;login:

WINTER 2020 VOL. 45, NO. 4

Characterizations of Cloud Functions Workloads
Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, and Ricardo Bianchini

RLBox: Simplifying In-Process Sandboxing
Tal Garfinkel, Shravan Narayan, Craig Disselkoen, Hovav Shacham, and Deian Stefan

BIBIFI Contests: Motivated Developers Still Make Security Mistakes
Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks

SRE Best Practices for Capacity Management
Luis Quesada Torres and Doug Colish

Columns

Review of Alex Hidalgo's Book about SLOs
Laura Nolan

Understanding Linux Containers
Corey Lueninghoener

BPF and Histograms
Dave Josephsen

Cryptographic Hash Functions
Simson L. Garfinkel

Software Supply Chain Security
Dan Geer, Bentz Tozer, and John Speed Meyers

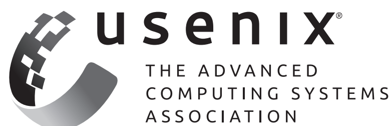
Final Print Issue Specials

Favorite Articles
Rik Farrow, Laura Nolan, and Arvind Krishnamurthy

Interview with USENIX Member #7
Rik Farrow

Interview with Kirk McKusick
Rik Farrow

Open Access
Laura Nolan



Thanks to our USENIX Supporters!

USENIX appreciates the financial assistance our Supporters provide to subsidize our day-to-day operations and to continue our non-profit mission. Our supporters help ensure:

- Free and open access to technical information
- Student Grants and Diversity Grants to participate in USENIX conferences
- The nexus between academic research and industry practice
- Diversity and representation in the technical workplace

We need you now more than ever! Contact us at sponsorship@usenix.org.

USENIX PATRONS

Bloomberg FACEBOOK Google

USENIX BENEFACTORS

ORACLE

USENIX PARTNERS

TOP10VPN

We offer our heartfelt appreciation to the following sponsors and champions of conference diversity, open access, and our SREcon communities via their sponsorship of multiple conferences:

Ethyca	Equinix Metal	Microsoft Azure
Goldman Sachs	LinkedIn	Salesforce

More information at www.usenix.org/supporters

;login:

WINTER 2020 VOL. 45, NO. 4

EDITORIAL

2 **Musings** *Rik Farrow*

OPINION

6 **Video Conferencing Must Evolve** *Michael Mattioli*

SECURITY

9 **Build It, Break It, Fix It Contests: Motivated Developers Still Make Security Mistakes** *Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks*

15 **The Road to Less Trusted Code: Lowering the Barrier to In-Process Sandboxing** *Tal Garfinkel, Shravan Narayan, Craig Disselkoen, Hovav Shacham, and Deian Stefan*

23 **Using Safety Properties to Generate Vulnerability Patches** *Zhen Huang, David Lie, Gang Tan, and Trent Jaeger*

29 **Interview with Sergey Bratus** *Rik Farrow*

SYSTEMS

35 **Characterization and Optimization of the Serverless Workload at a Large Cloud Provider** *Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, and Ricardo Bianchini*

40 **Posh: A Data-Aware Shell** *Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia*

46 **Interview with Margo Seltzer** *Rik Farrow*

SRE

49 **SRE Best Practices for Capacity Management** *Luis Quesada Torres and Doug Colish*

57 **The Case for CS Knowledge in SRE** *Adam McKaig*

COLUMNS

61 **Book Review: *Implementing Service Level Objectives* by Alex Hidalgo** *Laura Nolan*

64 **Systems Notebook: What's in That Container?** *Cory Lueninghoener*

68 **iVoyeur: BPF and Histograms** *Dave Josephsen*

71 **SIGINFO: The Tricky Cryptographic Hash Function** *Simson L. Garfinkel*

76 **Programming Workbench: Compressed Sparse Row Format for Representing Graphs** *Terence Kelly*

83 **For Good Measure—Counting Broken Links: A Quant's View of Software Supply Chain Security** *Dan Geer, Bentz Tozer, and John Speed Meyers*

87 **/dev/random: Discontent Creator** *Robert G. Ferrell*

BOOKS

89 **Book Reviews** *Mark Lamourine and Rik Farrow*

USENIX NOTES

95 ***;login:* Enters a New Phase of Its Evolution** *Cat Allman, Rik Farrow, Casey Henderson, Arvind Krishnamurthy, and Laura Nolan*

95 **Interview with Clem Cole** *Rik Farrow*

98 **Interview with Kirk McKusick** *Rik Farrow*

99 ***;login:* and Open Access** *Laura Nolan*

100 **Our Favorite *;login:* Articles, 2005-2019** *Rik Farrow, Laura Nolan, and Arvind Krishnamurthy*

EDITOR
Rik Farrow

MANAGING EDITOR
Michele Nelson

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Ann Heron
Jasmine Murcia
Olivia Vernetti

TYPESETTER
Linda Davis

USENIX ASSOCIATION
2560 Ninth Street, Suite 215
Berkeley, California 94710, USA
Phone: +1 510.528.8649
login@usenix.org

www.usenix.org

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

©2020 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

Cover Image by BUMIPUTRA, distributed by Pixabay.

EDITORIAL

Musings

RIK FARROW

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

I've sometimes been asked why computers are still so insecure, so eminently hackable. Didn't Bill Gates once shut down development at Microsoft so they could improve the security of Windows decades ago? While not quite decades ago, Gates really did shut down Windows development in 2002 and sent 7,000 systems programmers to special security training with the goal of "Trustworthy Computing."

It didn't work. While some things got better, and the rampage of worms slowed down, administrators and users continued to have to install patches frequently. In 2003, Patch Tuesday became a regular feature, followed by Exploit Wednesday for those who had ignored the routine of installing patches on the second Tuesday of the month.

Part of Microsoft's problem was a matter of programming culture, with a focus on new features. Exchange server, the email server product, actually had a good record for security, while the IIS web server certainly did not. Two distinct groups, with a different culture, worked on these products, resulting in very different security outcomes.

But the problem of insecurity is not unique to Microsoft. Sun Microsystems started delivering insecure workstations in the early 1980s, and continued to do so through the '90s. Sun employees announced at USENIX Security that they had a program for securing SunOS, but it was for internal use only. Dan Farmer, Brad Powell, and Matt Archibald released Titan for Solaris in 1998 as a public solution to tightening and securing Solaris. Linux was having severe issues with security at the end of the '90s but quickly improved over the next couple of years. But today, people build malware specifically for Linux, as Linux servers and desktops have become important targets for invading networks.

So far, all I've done is write about how the struggle to defend software against exploits has been a failure, but not why. The answer lies partially in the nature of software and largely because of our hardware designs.

First, programming is hard. I am constantly amazed at people announcing that they intend to turn everyone into a programmer. Perhaps these well-meaning projects can turn some people into middling programmers, but not ones who will be writing the next generation of services. I have had the misfortune of consulting in IT shops and have seen the carnage firsthand. On the plus side, when I turned out a handful of lines of shell script that did what they had failed to do in weeks, it made me look like a wizard. I have said this before: most programmers, by definition, have an average skill level, and half are below average. This is hard to remember when you work in Silicon Valley or at a top-ten university and all of your coworkers are geniuses.

Second, our computer systems were not designed for security. They were designed to be flexible. There are hardware security mechanisms that are important to security, such as the so-called rings, with the lowest numbered ring having the most access to hardware, and higher rings being reserved for "untrusted" code. Yet the largest and most complex programs run on most systems are the operating systems, and these run at the innermost ring. That makes the operating system the most important target for any attacker.

Microsoft has taken advantage of the ring added to support virtualization, called ADM-V or Intel VT, in Windows 10. They load kernel modules using Virtual Secure Mode, where the

operating system and critical system modules get executed in virtual containers. This beats the pants off the Linux model, where the kernel resides in a single address space, but still hasn't prevented bootkits from being installed in Windows 10 systems. This is supposed to be prevented by UEFI, but this can be worked around using firmware rootkits and on many motherboards because of the wrong settings being used.

Memory management is the next level of protection, but it was designed to protect programs running in one process from programs running in another process. Through abuse of the operating system, usually after an exploit, memory management can be bypassed.

Intel has introduced another level of protection, although this one is largely unused today. MPK (memory protection keys) allows programmers to split a single process's memory space into 16 different regions with the same protection provided by page tables [2]. Sixteen regions doesn't sound like a lot, but as a method for isolating threads, or portions of a program involved in parsing input, MPK could help.

The CHERI researchers have taken a slightly different tack by creating CPU designs with segment registers. MULTICS used segment registers to separate portions of programs, with a segment having a base address and a range, and accesses outside of this base and range being prohibited. CHERI represents another great idea, one that's been in development over a decade, making segments associated with capabilities, and one quite unlikely to be adopted by most programmers.

I guess I should mention enclaves, the tiny, encrypted execution domains, so I can also mention Meltdown, Spectre, and Load Value Injection [1]. Enclaves will not be of use to most programmers, and transient execution flaws have painted targets on them already.

