some place cool like Duo used to be.

I did the "funemployment" thing after Cisco and started blogging and whatnot. I probably could have done that for a year or so since I had my buyout money. Not a lot of money but "forget working for a while" money. Then I got a call from Kathy Wang, whom I've known from the hacker and security conference scene for nearly two decades, and she told me about GitLab, and it sounded too good to pass up. Plus the employee base is 100% remote, all cloud-based corporate infrastructure, so it has a modern and forward-thinking unicorn startup vibe and everyone is pretty damn awesome. I had to go for it.

I just started there, and I'll be working on research in areas where my skills are, and doing other stuff like conferences, blogging, and whatever. It's nice to work at a place that is extremely open and really heading in a good direction. I am still going to blog and speak at conferences, and life should be a lot of fun.

Reference
[1] Physical OPSEC as a Metaphor for Infosec, Enigma 2019: https://www.usenix.org/conference/enigma2019/presentation/loveless.

Datacenter RPCs Can Be General and Fast

ANUJ KALIA, MICHAEL KAMINSKY, AND DAVID G. ANDERSEN

Anuj Kalia is a PhD student in the Computer Science Department at Carnegie Mellon University, advised by David Andersen and Michael Kaminsky. He is interested in high-performance computer systems and networks. akalia@cs.cmu.edu

David G. Andersen is an Associate Professor of Computer Science at Carnegie Mellon University. He completed his MS and PhD degrees at MIT, and holds BS degrees in biology and computer science from the University of Utah. In 1995, he co-founded an Internet service provider in Salt Lake City, Utah. dga@cs.cmu.edu

Michael Kaminsky is a Senior Research Scientist at Intel Labs and an adjunct faculty member of the Computer Science Department at Carnegie Mellon University. He is part of the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS), based in Pittsburgh, PA. His research interests include distributed systems, operating systems, and networking. michael.e.kaminsky@intel.com

"Using performance to justify placing functions in a low-level subsystem must be done carefully. Sometimes, by examining the problem thoroughly, the same or better performance can be achieved at the high level."

—"End-to-End Arguments in System Design," J. H. Saltzer, D. P. Reed, and D. D. Clark, 1984

It is commonly believed that datacenter networking software must sacrifice generality to attain high performance. The popularity of specialized distributed systems designed specifically for niche technologies such as RDMA, lossless networks, FPGAs, and programmable switches testifies to this belief. In this article, we show that such specialization is not necessary. eRPC is a new general-purpose remote procedure call (RPC) library that offers performance comparable to specialized systems while running on commodity CPUs in traditional datacenter networks based on either lossy Ethernet or lossless fabrics.

eRPC performs well in three key metrics: message rate for small messages; bandwidth for large messages; and scalability to a large number of nodes and CPU cores. It handles packet loss, congestion, and background request execution. In microbenchmarks, one CPU core can handle up to 10 million small RPCs per second or send large messages at 75 Gbps. We port a production-grade implementation of Raft state machine replication to eRPC without modifying the core Raft source code. We achieve 5.5 μs of replication latency on lossy Ethernet, which is faster than or comparable to specialized replication systems that use programmable switches, FPGAs, or RDMA.

Squeezing the best performance out of modern, high-speed datacenter networks has meant painstaking specialization that breaks down the abstraction barriers between software and hardware layers. The result has been an explosion of co-designed distributed systems that depend on niche network technologies, including RDMA, FPGAs, and programmable switches. Add to that new distributed protocols with incomplete specifications, the inability to reuse existing software, hacks to enable consistent views of remote memory—and the typical developer is likely to give up and just use kernel-based TCP.

These specialized technologies were deployed with the belief that placing their functionality in the network would yield a large performance gain. Our work shows that a general-purpose RPC library called eRPC can provide state-of-the-art performance on commodity Ethernet datacenter networks without additional network support. This helps inform the debate about the utility of additional in-network functionality versus purely end-to-end solutions for datacenter applications.

The goal of our work is to answer the question: can a general-purpose RPC library provide performance comparable to specialized systems? Our solution is based on two key insights. First, we optimize for the common case, i.e., when messages are small, the network is congestion-free, and RPC handlers are short. Handling large messages, congestion, and long-running RPC handlers requires expensive code paths, which eRPC avoids whenever possible.

Several eRPC components, including its API, message format, and wire protocol, are optimized for the common case. Second, restricting each flow to at most one bandwidth-delay product (BDP) of outstanding data effectively prevents packet loss caused by switch buffer overflow for common traffic patterns. This is because datacenter switch buffers are much larger than the network's BDP. For example, in our two-layer testbed that resembles real deployments, each switch has 12 MB of dynamic buffer, while the BDP is only 19 kB.

eRPC (efficient RPC) is available at https://erpc.io.

Background and Motivation

We first discuss aspects of modern datacenter networks relevant to eRPC and the limitations of existing networking options that underlie the need for eRPC.

High-Speed Datacenter Networking

Modern datacenter networks provide tens of Gbps per-port bandwidth and a few microseconds of round-trip latency. They support polling-based network I/O from userspace, eliminating interrupts and system call overhead from the datapath. eRPC uses userspace networking with polling, as in most prior high-performance networked systems.

Lossless fabrics. Lossless packet delivery is a link-level feature that prevents congestion-based packet drops. For example, priority-based flow control (PFC) for Ethernet prevents a link's sender from overflowing the receiver's buffer by using pause frames. Some datacenter operators, including Microsoft, have deployed PFC at scale. Unfortunately, PFC comes with a host of problems, including head-of-line blocking, deadlocks due to cyclic buffer dependencies, and complex switch configuration. In our experience, datacenter operators are often unwilling to deploy PFC due to these problems.

Switch buffer >> BDP. The increase in datacenter bandwidth has been accompanied by a corresponding decrease in round-trip time (RTT), resulting in a small BDP. Switch buffers have grown in size to the point where "shallow-buffered" switches that use SRAM for buffering now provide tens of megabytes of shared buffer. Much of this buffer is dynamic, i.e., it can be dedicated to an incast's target port, preventing packet drops from buffer overflow. For example, in our two-layer 25 GbE testbed that resembles real datacenters, the RTT between two nodes connected to different top-of-rack (ToR) switches is 6 μs, so the BDP is 19 kB. In contrast to the small BDP, the Mellanox Spectrum switches in our cluster have 12 MB in their dynamic buffer pool. Therefore, the switch can ideally tolerate a 640-way incast. The popular Broadcom Trident-II chip used in datacenters at Microsoft and Facebook has a 9 MB dynamic buffer.

In practice, we wish to support approximately 50-way incasts: congestion control protocols deployed in real datacenters are tested against comparable incast degrees. This is much smaller than 640, allowing substantial tolerance to technology variations, i.e., we expect the switch buffer to be large enough to prevent most packet drops in datacenters with different BDPs and switch buffer sizes.

Limitations of Existing Options

Software options. Existing datacenter networking software options sacrifice performance or generality, preventing unmodified applications from using the network efficiently. On the one hand, fully general networking stacks such as mTCP [4] allow legacy sockets-based applications to run unmodified. Unfortunately, they leave substantial performance on the table, especially for small messages. On the other extreme, fast packet I/O libraries such as DPDK provide only unreliable packet delivery.

Our prior RPC design—FaSST RPCs [6]—was the precursor to eRPC. Like eRPC, FaSST RPCs use datagram packet I/O, but they assume a lossless network and lack several features such as multi-packet messages and congestion control. eRPC's main contribution is a design that performs well in lossy networks and supports the aforementioned features with low overhead.

Hardware options. Numerous recent research proposals co-design distributed systems with special network hardware technologies like RDMA, FPGAs, and programmable switches for fast communication. While there are advantages of co-design, such specialized systems are unfortunately very difficult to design, implement, and deploy. Specialization breaks abstraction boundaries between components, which prevents reuse of components and increases software complexity. Building distributed systems requires tremendous programmer effort, and co-design typically mandates starting from scratch, with new data structures, consensus protocols, or transaction protocols. Co-designed systems often cannot reuse existing codebases or protocols, tests, formal specifications, programmer hours, and feature sets.

eRPC Overview

eRPC implements RPCs on top of a transport layer that provides basic unreliable packet I/O, such as UDP over Ethernet networks or InfiniBand's Unreliable Datagram transport. A userspace NIC driver is required for good performance. Our primary contribution is the design and implementation of end-host mechanisms and a network transport (e.g., wire protocol and congestion control) for RPCs.

eRPC's requests execute at most once and are asynchronous to avoid stalling on network round trips; intra-thread concurrency is provided using an event loop. RPC servers register request handler functions with unique request types; clients use these request types when issuing RPCs, and get continuation callbacks on RPC completion. Users store RPC messages in opaque, DMA-capable buffers provided by eRPC; a library that provides marshalling and unmarshalling can be used as a layer on top of eRPC.

Each user thread that sends or receives requests creates an exclusive RPC endpoint. Each endpoint contains an RX and TX queue for packet I/O, an event loop, and several sessions. A *session* is a one-to-one connection between two RPC endpoints, i.e., two user threads. The client endpoint of a session is used to send requests to the user thread at the other end. A user thread may participate in multiple sessions, possibly playing different roles (i.e., client or server) in different sessions.

User threads act as "dispatch" threads: they must periodically run their endpoint's event loop to make progress. The event loop performs the bulk of eRPC's work, including packet I/O, invoking request handlers and continuations, congestion control, and management functions. To avoid blocking on a long-running request handler, eRPC provides a pool of background threads to handle request types that are annotated by the user as long-running, typically over a few microseconds.

Client control flow. `rpc->enqueue_request()` queues a request on a session, which is transmitted when the user runs `rpc`'s event loop. On receiving the response, the event loop copies it to the client's response buffer and invokes the continuation callback.

Server control flow. The event loop of the `rpc` that owns the server session invokes (or dispatches) a request handler on receiving a request. We allow *nested* RPCs, i.e., the handler need not enqueue a response before returning. It may issue its own RPCs and call `enqueue_response()` for the first request later when all dependencies complete.

eRPC design

Achieving eRPC's performance goals requires careful design and implementation. We discuss three aspects of eRPC's design in this section: scalability of our networking primitives, the challenges involved in supporting zero-copy transfers, and the design of sessions. The next section discusses eRPC's wire protocol and congestion control. A recurring theme in eRPC's design is that we optimize for the common case, i.e., when request handlers run in dispatch threads, RPCs are small and the network is congestion-free.

Scalability considerations. We chose plain packet I/O instead of RDMA writes to send messages in eRPC. eRPC holds connection state in large CPU caches, which allows scaling to a large number of connections. In contrast, RDMA requires maintaining per-connection in much smaller (~2 MB) on-NIC caches, which does not scale well to large clusters. Our experiments show that whereas RDMA performance drops by up to 50% with 5000 connections, eRPC's performance remains constant with even 20,000 connections. In addition, eRPC uses modern NIC features (e.g., multi-packet receive queues) to guarantee a constant NIC memory footprint per local CPU core.

Zero-copy challenges. eRPC supports zero-copy transfers from DMA-capable buffers provided to applications. Supporting zero-copy along with eRPC's feature set required solving several challenges, such as reasoning about DMA buffer ownership in the presence of retransmissions. Since eRPC transfers packets directly from application-owned buffers, care must be taken so that buffer references are never used by eRPC after buffer ownership is returned to the application. The following example demonstrates the problem: Consider a client that falsely suspects packet loss and retransmits its request. The server, however, received the first copy of the request, and its response reaches the client before the retransmitted request is sent out by the client's NIC. Before processing the response and invoking the continuation, we must ensure that there are no references to the request buffer in the client's NIC DMA queue.

The conventional approach to ensure DMA completion is to use "signaled" packet transmission, in which the NIC writes completion entries to the TX completion queue. Unfortunately, doing so increases NIC and PCIe resource use, so we use unsignaled packet transmission in eRPC. Our method of ensuring DMA completion with unsignaled transmission is in line with our design philosophy: we choose to make the common case (no retransmission) fast, at the expense of invoking a more-expensive mechanism to handle the rare cases. We flush the TX DMA queue after queueing a retransmitted packet, which blocks until all queued packets are DMA-ed. This flush is moderately expensive (≈2 μs), but it is called only during rare retransmissions.

Sessions. Each session maintains multiple outstanding requests to keep the network pipe full. Concurrent requests on a session can complete *out-of-order* with respect to each other. This avoids blocking dispatch-mode RPCs behind a long-running background RPC. We support a constant number of concurrent requests (default = 8) per session; additional requests are transparently queued by eRPC.

eRPC limits the number of unacknowledged packets on a session to implement end-to-end flow control, which reduces switch queueing. Allowing *BDP/MTU* unacknowledged packets per session ensures that each session can achieve line rate.

Transport Layer

One of eRPC's main contributions is the design of low-overhead transport layer components, including end-to-end reliability and congestion control, discussed next. eRPC uses a *client-driven* protocol, meaning that each packet sent by the server is in response to a client packet. This shifts most transport complexity to clients, freeing server CPU that is often more valuable.

End-to-end reliability. For simplicity, eRPC treats reordered packets as losses by dropping them. Datacenter networks typically preserve intra-flow ordering even with network load balancing (e.g., ECMP), except during rare route churn events. On suspecting a lost packet, the client rolls back the request's wire protocol state using a simple Go-Back-N mechanism, and retransmits from the updated state. The server never runs the request handler for a request twice, guaranteeing at-most-once RPC semantics.

Congestion control. Congestion control for datacenter networks aims to reduce switch queueing, thereby preventing packet drops and reducing RTT. While software-based congestion control has been considered to be slow in the past, we show that optimizing for uncongested networks, and recent advances in software rate limiting allow congestion control with little overhead.

eRPC uses a congestion control algorithm for high-speed datacenter networks called Timely [7], although other algorithms may also be supported in the future. Timely uses packet RTT as the congestion signal, and it updates session transmission rates based on RTT statistics. We use a software rate limiter for enforcing the transmission rate suggested by Timely.

Datacenter networks are typically uncongested, so we optimize congestion control for uncongested networks. Recent datacenter studies support this claim. For example, Roy et al. [9] report that 99% of all Facebook datacenter links are less than 10% utilized at one-minute timescales.

When a session is uncongested, RTTs are low and Timely's computed rate for the session stays at the link's maximum rate; we refer to such sessions as uncongested. If the RTT of a packet received on an uncongested session is smaller than Timely's low threshold (~50 μs), below which it performs additive increase, we do not perform a rate update. For uncongested sessions, we transmit packets directly instead of placing them in the rate limiter.

Evaluation

eRPC is implemented in 6200 SLOC of C++, excluding tests and benchmarks. We evaluated eRPC's performance both in microbenchmarks and real applications. The numbers presented here were measured on an eight-node cluster with Intel Xeon servers, with Mellanox ConnectX-5 NICs connected to a 40 GbE switch.

Microbenchmarks

Latency. For small 32-byte RPCs, eRPC's median latency is 2.3 μs, which is only 300 ns more than 32-byte RDMA reads.

Bandwidth. To measure eRPC's bandwidth for large messages, we use a client that sends large requests to a server thread, which replies with small, 32-byte responses. With 8 MB requests, eRPC saturates the network's 40 Gbps with one client thread. On a faster 100 Gbps InfiniBand network, we measured that eRPC can achieve 75 Gbps in the same experiment.

In addition, our microbenchmarks showed that eRPC also provides:

- **High scalability**. On a large 100-node cluster, eRPC's performance scales to 20,000 connections per-node.
- **Incast tolerance**. eRPC's congestion control successfully reduces switch queueing with up to 50-way incasts.
- **Packet loss tolerance**. eRPC delivers good bandwidth with a packet loss rate of up to 10^{-5}.

Raft over eRPC

To evaluate whether eRPC can be used in real applications with unmodified existing storage software, we built a state machine replication system using an open-source implementation of Raft [8].

State machine replication (SMR) is used to build fault-tolerant services. An SMR service consists of a group of server nodes that receive commands from clients. SMR protocols ensure that each server executes the same sequence of commands and that the service remains available if servers fail. Raft is such a protocol that takes a *leader*-based approach: absent failures, the Raft replicas have a stable leader to which clients send commands; if the leader fails, the remaining Raft servers elect a new one. The leader appends the command to replicas' logs, and it replies to the client after receiving ACKs from a majority of replicas.

SMR is difficult to design and implement correctly: the protocol must have a specification and a proof (e.g., in TLA+), and the implementation must adhere to the specification. We avoid this difficulty by using an existing implementation of Raft [1]. (It had no distinct name, so we term it LibRaft.) We did not write LibRaft ourselves; we found it on GitHub and used it as is. LibRaft is well tested with fuzzing over a network simulator and 150+ unit tests. Its only requirement is that the user provide callbacks for sending and handling RPCs—which we implement using eRPC. Porting to eRPC required no changes to LibRaft's code.

Measurement	**System**	**Median** (μs)	**99%** (μs)
Measured at client	NetChain	9.7	N/A
	eRPC	5.5	6.3
Measured at leader	ZabFPGA	3.0	3.0
	eRPC	3.1	3.4

Table 1: Latency comparison for replicated PUTs

We compare against recent consistent replication systems that are built from scratch for two specialized hardware types. First, NetChain [5] implements chain replication over programmable switches. Second, Consensus in a Box [3] (called ZabFPGA here) implements ZooKeeper's atomic broadcast protocol [2] on FPGAs.

Workloads. We mimic NetChain and ZabFPGA's experiment setups for latency measurement: we implement a three-way replicated in-memory key-value store with 16-byte keys and 64-byte values, and use one client to issue PUT requests. The replicas' command logs and key-value store are stored in DRAM. We compare eRPC's performance on CX5 against their published numbers because we do not have the hardware to run NetChain or ZabFPGA. Table 1 compares the latencies of the three systems.

Comparison with NetChain. NetChain's key assumption is that software networking adds 1–2 orders of magnitude more latency than switches [5]. However, our experiments show that eRPC adds 850 ns, which is comparable to latency added by current programmable switches (~800 ns).

Raft's latency over eRPC is 5.5 μs, which is substantially lower than NetChain's 9.7 μs. This result must be taken with a grain of salt: on the one hand, NetChain uses NICs that have higher latency than our ConnectX-5 NICs. On the other hand, it has numerous limitations, including key-value size and capacity constraints, serial chain replication whose latency increases linearly with the number of replicas, absence of congestion control, and reliance on a complex and external failure detector.

Comparison with ZabFPGA. Although ZabFPGA's SMR servers are FPGAs, the clients are commodity workstations that communicate with the FPGAs over slow kernel-based TCP. For a challenging comparison, we compare against ZabFPGA's commit latency measured at the leader, which involves only FPGAs. In addition, we consider its "direct connect" mode, where FPGAs communicate over point-to-point links (i.e., without a switch) via a custom protocol. Even so, eRPC's median leader commit latency is only 3% worse.

Conclusion

eRPC is a fast, general-purpose RPC system that provides an attractive alternative to putting more functions in network hardware and specialized system designs that depend on these functions. eRPC's speed comes from prioritizing common-case performance, carefully combining a wide range of old and new optimizations, and the observation that switch buffer capacity far exceeds datacenter BDP. eRPC delivers performance that was until now believed possible only with lossless RDMA fabrics or specialized network hardware, and it allows unmodified applications to perform close to the hardware limits.

References

[1] C Implementation of the Raft Consensus Protocol, 2019: https://github.com/willemt/raft.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, June 2010: https://www.usenix.org/legacy/event/usenix10/tech/full_papers/Hunt.pdf.

[3] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a Box: Inexpensive Coordination in Hardware," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, May 2016: https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-istvan.pdf.

[4] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, April 2014: https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-jeong.pdf.

[5] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-Free Sub-RTT Coordination," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, April 2018: https://www.usenix.org/system/files/conference/nsdi18/nsdi18-jin.pdf.

[6] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, November 2016: https://www.usenix.org/system/files/conference/osdi16/osdi16-kalia.pdf.

[7] R. Mittal, T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "TIMELY: RTT-Based Congestion Control for the Datacenter," in *Proceedings of the ACM SIGCOMM*, August 2015.

[8] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '14)*, June 2014: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf.

[9] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *Proceedings of the ACM SIGCOMM*, August 2015.

The Flipside

A Bit Flip Saved Is Power and Lifetime Earned

DANIEL BITTMAN, PETER ALVARO, DARRELL D. E. LONG, AND ETHAN L. MILLER

Daniel Bittman is a PhD candidate at the University of California, Santa Cruz, studying under Ethan Miller, Darrell Long, and Peter Alvaro. His research interests include operating systems, non-volatile memory, concurrency, and systems security. He is currently working on developing operating system techniques for improving the use of persistent memory, reducing power and wear for persistent memory, and studying non determinism in distributed systems. dbittman@ucsc.edu

Peter Alvaro is an Assistant Professor of Computer Science at the University of California, Santa Cruz, where he leads the Disorderly Labs research group (disorderlylabs.github.io). His research focuses on using datacentric languages and analysis techniques to build and reason about data-intensive distributed systems in order to make them scalable, predictable, and robust to the failures and nondeterminism endemic to large-scale distribution. Peter earned his PhD at the University of California, Berkeley, where he studied with Joseph M. Hellerstein. He is a recipient of the NSF CAREER Award and the Facebook Research Award. palvaro@ucsc.edu

We have an opportunity to rethink, from scratch, the design of our data structures. New byte-addressable non-volatile memory (BNVM) technologies promise the construction of systems with large persistent memories, potentially improving reliability and performance. With these technologies come new characteristics that deviate from those of flash and spinning disk—and with new characteristics come new optimization goals. In particular, the read/write cost disparity and fine granularity of updates allows us to save power and wear by reducing the bits flipped during writes to memory. Targeting *these* optimizations by formulating new data structure design and implementation strategies instead of relying on existing ideas will be vital for BNVM technology to reach its full potential. We modified a full-system simulator to count bit flips during program operation, opening the door for future research to design, construct, and evaluate data structures for these new goals.

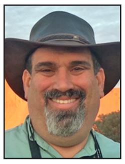
New Optimization Targets

As byte-addressable non-volatile memories (BNVMs) become common, it is increasingly important that systems are optimized to leverage their strengths and avoid stressing their weaknesses. Historically, such optimizations have included reducing the number of writes performed, either by designing data structures that require fewer writes or by using hardware techniques such as caching to reduce writes. While still worthwhile, write-reduction fails to take advantage of a key optimization made by the memory controller in those non-volatile memories.

Some technologies, including phase-change memory (PCM), have a significant disparity between the cost—be it power, time, or wear—of reading a cell and writing a cell. When these technologies also support fine granularity updates, they can make use of a clever optimization: checking if a cell already contains the new, target value [10] instead of blindly overwriting it. Such an optimization yields a change in perspective on what is costly when operating on BNVM; it is not the writes themselves so much as *bits flipped* during the writes. In PCM, for example, changing a cell consumes 15.7–22.5x more power than reading a cell [5, 6] in addition to causing wear-out (a significant problem for PCM as it has limited endurance).

Therefore, system designers ought to consider the effects of bit flips when building systems for BNVM, both when considering the target use-case for the hardware and picking an appropriate combination of BNVM and DRAM, but also when considering the design of the *software* that issues the writes in the first place. To get a sense of how write patterns might affect power consumption, Figure 1 shows a model of power consumption of DRAM and PCM under a varying number of bit flips per second. The power consumption of PCM depends heavily on the bit flips per second, while DRAM's power consumption is relatively independent. We also see that DRAM requires a high "maintenance" power (due to the need to refresh), whereas PCM does not. The choice to use a particular technology could depend,

Dr. Darrell D. E. Long is Distinguished Professor of Engineering at the University of California, Santa Cruz. He holds the Kumar Malavalli Endowed Chair of Storage Systems Research and is Director Emeritus of the Storage Systems Research Center. His broad research interests include many areas of mathematics and science, and in the area of computer science include data storage systems, operating systems, distributed computing, reliability and fault tolerance, and computer security. He is currently Editor-in-Chief of the *Letters of the Computer Society*, and Editor-in-Chief Emeritus of the *ACM Transactions on Storage*.
darrell@ucsc.edu


Ethan L. Miller is a Professor of Computer Science in the Jack Baskin School of Engineering, where he holds the Veritas Presidential Chair in Storage. He is the Director of the NSF I/UCRC Center for Research in Storage Systems and the Director of the Storage Systems Research Center. He was a member of the RAID project at UC Berkeley, where he did his PhD on a decentralized parallel file system for high-end scientific computing. His current research interests include archival storage systems, file systems for storage-class memories, scalable view-based metadata management, and issues in reliability, scalability, and security, both for short-term and archival storage.
elm@ucsc.edu

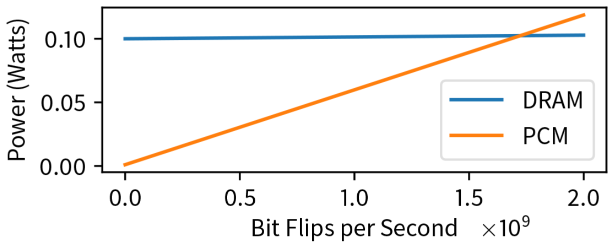


Figure 1: Power use of 1 GB devices as a function of flips per second [2]. DRAM's power consumption is largely proportional to memory size whereas PCM's is largely proportional to bit flip rate.

therefore, on the expected write patterns to memory, since there is a crossover point on the graph. This is particularly important for Internet of Things (IoT) devices, where power consumption and conservation is critical.

Another significant advantage to avoiding bit flips is reducing memory cell wear-out. BNVM technologies typically have a maximum number of lifetime writes, and fewer writes means a longer lifetime. However, we can make use of hardware techniques such as row shifting [11] to *spread out* the "hot spots," thus translating a reduction of bit flips in part of a word to an average reduction across the entire word.

Optimizing software for a novel optimization goal such as bit flipping requires rethinking some core design ideas. The need to incorporate an underlying technology's characteristics into software is not new; indeed, it has been seen with block-oriented sequential access data structures for disk and trading writes for random reads in flash. For BNVM, research has focused on reducing writes while often ignoring the importance of the bits flipped by the writes. Prior work that looks at the bit flips directly either merely considers hardware solutions [4, 7, 8] or suggests that write reduction is a good analog for bit flip reduction [3]. While hardware techniques are certainly a more *general* solution to the problem, they lack the semantic knowledge available to software to improve bit flip reduction. Similarly, write reduction by itself *may* reduce bit flips, but we have found that this is not always the case [1, 2].

Once we accept that bit flips play a significant role in the power consumption and wear of BNVM technologies, we must ask the questions, what changes can we make to software to improve bit flip reduction, and how do we measure our work? We approached this problem by focusing on optimizing *data structures* for bit flip reduction, since data organization plays a large role in the writes that make it to memory. Although data writes themselves significantly affect bit flips, these writes are often unavoidable (since the data must be written), while data structure writes are more easily optimized (as we see in existing BNVM data structure research). Furthermore, data structures often require a significant number of updates over time, while data is often written once (since we can reduce writes by updating pointers instead of moving data). Thus the overall proportion of bit flips caused by data writes may drop over time as data structures are updated.

To show that bit flips can be optimized for, and to explore several techniques we thought of to do so, we designed and built several data structures and evaluated them by counting their bit flips and writes at the memory controller, as well as measuring the performance of each. While our earlier work [2] focused on manual instrumentation of code to count bit flips, we decided to use a full-system simulator (Gem5) to count bit flips so we could take into account caching layers and compiler optimizations. More details for our current work, including more experiments, data structures, and bit flip reduction techniques, are available [1].

Pointer Distance in Data Structures

Data structures are often made up of a significant number of pointers. Take the doubly linked list, for instance: each node contains two pointers, one forward and one back. A clever technique to reduce the memory footprint is to XOR the pointers together, storing pointer *distance* instead of absolute addresses. This is known as an XOR linked list [9]. The program can still traverse the list in either direction with two adjacent pointers, but the overhead of the node is halved. When XOR linked lists were originally proposed, there wasn't much of an advantage to using them beyond a modest memory saving. However, they reduce bit flips by not only cutting the number of writes in half but also zeroing-out many of the bits contained within a standard pointer value.

We can extend XOR linked lists into the domain of indexing structures by reapplying the pointer distance technique to binary search trees. Binary search trees are commonly used for data indexing and support range queries, and they allow efficient lookup and modification, as long as they are balanced. In a standard red-black tree (RBT), for example, a node stores a left child pointer, a right child pointer, and a parent pointer. We can instead store "xleft" and "xright" by XORing the left child pointer with the parent pointer and the right child pointer with the parent pointer, respectively. This reduces the size of the node from three pointers to two pointers while still allowing easy up and down traversal (and thus keeping the benefits of the three-pointer approach), and saves bit flips for the same reason as the XOR linked list.

Tree traversal and update operations in the XOR red-black tree are largely the same as in a standard red-black tree implementation. However, since we are storing XORs of pointers and not the pointers themselves, some additional effort from the programmer is required to "decode" the stored values into a "true" address. Additionally, while traversal *down* the tree is straightforward (given a parent node pointer and a current node's xleft value, we can traverse to the left child by XORing together the parent pointer and the xleft value), traversing up the tree is more difficult. Given a current node and one of its children, the traversal algorithm needs to know *which* child it is. Fortunately, we can make use of the node ordering of a binary search tree to determine which child we have, thus enabling upward traversal.

Results and Discussion

We implemented our XOR red-black tree design alongside a traditional red-black tree and evaluated both under a full-system simulator—Gem5—which simulates the cache hierarchy and allowed us to collect bit flip numbers on unmodified code, thus more faithfully representing the behavior of a system. We found that the programmer overhead required for dealing with pointer distance was not high, especially when considering the abundance of tooling that could be used and harnessed to make debugging easier. The patch to Gem5 to enable bit flip counting at the memory controller was similarly straightforward, but opens up a significant amount of evaluation and research that can be done to evaluate the bit flipping characteristics of existing systems and data structure design (https://gitlab.soe.ucsc.edu/gitlab/crss/opensource-bitflipping-fast19).

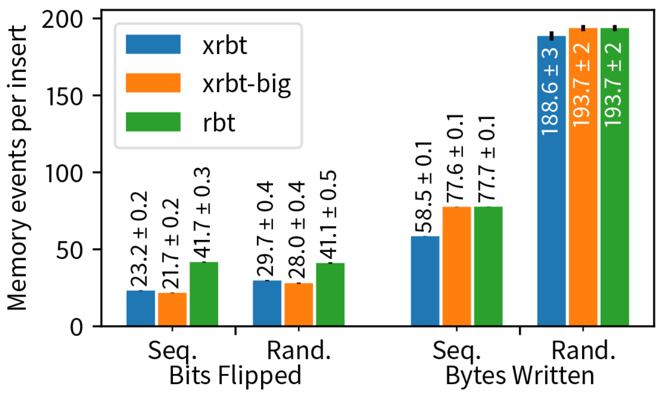


Figure 2: Memory characteristics of XOR red-black trees compared to normal red-black trees (lower is better). The XOR technique significantly reduces bit flips.

Figure 2 shows the bit flips and bytes written of `xrbt` (our XOR RBT implementation) and `rbt` (our standard RBT) under sequential and random inserts of one million unique items. We also evaluated `xrbt-big`, which was the same implementation as `xrbt` but with the same node size as `rbt` (to control for node-size in our results). Both `xrbt` and `xrbt-big` cut bit flips by 1.92x (nearly in half) in the case of sequential inserts and by 1.47x in the case of random inserts, a dramatic improvement for a simple implementation change. We can also compare the bytes written, noting that due to the cache absorbing writes, `xrbt-big` and `rbt` write the same number of bytes to memory in all cases, even though `rbt` writes more pointers during its operation.

Because this new optimization target adds additional overhead, we wanted to get an idea of the performance impact of our changes. Figure 3 shows the latency per insert operation for all three variants for both sequential and random insert. Somewhat surprisingly (at first), the `xrbt` is *faster* than `rbt`! But, when looking at `xrbt-big`, this makes some sense. There are two conflicting effects in play: the performance *cost* of doing the extra XOR operations, and the performance *gain* from reducing the size of the node. The interval labeled "a" in Figure 3 is the former, while the interval labeled "b" is the latter. The two nearly cancel out, and we see a similar result for lookup latency.

These results indicate that bit flips can and should be reasoned about directly. Not only is it possible to do so, but the methods presented here are straightforward once this goal is in mind, and they come at little cost to performance and low program-

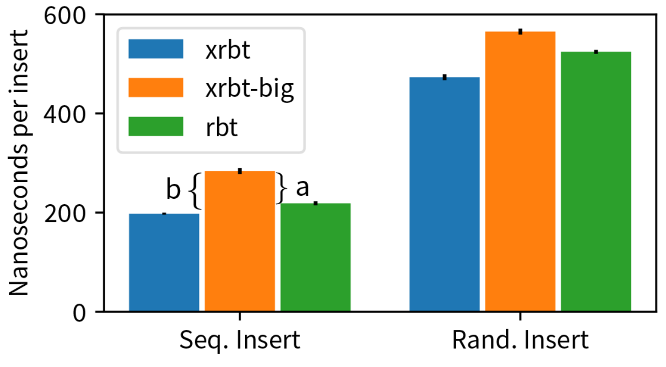


Figure 3: Insert latency for XOR red-black trees compared to normal red-black trees (lower is better). The label "a" shows the cost of the XORs (small), while "b" shows the cost of the larger node.

ming overhead. Furthermore, while reducing writes *can* reduce bit flips, we have confirmed that this is not *always* true—`xrbt` reduced writes over `xrbt-big` at the cost of increasing bit flips.

We can use the results of prior research reporting on power consumption and wear-out of PCM to estimate the effects of our XOR red-black tree. Since PCM power consumption is largely dependent on bit flip rate, we estimate that the power consumption per second of `rbt` and `xrbt` running at full speed are 13mW and 6.6mW, respectively—a ratio of nearly two.

Lifetime is more complex, but a quick calculation taking into account row-shifting and the differences in bytes written by the two variants shows a savings of 1.83x, assuming that the memory controller spreads out writes in larger regions [11]. These savings are estimates, and we may see more savings since potential nonlinearity in power consumption due to heat could improve the power savings from bit flip reduction, and the overall operational power use of controllers may reduce slightly along with the number of writes.

Discussion and Future Research

The data structures presented here emerge from both old and new ideas. While not algorithmically different from existing implementations (both `xrbt` and `rbt` use the same, standard red-black tree algorithms), they present a new approach to implementation with optimizations for bit flipping. This has not been sufficiently studied before in the context of software optimization; after all, there is no theoretical advance nor is there an overwhelming practical advantage to these data structures outside of the bit flip reduction, an optimization goal that is new with BNVM. They do little to impact performance, but performance increases are not the direct goal of this work. Instead, these modest changes can gain us a significant reduction in bit flips that corresponds directly to power and wear reductions, a worthwhile effort even if the saving is small (which, in our work, it is not).