Software

That leaves us with software. Software can either make computers more secure or less secure, and our favorite languages make our systems less secure.

```
.cfi_startproc
pushq   %rax
.cfi_def_cfa_offset 16
movslq  %edi, %rax
leaq    _ZN5hello4main17hd078db076938ab99E(%rip), %rdi
movq    %rsi, (%rsp)
movq    %rax, %rsi
movq    (%rsp), %rdx
callq   _ZN3std2rt10lang_start17he5a718dea3bb834eE
popq    %rcx
.cfi_def_cfa_offset 8
retq
```

Listing 1: Some assembler

Listing 1 depicts the `main()` function for a "Hello World!" program. Compilers produce assembler as an intermediate format, and that's what appears in Listing 1. You can learn to program in assembler, but you have to handle things that compilers make easy to do, like choosing the register to use (anything beginning with `%`), managing the stack, managing memory. Each CPU architecture has a different set of registers and assembly instructions, although assemblers themselves, like `as`, work the same. You still have comments, but assembler is hard to read and is not portable between CPU architectures.

That's why the geniuses who created UNIX created the C language: they needed a language that made porting an operating system easier. They also wanted something that would be fast and that provides little in the way of handholding. If you don't know better, you can easily make "fatal" mistakes, like using a pointer after the memory it points to has been freed or writing into memory beyond the end of an array. On the other hand, you can treat pointers as function entry points and perform arithmetic on pointers, very handy things to have when writing an operating system—especially one that runs on hardware with 32K words of RAM.

C is my favorite language, but it is a language without seatbelts, airbags, or even bumpers. C, and its younger cousin C++, assume that you know what you are doing and you never make a mistake. The first of these points is rarely true, and the second is never true—even the best programmers make mistakes.

There are safer languages to use, ones with safety features. Generally, these languages remove access to pointers and provide strong types. Go and Rust are examples of safer languages, with Rust being designed particularly for safety. Go is not as fast as C or C++, but perhaps a 10–15% penalty for a lot of execution safety is worthwhile. Rust, meanwhile is nearly as fast as C, and perhaps will be when LLVM can produce code as performant as GCC.

Safer languages leverage hardware support for security by making it much more difficult to write programs that are terribly insecure. I think this is a very good idea, especially if we are going to teach everyone to program.

The Lineup

We start out this issue with an opinion piece by Michael Mattioli, who feels that tools like Zoom, Meet, Teams, and so on are missing something important.

Next, I picked two papers from USENIX Security '20 that were clearly written and included points that I felt were especially worth sharing. There were another half-dozen papers that I really liked, but those either weren't as well written, had deep dives into statistics, or were too narrow for the wide audience represented by the USENIX membership.

Votipka et al. examine programmer mistakes, but not just any type of errors. They used the Build It, Break It, Fix It (BIBIFI) program as their data source. BIBIFI challenges programmers who have had training or work experience to write three, non-trivial programs with some security requirements, share the sources to these programs with other teams, and then analyze the programs and the faults found by the teams. What they found was distressing to me and is part of the reason why security is so hard to get right.

Garfinkel et al. have written a tool, RLBox, that makes sandboxing libraries easier. Most programs incorporate libraries, and many of these libraries process input that may come from attackers, such as image or video decoders. RLBox simplifies the process of sandboxing these libraries. The authors worked with Mozilla to sandbox several key libraries, and their tool will work for other programs as well.

Huang et al. volunteered to write about their research project, Senx, an automatic program repair tool. The authors argue that waiting for security patches to appear often takes much too long, and with access to source code and an example of an attack, Senx can create patches for three different types of vulnerabilities.

I interviewed Sergey Bratus. There were several papers at USENIX Security '20 that appeared to be directly related to Language Security principles, or LangSec. Bratus has written for *;login:* before, has been running a workshop on LangSec for years, and seemed to me to be the perfect person to explain LangSec principles. And this worked, as LangSec seems much clearer to me now and is important if we are ever going to be able to write secure software.

The USENIX Annual Technical Conference also happened this summer, and I chose two papers and one talk as the basis for articles. Shahrad et al. explain a key feature of running cloud functions: deciding how long a function should be kept warm, that is, ready to run. They provide examples taken from Azure and a new scheme that improves performance and efficiency.

Raghavan et al. discuss Posh, a distributed shell. To me, Posh is a great example in the tradition of USENIX ATC, an improvement on the shell that works by moving execution closer to the sources of data, when that data is available over NFS. Posh can also add parallelism to shell scripts without rewriting the scripts.

I interviewed Margo Seltzer, who gave an afternoon keynote at USENIX ATC '20. Seltzer encouraged her audience to explore beyond the safe confines of their personal specialties and consider "fringe" ideas. Seltzer provides several examples of doing this in her own highly successful career.

Torres and Colish cover capacity planning for SREs. They divide capacity planning into two areas: resource provisioning and capacity planning to safeguard the future potential of a service. The authors cover redundancy for reliability and how this must include back-end services as well.

Adam McKaig explains why he thinks that it's important for SREs to understand algorithms and data structures. McKaig takes us through three examples of a service that initially is performing well, uncovering the reasons why the service starts failing SLOs, and explaining the solutions that he and the teams he worked with came up with for repairing the service.

Laura Nolan has written a book review of Alex Hidalgo's recently published book about SLOs. Nolan explains why she considers Hidalgo's book one of the most important for SREs. Hidalgo wrote an article for *;login:* in the Summer 2020 issue, so you can also sample his writing there.

Cory Lueninghoener shows us how to create different aspects of containers from the command line. While you may be more likely to use a tool like Docker for this, you will gain understanding of what Docker is doing by trying Lueninghoener's examples.

Dave Josephsen continues his exploration of eBPF, this time focusing on histograms as a clever technique for displaying potential performance issues. Josephsen dives into how to select bin sizes for histograms and exactly why histograms are so good at unveiling problems that would be buried in data otherwise.

Simson Garfinkel covers the history and uses of cryptographic hashes. While the use of hashes has become commonplace in programming, cryptographic hashes provide the foundation for assuring the authenticity of code or messages, timestamps for documents, and in forensics.

Terence Kelly demonstrates a technique for storing graphs as compressed sparse row format. First, Kelly shows the most commonly used ways of storing graphs, explains why these methods waste memory, and then details how to use the compressed sparse row format and when other formats will work better.

Dan Geer, along with coworkers John Speed Meyers and Bentz Tozer, has researched software supply chain insecurity. They have collected data about how often attackers have modified the source code for open source libraries as well as how often this has resulted in successful attacks, work that I believe is really important so long as we continue to include other people's code, via libraries, in our own programs.

Robert Ferrell distracts us with his views on social media *influencers*. Ferrell deletes himself from this clan, while pondering on the usefulness of content that is itself nothing more than advertising.

Mark Lamourine has reviewed three books this time, *Effective Python*, *Dependency Injections, Practices, and Patterns,* and *Building Secure and Reliable Systems*. I reviewed a book about rootkits for Windows.

Most of us have little to no influence on hardware design. To be honest, most of us won't have the type of ideas necessary to even get the CPU industry to move at all toward better security. Personally, I'd like to see designs that support message passing without involving context switches, as that would allow our servers to appear more like clouds than 1970s mainframes.

We do have choices we can make about the programming languages we use. Well, some of us do, while those working at corporations often have that decision made by someone far off in the top of the management hierarchy, based on the latest buzz. For those who have choices, I recommend languages like Rust that emphasize both security and performance. For a different way of looking at things, I found this article at Northeastern an interesting way to view programming languages [3]. Hopefully, someone will keep this page up-to-date so we don't have to rely on sites like TIOBE.

And as for making systems more secure, we do need to stop handing out assault rifles like C++ and get more people to use inherently safer programming languages like Rust or Go. Python has its faults, like the lack of strong types and being single threaded like JavaScript, but it doesn't have pointers and the types of memory issues that C and C++ have had for decades. Decisions at institutions of higher education do have an influence over the future security, or insecurity, of computers.

Remember Listing 1, the assembly language example? That was `helloworld.rs`, the Rust version, but you can hardly tell by looking at the intermediate assembly code. All programming languages wind up as machine code, and while that may sound like all languages are equal, they are not. Some languages take advantage of advances in compiler designs so they make it much easier to write secure code. You can choose the 1970s model with some upgrades, or learn something new that can help make the world a safer place.

The Future of *;login:*

This issue marks the end of print *;login:*. You can read about why this is happening and learn more about how the digital version of *;login:* will work in *USENIX Notes*, beginning on page 95.

Some people have found the reference to "peer-reviewed" in this description a bit confusing, thinking that the digital version of *;login:* will be like a journal. That's not true. The peer-review has long been a part of editing *;login:,* and consisted of PC members who accepted the papers that many articles are based upon. For the rest, I was the "peer," with responsibility for accepting articles only from subject matter experts. I did rely on other experts in areas where I was unfamiliar with the authors. The digital version will expand the number of peers, so I will no longer be responsible for culling out articles that should not be published in *;login:*.

Another advantage of a digital *;login:* will be shorter elapsed time between submitting an article and its appearance online. Printing *;login*: takes a long time—just dealing with the printing process itself took almost three weeks. While I might see a draft, get a final version, format it and turn it in in just one week, the process that includes copy editing and typesetting takes a great deal longer. Michele Nelson, the Managing Editor, received articles from me and sheparded them through this long process.

I think we will miss our copy editors, Steve Gilmartin and Amber Ankerholz. Good copy editors improve your writing, often taking something not written that well and turning it into something that makes you start believing you really can write well. The copy editor must improve your written English without distorting your meaning, and Steve did a great job. Amber's task was to approve Steve's edits from a technical standpoint. That process, and proofreading, added three weeks to the process. Typesetting, expertly done by Linda Davis, added yet another week. When you add all of this up, and start from the point when I ask authors to write or get a proposal, the process can take over four months. I don't even want to think about how long your *;login:* magazine sat in a pile before you started reading it....

The digital *;login:* will be open access. Laura Nolan has written about the value of open access in this issue. All articles will be open access when posted, as opposed to members-only for one year. Only USENIX members will be able to comment on articles, something we hope will lead to discussion about articles and feedback to authors. With print *;login:*, about the only time authors get feedback is during in-person conferences, and from personal experience I can tell you that even that is rare. I hope the ability to respond to articles results in useful feedback, or at least acknowledgement that someone has read and appreciates the work someone put into an article.

We—that is the committee composed of three board members, Laura Nolan, Arvind Krishnamurthy, and Cat Allman, along with Casey Henderson—came up with several other ideas to celebrate this, the final print issue. I was assigned to interview two early USENIX members. Clem Cole has the honor of being USENIX member number seven, and I interviewed him first. Kirk McKusick represents, at least for me, the Berkeley side of UNIX and makes up the second interview. They both participated in the story of how USENIX helped Rick Adams start UUNET in the late 80s, as did Deborah Scherrer (the Board VP), Steve Johnson (Treasurer), and Rick Adams. Adams recommended reading Peter Salus' (Executive Director) article [4]. Adams also deserves thanks for donating UUNET stock from his fledgling company that later became the foundation for the endowment that is keeping USENIX alive during COVID-19.

Finally, Laura Nolan, Arvind Krishnamurthy and I picked out our favorite articles from *;login:* issues starting with 2005. I learned that my ability to edit *;login:* has improved over the years. I had started to edit special security-focused issues of *;login:* in 1998, but to my eyes, the first five years of being the regular editor, starting in 2005, seem pretty rough.

You might be wondering what I plan to do with all the time I will have because I will be sharing the editorial responsiblities. I plan on writing some science fiction, and hope to have at least one short story up at https:/rikfarrow.com/fiction/ by the time this issue appears. I've started at least five stories, and have one close to completion—about computers and future myths, of course.

References

[1] D. Goodin, "Intel SGX Is Vulnerable to an Unfixable Flaw That Can Steal Crypto Keys and More," Ars Technica, March 10, 2020: https://arstechnica.com/information-technology/2020/03/hackers-can-steal-secret-data-stored-in-intels-sgx-secure-enclave/.

[2] Memory Protection Keys: https://www.kernel.org/doc/html/latest/core-api/protection-keys.html.

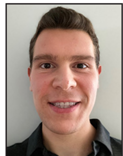
[3] B. Eastwood, "The 10 Most Popular Programming Languages to Learn in 2020," Northeastern University Graduate Programs, June 18, 2020: https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/.

[4] P. Salus, "Distributing the News: UUCP to UUNET," *;login:*, vol. 40, no. 4 (August 2015): https://www.usenix.org/system/files/login/articles/login_aug15_09_salus.pdf.

OPINION

Video Conferencing Must Evolve

MICHAEL MATTIOLI



Michael leads the Hardware Engineering team within Goldman Sachs. He is responsible for the design and engineering of the firm's digital experiences and technologies. He is also responsible for the overall strategy and execution of hardware innovation both within the firm and within the broader technology industry. Michael.Mattioli@gs.com

The 2020 COVID-19 pandemic has forced hundreds of millions of people to use video conferencing tools to continue learning, educating, conducting business transactions and providing health care. These tools are poor conduits for true human-to-human communication. Posture, tone of voice, use of physical space, facial expressions, gestures, and more are all lost when communicating through these tools as we know them today. To replicate the in-person experiences we've all come to know, these tools need to evolve from what is possible today with simple sound and some pixels.

Zoom, Meet, Teams, Skype, Webex, FaceTime—take your pick. Whichever you choose, they all boil down to nothing more than one or two audio channels and a few hundred thousand pixels. That's all you get. Since March of 2020, the COVID-19 pandemic has forced hundreds of millions of people to communicate with each other using myriad video conferencing tools. Human-to-human communication is much more than just hearing what we say and seeing some pixels arranged to represent faces. These tools were designed in a world much different from the one we live in now. Don't get me wrong; they've been crucial in the continuity of health care, education, and business over the last few months. I am not trying to diminish or downplay their importance; quite the opposite. Because they are so important, they need to evolve in order to become more effective vehicles for human-to-human communication.

Health Care

There's a reason why bedside manner is so strongly emphasized in training health care professionals. Intimate conversations with patients transpire: conversations about their health, well-being—*their literal lives*. These conversations require connection and engagement. Patients need to establish trust with their physician or nurse [6]. They want to be certain that

they are receiving nothing less than the absolute best care to be offered. Simple things like sitting down or standing up to have these conversations can make a world of difference [8].

When explaining a course of treatment, the tone in a physician's voice can convey confidence or doubt. A patient's nonverbal reactions (e.g., facial expressions, posture) to information give the physician more clarity on how the information is received than the patient's verbal response. These aren't things that can be effectively conveyed with the digital tools we have today.

Education

Students, instructors, and parents all agree that remote learning is nowhere near as engaging as in-person instruction. Instructors cannot accurately determine how their material is being received. Do the students react by sitting up or slouching? Are they responding confidently or insecurely in their answers? Are they even paying attention? A Dallas middle-school Spanish teacher struggles to find ways to read her students' body language: "In the classroom, I can look around and see body language and know when some of my students not fluent in Spanish need me to switch to English. I can't do that online. We need the interaction with the kids, face-to-face" [3]. Students face similar challenges with parents claiming that remote instruction "lacks substance" and with some parents even considering having their children repeat this past year's coursework [4]. Even video games are more engaging; the physical feedback (haptics) and input (buttons, joysticks, etc.) using a controller provide an entirely different sensory experience. Video games stimulate three of the five human senses (sight, sound, and touch), whereas modern video conferencing tools only provide interfaces for two (sight and sound).

Businesses

Know Who You're Dealing With

It's commonplace for high-stakes business interactions to take place in-person. In a study by Great Business Schools, 82% of people say that in-person meetings are essential for important contracts [7]. Any decision involving large sums of money, contracts, careers, and anything in between can be devastating if made incorrectly. Negotiations typically take place in-person so that each party has an opportunity to better understand the other prior to transacting. Use of body language and the physical space in the meeting place are key [5]. Lack of eye contact could suggest deceit. Folding one's arms could suggest defensiveness. Short and curt answers could be indicative of disinterest. Sitting down suggests one is confident and relaxed, whereas pacing around the room suggests one is anxious. When deciding whether or not to close on a new home, extend an employment offer, or enter into a contract, it's fair to want as much information as possible to truly assess the situation before making a final, binding decision.

Build Relationships

Arguably the most important component of a business transaction is not just the transaction itself but the long-term relationships that are formed between parties. The same study by Great Business Schools also reported that 85% of people found that in-person meetings built strong, more meaningful business relationships. Meaningful relationships require connection and engagement on a human level—a difficult task if relying solely on an audio device's representation of someone's voice and a display's representation of someone's face. Also consider the level of effort it takes to communicate with someone via video conferencing (rather low) as opposed to an in-person meeting (potentially rather high depending on various factors). "Going the extra mile" (quite literally, in some cases) helps establish a foundation of trust between parties and suggests that the relationship is of high importance [1].

Conclusion

We've all been making do with what we have in this time of crisis. The current situation is far from ideal. We've been using tools that were designed as a convenience or a luxury, but it's clear now that they need to be classified as a necessity.

What separates video conferencing from a phone call? A few hundred thousand pixels and, if you're lucky, slightly higher fidelity audio—nothing more. Video conferencing excels in situations where the human aspect of communication is not critical, such as brief conversations and informal discussions. These tools need to make the generational leap that provides more natural human-to-human communication.

Of course, the concept of going to a physical location to learn, conduct business transactions, or consult with a physician may already seem archaic. Just ask the hundreds of millions of people who used to cram themselves into a bus or a train (or a combination of the two) for hours each day. In a study performed by Morning Consult, 32% of adults in the United States would prefer to never commute again and work remotely every day, and only 24% would want to continue to commute every day [2].

It's time to move on, but the tools we have are holding us back. The tone or volume of someone's voice, whether or not they make eye contact, use facial expressions, posture, etc. all need to be accurately conveyed through digital means in order to reproduce the in-person experiences we've all come to know. What is the underlying technology that will help us get there? How does this new level of communication affect how we will approach privacy and security? These are some of the questions we, as engineers, need to ask ourselves. No one has all the answers right now. Who knows—years from now, when we get there, we may not even refer to it as "video conferencing" anymore.

References

[1] Ashton College, "The Importance of Face-to-Face Communication": https://www.ashtoncollege.ca/the-importance-of-face-to-face-communication/.

[2] Morning Consult, "The Future of Work—How the Pandemic Has Altered Expectations of Remote Work," June 2020: https://go.morningconsult.com/rs/850-TAA-511/images/Remote%20Work%20Report%20-%20Morning%20Consult%20-%20Final.pdf.

[3] T. D. Hobbs, and L. Hawkins, "The Results Are in for Remote Learning: It Didn't Work," *Wall Street Journal*, June 5, 2020: https://www.wsj.com/articles/schools-coronavirus-remote-learning-lockdown-tech-11591375078.