The implications are far-reaching when considering the promise of BNVM and the potential for disruption throughout the system stack. This work is merely the beginning, and we hope that there are future bit flip reduction techniques discovered that we have not considered here. By providing a framework that counts bit flips on data structures, we hope to open an avenue into developing more sophisticated profiling tools that help navigate the tradeoffs between performance, consistency, power consumption, and wear-out.

Considering these results in the context of larger systems is important to understanding the overall effect of bit flip reduction. For example, it would be useful to compare existing key-value stores and observe their memory behavior. However, applying the data structures discussed here as a drop-in replacement for data structures in an existing system would sell them short. Since current systems are designed for non-BNVM technologies, they would fail to make basic optimizations and structural changes that one would expect in a BNVM-optimized system *even without* taking bit flips into consideration. A more effective evaluation would be to construct a BNVM-optimized system from scratch, taking into account write reduction, consistency, *and* bit flips, and then compare it to an existing, unmodified system.

There are a number of implementation details in real hardware that might affect bit flip optimizations. While the basic optimization of avoiding unnecessary overwrites would remain, there are several questions that we do not know the answers to when it comes to bit flip reduction on real hardware. First, what is the *actual* power cost? We will need to wait for real hardware to become available to test this. Second, is there a difference between flipping from a 0 to a 1 compared to flipping from a 1 to a 0? If there is, a new contract between hardware and software would need to contain information that ensures software can predict which is cheaper. Third, is there a performance difference between a write that flips few bits compared to many bits? This depends on hardware implementation details, but if there is, it might make the benefits from bit flip reduction even more significant.

Data structures are not the only causes of memory writes, of course. The obvious candidate for targeted bit flip reduction is the data itself, for which we could rely on existing hardware reduction techniques to work in tandem with software techniques. Another significant source of writes is from the program stack, especially when considering the desire for efficient restart that BNVM offers. We evaluated potential backward-compatible ABI modifications [1], but plenty more work can be done to study these modifications in a real compiler or take them further.

Finally, there are many existing data organization techniques that can be evaluated and tweaked for bit flips. Not only data structures, but algorithms too can be evaluated. For example, if one were to sort a collection of items in BNVM, what would be the most efficient sorting algorithm in terms of bit flips? While it is likely one that minimizes the number of moves, this might not always be the case; we saw above that write reduction does not always correlate with bit flip reduction.

Conclusion

The pressures from new storage hardware trends compel us to explore new optimization goals as BNVM becomes more common as a persistent store; the read/write asymmetry of BNVM must be addressed by reducing bit flips. Reasoning about bit flips should be done at the application level instead of just in hardware to take into account the semantic knowledge of data structure operations, and we cannot get away with simply reducing writes if we strive to reduce power consumption and wear. While hardware techniques apply more broadly, software techniques open the door for significant future research at a variety of levels of the stack. Our work translates directly to power saving and lifetime improvements, both important optimizations for early adoption of new storage trends that will have lasting impact on systems, applications, and hardware.

Acknowledgments

This research was supported in part by the National Science Foundation grant number IIP-1266400 and by the industrial partners of the Center for Research in Storage Systems. The authors additionally thank the members of the Storage Systems Research Center for their support and feedback.

References

[1] D. Bittman, P. Alvaro, D. D. E. Long, and E. L. Miller, "Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, February 2019: https://www.usenix.org/system/files/fast19-bittman.pdf.

[2] D. Bittman, M. Gray, J. Raizes, S. Mukhopadhyay, M. Bryson, P. Alvaro, D. D. E. Long, and E. L. Miller, "Designing Data Structures to Minimize Bit Flips on NVM," in *Proceedings of the 7th IEEE Non-Volatile Memory Systems and Applications Symposium* (NVMSA 2018), August 2018.

[3] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking Database Algorithms for Phase Change Memory," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, January 2011, pp. 21–31.

[4] S. Cho and H. Lee, "Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 347–357.

[5] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *Proceedings of the 46th IEEE Design Automation Conference (DAC '09)*, 2009, pp. 664–669.

[6] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, July 2012.

[7] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin, "Coset Coding to Extend the Lifetime of Memory," in *Proceedings of High Performance Computer Architecture (HPCA '13)*, 2013, pp. 222–233.

[8] S. M. Seyedzadeh, R. Maddah, D. Kline, A. K. Jones, and R. Melhem, "Improving Bit Flip Reduction for Biased and Random Data," *IEEE Transactions on Computers*, vol. 65, no. 11, 2016, pp. 3345–3356.

[9] P. Sinha, "A Memory-Efficient Doubly Linked List," *Linux Journal*, vol. 129, 2004: http://www.linuxjournal.com/article/6828.

[10] B. D. Yang, J. E. Lee, J. S. Kim, J. Cho, S. Y. Lee, and B. G. Yu, "A Low Power Phase-Change Random Access Memory Using a Data-Comparison Write Scheme," in *Proceedings of IEEE International Symposium on Circuits and Systems*, May 2007.

[11] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009, pp. 14–23.

Structured Logging
Crafting Useful Message Content

VLADIMIR LEGEZA AND ANTON GOLUBTSOV WITH BETSY BEYER

Vladimir Legeza is a Technical Solutions Engineer at Google Cloud Japan. For the last few decades, he has worked for various companies in a variety of sizes and business spheres such as business consulting, Web portals development, online gaming, and TV broadcasting. Since 2010, Vladimir has primarily focused on large-scale, high-performance solutions. Before Google, he worked as an SRE on search services and platform infrastructure at Yandex and then in a similar position at Amazon Japan. lgz@google.com

Anton Golubtsov is a Software Development Engineer at Amazon Japan. Before Amazon, he worked at Yandex in a few roles: an SDE, a team leader, and a Technical Project Manager. zoomacode@zoomacode.ru

Betsy Beyer is a Technical Writer for Google Site Reliability Engineering in NYC and is the editor of *Site Reliability Engineering: How Google Runs Production Systems* and *Site Reliability Workbook*. She has previously written documentation for Google datacenters and hardware operations teams. She holds degrees from Stanford and Tulane. bbeyer@google.com

In the context of logging, the word "structured" typically refers to the way log records are represented in a machine-readable format, such as JSON or XML. In this article, we focus on another aspect of logging structure: the message content.

Computing today offers several automated ways of collecting, delivering, and processing log records from different types of systems. But modern technologies are not supportive if the information describing a specific event is insufficient or otherwise not helpful.

To approach this topic, it's useful to understand the most common logging issues, why they occur and possible solutions. By discussing some representative use cases, we aim to provide practical insights and approaches to improving the structure of your logs. As with most advice, our proposed solutions are just one way of approaching a problem space—feel free to either apply our suggestions wholesale or pick and choose the pieces that suit your needs.

Reasons to Invest in Well-Structured Logging

Before diving into specifics: why should you invest time and effort on designing and implementing a sound logging strategy?

Imagine a scenario in which you're trying to investigate event statements to determine why your main service isn't responding. Meanwhile, angry customers are reaching out to you via every possible communication channel, and upper management is shouting at you to resolve the situation quickly. Every corner of the office seems to be consumed with anxiety and pressure, but your only reasonable response is, "I couldn't find any useful information in our logs…I'll need to reproduce this entire event on a staging environment."

For many companies, every minute of downtime results in a certain amount of harm: outages entail both financial costs and damage to your reputation. When customers and investors experience a serious scare, the entire business may be at risk.

Well-structured logging can make a world of difference in the above scenario. After a few years in the industry, our experience has shown that investing time and effort in improving the logging process is worthwhile. When a crisis occurs, the alternative is too costly and painful.

Anatomy of a Log Entry

Logs can be split into three broad categories:

- **Operational logs:** contain information about service usage, such as user requests and transactions.
- **Telemetry logs:** contain application-internal metrics, expressed in the form of log records.
- **Behavioral logs:** show what is happening inside the application.

Operational and telemetry logs are typically generated from a pre-formatted template, while behavioral logs incorporate manually crafted components that are unique to each record. We'll focus on behavioral logs, the most widely used and complex of the three, but you can apply the solutions we discuss to operational and telemetry logs as well.

Message Type	Severity Level
Milestone	INFO
Alert	WARNING or ERROR
Data samples, additional details	DEBUG

Table 1: Message types mapped to severity levels

Types of Records/Messages

You might implement logging for a variety of purposes, most of which enable effective root cause analysis (RCA):

- **Tracking milestones:** To show an application's current state or a specified milestone. This type of information is useful in a couple scenarios:
  - When you want to understand what the application is doing right now—whether or not it reaches a particular milestone. **Example milestones:** *Operation A is completed; starting operation B. No more data to process.*
  - When you want to see a sequence of state changes in order to understand the application's end-to-end behavior. **Example implementation**: *The* `ssh` *binary expresses certain information into STDERR if executed with the* `-v` *flag.*
- **Alerting**: To emit alert notifications when something goes amiss.
- **Debugging/sampling**: To capture the data samples and statements with which a program is operating. These types of messages appear in the code in early development stages, when you simply want to see if an application is behaving as expected.

These message types map to a set of standard logging severity levels, as shown in Table 1.

Granular Decomposition

It's helpful to think of all log messages as a collection of answers about *why* an issue is happening. To answer *why*, we also need more information about *when* an event happened, *where* it happened, and *what* exactly happened. (Of course, not every type of log will answer all four "W" questions.)

Regardless of the type and category, every log record consists of two parts:

- **Metadata**: Information about the event statement
- **Content**: The statement itself

Metadata is generated automatically, whereas content is manually crafted. Each part answers its own set of "W" questions, as represented in Figure 1.

Complications

Problems with logging fall into two main buckets: inconsistency and missing data.

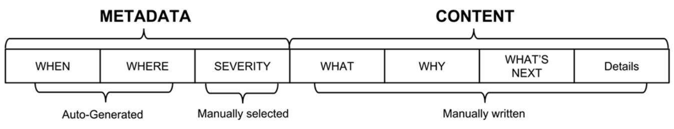


Figure 1: Anatomy of a log entry (simple)

Inconsistency

Modern systems that use microservice architectures typically aggregate logs from multiple applications. The inconsistency problem arises at the edge between apps when you attempt to correlate events across several services. Because these problems largely pertain to the auto-generated portion of logs, you can address them systematically by establishing a set of rules about metadata and content.

For example, while one engineer might assign a severity level of INFO to a given error notification, another might classify that notification as an ERROR. The same uncertainty might apply to WARNING versus ERROR classifications, ERROR versus CRITICAL classifications, and so on. You can overcome this inconsistency problem by establishing clear guidelines around appropriate severity levels.

It can also be difficult to track *where* events occur across microservices, as identity elements like process and thread IDs, host, service, executable names, and source code pointers tend to be tightly coupled to specific service instances. To overcome this inconsistency, consider introducing global variables, such as a unique Request ID that's randomly generated in a front line server and passed along all data paths.

Missing Data

Sometimes the content portion of a log record is missing a chunk of valuable information, which means the log entry is less meaningful or even meaningless. Overcoming this issue is the difference between implementing log messages and making sure that their recipients can read meaningful information.

In order to craft meaningful log messages, put yourself in the shoes of potential readers. A log entry's audience likely needs much more context about the application than the engineer who crafts the message. In concrete terms, consider two of the previously mentioned "W" questions: *what* and *why* (as *where* and *when* were automatically generated). After writing a baseline message, recursively iterate over each "W" question until the message has only one possible meaning.

Tips and Tricks

Now that you're familiar with the anatomy of a log entry and the broad categories of complications, we'll address some of the most common problems with the structure of logging contents.

Metadata Issues

Time Zone

Some timestamp formats may not include critical details, or may include non-essential information. For example, it's very important for log messages to include time zone information—particularly if your organization spans or will someday span more than one location—whereas including the weekday is simply a waste of space. Daylight saving time is another hidden issue: even if you use a single time zone for all systems and services, this quirk can translate into either an empty hour of data or two independent sets of records captured for the same hour.

To address these issues, the timestamp identifier should always include the time zone. We recommend basing your time zone on UTC. Unlike GMT, UTC is not impacted by daylight saving time. You likely also need to implement a time converter to align logs across time zones.

Severity Level

As previously mentioned, there's a lot of room for discordance in imprecise name-based severity-level definitions. Overcome this friction by establishing clear organization-wide guidelines.

INFO vs. DEBUG

You can define the boundary between INFO and DEBUG buckets by restricting the INFO level to represent milestone information. You also need to account for two known gray areas:

◆ Logs that describe decisions that an application made. For example:

```
Descale cluster 'abc' from 7 to 5 nodes. Reason: average node
load < 30% for the last 10min.
```

Note: Here, and for most of the examples that follow, we've omitted the metadata from log entries since this metadata takes up space and isn't particularly relevant.

◆ Operational logs that require an attached severity level. For example, if user request logs pass through the same logging mechanism as other entries.

You might classify both of these cases as INFO messages: while they're not clear milestones, they roughly represent a logical point in the code, reached during execution.

If a given log record contains only statement information, but you also need to provide additional knowledge about that information, we recommend distributing the information across two log messages: the statement itself as an INFO message, and the additional information as a DEBUG message. This approach maintains clear information on every level, and the person who reads the logs can easily decide which level of detail they want to see. DEBUG messages are ideal for hosting information that doesn't fit into other level criteria or information that may be valuable in the future.

WARNING vs. ERROR Messages

We recommend differentiating between WARNING versus ERROR messages according to application behavior:

◆ **WARNING**: If the app can automatically recover from this state

◆ **ERROR:** If the app can't automatically recover from this state

You can implement these classifications on a more granular level—e.g., on an individual thread, branch, or transaction level.

For example, the following issue occurred in the middle of the request processing:

```
Request "/api/v1/get?obj_id=12345678"
Attempt to retrieve from cache.
Unable to resolve "cache-farm.example.com": Not found. Abort.
Cache miss.
Attempt to retrieve from origin.
Object obtained.
Response sent.
200 OK "/api/v1/get?obj_id=12345678"
```

If you're treating signals on a thread basis, you should treat the highlighted alert as a WARNING because processing isn't blocked. If you're treating signals from a branch perspective, you should mark the alert as an ERROR because the operational branch that retrieves objects from the cache was aborted and reached its logical end. By handling alerts based upon the branch depth, some ERRORs don't result as an error in the overall request processing. However, you can identify smaller anomalies faster.

In the following example, which attempts to reach an unavailable API, a set of WARNING messages precedes the final ERROR message. The service attempts to connect five times before giving up. The overall procedure is not yet aborted during retries, so the first four requests are marked with WARNING messages; only the last attempt is stated as an ERROR. If your logs only accounted for ERROR messages, you'd only see the final message, which doesn't tell the entire story. It's key here that the ERROR message explicitly references the five previous attempts and four WARNING messages—otherwise, the WARNING messages may be buried among hundreds of other unrelated messages, and the reader might not even realize that there were previous attempts to connect to the API.

```
 ...
WARNING "Unable to connect to Awesome API. Connection timed
out. Attempt 3/5"
WARNING "Unable to connect to Awesome API. Connection timed
out. Attempt 4/5"
ERROR "Unable to connect to Awesome API after 5 attempts.
Connection timed out. Exiting."
```

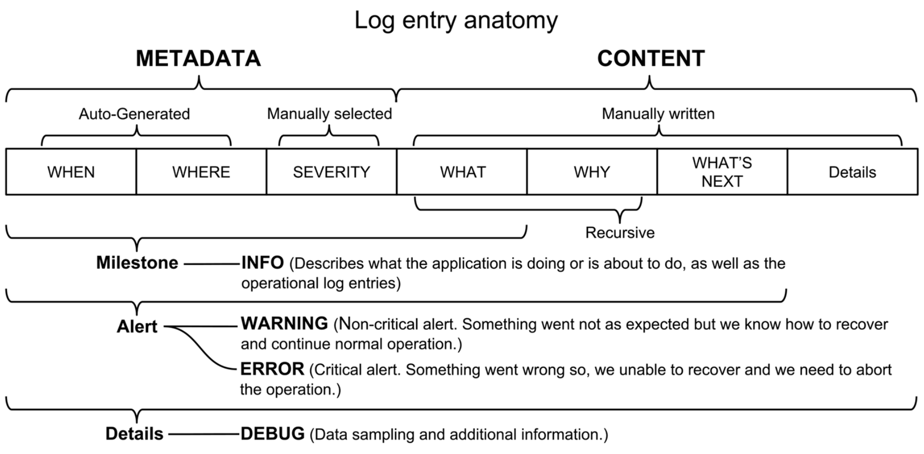

Log entry anatomy
METADATA
CONTENT
Auto-Generated
Manually selected
Manually written
WHEN
WHERE
SEVERITY
WHAT
WHY
WHAT'S NEXT
Details
Recursive
Milestone
INFO (Describes what the application is doing or is about to do, as well as the operational log entries)
Alert
WARNING (Non-critical alert. Something went not as expected but we know how to recover and continue normal operation.)
ERROR (Critical alert. Something went wrong so, we unable to recover and we need to abort the operation.)
Details
DEBUG (Data sampling and additional information.)

Figure 2: Anatomy of a log entry (complex)

ERROR vs. CRITICAL Severity

While a CRITICAL severity doesn't necessarily indicate that something bad happened, an ERROR severity unequivocally does.

We've personally found that CRITICAL severity isn't useful in most scenarios and have chosen to do away with that severity entirely. However, you may find useful scenarios for using the CRITICAL severity level. Be sure to determine a precise definition of what information is deemed CRITICAL and how to clearly distinguish that information.

Figure 2 consolidates a proposed schema for severity levels and their meanings.

Content Issues

INFO

INFO is a simple statement addressing *what* happened or is about to happen. Your aim in crafting this message should be to provide clarity.

Consider the following message, which, although accurate, isn't sufficiently descriptive:

```
Server has started.
```

This message fails to indicate why this information is important. Your *what* questions should only have one answer. In this case, is there more than one server that could have started? If so, which server started?

The following message is a marked improvement:

```
HTTP API Server has started.
```

You can improve further upon an INFO message by asking, *What is the most valuable information about the subject of this event?* In this case, *What is the most valuable information about the HTTP API server?* For any network communication HTTP server, the answer to this question is the entry point:

```
API Server start listening on http://0.0.0.0:80.
```

This event statement is three times more useful than the original.

ERROR and WARNING Messages

ERROR and WARNING messages are categorized as alerts, which describe the difference between the expected and actual behavior: *We expected A, but got B.* To craft meaningful ERROR and WARNING messages, ask yourself:

- What happened?
- Why is there a difference between the expected and actual conditions?

In the following example, we attempted to call an HTTP API and received an error. We'll iterate a couple of times on *What* and *Why* in order to demystify details.

What happened?

```
Unsuccessful API call.
```

What was the reason for this call?

```
Unable to retrieve the data object via API.
```

What data object?

```
Unable to retrieve a file's metadata via API.
```

What file?

```
Unable to retrieve metadata for the "abc123" file via API.
```

What API?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123".
```

Why?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123": Failed to parse
server response.
```

What further information do we know about the server response?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123": Failed to parse JSON
response.
```

Why?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123": Failed to parse JSON
response. No JSON object could be decoded.
```

What happens next?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123": Failed to parse JSON
response. No JSON object could be decoded. Aborting.
```

Working from the original message text, gathering all this information from the live system would take minutes or even hours. By iterating through this series of questions, we gather information that the original message didn't provide:

- **Exact file name:** We can make a request for this object in a separate system to clarify the current object condition, including the state of its metadata.
- **Exact URL of a metadata request:** Now we can quickly request the metadata again for further inspection without having to craft this request from scratch. We can also immediately verify that the data was requested from the correct place and with the correct environment (API version, additional parameters, modifiers, etc.).
- **The data source service:** We know who to page in case we encounter a massive issue.
- **Metadata encoding format:** JSON.
- **Final action:** No additional retries were made, and the requester did not get the data they needed.

It's worth emphasizing the *What's next?* question here, which comes in handy when you encounter a vague statement. In this example, the reader was left with questions like, *Will the system retry requesting data from another source? Was this the only attempt?* and *Did the original request ever succeed?* The "Aborting" statement, which reports the application's next expected action, removes all these uncertainties.

Accounting for sensitive information in DEBUG messages

Take care in choosing how to represent the original alert as a DEBUG message. Whenever you log a working data sample, even partially, make sure that the sample doesn't contain sensitive information. Publishing such information accidently can pose a threat and security risk to people and systems!

Alternatively, when an error requires a long explanation (for example, with potential causes and suggestions for various methods of mitigation), instead of packing the entire text into the message itself, consider assigning a unique ID and providing a reference to a full explanation. The disadvantage of this method is that the log entry may not explain the situation. On the other hand, the description can be highly enriched with background information, solution playbooks, examples, and so on.

DEBUG

The debugging level serves two main purposes: increasing output verbosity and providing data samples.

When it comes to increasing output verbosity, DEBUG messages can better detail lower-level milestones and illustrate ongoing values, which are useful to real-time application tracing. For example, consider a well-known open-source "OpenSSH" utility. Both the server and the client support extended verbosity that displays the files being read, network connection establishment details, cryptography negotiation, and much more, thereby helping the reader understand how the utility works.

Data sampling is relatively straightforward. Consider it in the context of the previous file metadata retrieval example. The next logical step for a reader faced with the message "No JSON object could be decoded" is to examine the content of this object. We can make the reader's life easier by placing this data as a DEBUG message that follows the original alert.

You need to decide whether or not to include the original alert information in the DEBUG message—does it suit your purpose better to optimize for saving storage or to optimize for individual message clarity? Because you can link these messages using metadata, there's no strong need to include the original message. However, longer records can save valuable time during incident investigations, and they aren't filtered out from non-metadata searches.

Our final DEBUG entry perfectly aligns with the log entry anatomy scheme:

Complete entry:

```
2019.01.15 00:55:12.345012 UTC 43526 62837563 file_manager
[.../example_api/__init__.py:1024] DEBUG Unable to retrieve
metadata for the "abc123" file from "https://api.example.com/
v1/get_meta?obj=abc123": Failed to parse JSON response. No
JSON object could be decoded. Aborting. Response: 'Internal
Server Error.'
```

Breakdown:

`2019.01.15 00:55:12.345012 UTC`	WHEN	
`43526`	Process ID	WHERE
`62837563`	Thread ID	
`file_manager`	Binary Name	
`[.../example_api/__init__.py:1024]`	Code Pointer	
`DEBUG`	SEVERITY	
`Unable to retrieve metadata for the "abc123" file from "https://api.example.com/v1/get_meta?obj=abc123"`	WHAT	
`Failed to parse JSON response. No JSON object could be decoded.`	WHY	
`Aborting.`	WHAT'S NEXT	
`Response: 'Internal Server Error.'`	Details	

Message Formatting and Processing

Formatting

You can use formatting to coherently represent metadata. Aim to keep your formatting brief but sufficiently explicit.

Here's an example of formatting that's applied to Kubernetes log metadata (you can see the original format description at https://github.com/kubernetes/klog/blob/master/klog.go):

```
I0115 02:31:05.029108 1083 server.go:796] GET /stats/summary/:
(10.507359ms) 200 [[Go-http-client/1.1] 10.44.1.11:60556]
```

Note the following:

- The severity level is collapsed to a single capitalized character, *I*, which stands for INFO. You can represent the other severity levels with the letters *W (WARNING)*, *E (ERROR)*, and *D (DEBUG)*. Because the severity is the first character in the line, it can be easily expressed in a regular expression.
- The four digits concatenated with the severity level represent a date: 0115 refers to January 15, and the year is omitted, likely because records aren't stored for more than 12 months or because this information is added during entry processing.
- A single closing square bracket (*]*) separates the metadata from the content.

You could improve this formatting by accounting for time zones. For example, Google Cloud Platform collects the full timestamp, along with additional metadata.

By consolidating the date format, this record is readable, contains *almost* all the data we need, and saves about 10 bytes of space per record.

Output

Messages to the standard file descriptor should be delivered to STDERR, as opposed to STDOUT. Because users and various tools expect messages to appear on the STDERR, messages directed to STDOUT will be ignored or potentially cause harm by injecting data into a data flow pipeline.

You can save time and ensure log format consistency by creating a small set of libraries for various languages, which can coordinate all logging configuration out of the box.

Multi-line Messages

Multi-line entries can be problematic when log processing software treats messages as one entry per line. You can mitigate this problem by performing an additional layer of pre-processing on the application level: you can adjust every outgoing string by replacing the newline character with another unique string (`\n`, for example), so that the message can be restored by reverse conversion.

This solution's only drawback is message length. For example, a Java stack trace may run up against the maximum permitted message size in the processing or delivery stage. If you run into this problem, you can consider splitting one message into a sequence of several messages.

As a practical example, consider the shell script below. Shell scripts notoriously suffer from poor logging. We can improve the script by replacing the simple `echo` function with the more meaningful `log_info`. By logging INFO messages, we address the previously discussed time zone and output issues, thereby accommodating compact formatting and multi-line entries.

Here's an example implementation, distributed under the Apache 2.0 license:

```
$ cat standardized_bash_log.sh
#!/bin/bash
date_fmt='%m%d %H:%M:%S'
tz='UTC'

log_preproc(){
   echo "$@" | awk -v ORS='' '{if (NR!=1) $0 = "\\n" $0};{print}'
}
```

```
log(){
    metadata="${1}$(TZ=$tz date "+$date_fmt") $tz $$ \
$(basename $0)"
    content=$(log_preproc "$2")
    echo "${metadata}] $content" >&2
}

log_info(){
    log 'I' "$@"
}
```

Usage:

```
$ cat logging_example.sh
#!/bin/bash
source standardized_bash_log.sh
log_info 'The first line of text;
    The second;
    And finally, the third one.'
```

Execution with multi-line restoration:

```
$ echo -e $(./logging_example.sh 2>&1 | grep "second")
I0116 07:42:59 UTC 40844 logging_example.sh] The first line of
text;
 The second;
 And finally, the third one.
```

Storage

Root cause analysis (RCA) benefits from robust logging data. However, crafting and storing a comprehensive set of logging records requires a prohibitively large amount of storage. Does storing all logging data from all apps and environments in full really make sense?

When performing any kind of RCA, each investigation is initiated by an error. An investigator needs the information surrounding this problematic event. In reality, you need to store operational logs (user requests and decision-making information) in full, as these events are unique and unrelated to each other. The behavioral logs can be partially truncated because event sequences are repetitive.

To reduce storage volume, you can place a filter between an application and the delivery mechanism. For example, a filter can accumulate all messages for the last two to five minutes of operation in a buffer; when an error occurs, the filter dumps the entire buffer to remote storage. An engineer can then find all error-related records as well as all potentially correlated events that were in flight during the incident.

This approach has a couple of positive side effects. Because the price of log storage depends directly on the number of errors the service experiences, the fewer errors your service undergoes, the less storage you need. The filter can also prevent the system from flooding the logging system with identical errors.

If you're concerned about potentially problematic situations that don't produce errors and hence can't be easily detected, you can keep the entire logging set locally on the host with a shorter retention period.

If you want to minimize storage use when conducting other research and development, you can narrow the observation scope to a single application instance for a certain period of time. That way, you can easily reclaim the space occupied by locally stored data once the experiment is complete.

Conclusion

Many of the problems with logging in modern computing can be addressed by bridging the gap between the people writing and reading the logs. You can narrow this distance by clarifying and restricting the meanings of various terms and by using a question-based approach to ensure that you express all of the necessary data. We hope you find the recommendations in this article useful and that you adjust our approaches according to your preferences and experience, improve them, and share your further ideas and best practices with your team and beyond.

Complex
The Most Overloaded Word in Technology

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly *Site Reliability Engineering* book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura Nolan is a production engineer at Slack. laura.nolan@gmail.com

"Site reliability engineers shall predict the behavior of complex systems."

This sentence, which I recently came across in a job description, is fascinating because it is one that, depending on your background, could seem either reasonable enough or an utter glaring contradiction in terms.

Most people use the word *complex* as a synonym of *complicated* or *intricate*—something with a lot of parts that's hard to fully grasp. Understanding something complicated may be hard, but make the effort and you can, at least potentially, do it.

However, both software engineers and systems engineers use the word *complex* as a specific term of art. Software engineers in fact use it in several different ways, distinct from the systems meaning. Software engineers and systems engineers (please read that term throughout this article to mean SREs, production engineers, systems administrators, DevOps practitioners, etc.) are overlapping groups of people who work together. We all need to understand which meaning is in use at any given time so we can communicate clearly.

First, software engineers talk about time and space complexity: in other words, Big-O. In this context, complexity refers to how the time or space requirements to execute an algorithm scale with the properties of the input. There are also code complexity metrics like McCabe's Cyclomatic Complexity—that metric counts the number of independent code paths in a piece of software. But neither of these are what most of us mean when we discuss complexity or its inverse, simplicity.

Software Complexity

Complexity has been the enemy of the software engineer for decades now. Fred Brooks' classic essay "No Silver Bullet" [1] divided software's complexity into two parts: essential complexity and accidental complexity. Essential complexity is that related solely to specifying the problem and how it should be solved. Accidental complexity is related to the details of implementation. Writing your business logic and unit testing it is (hopefully) mostly essential complexity, but HTTP and managing concurrency and garbage collection and deployment to production are largely accidental complexity. The overwhelming majority of the work of technology operation is about accidental complexity.

But this doesn't tell us what software engineers mean by complexity. Fundamentally, complexity is that which makes software difficult to fully understand and to correctly reason about. Moseley and Marks' paper "Out of the Tarpit" [2] discusses several sources of complexity. The biggest, and hardest to deal with, is state—state influences the flow of control of a program, but the number of potential states a piece of software can be in increases exponentially with the number of variables. Dealing with this is such a difficult problem that we basically handwave past it: we normally run all tests on modules in known states, and we routinely restart misbehaving programs in order to restore them to a known good internal state.

Other major sources of complexity are sheer code volume and the fact that programs, unlike complex physical structures, cannot be visually inspected. Mental models of the program must be constructed from the source code. This can of course be easier or harder depending on how the code is structured. John Osterhout's book *A Philosophy of Software Design* [3] is all about making the design of software systems less complex, and he advocates very strongly for relatively few *deep* modules, each of which implements powerful functionality behind a simple interface. This is much like the UNIX philosophy—write small programs that do one thing well and can be used together.

Systems Complexity

Systems engineers tend to have a completely different idea of complexity, stemming from systems theory. Systems theory is a distinct area of research, spanning all kinds of manmade or natural systems—everything from an anthill to a nuclear power plant—and complex systems theory is a subset of it. Complex systems have particular characteristics: multiple interacting parts, system state (i.e., a memory of some kind), and feedback loops. They display emergent phenomena, have nonlinear relationships (small changes in one part can lead to large deviations in overall system behavior) and tend to be prone to cascading failures or "vicious cycles." Complex system behavior cannot be predicted reliably.

An amusing example of a complex systems failure is the incident that led to two interacting book pricing bots driving the price of a book on the genetics of flies to over 23 million dollars [4]. One bot was designed to set its price to undercut its competition by 2%, and another bot was coded to price books it didn't have in stock at 27% above the price it found in the market (in order to make a profit reselling them). In the case of one rare book, each bot set its price based on the other bot's price on a daily basis, leading to a vicious cycle of compounding prices. This system has multiple interacting parts, state and feedback loops—it is a complex system, albeit a trivial example of one.

All computing systems are complex systems. Even if a system is running on a single physical machine you are still dealing with the interactions of multiple pieces of software, all of which are likely complex systems in their own right, running on complex hardware. Each running program may have multiple threads of control, state, interactions with the operating system and other programs—even if not explicitly then via shared resources.

The "Stella Report" [5] describes several real-world examples of the kinds of deviations and failures that are commonly experienced in complex computing systems. In one example from the report, the combination of centralized logging with the ELK stack plus installation of a keylogger for audit purposes resulted in system failure when the remote Logstash program experienced intermittent failure. The issue was compounded by the terminal becoming unresponsive (waiting for the logging system), hindering debugging. That outcome is hard to predict ahead of time by reasoning about system behavior. This is why chaos testing has become popular. It's easier, and far more reliable, to add latency to a component in a controlled fashion and see what is affected than to attempt to model all the possible interactions between system components.

This systems theory definition of complexity is the one often used by systems administrators, SREs, and DevOps practitioners—this is in no small part due to the impact of Richard Cook's paper "How Complex Systems Fail" [6] on the industry some years ago. Software engineers, on the other hand, mainly think in terms of code structure, interactions between modules, and interdependencies in their code bases. Software engineers' primary concern is the difficulty of making correct changes without introducing errors. Systems engineers' primary concern is stability of the deployed software in production.

This is why, when you ask a software engineer to promote simplicity as part of their job description, they look for opportunities to separate concerns and reduce coupling in their code base to refactor to well-known design patterns, create better-defined interactions between modules, and remove unused code.

When you ask systems engineers to do the same thing, they often look for ways to control extremes of the system's behavior (using load shedding and circuit breakers, for instance), or to make elements of the system more uniform. Dave Mangot's recent *;login:* article "Achieving Reliability with Boring Technology" [7] discusses the use of infrastructure-as-code techniques to make sure your production environments are standard and well-understood. That's a very good example of the kinds of ways that systems engineers can reduce complexity.

The two kinds of complexity that we discuss here are quite different, but they do also have one major thing in common: both software complexity and systems complexity make the task of understanding and predicting behavior impossible.

All of us—software engineers, systems administrators, site reliability engineers, production engineers, DevOps practitioners—we are all fighting the same two-faced demon named complexity. In both software and operations, complexity arises from state, from the sheer number of components or modules, from the number of interactions (both intended and unintended), as well as from the impossibility of direct inspection of the systems we work on.

Code and the running production system are two aspects of the same thing, and it's very unlikely we can run a stable, reliable, performant, maintainable system if either variety of complexity (code or systems) is not continually managed. Let's understand each other's language, and let's always have empathy for the challenges that our colleagues face.

References

[1] F. Brooks, "No Silver Bullet—Essence and Accident in Software Engineering," *Proceedings of the IFIP 10th World Computing Conference*, 1986.

[2] B. Moseley, P. Marks, "Out of the Tar Pit," BCS Software Practice Advancement (SPA 2006).

[3] J. Osterhout, *A Philosophy of Software Design* (Yaknyam Press, 2018).

[4] M. Masnick, "The Infinite Loop of Algorithmic Pricing on Amazon...Or How a Book on Flies Cost $23,698,655.93," Techdirt: http://bit.ly/2FagxMz (accessed March 18, 2019).

[5] D. Woods, "STELLA Report," SNAFUcatchers Workshop on Coping with Complexity, 2017.

[6] R. I. Cook, MD, "How Complex Systems Fail," Cognitive Technologies Lab, University of Chicago, 2002.

[7] D. Mangot, "Achieving Reliability with Boring Technology," *;login:*, vol. 44, no. 1 (Spring 2019): https://www.usenix.org/publications/login/spring2019/mangot.

Other Faces of Python

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. Even though he is a native New Yorker, he is currently living and working from home in the northeast of Brazil. pcnorton@rbox.co.

I'd like to talk about uses for serialized data this time, looking at them through contrasting language-neutral formats: YAML and protocol buffers. These will be the basis for discussing an interesting Python interpreter, specially built to make working with protocol buffers easier.

Wikipedia (https://en.wikipedia.org/wiki/Serialization) has a really great, straightforward definition of serialization: "the process of translating data structures or object state into a format that can be stored." YAML is a really easy format for serialization/deserialization for simple Python data types since it represents data structures in a way that's really similar to how Python does; in my experience, however, this is not so much the case for defined types.

YAML

So let's talk about YAML. YAML (standing for YAML Ain't Markup Language, or possibly Yet Another Markup Language, or maybe something else) is recognizable in the wild as the prolific format where the whitespace is relevant and indentation is incredibly important, and which breaks if someone naively makes a single whitespace change (like many people's first impression of Python!). Its goal is to be able to serialize and deserialize data in a format that is human-readable (text) and comprehensible (line breaks matter in a way that is similar to written language, indentation guides the structure, etc.).

YAML also has all sorts of interesting features, like the ability to name a structure and reuse it multiple times, and graft that onto various other locations, similar to using variables. (Some interesting discussion about the full range of what it can do is available at http://yaml.org.) YAML is often used as more than just a serialization format since it has the ability to, for example, declare blocks, repeat them, etc. A recent post at https://blog.atomist.com/in-defense-of-yaml/ reminded me of some of the work I've been doing. In short, YAML is hugely useful, but it also has limits that should be respected.

One trivial example of its usefulness is:

```
this:
  is: a mapping with
  different: value types
  here: 3
```

which would look like this in Python:

```
{"this": {"is": "a mapping with", "different": "value types", "here": 3}}
```

Declaring a reusable block (called an anchor) is this simple, and you can see how it's expanded by running this in the Python REPL using the pyyaml module (see http://pyyaml.org for more info):

```
>>> import yaml
>>> yaml.load("""
... this: &use_this_anchor
...   is: cool
...
... here: *use_this_anchor
... """)
{'this': {'is': 'cool'}, 'here': {'is': 'cool'}}
```

This can greatly reduce size and repetition. It's clear that human-readable and understandable formats like YAML have been a huge positive change. Because of their widespread use and acceptance, people feel less need to create poorly defined ad-hoc configuration formats. The fact that software is shipped using YAML means that they're being configured via plaintext data structures. That's a big win!

YAML and Configuration

These formats make your configuration much easier to comprehend. You almost don't have to do any work. It also means that your configuration often seems to be self-documenting—we can read about specific data types, quantities, etc., and with only a little familiarity with the system you're working with, it's almost obvious what you (or the program) are trying to express. For example, the following is probably going to make sense if I tell you that it's a section of YAML-formatted configuration for the Envoy proxy, a Layer 7 proxy (sometimes called a *service mesh*; see envoyproxy.io for more info):

```
static_resources:
  clusters:
  - circuit_breakers:
      thresholds:
      - max_pending_requests: 8192
        max_requests: 8192
        max_retries: 1000
        priority: DEFAULT
      - max_pending_requests: 8192
        max_requests: 8192
        max_retries: 1000
        priority: HIGH
    connect_timeout: 0.5s
    hosts:
    - socket_address:
        address: foohost-ssl
        port_value: 443
    lb_policy: ROUND_ROBIN
    name: foohost
    per_connection_buffer_limit_bytes: 3100000
    tls_context: {}
    type: STRICT_DNS
```

It doesn't provide the person reading it with the larger picture, but you can use this as a starting point—it's probably configuration that governs the behavior of a listening port and multiple hosts behind a load-balancer .

One limit to YAML's flexibility, though, is that small nested changes prevent the use of anchors. So there are two threshold entries that look almost exactly alike. But the difference in the `priority` key means that the entire structure must be repeated. As you can imagine, this sort of inconsistency can become irritating as the size of the data gets larger.

Using YAML as the representation of the data comes with another weakness: there is no built-in checking that a message has the right shape or the right structure—essentially it doesn't come with any type checking. Let's focus on this, because better type checking is great, especially when it is easily achievable at a low cost.

Skycfg, Protocol Buffers, and YAML

So how can someone do better than YAML? One answer is to use protocol buffers (usually just called *protobufs*). Protocol buffers are also widely used, and one important role they play is in defining APIs. Two examples that have been increasingly adopted over the past few years are the Envoy proxy (mentioned above) and Kubernetes (https://kubernetes.io). In both cases, protocol buffers are used to define the structures used by the API internally, while their external-facing REST API and configuration will accept messages in other formats (e.g., YAML) but translate them and check them against the API definition. This means that a REST API may be used with YAML data, but when this data gets into the system and is deserialized, it'll get checked against the protocol buffer definitions, which are the real source of truth.

In order to make using protobufs easier, the folks at Stripe have created *Skycfg*, which is based on a special-purpose language whose syntax and behavior are derived from Python. While Python is usually considered a "general-purpose" language, Skycfg has an entirely different reason for existing. It is based on a variant of Python whose primary goal is to be as easy to use as the standard CPython but to be limited in a way that focuses on enhancing the process of configuring large software systems. The language Skycfg is based on was once called "Skylark" but was renamed "Starlark" (https://blog.bazel.build/2018/08/17/starlark.html) and released as part of the Bazel build system (http://bazel.build).

With Skycfg, protocol buffer messages are compiled from a neutral format into a Golang-specific library and imported into Skycfg, and your own variation of Skycfg is built for your own use. When your custom interpreter is run, you can create objects using their protocol buffer message definitions, and they maintain their type information per the underlying Golang runtime. The intent is that the protobuf data structures remain strongly typed and will not have implicit conversions done to them. Messages are defined ahead of time; they are created, updated, compared, etc. using the syntax of Python (Skycfg), and doing things this way maintains a strongly typed, statically checkable configuration.

Some Examples

So let's have some show and tell.

This bit of YAML is pretty easy to comprehend:

```
access_log_path: /var/log/envoy/admin_access_log
address:
  socket_address:
    address: 127.0.0.1
    port_value: 1234
```

This is short and sweet, and as configuration it seems pretty straightforward. As mentioned earlier, the user/operator must make sure to avoid some common mistakes. If I add a tab instead of spaces, it breaks in a way that may not be obvious. If I make the port value >65k, I may not notice it, but it's clearly outside the range of available ports. If I mistype something it's still valid YAML, but it doesn't mean anything to the program that reads it.

By contrast, generating this in Skycfg code has the upfront cost of writing some Python, with a disproportionately large benefit: I can create configuration messages where the type of the message is known and statically checked. So, unlike YAML, this doesn't allow us to graft the wrong message into the wrong place. In addition, the fields of the messages are also type checked, and we can create these messages with proper functions instead of being YAML anchors, in which you can't replace at the granularity of one element of a list or a mapping.

Just in case you are interested in the entire v2 API that Envoy provides, the messages being generated below are documented further at https://www.envoyproxy.io/docs/envoy/latest/api-v2/api.

```
# -*- Python -*-
v2_bootstrap = proto.package("envoy.config.bootstrap.v2")
# Code we write, the "//" is specific to Skycfg/starlark
load("//common_helpers.sky", "to_struct")
load("//common_helpers.sky", "envoy_address")
# this gets code the envoy maintainers wrote,
# built into the main.go
v2_core = proto.package("envoy.api.v2.core")

# Bootstrap message sections
def admin_msg(access_log_path, address, port):
    """This generates the :admin:
    section, including the access log path
    and the listen address of this server.
    """
    admin = v2_bootstrap.Admin(
        access_log_path=access_log_path,
        address=envoy_address(address, port))
    return admin

def node_msg(cluster, node_id):
    """The cluster name should match whatever
    we're using to identify the cluster, the
    node_id should match the IP address or
    hostname.
    """
    return v2_core.Node(
        id=node_id,
        cluster=cluster)

def build_bootstrap_msg(
    admin, node, static_resources, stats_sinks):
    """The core initial config is the
    bootstrap message - this is essentially
    the jumping-off point that we plant in
    `/etc/envoy/envoy.yaml`
    """
    return v2_bootstrap.Bootstrap(
        admin=admin,
        node=node,
        static_resources=static_resources,
        stats_sinks=stats_sinks)
```

To use this, you need to build the interpreter, which is really simple. Look at https://github.com/pcn/followprotocol for the code and try it out.

This is a pretty neat trick, and the benefits become clearer once you consider the power of the combination of Python's syntax to easily and dynamically script up the configuration and then add strong type checking, where the definitions are supplied by the authors of the server side, so you don't have to track changes. So, for instance, when a breaking change appears in a newer API version, it will be made clear to you just by generating the configuration. The server doesn't have to try the bad configuration and reject it.

What's more, protocol buffers provide a way to update message formats by adding to the end of a structure. This allows for compatibility as you change your messages.

Also, notice that the above snippets do the right thing when type bounds are violated. So if I change the port number in the `bootstrap.sky` to something far outside of the bounds of a TCP port, the following would happen:

```
followprotocol$ ./followprotocol envoy.sky
panic: ValueError: value 12345678910 overflows type 'uint32'.
goroutine 1 [running]:
main.main()
        /home/spacey/go/src/github.com/pcn/
followprotocol/main.go:94 +0x7c6
```

Full disclosure: the definition of the message specifies that this uint32 must be <= 65536, but as of this writing, there seems to be an issue with this, so I overflowed with a larger number for this example to contrast it with CPython behavior.

Any time this sort of check catches an error, it is like free time being given back to you! One of the most common problems with configuring programs is that in order to know whether a configuration is even valid—things that are supposed to be strings look and act like strings, numbers are numbers, etc.—you need to pass them into a running process where that process validates it (e.g., in the best case with a `--check-config` flag or something along those lines). But even a checker often won't be able to tell you that you've violated a constraint that has to do with the type and not the format. Some things that a strongly typed checker can know is that, for example, you've configured a number value to be larger than an unsigned 8-bit integer, and it will only accept a signed 8-bit integer. Or you create a list that contains strings and numbers, but for a situation where the required list is only able to accept strings. These, in addition to the actual syntax errors, are much, much more difficult to catch, and the goal is that the process of creating the configuration makes it clear that these errors are present. It also turns the problem of perhaps indenting YAML a bit oddly into a problem of Python indentation. Since the syntax is Python, you can use Python syntax checkers to your advantage, though they're imperfect. In any case, the fact that these messages are declared and dealt with by the Skycfg protocol buffer handling means a whole class of checks is largely done for you.

Another effect of this is that since Skycfg isn't a general-purpose language, once a message is created, handling it is done outside of Python syntax. With only a little bit of experience with Golang, you can take the messages that are generated by Skycfg and do something with them there. They could also be saved to a file or shipped out over a network socket—you do need to add this in for yourself. Oddly, you may find that after doing all this work, you end up writing everything out as YAML, per my example repository. So it's always a good idea to keep that option in mind.

Templating

Lastly, let's discuss the Skycfg approach as compared to another method that's often used to generate configuration: use a templating language/macro language like Jinja2, Mustache, or maybe Erb if you're using Ruby. Configuration syntax for simpler things tend to be quite comprehensible at first, but some parts can grow and change to the point where you end up gaining domain-specific knowledge about particular sections of configuration that because of their irregularity have nothing to do with anything else—they sort of make up their own rules as they go along. The configurations for Apache and Nginx are very expressive, but they also make it very challenging to just confirm that they are acceptably formatted. Using a templating approach works very, very well when the problem is simple, and, fortunately, most configurations can be made to be simple and can work by fitting them into a pretty simple template.

Unfortunately, generating YAML with templates, even with regular, simple YAML, gets tricky as soon as you attempt to graft new data structures onto the existing text of a partial message via appending templates. It doesn't seem like it should be so hard, but it turns out that it often is.

By defining a service in terms of protocol buffers, and by using that to make systems that are meant to be operated programmatically via an API, the authors of Envoy and Kubernetes (among others) are inviting the use of a solution like Skycfg in order to generate the desired configuration faster and more safely. I recommend taking a look at Skycfg if you find yourself working with a system that defines itself via protobuf messages.

Note about the last column: In the last column, I mentioned that I'd look to see whether there's a way to make a change to something like the `zip` built-in work throughout a codebase. So far, I haven't found a way to do that well (the idea I had in my head failed so hard…), so I'll look a bit more to see if it does, in fact, seem possible.

Passwords

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

Passwords. Everybody hates them, but everyone still uses them.

While there are pushes for certificates, OTP, and other forms of authentications, the password is still king. In addition to the ubiquity of passwords, good practice dictates that we use different passwords for every account silo, and (controversially) we are supposed to change them often.

Most corporate environments use a single account silo, and the use of single sign-on systems has the promise of not needing to authenticate regularly. However, most of the time, these systems are more consistent sign-on instead of single sign-on, so you end up typing your password over and over again. On the plus side, this is convenient in quickly updating muscle memory following password changes.

The above reasons have given rise to the heavy use of personal password storage. There are online services which provide this in bulk. Most operating systems provide some form of personal password storage: Windows has credential manager, OSX has Keychain, and Linux has several depending on the distribution that you're working on. Even the mobile OSes have some form of native secrets manager that is now being opened up to the applications.

In this exercise, we're going to examine Go libraries for OSX password manager, a Windows password manager, and one that is cross-platform. We'll be looking at how to interact with them and how they store the passwords via the libraries.

The code for these examples can be found at https://github.com/cmceniry/login in the passwords directory. This code is using `dep` for dependency management, but this should work with Go modules as well. After downloading the code, change into the example's directory (`keychain`, `credmgr`, `keyring`) and run the example with `go run main.go`.

Caveats

The built-in password storage mechanisms on most OSes authenticate the application running as well as the user. With some storage types, you can grant permissions for the application to bypass the user authentication. The target of this grant depends on the OS—e.g., it can be the application binary, the user + binary, the name or file path to the binary, etc.

Unfortunately, this does not play as well with `go run` since that produces a different binary and identifier every time you run it.

Because of that, I recommend that you do not apply any "AllowAlways" rules for any of the runs, and that you only use test examples for these runs. At the end, you should clean up the examples that are created. In a production situation, once your binary has been built and distributed, then and only then, should you decide whether "AllowAlways" is worth the risk.

Some of the libraries attempt to simplify the overall interface to the native password stores. This simplification sometimes creates incomplete maps. In addition, the multiple libraries do not map the fields consistently, and the ones that support multiple native implementations may map each of those differently. This does make it a challenge to understand which is the correct invocation for each library and native store. Be sure to double check the documentation.

I've attempted to make it obvious in these examples, but you will see that that is not always an easy prospect.

Most of the libraries focus on the password or generic (`[]byte`) secret. Many of the native stores support additional typing for their secrets, but most of these are not supported by the libraries. Accordingly, we're going to focus on generic passwords in these exercises.

OSX Keychain

There are multiple implementations that interact with the OSX Keychain. We're going to explore the `go-keychain` library from `keybase`. This will have the import line (with alias to avoid naming issues):

```
keychain "github.com/keybase/go-keychain"
```

Keychain stores "Items" which are a password blob combined with metadata.

As mentioned, Keychain is capable of storing multiple Item types inside of it; however, the library support, and hence our focus, is limited to passwords, specifically "application password" (term inside of Keychain) or `GenericPassword` (term inside of the library).

`go-keychain` supports four pieces of metadata: the "Name" or "Label," the "Account," the "Service," and the "Access Group." The Name is what this Item shows up under in Keychain itself, and the library refers to this name as the Label. The Account is a string for the username associated with the password. The Service is a string for where (e.g., the URL) you want to use the password (the underlying Keychain field is literally called "Where"). You can have multiple Items with the same Name as long as the Account is different. Since the other libraries do not support a distinction between the "Name"/"Label" and the "Service" fields, we're going to set them to be the same thing.

The Access Group is a way of collecting multiple applications and multiple passwords and administering their access together. This is not used by other libraries and other OSes, so we will not use it here.

To create a simple password, we pass a `NewGenericPassword` to the `AddItem` library func:

`keychain/main.go: create.`

```
err := keychain.AddItem(
   keychain.NewGenericPassword(
      ";login example,"
      "falken,"
      ";login example,"
      []byte("joshua"),
      ","
   ),
)
```

Since we have to ensure that the Name and Account are unique, it is possible to encounter a duplicate. For this simple example, we're going check our errors for that and ignore just that error while responding to any other errors.

`keychain/main.go: duperr.`

```
if err != nil && err != keychain.ErrorDuplicateItem {
```

If all goes well, we've stored it into the password store, and now we need to retrieve it. There are two ways to retrieve the secret: the get helper, and a full query.

For just getting a password with known location, there's a convenient `GetGenericPassword` func that will grab it for us, and we can print it out. To use it, we have to identify the metadata for it: Name/Label, Account, Service, and Access Group. Since this function is general purpose, we have to fill in all four fields, even with empty strings, to match the signature.

`keychain/main.go: getgeneric.`

```
item, err := keychain.GetGenericPassword(
   ";login example,"
   "falken,"
   ";login example,"
   ","
)
...
fmt.Println(string(item))
```

The second method is to `query` for it. The `query` is useful when looking for multiple Items. We set the parameters that we need to match on, indicate that we're looking for one or multiple answers, ask for it to return the data rather than just the metadata, and then perform our actual query. Since this can return multiple responses (though in this case only one will return), we still want to iterate over the results.

`keychain/main.go: query.`

```
query := keychain.NewItem()
query.SetSecClass(keychain.SecClassGenericPassword)
query.SetLabel(";login example")
query.SetAccount("falken")
query.SetService(";login example")
query.SetMatchLimit(keychain.MatchLimitOne)
query.SetReturnData(true)
results, err := keychain.QueryItem(query)
...
for _, i := range results {
   fmt.Println(string(i.Data))
}
```

This works well if we're looking for a specific password by account. Try removing the `SetService` or `SetLabel` calls for the query to see what is returned.

Credential Manager

Next, we're going to interface with Window's Credential Manager using the `wincred` library from GitHub user `danieljoos`.

It has the import path:

```
"github.com/danieljoos/wincred"
```

Like Keychain, Credential Manager maintains some metadata for its secrets. It requires a "TargetName," which has to be unique, and allows for an optional Username field. For this example, we're going to limit it to just the TargetName and secret itself.

The creation of new passwords is relatively straightforward. We create a new record by its name with `NewGenericCredential`, assign the password itself and the optional metadata, and then write that to the store.

credmgr/main.go: create.