[4] L. Brody, "Struggling with Remote Learning, Some Families Cut Class," *Wall Street Journal*, May 1, 2020: https://www.wsj.com/articles/struggling-with-remote-learning-some-families-cut-class-11588334403.

[5] Benchmark Meetings, "5 Reasons Virtual Meetings Can't Replace Face-to-Face Meetings," April 2019: https://www.benchmarkmeetings.com/face-to-face-meeting-advantages/.

[6] UCLA David Geffen School of Medicine, "The Importance of Bedside Manner to Trust and Patient Engagement," July 2016: https://medschool.ucla.edu/body.cfm?id=1158&action=detail&ref=699.

[7] Great Business Schools, "Face Squared—The Numbers Behind Face to Face Networking," January 2014: https://www.greatbusinessschools.org/networking/.

[8] St. George's University, "Developing Good Bedside Manner: 9 Tips for Doctors," January 2019: https://www.sgu.edu/blog/medical/how-to-develop-good-bedside-manner/.

FAST '21

19th USENIX Conference on File and Storage Technologies

FEBRUARY 23–25, 2021 | VIRTUAL EVENT

The 19th USENIX Conference on File and Storage Technologies (FAST '21) brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems.

PROGRAM CO-CHAIRS

Marcos K. Aguilera
VMware Research

Gala Yadgar
Technion—Israel Institute of Technology

www.usenix.org/fast21

SECURITY

Build It, Break It, Fix It Contests

Motivated Developers Still Make Security Mistakes

DANIEL VOTIPKA, KELSEY R. FULTON, JAMES PARKER, MATTHEW HOU, MICHELLE L. MAZUREK, AND MICHAEL HICKS

Daniel Votipka is a computer science PhD candidate at the University of Maryland. His research focuses on information security, with an emphasis on the human factors affecting security professionals. His most recent work focuses on understanding the processes and mental models of software vulnerability discovery to provide research-based improvements for education and automation to help develop and leverage human expertise. dvotipka@cs.umd.edu

Kelsey Fulton is a computer science PhD student at the University of Maryland. Her research explores the human factors of information security, with a focus on software developers and security professionals. Her most recent work centers on the barriers to adoption of secure programming languages in order to provide an empirical foundation for the future design of secure languages, APIs, and tools. kfulton@cs.umd.edu

James Parker is a Software Research Engineer at Galois. James earned his PhD in 2020 and was advised by Michael Hicks. His research spans verifying information flow control mechanisms, guaranteeing correctness of distributed systems, and studying secure development practices. jprider@cs.umd.edu

Secure software development is a challenging task requiring consideration of many possible threats and mitigations. We reviewed code submitted by 94 teams in a secure-programming contest designed to mimic real-world constraints—correctness, performance, and security. We found that the competitors, many of whom were experienced programmers and had just completed a 24-week cybersecurity course sequence with specific instruction on secure coding and cryptography, still introduced several vulnerabilities (182 across all teams), mostly due to misunderstandings of security concepts. We explain our methodology, discuss trends in the types of vulnerabilities introduced, and offer suggestions for avoiding the kinds of problems we encountered.

Developing secure software remains challenging, as evidenced by the numerous vulnerabilities still regularly discovered in production code [6]. There are many approaches that could be—and often have been—taken to improve this situation: building and deploying more automated tools for vulnerability discovery, expanding security education, or improving secure development processes.

But which of these interventions should we prioritize? While all are potentially helpful, we must carefully consider which provide the best return on investment, maximizing security while minimizing time, effort, and other resources, all of which are in short supply as developers are pressured to produce more new services and features.

A key part of this consideration is to understand the kinds and frequency of vulnerabilities that occur, and why developers introduce them, so that the root causes can be addressed. To this end, we performed a systematic, in-depth examination using best practices developed for qualitative assessments of vulnerabilities present in 94 project submissions by teams made up mostly of experienced programmers—many of whom had just completed a four-course program on secure development—to the Build It, Break It, Fix It (BIBIFI) secure-coding competition series [8, 10]. Our six-month examination considered each project's code and 866 total exploit submissions, corresponding to 182 unique security vulnerabilities associated with those projects.

Our findings suggest rethinking strategies to prevent and detect vulnerabilities, with more emphasis on conceptual difficulties rather than mistakes. This article provides an overview of our work. A more in-depth discussion of the methods followed, survey of related literature, and description of results can be found in our recent USENIX Security paper [10].

Build It, Break It, Fix It: A Happy Medium to Study

Our work to examine vulnerabilities introduced by software developers complements many prior efforts. Some researchers have performed large-scale analyses of open-source code and CVE reports, categorizing vulnerabilities found in production code [2, 3]; others have explored specific possible sources of error using controlled experiments with small, security-focused tasks [1, 7]. These field measures and lab studies represent two ends of a methodological spectrum. Field measures provide strong ecological validity, reflecting real-world

Matthew Hou is a first year computer science graduate student at the University of Maryland and is expecting to complete his master's degree next May. He recently graduated with honors from the University of Maryland with a BSc in computer science. His focus is on machine learning and artificial intelligence, leveraging cybersecurity principles. mhou1@cs.umd.edu

Michelle L. Mazurek is an Associate Professor in the Computer Science Department at the University of Maryland. Her research explores human aspects of information security and privacy, with a recent focus on improving security tools and processes for professionals, including software developers, network administrators, and reverse engineers. She also investigates how and why end users learn and adopt security and privacy behaviors, and she develops tools to increase transparency in online tracking and inferencing. mmazurek@cs.umd.edu

Michael Hicks is a Professor in the Computer Science Department at the University of Maryland. His research explores ways to make software more secure. He has a particular interest in securing low-level systems software, with a nearly 20-year stretch of work that started with the Cyclone safe C-like programming language (a significant influence on today's Rust programming language) and now involves contributions to Checked C, a safe-C extension based on clang/LLVM. He is also exploring synergies between cryptography and programming languages; techniques for better random (fuzz) testing and probabilistic reasoning; and high-assurance tools and languages for quantum computing. He blogs at https://www.pl-enthusiast.net/. mwh@cs.umd.edu

contexts, but provide no control over conditions like developer motivation and functionality being implemented that can affect results. In contrast, lab studies provide high levels of control but only limited ecological validity.

We attempt to balance ecological validity and experimental control by studying vulnerabilities in the context of BIBIFI competition projects. A BIBIFI competition has three phases. In the *build it* phase, teams are given just under two weeks to build a project that (securely) meets a given specification. Team scores depend on the project's correctness and efficiency, based on provided test cases. Submitted projects may be written in any programming language and can use any open-source libraries, as long as they can be built on a standard Ubuntu Linux VM. In the *break it* phase, teams receive access to their competitors' source code in order to search for vulnerabilities. Teams can submit test cases, known as *breaks*, to demonstrate exploitation. Successful breaks add to the exploiting team's break-it score, while reducing the victim's build-it score. The final *fix-it* phase allows teams to fix identified vulnerabilities in order to gain back a portion of their lost build-it points.

BIBIFI data therefore strikes a unique balance between ecological validity and control. Many implementations of the same functionality, created under similar circumstances, provide more confidence than field data does to help us understand what happened and why. On the other hand, teams had weeks (rather than hours) to develop their projects, could use their choice of languages and libraries, and were incentivized to consider constraints like performance and functionality as well as security, creating more ecological validity than many lab studies. While we know BIBIFI does not provide a perfect view into the development process (see our original paper [10] for a detailed discussion of limitations), it provides a new and valuable vantage point for examining the vulnerability landscape and informing future work.

The Competition's Projects

We analyzed projects from four BIBIFI competitions, covering three different programming problems: *secure log*, *secure communication*, and *multiuser database*. Each problem specification required the teams to consider different security challenges and attacker models.

Secure log (SL). This problem requires teams to implement two programs: one to securely append records to a log, and one to query the log's contents. Teams must protect against a malicious adversary with access to the log and the ability to modify it. The adversary does not have access to the keys used to create the log. Teams are expected (but not told explicitly) to utilize cryptographic functions to encrypt the log and protect its integrity.

Secure communication (SC). This problem requires teams to create client/server programs representing a bank and an ATM. The ATM initiates transactions, including account creation, deposits, and withdrawals.

Teams must protect bank data integrity and confidentiality against an adversary acting as a man-in-the-middle (MITM), with the ability to read and manipulate communications between the client and server. Once again, build teams were expected to use cryptographic functions and to consider challenges such as replay attacks and side-channels.

Multiuser database (MD). This problem requires teams to create a server that maintains a secure key-value store. Clients submit scripts written in a domain-specific language. A script authenticates with the server and then submits a series of commands to read and write data stored there. Data is protected by role-based access control policies customizable by the data owner, who may (transitively) delegate access control decisions to other principals.

The problem assumes that an attacker can submit commands to the server but not snoop on communications.

Vulnerabilities: Type and Prevalence

We manually analyzed 94 (out of 142) BIBIFI projects and 866 exploit submissions against them, ultimately identifying 182 unique vulnerabilities (some of which had not been identified during the contests). We grouped these vulnerabilities according to three main types: *no implementation*, *misunderstanding*, and *mistake*. Table 1 shows how many vulnerabilities, from how many projects, we identified for each type. This section describes each type, with examples.