```
cred := wincred.NewGenericCredential("loginExample")
cred.UserName = "falken"
cred.CredentialBlob = []byte("joshua")
err := cred.Write()
```

In the event of duplicates, Credential Manager will overwrite what is there.

Now we can retrieve that password out of the Credential Manager. Since we're fetching by TargetName, it's a simple `get` command followed by print.

credmgr/main.go: get.

```
cred, err := wincred.GetGenericCredential("loginExample")
...
fmt.Println(cred.UserName)
fmt.Println(strings(cred.CredentialBlob))
```

The `wincred` library and underlying interface to Window's Credential Manager is quite a bit more intuitive than the Keychain interface, but it also does not allow for more complex cases that use duplicate metadata values for records (e.g., retaining multiple versions).

Cross-Platform

Now, let's combine those and use a common library to try to make it cross-platform. To be specific, our definition of cross-platform use is to be able to use the same binary/code across multiple OSes; it is not about moving the password data across multiple OSes. Directly porting over the password data is complicated and needs some transformation since Keychain and Credential Manager and other native stores have different semantics.

For cross-platform usage, we're going to look at the `keyring` library by `99designs`. It supports storing secrets in multiple backends: Keychain, Credential Manager, the Gnome secrets service, KDE Wallet, and others. It has the import path:

```
"github.com/99designs/keyring"
```

Many of the overall options for `go-keychain` and `wincred` are stripped out from `keyring`. With it, you can specify a container, the "ServiceName," and a "Key" for a specific entry. In the underlying password store, these two fields may be mirrored onto other fields (e.g., when backing with Keychain, it sets Label and Service to both be the ServiceName), but `keyring` only allows for these two fields.

For creation and retrieving, we start by opening (read declaring) our password container by its ServiceName:

keyring/main.go: open.

```
kr, err := keyring.Open(keyring.Config{
    ServiceName: ";login example,"
})
```

After that, we are able to commit it to the password store with its value. We must also identify the unique Key inside of this ServiceName for this password. Note that unlike some of the native libraries where this is a "construct then write" method, the `keyring` library does this as one command (again, different semantics).

keyring/main.go:create.

```
_ = kr.Set(keyring.Item{
    Key:  "falken,"
    Data: []byte("joshua"),
})
```

Much like `wincred`, `keyring` overwrites existing values instead of indicating a duplicate Item.

Now that we have the data stored, we can pull it out and display it. As with the single function to write the Item, the `keyring` library uses a single function to retrieve data.

keyring/main.go: get.

```
p, err := kr.Get("falken")
if err != nil {
        panic(err)
}
fmt.Println(string(p.Data))
```

Library Compatibility

Making it so that we can use `keyring` and either `go-keychain` or `wincred` together requires us to match up the appropriate metadata.

For `go-keychain`, this is a matter of matching the `keyring` ServiceName with both the `go-keychain` Label and Service and the `keyring` Key with the `go-keychain` Account.

For `wincred`, we must match the `keyring` ServiceName with the `wincred` TargetName which currently has to include some additional markup. As of this writing, the `keyring` library will append `aws-vault:` to the ServiceName and Key to store as the TargetName in `wincred`. (The `keyring` library was originally named `aws-vault` and so retains a few vestiges of that. There is an open issue for this.)

Conclusion

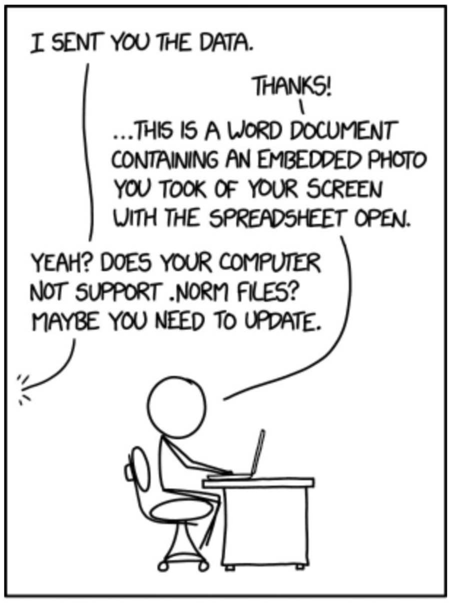
Password management is hard.

The current level of cross-platform compatibility is low. There is no standard for password storage—particularly the identity and metadata for finding the passwords. There are significant inconsistencies in the native password stores as well as inconsistencies in the way that the libraries map to those password stores. Applications can be ported across multiple OSes, but the passwords saved are hit or miss if you try to use them across multiple applications. It's best right now to pick one library and stick with it.

When considering the user, it's a constant tradeoff between security and ease of use. You have to decide for yourself what is your level of risk. If storing passwords in the native password stores encourages other secure activities (e.g., shorter sessions, not using static API keys, etc.), then this may be right up your alley. If it's a matter of storing the passwords in the native passwords stores versus your own securely wrapped (encrypted, permissions, etc.) files, it's better to use the former as a significant amount of engineering effort has been put in place that makes the native password stores probably more secure than what most of us would do otherwise.

Despite the issues, I hope that this column has given you some insight into how to handle these in reasonable ways and the confidence to do so. Anything that we can do to improve the state of password management by making it easier on the user and more secure is a boon for our field. Native password stores are just one of those methods, and I encourage you to use them. Good luck and be safe!

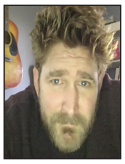
XKCD xkcd.com


I SENT YOU THE DATA.
THANKS!
...THIS IS A WORD DOCUMENT CONTAINING AN EMBEDDED PHOTO YOU TOOK OF YOUR SCREEN WITH THE SPREADSHEET OPEN.
YEAH? DOES YOUR COMPUTER NOT SUPPORT .NORM FILES? MAYBE YOU NEED TO UPDATE.

SINCE EVERYONE SENDS STUFF THIS WAY ANYWAY, WE SHOULD JUST FORMALIZE IT AS A STANDARD.

iVoyeur
Prometheus

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Sparkpost. His continuing mission: to help engineers worldwide close the feedback loop.
dave-usenix@skeptech.org

I keep having this conversation—a byproduct of my own incessant humble-bragging about living in Montana. I will be talking to someone I've just met, at a nerd meet-up say, or maybe a coffee shop or airplane, and the topic will just sort of come up. Arise. Spring forth into being.

It probably has something to do with my handshake. The way I grab people by their outstretched hand, not so much heartily greeting them as capturing them in place, and ensuring they cannot politely escape as I loudly exclaim something to the effect of:

"HI-I'M-DAVE-I-LIVE-IN-MONTANA!"

And then, as if by magic, the conversation careens away from whatever direction it probably should have been headed directly north into the tree-lined hinterlands.

Obviously, as the architect of this colloquial digression, I'm more or less rudely declaring my preference for hinterlands over the unknowable set of topics which could be derived from the unique soup of ingredients of your humanity combined with mine.

I mean, we were probably just going to talk about the weather anyway, but it does bother me. This notion that I have robbed us of the opportunity for spontaneous conversation in exchange for my personal, known baseline of locutionary enjoyment. Had I not grabbed the steering wheel and swerved, we might be talking about bread-making right now or tuning diesel engines. Maybe you're 3/4 of the way through the *Manga Guide to Linear Algebra* and are just dying to talk with someone about it. That would have been really fun, and I failed to allow that possibility to blossom between us. I didn't give you the credit you deserve.

I recognize that. I do. So although I brought us here, I won't insist on us staying. What often happens is that you will know of a person or place in Montana. You used to visit your grandma in Pony, or maybe your ex frequents the ice-climbing festivals in Hyalite Canyon. And so we are brought closer together by virtue of a shared experience with place.

Maybe the notion of wilderness itself resonates with you. Your heart pounds for the dry-hot, windy openness of the West: Arizona, New Mexico, Utah. Maybe you've wandered central Africa or sailed the coast of South America. You've never been to Montana per se, but there is a remote place threaded into the fabric of your soul. A hard-to-get-to place into which you could happily disappear and, just as happily, never return.

But just as often, you happen to be someone to whom my happy place parses as a hellish sort of prison. A person who finds it hard to imagine a fate worse than banishment to the icy, deserted foothills of the Absaroka-Beartooth Wilderness, where the elk outnumber people, there is no decent pizza, and the preponderance of what little conversation there is to be had revolves around, well, bread-making and tuning diesel engines.

Thus, a sort of conversational reset transpires. We can't, after all just throw our hands in the air and run away from each other. That'd be weird. So we do what all good engineers do: we turn it off and then turn it back on again. You say something like, "Wow, you really must like getting away from it all," and I chuckle politely and hand over the wheel. First law of improv and all that; I'm sure we can find common ground in some other subject matter that is hopefully not the weather.

But that phrase *getting away from it all*; it'll still bug me weeks later. The unspoken thought that I am willingly abandoning the richness of a full place for one of emptiness. That I'm escaping or ejecting. Getting away from "it," whatever it is.

I guess maybe we have different *it*s. And I think I get yours, the cultural it, with the third-wave coffee and the music and the bright laughter, and liberality of ideas. The golden aura that glows around healthy, sophisticated, urban coexistence.

But you don't need to come to the hinterlands to experience my it. You can find it within yours. *Between it*. In empty offices after dark and in shops that haven't opened yet. In any kind of place where, if you were to stand in place and turn time like a knob, you would see that the place always exists, but the people who inhabit it exist only between the hours of 8 and 4 say, only as long as the band is playing or until the bell rings.

With apologies for polluting an otherwise objective and scientific journal with magical thinking: there's something that lives in the spaces we vacate. Not an emptiness, but a kind of presence or potential. A thing accentuated by our absence. A signal underlying the obvious stuff, there for anyone who cares enough to listen. A wilderness. You can feel it vibrating there if you hold still, in the in-between. That's my it.

Prometheus

I have mentioned Prometheus (http://prometheus.io) here and there in my recent articles, and I think the time has finally come to do a short series on it. There are myriad details of note. Its built-in time series database and cloud-native focus. Its reinvention (or maybe revitalization?) of the pull-model so abhorred by DevOps acolytes, and its query-centric operation and accompanying domain-specific language: PromQL.

But rather than charging head-first into any of those technical frontiers, I'd like to take a moment to make a point about what systems like Prometheus represent in the context of the evolution of systems monitoring. There is a continuum where, on one end, exist the monitoring systems of the "past." They were built, in a way, unconcerned with what they would eventually be tasked with monitoring. Which is fair, since the things they monitored were equally unconcerned with being monitored.

If we built airplanes like this, it would be as if we built one airplane with no instrument panel and then a second altogether separate airplane to fly beside it, taking note every so often of the first plane's speed and the temperature of its exhaust. Whatever could be seen from the outside. The first airplane flying happily, obliviously along until it didn't. And in the aftermath we'd ignore the wreckage, turning our attention instead to the plane that *didn't crash*. Scratching our heads and combining its samples with our intuition to guess at what might have happened to the plane that did.

On the other end of the continuum, every piece of software written worldwide *is a monitoring system* and, oh by the way, also possessed of some additional, arbitrary functionality. As if everything we make is instrument panel, upon which we sometimes bolt wheels or wings.

Both ends of this continuum are dystopian in their special way, but also instructive in that they hint at some middle ground in between. Imagine a place where applications run, interactions happen, and customers are serviced, and in between all of that, quite unbeknownst to everyone, a telemetry signal is emitted, underlying the obvious stuff. There for anyone who cares enough to listen.

We aren't building a monitoring system into everything we make, but we are acknowledging the importance of feedback, upfront in the development process, and providing an organizational answer to the question of where to send operational telemetry.

Prometheus server is probably the first centralized poller that assumes the presence of such a signal. It's a traditional poller in that it wakes up on a configurable interval, polls metrics, and stores them internally. Its contribution to the field of pollers is subtle but important: it *only* polls this specific kind of signal, emitted via HTTP on one or more TCP ports on every host.

If you've worked with other monitoring systems, this might seem like a limitation at first, but it's really quite liberating. There is no particular agent to install; no collector-side of its body to configure and reunite with its head. Instead, you are encouraged to participate directly in the data-model by taking whatever you want Prometheus to slurp up, packing it into a text format, and making it available via HTTP on a pre-arranged TCP port.

So we are left to create this signal. We can emit it from within—inside a running process or thread that is our application—or from the outside using a piece of software dedicated to collecting metrics. We can even post-process logs into the correct format. Obviously, we can also use a combination of techniques and, in practice, pretty much always do. Here's a simple example of its text format.

```
# HELP go_threads Number of OS threads created.
# TYPE go_threads gauge
go_threads 71
# HELP process_cpu_seconds_total Total user and system \
        CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 24738.72
# HELP process_max_fds Maximum number of open file \
        descriptors.
# TYPE process_max_fds gauge
process_max_fds 1024
```

These metrics were emitted from inside a Go program, using the Go exporter (https://github.com/prometheus/client_golang). If you're used to dealing with metrics systems, the concept of a named value like `process_max_fds 1024` is no doubt very familiar to you. Note also the metadata fields that provide a datatype and a human-readable description for each metric; although these appear to be comments, they are in fact required fields.

In Prometheus land, the software that emits metrics signals is always called an *exporter*. Getting started, you probably won't need to write one yourself because there are already off-the-shelf exporters for just about everything you can imagine, including a "node exporter," which works like a traditional monitoring agent, reading values out of /proc and /sys and exporting these as Prometheus signals for you.

Here's another, more complicated version of Prometheus's data format:

```
# HELP nginx_requests_total Total number of reqs by HTTP \
        status code.
# TYPE nginx_requests_total counter
promhttp_metric_handler_requests_total{code="200"}
1.654693e+06
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
```

This example adds "labels," enclosed in curly braces after the metric name. Arbitrary metrics dimensionality can be achieved with labels, but the Prometheus documentation (and anyone who has used it in anger) warns against their overuse (https://prometheus.io/docs/practices/naming/). In real life, metrics should be restricted to labels with fewer than 10 dimensions: e.g., recording requests by HTTP type is fine, but attempting to record requests by customer ID will quickly blow up in your face.

I can tell you firsthand, the act of creating and nurturing this signal proliferates quickly through an organization, with application engineers using libraries to export metrics directly from their services as well as more operations-oriented engineers writing shell-scripts or other little pieces of automation to add more host-oriented metrics. Once you start working with it and relying on it, it quickly becomes habit-forming. At Fastly we've written a discovery tool (https://vimeo.com/289893972) called *PromSD*, which makes it easy for Prometheus (and people like me) to discover and explore the metrics backchannel in the in-between space of our network.

The data model is probably my favorite thing about Prometheus, this lovely notion of a monitoring subtext quietly woven into the fabric of the "important stuff." A single text format we can all agree on, that multiple monitoring systems or even human beings can consume. It feels like our little secret, and I think it's an important step forward for the field of monitoring in general. Tune in next time when we'll explore Prometheus's local storage, and PromQL, the query language you can use to interrogate the storage layer and draw graphs.

Take it easy!

For Good Measure
Curves of Error

DAN GEER

Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

The major difference between a thing that might go wron-g and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair. —*Douglas Adams*

One hears often enough that the error rate for software is so many flaws per thousand lines of code or the like. A fraction of those flaws turn out to create vulnerabilities. A fraction of those vulnerabilities get exploited. And "we" learn about a fraction of those exploits. Let's call it

$$S * F * V * E * P$$

In other words, we create S lines of new code, F of which are wrong, V of which are vulnerabilities, E of which are weaponized, and P of which come to our attention. Let's stipulate one thing: arguing about what constitutes a line of code is irrelevant. While we're at it, let's stipulate that everything here is subject to argument about definitions and what goes in what set.

That kind of formulation is similar to the kinds of rough calculations around whether there is other intelligent life in the galaxy. On the one hand, there are something like 100 billion stars in the Milky Way. On the other hand, intelligent life requires a bunch of pretty unlikely coincidences (probabilities) multiplied by that 100 billion. You only need five 1% probabilities multiplied together to get down to as many intelligent life planets in the Milky Way as you have fingers. Six such conjunctions and the odds turn against our very existence.

So, how many lines of code? That is harder to estimate than the number of stars in the Milky Way. An unsubstantiated claim in CSO magazine [1] was that 111 billion lines of code (LOC) were created in 2017. Elsewhere, there's a slightly more substantiated estimate of 20 million developers at work today [2]. The old rule of thumb for a developer is/was 50 LOC/day or 10–15 KLOC/year. Multiplied times 20 million devs, that's 20–30% of that 100+ billion LOC claim. Of course, some code is not written by hand but by machine, but is it really 70–80%? Or are there more than 20 million devs? Or are they more productive than 10–15 KLOC/year? With almost 8 billion people on earth, does it sound right that 1 in 400 is a dev? Let's take $S = 10^{11}$ for the moment.

How many flaws are in that code? The old rule of thumb is/was 25–50 flaws per thousand lines of code (KLOC), although the various measurements that come to that range of numbers were for software products that are deployed en masse after a defined build process as opposed to continuously thrashed web applications. Anyhow, if we use 40 flaws/KLOC and multiply it by 100 billion LOC, we are looking at 4 billion new flaws per year.

As a kind of comparative calibration, the top six open source package repositories account for 80% of all open source repositories [3], a combined total of 1.75 million packages, which number is increasing by 1,000 per day. That's 23% annual growth, and we're not even talking about GitHub or SourceForge.

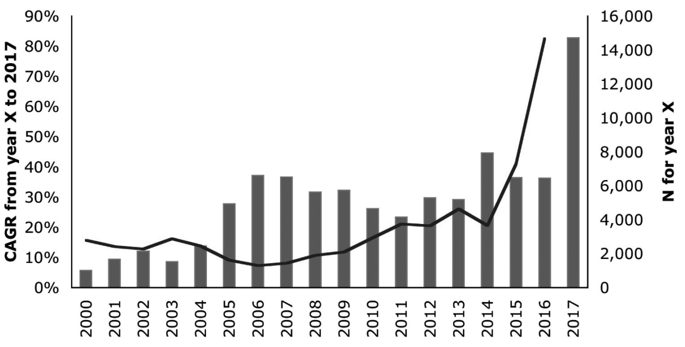


Figure 1: Number of CVE entries and CAGR from each year to the 2017 value

Turning to what fraction of uncategorized flaws are security flaws, i.e., what fraction of bugs create vulnerabilities, we find two schools of thought. School One: any and all bugs are vulnerabilities unless and until proven otherwise. School Two: only a small fraction of all bugs are security bugs. For today's purpose, we'll side with School Two, taking the line that vulnerabilities are a small percentage of total flaws. That may well be incorrect in the sense of "failing to make the conservative assumption" (conservative with respect to security outcomes, that is).

In any case, the vulnerability to flaw ratio is related to Bruce Schneier's foundational question of whether, in truth, vulnerabilities in software are sparse or dense [4]. Chris Wysopal says that Veracode [5] finds 0.1 vulnerabilities per KLOC, so a 40 flaws/KLOC starting point means that 0.1/40 = 0. 0025 or 1/4 of 1% of flaws are actually vulnerabilities. That makes it conservative to say that 1% of all software bugs are vulnerabilities, so we'll go with that for the moment.

Then there is the probability that a given vulnerability can and will be weaponized, which is to say turned into a deployed exploit. As Dave Aitel [6] has repeatedly argued, what is sparse is not vulnerabilities that could be weaponized but the people who can weaponize vulnerabilities. Add to that that good exploits may require more than one vulnerability, i.e., the conversion rate of vulnerabilities to exploits may be lower still. Plus in a whole-world setting where the installed base of software is growing faster than the human population, then the fraction of vulns that are weaponized might actually be falling, not for want of opportunity but for want of labor (back to Aitel). In any case, and for the purpose of argument here, let's call it 1-in-200 or .005 that a given vuln will be weaponized.

That 1-in-200 is almost surely way conservative. Brian Martin of Risk Based Security [7] has data showing that out of 199,311 vulns with a CVSSv2 9.3 or higher, 6,244 have a public exploit, 2,350 have a proof of concept, and 3,048 have a private exploit. That works out to 5.8% of those (very serious) vulns are known to have, or be capable of, exploits. That's an order of magnitude higher than 1-in-200, but we'll stick with 1-in-200 for the moment. While we're at it, HackerOne says that in 2018 they managed 78,275 reports [8]. If even half of those are valid security bugs, then it would more than double the 2018 CVE or VulnDB count.

Of course, some weaponized vulns will never come to our attention. For this estimate, we must admit that we are in the murk; zero-days don't get counted since they aren't 0days if we can count them, nor are exploits that are use-once for precious targets something we'll ever see. This is what Ablon and Bogart covered so well for RAND [9]. For the sake of argument, let's pick maximum ignorance priors, i.e., say that 50% of vulnerability weaponization is unobservable while 50% comes to some kind of public attention. This brings us back to the top, viz.,

$$S * F * V * E * P$$

which we'll rewrite with $S = 10^{11}$, $F = .04$, $V = .01$, $E = .005$, and $P = .50$

$$10^{11} * .04 * .01 * .005 * .50 = 100{,}000/\text{yr}$$

A number like 100,000 de novo, non-targeted exploits in the wild per year is certainly a stunning number. But is it real? Does it carry policy freight?

In calendar 2017, there were 14,714 new CVE reports made. Going back to year 2000, there were 1,020. That works out to a compound annual growth rate (CAGR) from 2000 to 2017 of 15.7%. For 2001, there were 1,677 reports, which amounts to a CAGR from 2001 to 2017 of 13.6%. Figure 1 lays this all out; for each year, the column represents the number of CVE reports made, and the line is the value for the compound annual growth rate between that year to 2017.

Focusing on the longest term, from 2000 to 2017, that CAGR of 15.7% begs the question: how fast is the total body of installed code growing? If that total installed base is growing faster than 15.7%, then CVE would say that either we're collectively getting better at making new software secure or we're collectively getting worse at finding (and recording) security problems in new software.

Generally speaking, a measure that has constant error will return value estimates that are wrong, but its trend line will at least have the right shape. Therefore, we might ask the question, "Does CVE has relatively constant error?" Unfortunately, the answer to that question is almost surely "No"—CVE's coverage has been shown to be poor, and the recent spike in its listing of new vulnerabilities is more a response to embarrassing congressional hearings in late 2016 than anything else: i.e., the big jump in 2017 is an artefact.

We seem to have no good measure, then, of the shape of the curve of error. Bug bounty payouts are not it (it being the way to measure the fraction of vulnerabilities that are exploitable). So we are back to the number to noodle over: is finding 100,000 de novo, non-targeted exploits in the wild per year a round-number estimate good enough to inform policy?

Obviously, spreading 100,000 new exploits through 100 billion new lines of code constitutes a seemingly low density—literally one in a million ($10^5/10^{11}$). That ought to be reassuring. Or should it? The 2016 Ford F150 pickup truck is said to have 150 million lines of code [10]. By our working guess of 1-in-a-million, that $150 * 10^6$ LOC might be expected to set the stage for 150 exploits. Perhaps thankfully, the Boeing 787 Dreamliner is said to have 7 million LOC which gives the naive estimate of a half-dozen exploits waiting to come "out of the nowhere into the here."

If the reader thinks software errors are some flavor of inevitable and are not designed in by a present-day Illuminati, then those errors are sprinkled over software products like some kind of pixie dust. Even if the estimates above are off by an order of magnitude (in either direction), the implication, both personal and policy, might well be this: the more software there is in a product, the less you should depend upon it. The more a given supplier lards up the product with features, the less you should want to depend on it. The more often the software base turns over, the less the software in it has been burned in.

For the present author, a state of security is the absence of unmitigatable surprise, hence the Douglas Adams quote at the start, the conservative assumptions throughout, and the previous paragraph's appeal to retaining alternative—analog if you prefer—mechanisms. That position, itself, would be shown to be conservative if finding bugs with AI turns out to be as effective as it might be. While AI uber-bug-finding would likely depress the number of latent, as yet invisible threats, it is hard to imagine that any substantial entity would be prepared for their AI to autonomously fix bugs it had autonomously found, so, in turn, the number of known bugs could well increase faster than the market cycle could accommodate fixing them. And then we'd be writing about this in parallel to the vaccination of school-age children as a state-imposed access requirement to a public good, only now it would be autonomous update as a state-imposed access requirement to a different public good.

References

[1] S. Morgan, "World Will Need to Secure 111 Billions Lines of New Software Code in 2017": https://www.csoonline.com/article/3151003/world-will-need-to-secure-111-billion-lines-of-new-software-code-in-2017.html.

[2] R. Cox, "How Many Go Developers Are There?": https://research.swtch.com/gophercount.

[3] Open source repository timeline: http://www.modulecounts.com/modulecounts.csv.

[4] B. Schneier, "Should U.S. Hackers Fix Cybersecurity Holes or Exploit Them?" May 19, 2014: https://www.theatlantic.com/technology/archive/2014/05/should-hackers-fix-cyber-security-holes-or-exploit-them/371197/.

[5] Veracode, *State of Software Security*, vol. 9: https://www.veracode.com/state-of-software-security-report.

[6] D. Aitel, personal communication.

[7] B. Martin, personal communication; https://vulndb.cyberriskanalytics.com.

[8] "Hacker-Powered Security Report 2018," July 11, 2018: https://www.hackerone.com/blog/Hacker-Powered-Security-Report-2018.

[9] L. Ablon and A. Bogart, *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits*, RAND Corporation, 2017: https://www.rand.org/pubs/research_reports/RR1751.html. Also available in print form.

[10] R. Saracco, "Guess What Requires 150 Million Lines of Code...," January 13, 2016: https://www.eitdigital.eu/news-events/blog/article/guess-what-requires-150-million-lines-of-code/.

/dev/random
Techno-illogical

ROBERT G. FERRELL

Robert G. Ferrell, author of *The Tol Chronicles*, spends most of his time writing humor, fantasy, and science fiction. rgferrell@gmail.com

I recently stumbled over an online debate attempting to assign blame for the failure of technology to live up to society's expectations (although who, precisely, established those expectative benchmarks wasn't clear, possibly because, as is my custom, I only skimmed the first few paragraphs: tl;dr is my motto these days). Was it the fault of technology itself or of the companies that make it? In other words, is the reason we don't have flying cars because the technology is still out of reach or simply because no company has bothered to make full use of what we already have available?

If you're going to remind me at this juncture that we do in fact have flying cars, let me stop you right there. We both know that the flying cars we were promised were something the Jetsons would recognize as such, not these giant drone abominations that you could fly inverted and mow the lawn. Nor are the "hoverboards" currently available anything like the models we were expecting. In fact, no component of them does any actual hovering. They're more akin to self-propelled skateboards with their wheels mounted sideways, as though they were designed and assembled under the influence of peyote.

There was a time, I'm led to understand, when technology was still a shining beacon of hope on the horizon, promising solutions to problems we hadn't yet even created for ourselves. That was an era of optimism and blind, cheerful trust in the implicit genius of the technologists who were going to make life so much easier and more pleasant for us all. The utopian society of the future would be populated by buttons, gadgets, and machines that go "ping," all working in seamless unison to enable the humans of the household to go about their day in a state of blissful freedom, unencumbered by the necessity for manual labor or thinking of any sort.

That's what we were promised. What we got, at least up to the present day, falls a bit short of that ideal. The buttons summon products we don't really need or could easily obtain by more conventional avenues that don't require relinquishing our last vestige of privacy; the gadgets spy on us unceasingly and bombard us with a barrage of ads for yet more gizmos we don't need; the machines trap us in a closed loop of surrendering basic control of our domestic landscape for the sake of perceived convenience. Hooray for progress.

If future projections aren't quite being met on the home front, they're even more divergent when seen through a wider lens. We've already covered the disappointments that are the flying car and hoverboard, but these only scratch the surface of our poor record at projecting technological achievements. Perhaps, however (as I intimated in the first paragraph), it isn't our predictive acumen that's to blame but some aspect of technology itself.

We tend to ascribe virtually no limits to the wonders technology can summon. In point of fact, the very word "technology" is more or less useless as a substantive noun. It doesn't mean much, other than the *application of knowledge for practical ends*. So, can the application of knowledge be indicted for not applying itself? Blaming technology for failing to live up to our nebulous and in many cases wholly unrealistic fantasies is hardly rational. If indeed there is a culprit in all this, it must therefore be the company that develops said technology.

Now that we've established culpability for our disappointment, let's move on to creating some sane expectations. Technology isn't a broad-spectrum magic bullet, and it can't pull solutions out of thin air. Pretty much every technological advance generates as many new questions as it provides old answers, in the grand tradition of progress. Moreover, technology has proven time and again that it is not easily predictable in any granular sense.

I read a study just today that claims a substantial proportion of millennials interviewed considered cracked smartphone glass and a failure to receive "likes" on their social media posts as major stressors in their lives. Leaving aside the questionable priorities thus illuminated, what this statistic reveals to me is that technological advances my generation considers relatively minor—a smartphone is just the latest iteration of the telephone, after all, and that's been around since the 19th century—have assumed a central position in an entire generation's lives. They rely on their phones for, well, *everything*.

I dropped my iPhone in a pond several years ago and went 133 days without it. It was frustrating at times, yes, but really not much more than an annoyance. I could still send email, and I never really used the phone part of my smartphone much, anyway; it served mostly as just a terminus for spam calls. To someone of the current generation, however, I'm told that same situation might prove utterly devastating, on par with sudden amputation of a limb or loss of family members in some horrific accident. Technology here has clearly moved from servant to master.

Perhaps that's the crux of this entire discussion: is technology meant to be something that enriches our lives or exerts control over them? Will we even be able to make that distinction before long? The technology we want is occulted by the technology we realize, which then eventually replaces our expectations with itself. The substituted technology often bears little resemblance to what we visualized, but sooner or later that disparity loses any relevance.

It's probably best for our collective blood pressure if we accept that technology moves in its own direction at its own pace, whether or not we find that movement agreeable. Don't establish inflexible expectations and you won't be disappointed when they fail to come to fruition. If you can't have the flying car you love, honey, love the flying car you get.

Book Reviews

MARK LAMOURINE AND RIK FARROW

Refactoring, 2nd ed.
Martin Fowler, with contributions by Kent Beck
Pearson Education, Inc., 2019, 418 pages
ISBN 978-0-13-475759-9

Reviewed by Mark Lamourine

Martin Fowler released the first edition of *Refactoring* in 1999. It would be nice to think that after 20 years it wasn't necessary to promote the idea of refactoring as a best practice for software development. Sadly, I still find that people blink at the idea of modifying code without changing its behavior, solely to make it easier to read and maintain.

Refactoring is a practice, a technique for improving code and, at least in part, a philosophy. The main idea is that, especially in the "fail fast" world of Agile methodologies, everyone is likely to write a lot of ugly, non-optimal code. That's not a problem by itself because this working-but-incomplete code is revisited over and over, eventually resulting in a polished product. Unfortunately, often the pressure to add features overwhelms the need to clean up the structure of the code, resulting in layers of hacks on hacks that, while strictly functional, become increasingly difficult to maintain and extend.

This (valid) criticism of Agile development styles leads Fowler to compile a set of change patterns (reminiscent to me of *Design Patterns* [1]) which "smell" code that can be improved, based on prior work by William Griswold [2], William Opdyke [3], and others. Following Opdyke's lead, Fowler called them "refactorings."

This second edition largely follows the format of the first, though there are numerous updates that justify revising the original. The examples are presented using JavaScript rather than the original Java. This makes the examples accessible to a larger audience and cuts the amount of boilerplate significantly. Technology has changed in 20 years, and the list of refactorings has been adapted, including some entries for functional programming and some for refactoring without classes. Fowler removed several chapters from the end of the first edition that discussed tools (which go obsolete) and "large refactorings," which are, in reality, redesigned from scratch.

In the first section, Fowler presents the concept and purpose of refactoring. The opening chapter walks through cleaning up a piece of sample code, now in JavaScript. The second chapter provides a definition and motivations for the formal practice. Chapter 3 introduces the idea of code "smells" as ways to recognize poor code. This may be a little subjective, but Fowler argues that these are only flags to look at code more closely. The final chapter of this opening section discusses testing, and how critical it is to be able to confirm that your refactoring actually *didn't* change your code behavior.

The main body of the book is really a catalog of refactoring patterns. This takes up three-fourths of the total space. Each refactoring (yes, it is a noun and a verb) is presented as a kind of encyclopedia entry, and each follows a specific format. An entry consists of five parts:

1. Name: a short description, 2-4 words
2. Sketch: a graphic depiction of the change in code
3. Motivation: a few paragraphs on when and when not to use this entry
4. Mechanics: how to apply the change (sometimes referring to other refactorings as intermediate steps)
5. Example: a sample of code to change, an explanation, and intermediate steps leading to a new result

The refactorings are grouped logically based on the type of change being made. All have an inverse, though not all of these are presented, because many have little value.