No Implementation

The first step in building a secure system is to *attempt* to implement necessary security mechanisms. Unfortunately, half of all teams introduced a *no implementation* vulnerability, failing in this first step for at least one required security mechanism. This is presumably because they did not realize the security mechanism was needed. We further divided *no implementation* vulnerabilities based on how *obvious* the need was, depending on whether it was directly mentioned in the problem specification or just implied. For example, in the secure log problem, where teams were asked to ensure an attacker with read/write file access could not read or make changes to a confidential log, we considered it obvious that encryption was needed to provide confidentiality, but unintuitive that a Message Authentication Code (MAC) should be used as an integrity check.

Unintuitive security requirements are commonly skipped. Of the *no implementation* vulnerabilities, we found that teams were much more likely to skip *unintuitive* security requirements (45% of projects) than their intuitive counterparts (16% of projects). This indicates that developers do attempt to provide security—at least when incentivized to do so—but struggle to consider all the unintuitive ways an adversary could attack a system. Therefore, they regularly leave out some necessary controls.

Misunderstandings

After realizing a security mechanism should be implemented, teams then needed to make sure they implemented it correctly. We found that most teams failed at this point in the secure development process, most commonly due to a conceptual misunderstanding (56% of projects). We sub-typed these as either *bad choice* or *conceptual error*.

A *bad choice* occurs when a team decides to use a known-insecure algorithm or library—likely because they did not realize its inherent flaw (12% of vulnerabilities). In another secure log problem example, one team realized they needed to encrypt their log, but chose to simply XOR key-length chunks of the log with the user-provided key to generate the final encrypted version of the log. This method of encryption is inherently insecure, as the attacker can simply extract two key-length chunks of the ciphertext, XOR them together, and produce the key, allowing them to decrypt the entire log easily.

Assuming a team did choose a secure algorithm or library, next they had to know how to use it properly. We observed several cases where teams introduced vulnerabilities by not using the algorithm or library as intended, owing to a conceptual misunderstanding (27% of vulnerabilities). We classified these as *conceptual error* vulnerabilities. For example, one team made the reasonable choice to use AES encryption but used a fixed value for its initialization vector (IV); see code in Listing 1. A fixed IV, rather than a random one, allows an attacker to break the encryption and read the secret log.

```
1  def fillercrypter (sharedkey, text):
2      ...
3      encryption_suite = AES.new (sharedkey,
4       AES.MODE_CBC, 'This is an IV456')
5.     ...
```

Listing 1: One team generated a *conceptual error* vulnerability by using a hardcoded IV.

Type	Sub-Type	Projects (94)	Vulnerabilities (182)
No implementation	Intuitive Unintuitive	15 (16%) 42 (45%)	23 (13%) 49 (27%)
	Total	47 (50%)	72 (40%)
Misunderstanding	Bad choice Conceptual error	20 (21%) 41 (44%)	22 (12%) 49 (27%)
	Total	53 (56%)	71 (39%)
Mistake	—	20 (21%)	39 (21%)

Table 1: Number of vulnerabilities for each type and the number of projects each vulnerability was introduced in. Note, because projects can have multiple vulnerabilities, the total number of projects introducing a vulnerability for each type may not be the sum of sub-type project counts.

```
1  self.db = self.sql.connect(filename, timeout=30)
2  self.db.execute('pragma key="' + token + '";')
3  self.db.execute('PRAGMA kdf_iter='
4    + str(Utils.KDF_ITER) + ';')
5  self.db.execute('PRAGMA cipher_use_MAC=OFF;')
6  ...
```

Listing 2: Another team disabled the automatic MAC in SQLCipher library.

In another interesting example, one team simply disabled protections provided transparently by their chosen library. They initially made a secure choice by using the SQLCipher library, which provides encryption and integrity checks in the background without developer effort, but then explicitly disabled the library's MAC protection; see line 5 in Listing 2.

Teams often used the right security primitives but did not know how to use them correctly. Among the *misunderstanding* vulnerabilities, we found that *conceptual error* vulnerabilities (44% of projects) were significantly more likely to occur than *bad choice* vulnerabilities (21% of projects). This indicates that if developers know what security controls to implement, they are often able to identify (or are guided to) the correct primitives to use. However, they do not always conform to the assumptions of "normal use" made by library developers.

Mistakes

Finally, some teams chose the correct algorithm or library, and appeared to understand how to correctly use it, but made a simple mistake that led to a vulnerability (21% of vulnerabilities). For example, some teams did not properly handle errors, leaving the program in an observably bad state. Other mistakes led to logically incorrect execution behaviors. Such mistakes were often related to control flow logic or missed steps in an algorithm. For example, if a team correctly encrypted their log, but accidentally wrote the plaintext log to file instead of the ciphertext, this would be a *mistake*.

Complexity breeds mistakes. We found that the frequency of *mistakes* was affected by complexity, within both the problem itself and also the approach taken by the team. First, we found that teams were 6.68× more likely to introduce *mistakes* in the *multiuser database* than in the *secure communication* problem. This likely reflects the fact that the multiuser database problem was the most complex, requiring teams to write a command parser, handle network communication, and implement nine different access control checks. Similarly, teams were only 0.06× as likely to make a mistake in the comparatively simple *secure log* problem compared to the *secure communication* problem.

Additionally, choosing not to reimplement security-relevant code multiple times was associated with only 0.36× as many *mistakes*, suggesting that violating the "Economy of Mechanism" principle [9] by adding unnecessary complexity leads to *mistakes*.

As an example of this effect, one team implemented their access control checks four times throughout the project. Unfortunately, when they realized the implementation was incorrect, they only updated it in one place.

Exploit Difficulty

In addition to examining vulnerability types and their frequency, we also assessed how *difficult* it would be for an attacker to find and exploit the vulnerability. Even if a vulnerability was quite common, if it was very difficult to identify, requiring esoteric knowledge or practically impossible to exploit, its resolution might be lower priority than a less common but more exploitable vulnerability.

We considered three metrics of difficulty: our qualitative assessment of the difficulty of finding the vulnerability (*discovery difficulty*); our qualitative assessment of the difficulty of exploiting the vulnerability (*exploit difficulty*); and whether a competitor team actually found and exploited the vulnerability (*actual exploitation*). For convenience of analysis, we binned *discovery difficulty* into *easy* (execution) and *hard* (source, deep insight). We similarly binned *exploit difficulty* into *easy* (single-step, few steps) and *hard* (many steps, deterministic or probabilistic). Figure 1 shows the number of vulnerabilities for each type with each bar divided by *exploit difficulty* and bars grouped by *discovery difficulty*.

***Misunderstandings* are rated as hard to find, while *no implementations* are rated as easy to find**. Identifying *misunderstanding* vulnerabilities often required the attacker to determine the developer's exact approach and have a good understanding of the algorithms, data structures, or libraries they used. As such, we rated *misunderstanding* vulnerabilities as hard to find significantly more often than other vulnerability types.

Unsurprisingly, a majority of *no implementation* vulnerabilities were considered easy to find. For example, in the secure log problem, an auditor could simply check whether encryption and an integrity check were used. If not, then the project can be exploited.

Easy to find doesn't mean easy to exploit. Interestingly, we did not observe a significant difference in actual exploitation between *misunderstandings* and *no implementations*. Some *misunderstandings* were rated as difficult to find, while others were rated as difficult to exploit. In one team's use of homemade encryption, the vulnerability took some time to find, because the implementation code was difficult to read. However, once an attacker realized the team had essentially reimplemented the Wired Equivalent Protocol (WEP), a simple check of Wikipedia revealed the exploit. Conversely, seeing that a non-random IV was used for encryption is easy, but successful exploitation of this flaw can require significant time and effort.

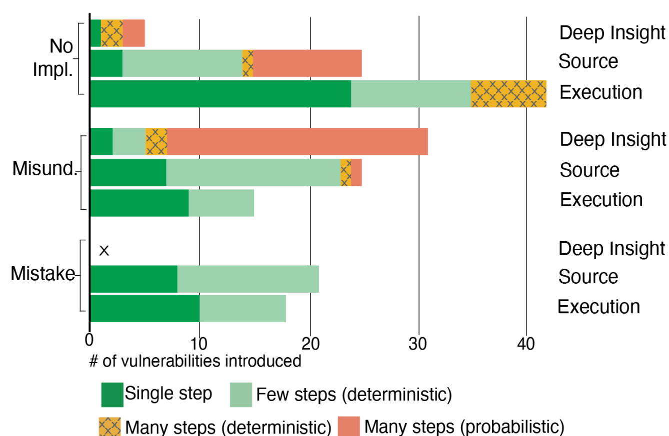


Figure 1: Number of vulnerabilities introduced for each type divided by *discovery difficulty* and *exploit difficulty*

As a *no implementation* example, one secure log team did not use a MAC to detect modifications to their encrypted files. This mistake is very simple to identify, but it was not exploited by any of the BIBIFI teams. This is likely because the team stored log data in a JSON blob before encrypting, meaning that any modifications to the encrypted text must maintain the JSON structure after decryption to succeed. This attack could require a large number of tests to find a suitable modification.

***Mistakes* are rated as easy to find and exploit**. We rated all *mistakes* as easy to exploit. This is significantly different from both *no implementation* and *misunderstanding* vulnerabilities, which were rated as easy to exploit less frequently. Similarly, *mistakes* were actually exploited during the Break It phase significantly more often than other vulnerability types. In fact, only one *mistake* was not actually exploited by any team. These results suggest that although *mistakes* were least common, any that do find their way into production code are likely to be found and exploited. Fortunately, our results also suggest that code review may be sufficient to find many of these vulnerabilities. We note that this assumes that the source is available, which may not be the case when a developer relies on third-party software.

Discussion and Recommendations

So what do these results mean for improving secure development? We believe they add weight to existing recommendations and suggest prioritizations of possible solutions.

Get the help of a security expert. In some large organizations, developers working with cryptography and other security-specific features might be required to use security-expert-determined tools and patterns or have a security expert perform a review. Our results reaffirm this practice, when possible, as participants were most likely to struggle with security concepts avoidable through expert review.

Security education. Better education should help developers better help themselves. However, across all vulnerability types, we observed no difference in vulnerabilities introduced related to prior security training or years of prior development experience. It therefore seems that increased development experience and (traditional) security training have, at most, a small impact.

Further, many of the BIBIFI teams had previously completed a four-course cybersecurity training during which all needed security controls were discussed, but a majority of these teams nevertheless botched *unintuitive* requirements. Were the topics not driven home sufficiently? An environment like BIBIFI, where developers practice implementing security concepts and receive feedback regarding mistakes, could help. Future work should consider how well competitors from one contest do in follow-on contests.

API design. Our results support the basic idea that security controls are best applied transparently, e.g., using simple APIs [4]. However, while many teams used APIs that provide security (e.g., encryption) transparently, they were still frequently misused (e.g., failing to initialize using a unique IV or failing to employ stream-based operation to avoid replay attacks). It may be beneficial to organize solutions around general use cases, so that developers only need to know the use case and not the security requirements.

API documentation. API usage problems could be a matter of documentation, as suggested by prior work [1, 7]. For example, two teams used TLS socket libraries but did not enable client-side authentication, necessary for the problem. This failure appears to have occurred because client-side authentication is disabled by default, but this fact is not mentioned in the documentation [11, 12]. Defaults within an API should be safe and without ambiguity [4]. Returning to the example from Listing 2, the team disabled the automatic integrity checks of the SQLCipher library. Their commit message stated, "Improve performance by disabling per-page MAC protection." We know this change was made to improve performance, but it is possible they assumed they were only disabling the "per-page" integrity check while a full database check remained. The documentation is unclear about this (https://www.zetetic.net/sqlcipher/sqlcipher-api/#cipher_use_MAC).

Vulnerability analysis tools. There is significant interest in automating security vulnerability discovery (or preventing vulnerability introduction) through the use of code analysis tools. Such tools may have found some of the vulnerabilities we examined in our study. For example, static analyses, symbolic executors, fuzzers, and dynamic analyses could have uncovered vulnerabilities relating to memory corruption, improper parameter use (like a fixed IV), and missing error checks. However,

they would not have applied to the majority of vulnerabilities we saw, which were often design-level, conceptual issues.

How could automation be used to address security requirements at design time? More research is needed, but one possible direction forward is to consider analysis development in tandem with improvements to API design. One example is Google's efforts to restrict the ways developers can potentially introduce certain vulnerabilities (e.g., XSS, SQL-injection) through API design, limiting the required complexity of vulnerability discovery analysis [5].

Conclusion

Secure software development is challenging, with many proposed remediations and improvements. To know which interventions are likely to have the most impact requires understanding which security errors programmers tend to make and why. In our review of 94 submissions to a secure-programming contest, each implementing one of three non-trivial, security-relevant programming problems, we found implementation mistakes were comparatively less common than failures in security understanding. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

References

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the Usability of Cryptographic APIs," in *Proceedings of the IEEE Symposium on Security and Privacy* (2017), pp. 154–171.

[2] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in *Proceedings of the ACM Conference on Computer and Communications Security* (2013), pp. 73–84.

[3] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software," in *Proceedings of the ACM Conference on Computer and Communications Security* (2012), pp. 38–49.

[4] M. Green and M. Smith, "Developers Are Not the Enemy!: The Need for Usable Security APIs," *IEEE Security & Privacy*, vol. 14, no. 5 (Sept.–Oct. 2016), pp. 40–46.

[5] C. Kern, "Preventing Security Bugs through Software Design," 24th USENIX Security Symposium: https://www.usenix.org/conference/usenixsecurity15/symposium-program/presentation/kern.

[6] D. R. Kuhn, M. S. Raunak, and R. Kacker, "An Analysis of Vulnerability Trends, 2008–2016," in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security Companion*, pp. 587–588.

[7] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study," in *Proceedings of the ACM Conference on Computer and Communications Security* (2017), pp. 311–328.

[8] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build It, Break It, Fix It: Contesting Secure Development," in *Proceedings of the ACM Conference on Computer and Communications Security* (2016), pp. 690–703.

[9] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," in *Proceedings of the Symposium on Operating System Principles* (ACM, 1975), pp. 1278–1308..

[10] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, "Understanding Security Mistakes Developers Make: Qualitative Analysis from Build It, Break It, Fix It," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, pp. 109–126.

[11] TLS socket documentation: https://golang.org/pkg/crypto/tls/#Listen and https://www.openssl.org/docs/manmaster/man3/SSL_new.html.

[12] SQLCipher documentation: https://www.zetetic.net/sqlcipher/sqlcipher-api/#cipher_use_MAC.

The Road to Less Trusted Code

Lowering the Barrier to In-Process Sandboxing

TAL GARFINKEL, SHRAVAN NARAYAN, CRAIG DISSELKOEN, HOVAV SHACHAM, AND DEIAN STEFAN

Tal Garfinkel is an independent researcher and consultant whose work focuses on the intersection of systems and security. He received his PhD from Stanford University in 2010 and is a co-founder of the USENIX Workshop on Offensive Technology. talg@cs.stanford.edu

Shravan Narayan is a fifth-year PhD student at UC San Diego working with Deian Stefan. His research focuses on in-process sandboxing, WebAssembly, browser security, and verified programming. He is the maintainer of the RLBox sandboxing framework. srn002@cs.ucsd.edu

Craig Disselkoen is a fifth-year PhD student at UC San Diego under Deian Stefan and Dean Tullsen. His research focuses on securing software through automatic vulnerability finding, program transformations, and secure runtimes. He is the author of the Haybale symbolic execution engine, written in Rust. cdisselk@cs.ucsd.edu

Hovav Shacham is a Professor of Computer Science at the University of Texas at Austin. His research interests are in applied cryptography, systems security, privacy-enhancing technologies, and technology policy. His work has been recognized with three "test of time" awards, including one at ACM CCS 2017 for his 2007 paper that introduced return-oriented programming. hovav@cs.utexas.edu

Firefox currently ships with a variety of third-party and in-house libraries running sandboxed using a new framework called RLBox. We explore how RLBox uses the C++ type system to simplify retrofitting sandboxing in existing code bases, and consider how better tooling and architecture support can enable a future where library sandboxing is a standard part of how we secure applications.

Users expect featureful software, and features, it hardly needs saying, come from code. The more features, the more code to implement them. And the more code, the more bugs—the more *security* bugs, in particular.

Whether it's the latest code rushed out before a marketing deadline, old code that hasn't been touched since the developer who wrote it retired, or a specialized module you licensed, attackers will scour them for bugs to use for exploiting your software and targeting your users.

The problem is especially acute with third-party open source libraries. You might care about one aspect of what the library does, but you ship the whole library, and bugs in any part of it can create security problems in your product. That is, unless you fork the library to remove the extraneous code, but who wants to maintain a fork forever? Worse, hackers who find a bug in a popular library can try to deploy it against every product that embeds the library—including yours.

Computer scientists have been thinking about software insecurity for 50 years, and they have come up with approaches to mitigate it. Rewrite your program (or parts of it) in a safer language! Refuse to ship new features and keep your program small! Formally verify the correctness of your software! "Privilege separate" your system by re-architecting it into multiple mutually distrusting processes! It's fair to say that none of these approaches has solved the problem. Insecure software is all around.

We believe that there is a practical path to improving software security. You can take software modules, including third-party libraries, and *sandbox* them to constrain what they can do—with low programmer effort, reasonable runtime overhead, and without wholesale rewriting or re-architecting—without even creating new OS processes. The sandboxed module will still have bugs, but those bugs will not (in most cases; see below) create security vulnerabilities in the enclosing program.

Consider an image decoding library like `libjpeg`. With sandboxing, we can restrict this library so it has access to the image it decodes and the bitmap it produces, *and that's it*. Or consider a spell-checking library like Hunspell. With sandboxing, we can restrict this library to just its dictionary and the text it checks. The application benefits from the library's features but doesn't inherit its security flaws.

Over the past two years we have worked with a team at Mozilla to build a tool, called RLBox, to support sandboxing and to migrate Firefox to a model where many third-party libraries run sandboxed. This new approach is now shipping in Firefox. Our experience suggests that once there is sufficient tooling support, then engineers can easily sandbox libraries, and they

Deian Stefan is an Assistant Professor of CSE at UC San Diego, where he co-leads the Security and Programming Systems groups. He received his PhD from Stanford University in 2016. Deian was a cofounder of Intrinsic, a web security start-up (acquired by VMware). His current research lies at the intersection of secure systems, programming languages, and verification. deian@cs.ucsd.edu

become increasingly comfortable with and excited by the opportunities this offers. For example, while the initial target of our sandboxing collaboration was a third-party font-shaping library, Graphite, now Firefox developers and security engineers are using RLBox to sandbox both third-party libraries and legacy Mozilla code in domains like media decoding, spell checking, and even speech synthesis.