The hardcover second edition, published under the Pearson imprint of Addison-Wesley, has about the same page count as the first, but it is only half the width on the shelf and significantly lighter. This does not mean lower quality in any way. The bindings are solid. The pages are smooth without being glossy. Fowler is able, with modern printing, to use color in graphics and to highlight code changes. Both include page marker ribbons in the bindings. The removal of almost 50 pages of summary at the end of the first edition allows the addition of new refactorings and the expansion of the sketch graphics for all of them.

Fowler will be the first to tell you that nearly all of us do refactoring without thinking about it on a daily basis. His purpose

References
[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* 1st ed. (Addison-Wesley, 1994).

[2] W. G. Griswold, "Program Restructuring as an Aid to Software Maintenance": http://cseweb.ucsd.edu/~wgg/Abstracts/gristhesis.pdf.

[3] W. F. Opdyke, "Refactoring Object-Oriented Framework" doctoral dissertation, 1992: http://dl.acm.org/citation.cfm?id=169783.

in writing about and promoting refactoring as a formal practice is to help us avoid the mess that can happen when we try to combine structural changes with feature additions. By keeping these separate in our minds (and in our code), we can both recognize what we are doing and focus on the immediate purpose of our work.

Concurrency in Go

Katherine Cox-Buday
O'Reilly and Associates, 2017, 224 pages
ISBN: 978-1-491-94119-5

Reviewed by Mark Lamourine

When Go was first released, one of the major selling points was the inclusion of concurrency primitives. Together, goroutines and channels were touted as the new solution to a major problem for many web services, or any service that must accept and process many communications at the same time. I read about goroutines and channels and, because that's not the kind of coding I tend to do, I shrugged and moved on. I have written a fair amount of Go, and, because I didn't really understand concurrency, I didn't try to use it.

One problem was that, while many sources showed mechanically how to use goroutines and channels, they all seemed to stop there, assuming that it would be evident that these were cool and useful and that the reader would Go Forth And Code. Another aspect that most examples neglect is how Go's primitives work under the hood. It's always helpful to know not just the overt consequences of some code feature, but the hidden ones as well.

In a mere 200 pages, Cox-Buday explains what concurrency means, how it's been done in the past, and how Go offers a new way. She devotes a full 30 pages of that to detailing why concurrency is important and why it's hard. Long ago I did some hard real-time programming, and this section brought back memories. The realms of web services and flight control systems are very different, but they share a need to process masses of input from many sources in limited time. In this first section she provides examples of race conditions, deadlock, livelock, and resource starvation as well as illustrating the traditional ways of managing them with shared memory and mutex locks. She introduces communicating sequential processes [1], which inspired Go's concurrency model, but also dispels the myth that mutex and shared memory are dead, even in Go.

Cox-Buday introduces the Go concurrency primitives and constructs in a mere 30 pages more. The narrative is clear, concise, and complete but without any fluff. I've seen all of this elsewhere, but the author interlaces code samples, graphics, and program output so that each reinforces the others to make her points.

The remainder of the book is a list of usage patterns. Pretty much every example offered something I hadn't considered, and it's going to take me a few times through and a lot of practice to really get them. There's a lot here to digest, but it's well enough written that it doesn't feel arcane or obscure.

You don't want to start learning Go from this book. Concurrency is fundamental to the language, but you won't get the most out of it by starting there. This is a book for developers who are experienced and want to expand into areas like microservices that will require high volume, low latency communications.

I've found Go to be useful and intuitive even without the concurrency constructs. Now that I have a taste for what they can do, I'm going to be coming back to *Concurrency in Go* often over the next few months to make sure I get the most out of it.

Reference

[1] C. Anthony and R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 1, no. 8 (August 1978): http://bit.ly/HoareCSP.

Cloud Native Go: Building Web Applications and Microservices for the Cloud with Go and React

Kevin Hoffman and Dan Nemeth
Addison-Wesley, 2017, 244 pages, ebook code samples add 180 pages
ISBN: 978-0-672-33779-6

Reviewed by Mark Lamourine

Some books I find with the intent to read and review them. *Cloud Native Go* I picked up because I have a pet project that I use for learning new languages and techniques. I was browsing one day for a book that treats web apps built with Go, MongoDB, and React, and I really was only looking for that last section. As it happens I learned a lot all the way through.

The book is subtitled *Building Web Applications and Microservices for the Cloud with Go and React*. As you might surmise for a book that tries to cover so much, Hoffman and Nemeth don't speak in depth on any topic. For example, they introduce Git in seven pages and Go in eighteen.

Cloud Native Go is really a survey of the suite of technologies and specific tools that are needed to create a comprehensive cloud-based service. The chapters range between 10 and 25 pages, but the longer ones often contain a few smaller sections describing more than one sub-service or component. The authors do provide source code examples on GitHub, although these have not been updated since May 2016, at least six months before publication.

As a survey the book does well, outlining the anatomy of a completely cloud-based service. All of the technologies the authors use have good online references and tutorials. At the end of that short primer on Go, they suggest that readers new to the language set their book aside for a while and get more familiar with the language before coming back. In each case, the authors explain their choice of a particular tool where choices exist (Werker for CI/CD, MongoDB for database, and RabbitMQ for messaging) but acknowledge that these are opinionated selections from a field of possibilities.

Hoffman and Nemeth are evangelists for cloud services and "the way of the cloud." In the first chapter they describe their philosophy, which is an evolution of Heroku's 12 factors [1] and a combination of other Agile tenets. This lightweight philosophy informs the rest of the book.

I'm not really sure who the audience is for this book. While the introductions to Git, Go, and Docker are clear and concise, they're not enough for a new learner and are unnecessary for the initiated. In nearly every chapter, Hoffman and Nemeth refer the reader to online resources to learn more. That's not bad, but sometimes it feels like that's all there is. I think this would make a good blog series.

The writing does work in one significant way, and perhaps that was the intent. For some readers it may be enough, but I found myself at the end of each chapter looking for more. Where there were links and references, I would go back and follow them. Where there was an example or challenge, I would look and sometimes try it. I have the names of some new tools and technologies that I didn't have before, and I have a sense of just how much more there is to a cloud application than a database, a web server, and some JavaScript.

Reference

[1] A. Wiggins, "The Twelve-Factor App": https://12factor.net/.

Designing an Internet
David D. Clark
The MIT Press, 2018, 419 pages
ISBN: 978-0-262-03680-7

Reviewed by Rik Farrow

David Clark was the head of the Internet Activities Board from 1980 to 1990. He has also served on committees investigating other designs for the Internet as well as writing papers about features of alternate designs. As you might expect, Clark is certainly an expert, if not *the* expert, in this area.

Clark wrote his book partially for policy-makers, those who might need to make decisions about future Internet designs. For that reason, he starts out with Internet basics, explains what he means when he writes about architecture, design, and requirements. While I found Clark's writing clear, I have been studying TCP/IP since 1989, and wonder just how well the book would work for a lawyer or someone working on a politician's staff. That said, Clark does use startlingly clear analogies that should help a determined reader understand at least the basics of internetworking.

Clark has been preparing the ground for Chapter 7, where he discusses alternative internetwork architectures. These vary widely, with some designs being closer to sketches and others much more detailed. The only one I recognized was Name Data Networking (NDN), which Clark often used as an example of applying a label instead of a global address in packet headers. When considering designs, a lot of what Clark has written focuses on just two areas: the expressive power in headers and the per-hop behavior of routers.

Expressive power has to do with the expressiveness used as a means for determining per-hop behavior. PHB is what we expect routers to do, to either forward packets or drop packets in the current Internet, but in future designs could cause other actions, such as requesting information from one or more central servers about how to set up a flow. The expressive power of IPv4 addressing has changed over time, from the initial flat address space, to the classful, then classless inter-domain routing (CIDR) architecture we see today.

Clark also discusses security in Chapter 10, something I was certainly looking forward to reading. For the most part, Clark provides persuasive arguments about why security is really a function of the application layer, with a few exceptions, DDoS being one and inter-domain routing being another. BGP has long been a problem, as any router can send updates about routing, a technique that has been used by nation-states and some attackers to subvert "normal" routing behavior. Clark deftly explains that we could be signing updates today, but the problem is political/social, as in who gets to run the public key server. Clark's suggestion is that regions could start by running regional servers, so that they can at least govern routing updates in their own regions.

Clark covers naming, addresses, availability, economics, and management and control. Most of these areas were not part of the design of the Internet, and Clark discusses why these weren't considered important or relevant in the early days, while providing suggestions for the future.

Clark provides an extensive glossary, pages covering three-letter-acronyms, 15 pages of references, a large index, and a 20-page appendix. The appendix seemed more like a summary of the history of internetworking and is excellent reading by itself, as long as you are willing to use the glossary to find explanations of terms that seemed to me to be unique to this field, like PHB.

As someone who learned about TCP/IP in the field, as it were, I occasionally found myself wondering if Clark and I lived in different realities. For example, UDP does not appear in the glossary and is mentioned just once in the book. Clark writes that he expects that core routers, because they have a very restricted set of responsibilities, shouldn't be as easily exploitable as desktop systems. He uses the notion that there haven't been public announcements of these exploits as proof. Yet a search of Cisco CVEs turns up dozens of router exploits, with few allowing complete control of the router, and most able to cause the router to crash. I asked my security-geek friends, and they pointed out that other core routers, like Juniper, are based on FreeBSD while still others are based on Linux. Huawei is rumored to have copied Cisco IOS firmware, and so shares the same buggy code that Cisco has been building since the late '80s.

But I make those comments because I would be enjoying reading Clark's writing, subtle humor, and careful explanations, only to be jolted by rare assertions that seemed to come from another realm of being. Other than that, I highly recommend his book to those interested in how we wound up with the Internet architecture we did, but not the corporate/political side of things, why certain decisions were made, and about the many ways that people in the US and Europe have been working to create new designs for a future Internet. Clark is certainly a master in the realm of Internet design.

Timefulness
Marcia Bjornerud
Princeton University Press, 2018, 208 pages
ISBN 978-0-691-18120-2

Reviewed by Rik Farrow

A geology professor writes beautifully about her passion, instilling in the reader the immensity of time as seen in the geological records. Bjornerud explains how geologists began by mapping time based on the fossil record, then added depth and precision by learning how to use natural radioactivity. She covers the terrifically slow changes in earth's crust, the faster changes at the surface, the balance that has protected atmospheric components, and how these aspects have worked over the aeons.

Bjornerud concludes with a chapter clearly demolishing hopes of using the current notions of geoengineering to solve the climate crisis. Appendix II provides wonderful tables demonstrating the scales of various geologic changes, from mountain growth and erosion (50-100 million years) to cycles and reoccurance intervals (supercontinent cycle, 500 million years). An easy read, one not requiring any background in geology or science—just an interested reader seeking to expand herself.

Empress of Forever
Max Gladstone
Tor Books, 2019, 480 pages
ISBN 978-0-765-39581-8

Reviewed by Rik Farrow

I read a preview of this book, after being attracted by comparisons to Iain M. Banks and William Gibson. While I don't agree with that, that's largely because the writing style is different and the pace much quicker than either of those authors.

The heroine, Viv Liao, is a Silicon Valley entrepreneur on the run, having attracted the attention of an increasingly authoritarian government. She seeks revenge by installing her nearly completed work, a conscious, aware intelligence requiring a distributed system of a trillion cores to run. But something goes awry during the break-in and installation at a datacenter, throwing her into what appears to be the end of time.

The Empress of Forever rules over all, destroying worlds if they threaten to attract the "Bleed," a rapacious force magnetized by high-technological development. What made this book relevant for me was the insight into how an imagined innovator and controller of vast companies thinks and how this allowed Viv to succeed in an unfamiliar reality, assemble a team of disparate allies, and take the fight to the all-powerful Empress's Citadel. A fun read, fast-paced, when what you need is a break from reading papers.

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's quarterly magazine, featuring technical articles, tips and techniques, book reviews, and practical columns on such topics as security, site reliability engineering, Perl, and networks and operating systems

Access to *;login:* online from December 1997 to the current issue: www.usenix.org/publications/login/

Registration discounts on standard technical sessions registration fees for selected USENIX-sponsored and co-sponsored events

The right to vote for board of director candidates as well as other matters affecting the Association.

For more information regarding membership or benefits, please see www.usenix.org/membership/, or contact us via email (membership@usenix.org) or telephone (+1 510.528.8649).

USENIX Board of Directors

Communicate directly with the USENIX Board of Directors by writing to board@usenix.org.

Reminder of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held on the evening of Monday, July 8, in Renton, WA, during the week of the 2019 USENIX Annual Technical Conference.

NSDI '19 Grant Recipients

Towards Enabling All Feasible Latency–Cost Tradeoffs in Geo-Distributed Storage
Poster sessions like the one held at NSDI '19 offer an excellent opportunity for conference attendees to engage one-on-one in conversations about important leading-edge research.

SREcon19 Americas co-chairs Liz Fong-Jones and Mike Rembetsy deliver remarks during the opening session.

SREcon19 Americas Grant Recipients

The Tuesday evening reception at SREcon19 Americas, sponsored by Packet, featured a live ice sculpting performance.

SREcon19 Americas Lightning Talks presenters

LISA19

October 28–30, 2019
Portland, OR, USA

Sponsored by USENIX, the Advanced Computing Systems Association

LISA19 will take place on October 28–30, 2019, at the Marriott Downtown Waterfront in Portland, OR, USA.

Important Dates

- Submissions due: **Tuesday, June 18, 2019, 11:59 pm PDT**
- Notification to submitters: **Friday, July 12, 2019, 11:59 pm PDT**

Both presenters and organizers may withdraw or decline proposals for any reason, even after initial acceptance. **Submissions must come from the speaker**. LISA does not accept submissions made through PR firms and other third parties.

Overview

System engineering, administration, operations—whatever your organization calls it—has evolved beyond merely making systems run. Today's systems engineers design, develop, and run more complex, optimized, and distributed systems at all levels of scale. Come learn the skills and knowledge for pushing your systems to the edge at LISA19!

LISA, now in its 33rd year, wants to help you share your stories and experience. Whether you're an infrastructure engineer, system administrator, SRE, academic, or developer, we highlight the best in systems engineering and the people who make it all come together.

Call for Participation

LISA is soliciting proposals that demonstrate the present and future of operations in all of its forms. Submissions should inspire and motivate attendees toward action that improves their day-to-day work as well as the tech industry as a whole.

We welcome and encourage participation from all individuals, including people who are underrepresented in, or excluded from, technology: people of color, women, LGBTQ people, people with disabilities, students, veterans, and others with unique characteristics, whether or not they are protected by law. Similarly, we welcome participants from diverse professional roles: QA testers, security teams, DBAs, network administrators, compliance experts, UX designers, government employees, scientists. Regardless of who you are or the job title you hold, if you are a technologist who faces unique challenges, we encourage you to be a part of LISA19.

LISA only accepts vendor-neutral proposals. If you wish to promote or pitch a product at LISA, please email the USENIX Sponsorship Department at sponsorship@usenix.org about exhibition and sponsor opportunities.

Themes and Topics at LISA

Design
- Cloud Computing
- Containerization
- Machine Learning/AI
- Big Data
- Infrastructure/Platform/Function as a Service
- Scalability and Resilience
- Infrastructure Design
- Performance Planning
- Backup and Recovery
- Strategic Vision
- What's Next

Develop
- Automation
- Continuous Integration
- Continuous Delivery
- System Engineering
- Monitoring and Instrumentation
- Performance Tuning
- Debugging
- Security Engineering
- Databases
- Programming
- Release Engineering

Run
- DevOps
- Open Source Communities
- Communication
- Standards and Regulatory Compliance
- On-Call Strategies
- Managing IT Teams
- Diversity and Inclusion
- Mentorship, Education, and Training
- Recruiting and Retention

Proposals We Are Seeking

- **Talks:** 35-minute talks with 5 minutes for optional Q&A (40 minutes total). See the Call for Participation at www.usenix.org/lisa19/cfp for links to sample talk submissions.
- **Training:** 90-minute training courses teaching practical, immediately applicable skills. See the Call for Participation at www.usenix.org/lisa19/cfp for a link to a sample training submission.
- **Panels:** Moderator-led groups of 3–5 experts answering moderator and audience questions on a particular topic. When submitting a panel, please include the panel line-up ahead of time, or how you will select panelists. Panels must meet before the conference to make introductions and go over questions and flow.

Questions?

The chairs can help! Email lisa19chairs@usenix.org.

Code of Conduct

LISA is an inclusive and equitable space that welcomes the perspectives of everyone. Our Conference Code of Conduct and Event Guidelines for Speakers specify our commitment to providing a safe and enjoyable event experience for all event participants and a welcoming environment for free discussion of ideas. We do not tolerate harassment of event participants in any form.

Program Co-Chairs

Patrick Cable, Threat Stack, Inc.
Mike Rembetsy, Bloomberg

Submit your proposal today!
www.usenix.org/lisa19/cfp

ENIGMA

A USENIX CONFERENCE

Submit a Talk

Enigma centers on a single track of engaging talks covering a wide range of topics in security and privacy. Our goal is to clearly explain emerging threats and defenses in the growing intersection of society and technology, and to foster an intelligent and informed conversation within the community and the world. We view diversity as a key enabler for this goal and actively work to ensure that the Enigma community encourages and welcomes participation from all employment sectors, racial and ethnic backgrounds, nationalities, and genders.

Enigma is committed to fostering an open, collaborative, and respectful environment. Enigma and USENIX are also dedicated to open science and open conversations, and all talk media is available to the public after the conference.

PROGRAM CO-CHAIRS

Ben Adida
VotingWorks

Daniela Oliveira
University of Florida

The submission deadline is August 21, 2019.

www.usenix.org/enigma2020

JAN 27–29, 2020

SAN FRANCISCO, CA, USA

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

28TH USENIX SECURITY SYMPOSIUM

AUGUST 14–16, 2019 • SANTA CLARA, CA, USA

The USENIX Security Symposium brings together researchers, practitioners, system administrators, system programmers, and others interested in the latest advances in the security and privacy of computer systems and networks. The Symposium will span three days, with a technical program including refereed papers, invited talks, posters, panel discussions, and Birds-of-a-Feather sessions (BoFs).

Keynote Address
The Keynote Speaker will be Alex Stamos, *Adjunct Professor, Stanford University; William J. Perry Fellow, Center for International Security and Cooperation; Fellow, Hoover Institution*

The following co-located events will occur before the Symposium:

SOUPS 2019: Fifteenth Symposium on Usable Privacy and Security

PEPR '19: 2019 USENIX Conference on Privacy Engineering Practice and Respect

WOOT '19: 13th USENIX Workshop on Offensive Technologies

CSET '19: 12th USENIX Workshop on Cyber Security Experimentation and Test

ScAINet '19: 2019 USENIX Security and AI Networking Conference

FOCI '19: 9th USENIX Workshop on Free and Open Communications on the Internet

HotSec '19: 2019 USENIX Summit on Hot Topics in Security

The full program and registration are now available.
Register by July 22 and save!
www.usenix.org/sec19