We believe that the opportunities extend far beyond Firefox. After all, secure messaging apps (e.g., Signal, WhatsApp, and iMessage), servers and runtimes (e.g., Apache and Node.js), and enterprise tools (e.g., Zoom, Slack, and VS Code) also rely on third-party libraries for various tasks—from media rendering, to parsing network protocols like HTTP, image processing (e.g., to blur faces), spell checking, and automated text completion. With RLBox, these systems' developers are empowered to sandbox modules and limit the damage their bugs can cause.

Recent advances in compilers and processor architectures have made efficient in-process isolation increasingly practical. As it turns out, though, preventing a module from reading or writing memory outside its data region isn't enough. Our initial efforts in manually sandboxing Firefox libraries are a case in point. Firefox had been written under the assumption that the libraries were trustworthy. Even when isolated, they could return data values that would cause the (unsandboxed) Firefox code to take unsafe actions, a scenario that security researchers describe as a confused deputy attack. We tried to add code to manually check return values for consistency, but repeatedly found that we had missed cases and left open avenues for attack.

That's where RLBox comes in. Using the C++ type system, RLBox automatically generates the boilerplate code required for sandbox interaction, and identifies *all* places where the programmer will have to add data-checking code. With RLBox, programmers have a framework that makes it easy to sandbox libraries (1) *securely*, ensuring the interface between the untrusted library and the application code is correct, and (2) with *minimal engineering effort*, so that the cost of migrating libraries and applications to sandboxing is not prohibitive.

In the rest of this article we describe the experience that led to RLBox, how RLBox works, how it leverages the C++ type system to make sandboxing practical, and how our type-driven approach can be used in other domains (e.g., trusted execution environments). Then we outline how this approach can translate to languages other than C/C++. Finally, we end with a vision of what software development could look like with broader first-class support for sandboxing.

Before closing, we should note that sandboxing is not a panacea. Some components must be *correct*, not just isolated, for the system as a whole to be secure. The JavaScript just-in-time compilers used by Web browsers are a notorious example. With RLBox, you can sandbox everything else, and focus developer time on getting these few critical modules right.

The Road to RLBox: Library Sandboxing in Firefox

Firefox, like other browsers, relies on dozens of third-party libraries to decode audio, images, fonts, and other content. These libraries have been a significant source of vulnerabilities in the browser (e.g., most of the vulnerabilities found by recent work using symbolic execution were in third-party libraries [2]). With collaborators at Mozilla, we sought to minimize the damage due to vulnerabilities in libraries by retrofitting Firefox to sandbox these libraries.

When we began this project roughly two years ago, we thought the hardest part would be adapting Google's Native Client (NaCl), a software-based isolation (SFI) toolkit, to sandbox libraries. NaCl is designed for sandboxing programs, not libraries. This turned out to be the easy part. Since then, WebAssembly (Wasm) toolkits—in particular the Lucet Wasm compiler—have made this even easier [5].

In fact, the hardest part was the *last mile*, retrofitting Firefox to account for the now-untrusted libraries. Firefox was written assuming libraries are trusted. To add sandbox-

ing, we had to change its threat model to assume sandboxed libraries are untrusted, and harden the browser-library interface. Hardening this interface in turn required sanitizing data and regulating control flow between sandboxed libraries and the browser, thus ensuring that malicious libraries could not break out of their sandbox.

Our first attempt at sandboxing libraries in Firefox involved manually hardening the library-application interface—this did not go well.

Security challenges. To see how things can go wrong, let's consider updating the `fill_input_buffer` JPEG decoder function. `libjpeg` calls this function whenever it needs more bytes from Firefox. As seen on line 16 of Listing 1, Firefox also saves the unused input bytes held by `libjpeg` to an internal back buffer, which it sends to `libjpeg` along with the new input bytes.

```
 1: void fill_input_buffer (j_decompress_ptr jd) {
 2:   struct jpeg_source_mgr* src = jd->src;
 3:   nsJPEGDecoder* decoder = jd->client_data;
 4:   ...
 5:   src->next_input_byte = new_buffer;
 6:   ...
 7:   if (/* buffer is too small */) {
 8:     JOCTET* buf = (JOCTET*) realloc(...);
 9:     if (!buf) {
10:       decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
11:       ...
12:     }
13:     ...
14:   }
15:   ...
16:   memmove(decoder->mBackBuffer + decoder->mBackBufferLen,
17:       src->next_input_byte, src->bytes_in_buffer);
18:   ...
19: }
```

Listing 1

When sandboxing `libjpeg`, we need to make the following changes:

- Sanitize `jd`, otherwise the read of `jd->src` on line 2 could become a read gadget.
- Sanitize `src`, otherwise the write to `src->next_input_byte` on line 5 becomes a write gadget and the `memmove()` on line 16 becomes an arbitrary read gadget.
- Sanitize `jd->client_data` on line 3 to ensure it points to a valid Firefox `nsJPEGDecoder` object; otherwise invoking a virtual method on it will hijack control flow.
- Sanitize the nested pointer `mInfo.err` on line 10 prior to dereferencing, else it becomes a write gadget.
- Sanitize the pointer `decoder->mBackBuffer + decoder->mBackBufferLen` used on the destination address to `memmove()` on line 16 to prevent overflows of the unused byte buffer.
- Adjust pointer representations for `mInfo.err` and `decoder->mBackBuffer`—both NaCl and Wasm have different pointer representations and we must translate (swizzle) these pointers accordingly.
- Ensure that multiple threads can't invoke the callback on the same image; otherwise we have a data race that results in a use-after-free vulnerability on line 8.

If we miss any of these checks—and these are only a limited sample of the kind of checks required [4]—an attacker could potentially bypass our sandbox through a confused deputy attack. Adding these checks to the hundreds of Firefox functions that use `libjpeg` was tedious. Worse, we frequently found checks we had overlooked.

Engineering effort. The upfront engineering effort of modifying the browser this way was huge. Beyond adding security checks, we also had to retrofit all library calls, adjust data structures to account for machine model (ABI) differences between the application and sandbox (a common issue with SFI toolchains), marshal data to and from the sandbox, etc. Only then could we run tests to ensure our retrofitting didn't break the application. Finally, since Firefox runs on many platforms—including platforms not yet supported by SFI toolkits like NaCl and Wasm—we had to do this alongside the existing code that uses the library unsandboxed, using the C preprocessor to select between the old code and the new code. The patches to do all this became so complicated and unwieldy that we couldn't imagine anybody maintaining our code, so we abandoned this manual approach, built RLBox, and started anew.

The RLBox Framework

RLBox is a C++ library designed to make it easier for developers to securely retrofit library sandboxing in existing applications. It does this by making data and control flow at the application-sandbox boundary explicit—using types—and by providing APIs to both mediate these flows and enforce security checks across the trust boundary.

RLBox mediates data flow using *tainted types*—it uses type wrappers to demarcate data originating from the sandbox, and ensure that application code cannot use this data unsafely. For example, while application code can add two `tainted<int>`s (to produce another `tainted<int>`), it cannot branch on such values or use them as indexes into an array. Instead, the application must validate tainted values before it can use them.

RLBox mediates control flow with explicit APIs for control transfers. Calls into the sandbox must use `sandbox_invoke(sbx_fn, args...)`. Callbacks into the application can only use functions registered with the `sandbox_callback(app_fn)` API. These APIs also impose a strict data flow discipline by forcing all sandbox function return values, and callback arguments, to be tainted.

As we show next, this tainted-type-driven approach addresses both the security and engineering challenges we outline above.

Using Tainted Types to Eliminate Confused Deputy Attacks

RLBox eliminates confused deputy attacks by turning unsafe control- and data-flows into type errors and, where possible, by performing automatic security checks. Concretely, RLBox automatically sanitizes sandbox-supplied (tainted) pointers to ensure they point to sandbox memory, swizzles pointers that cross the trust boundary, and statically identifies locations where tainted data must be validated before use.

Consider, for example, the JPEG decoder callback from before. RLBox type errors would guide us to (1) mark values from the sandbox as tainted (e.g., the `jd` argument and `src` variable on line 2, Listing 2) and (2) *copy and verify* (otherwise tainted) values we need to use (e.g., `jd->client_data` on line 3, Listing 2).

```
 1: void fill_input_buffer (rlbox_sandbox& sandbox,
      tainted<j_decompress_ptr> jd) {
 2:   tainted<jpeg_source_mgr*> src = jd->src;
 3:   nsJPEGDecoder* decoder =
      jd->client_data.copy_and_verify(...);
 4:   ...
 5:   src->next_input_byte = new_buffer;
 6:   ...
 7:   if (/* buffer is too small */) {
 8:     JOCTET* buf = (JOCTET*) realloc(...);
 9:     if (!buf) {
10:       decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
11:       ...
12:     }
13:     ...
14:   }
15:   ...
16:   size_t nr = src->bytes_in_buffer.copy_and_verify(...));
17:   memmove(decoder->mBackBuffer + decoder->mBackBufferLen,
18:       src->next_input_byte.copy_and_verify(...), nr);
19:   ...
20: }
```

Listing 2

In Listing 2, we need to write validators as C++ lambdas to the `copy_and_verify` method used on lines 3, 16, and 18. As we describe in [4], validators fall into one of two categories: preserving application invariants (e.g., memory safety) or enforcing library invariants. On line 3, for example, we must ensure that `decoder` points to a valid `nsJPEGDecoder` object not used by a concurrent thread, while on line 16 we need to ensure that copying `nr` bytes won't read past the `mBackBuffer` bounds.

We must get validators right—a bug in a validator is often a security bug. In practice, though, validators are rare and short. The six libraries we sandboxed in [4] required 2–14 validators each, and these validators averaged only 2–4 lines of code. Most importantly, by making these validators explicit, RLBox makes code reviews easier: security engineers only need to review these validators.

What's missing in Listing 2 is almost as important: we don't write any security checks on lines 2, 5, and 10, for example. Instead, RLBox uses runtime checks to automatically swizzle and sanitize the `src`, `src->next_input_byte`, and `decoder->mInfo.err` pointers to point to sandbox memory.

Using Tainted Types to Minimize Engineering Effort

Manually migrating an application to use library sandboxing is labor intensive and demands a great deal of specific knowledge about the isolation mechanism. RLBox abstracts away many of these specifics, making migration relatively simple and mechanical.

Incremental migration. While RLBox automates many tasks, we still need to change application code to use RLBox. In particular, we need to add a trust boundary at the library interface by turning all control transfers (i.e., library function calls and callbacks) into RLBox calls, and we need to write validators to sanitize data from the library, as we saw above. Making these changes all at once is frustrating, error-prone—overlooking a single change might suddenly result in crashes or more subtle malfunctions—and hard to debug.

RLBox addresses these challenges with *incremental migration*, allowing developers to modify application code to use the RLBox API one line at a time. A full migration involves multiple steps and is explained further in our paper [4]. However, the key idea is that RLBox provides *escape hatches* which let developers temporarily disable some checks while migrating their application code. Thus, at each step, the application can be compiled, run, and tested.

RLBox provides two escape hatches:

1. The **UNSAFE_unverified API** allows developers to temporarily remove the tainted type wrapper (e.g., to run and test their code). As the application is ported, calls to `UNSAFE_unverified` APIs are removed or replaced with validator functions that correctly sanitize tainted data.

2. The **RLBox noop sandbox** provides a pass-through sandbox that redirects function calls back to the unsandboxed version of the library, while still wrapping data as if it were received from a sandboxed library. This allows developers to use the RLBox APIs and test data validation separately from the actual isolation mechanism.

Compile-time type errors guide the developer by pointing to the next required code change—e.g., data that needs to be validated before use, or control transfer code that needs to change to use the RLBox APIs. By the end of the process, the application is still fully

functional, all the escape hatches have been removed, and the application-library interface has fully migrated to using tainted types.

We found that incremental migration greatly simplified the code review process. In Firefox, we could commit and get reviews for partial migrations to the RLBox API, since the Firefox browser continued to build and run as before. Additionally, we could explicitly include security reviews when writing the data validators for tainted data.

Beyond migration, we also found the noop sandbox to be useful for selectively enabling library sandboxing in conditional builds. For example, while Firefox on Linux and OS X uses Wasm for isolation, the Lucet Wasm compiler's support for Windows is incomplete and thus Firefox uses the noop sandbox on Windows builds; once Windows support is complete, a single line change will allow us to take advantage of the sandbox. This is useful beyond Firefox too: developers of the Tor Browser (a downstream project of Firefox for anonymous web browsing) are interested in sandboxing more libraries than mainline Firefox, since Tor users typically have a higher security-performance threshold. Using the noop sandbox will allow Tor developers to contribute upstream changes to sandbox libraries in mainline Firefox, using the noop sandbox to avoid noticeable overhead. Tor developers can then selectively enable additional sandboxing (again) with a one-line change, rather than having to maintain a major fork.

ABI translations. Isolation mechanisms can have different machine models and ABIs from the rest of the application. For example, Wasm uses a 32-bit machine model meaning that pointers, `ints`, and `longs` are 32 bits. However, this is a different machine model from that used by the host application. Handling such differences manually is laborious and error-prone.

Consider line 10 from the previous `fill_input_buffer` example in Listing 2:

```
// mInfo is an object of type jpeg_decompress_struct
decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
```

If we port this manually, the resulting code would be:

```
auto err_field = adjust_for_abi_get_minfo_field(decoder
      ->minfo, "err");
auto err_field_swizzled = adjust_for_abi_convert_pointer
      (err_field);
auto msg_field = adjust_for_abi_get_err_field
      (*err_field_swizzled, "msg_code");
assert(in_sandbox_memory(msg_field));
      // Ensure pointer is in sandbox memory
auto msg_field_swizzled = adjust_for_abi_convert_pointer
      (msg_field); // Assign the value
*msg_field_swizzled = adjust_for_abi(JERR_OUT_OF_MEMORY);
```

In contrast, RLBox requires no changes other than marking `mInfo` as tainted. RLBox automatically transforms pointers, and accounts for the difference in the size of `long` and `pointers`:

```
// mInfo is an object of type tainted<jpeg_decompress_struct>
decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
```

RLBox is able to abstract and automatically reconcile ABI differences since all control and data flow goes through its APIs and tainted types.

Using Tainted Types Outside of Library Sandboxing

The security challenges we face when sandboxing libraries are not unique to library sandboxing. Developers have to handle untrusted data and control flow in many other domains—and our tainted-type approach can help. We give three examples:

TEE runtimes. Applications running in trusted execution environments (TEEs), like Intel's SGX and ARM's TrustZone, interface with untrusted code by design—TEEs even consider the OS untrusted. Getting this code right is hard. And, indeed, TEE runtimes contain similar bugs: Van Bulck et al. [1], for example, found that most frameworks, across several TEEs, were vulnerable to bugs RLBox addresses by construction.

OS kernels. Operating system kernels handle untrusted data from userspace. Bugfinding tools—from MECA at the start of the century [10] to Sys this year [2]—have found many vulnerabilities in kernels due to unchecked (or improperly checked) userspace data (notably, pointers). Frameworks like RLBox could automatically identify where userspace data needs to be checked and even perform certain checks automatically (e.g., much like we ensure that sandbox pointers point to sandbox memory, we can ensure that userspace pointers point to userspace memory). Indeed, Johnson and Wagner's bugfinding tool [3] even used type inference to find such kernel bugs.

Browser IPC layers. Modern browser architectures privilege separate different parts of the browser into sandboxed processes. Almost all separate the *renderer* parts—the portion of the browser that handles untrusted user content from HTML parsing, to JavaScript execution, to image decoding and rendering—from the *chrome* parts—the trusted portion of the browser that can access the file system, network, etc.—and restrict communication to a well-typed inter-process communication (IPC) layer. Like OS kernels, the browser chrome must validate all values coming from untrusted renderer processes; like kernels, browsers have been exploited because of unchecked (and improperly checked) untrusted data. Here, again, tainted types can help—and as a step in this direction, Mozilla started integrating tainted types into the Firefox IPC layer, as part of the IPDL (IPC protocol definition language) used to generate boilerplate code for sending and receiving well-typed IPC messages [7].

This list is by no means exhaustive; others have similarly observed that tainting can be used to catch and prevent bugs when handling untrusted data (e.g., see [9]).

Beyond RLBox

We have thus far discussed RLBox in its current form—a framework that uses the C++ type system, template metaprogramming, and SFI toolkits like Wasm to securely sandbox libraries typically written in C. In the future, we hope to see extensions to other languages, support for sandboxing libraries written in arbitrary languages, and the adoption of processor features that can further lower in-process sandboxing overheads.

Beyond C++

We implemented RLBox in C++ because Firefox is predominantly written in C++. To extend RLBox to other languages, we need to understand how to implement RLBox's tainted type system.

Our C++ implementation uses templates to implement the generic `tainted<T>` type and takes advantage of function and operator overloading to make most of the tainted type interface transparent. For example, RLBox overloads pointer dereferencing—the `->` and `*` operators—to allow dereferencing `tainted<T*>` values safely by automatically sanitizing the underlying pointer to point to sandbox memory (line 10 in Listing 2). We also use template metaprogramming to enforce a custom type discipline.

Many languages have features that are expressive enough to implement our tainted type system directly or as part of the language toolchain, for example, with compiler plugins.

Statically typed languages. RLBox is a natural fit for languages that already enforce type safety statically. Statically typed languages typically offer some form of generics or templates that can be used to implement tainted types. Many also allow function and operator overloading which, like C++, would allow us to provide safe operations on tainted types while preserving the original syntax of the language.

Rust is a particularly compelling language. First, Rust's raison d'être is safety—indeed, the language is used in many settings where assurance is paramount—and RLBox can complement Rust's safety by, for example, making it easy for Rust programmers to safely integrate C/C++ code into their projects, which today is considered unsafe. Second, Rust's macro system and support for generics and operator overloading via traits allows tainted types to be implemented directly in the language. Finally, Rust's affine types can even simplify certain RLBox validators, like the validators used to prevent time-of-check to time-of-use and double fetch attacks [4].

Dynamically typed languages. In dynamically typed languages like JavaScript and Python, we can enforce tainted types dynamically. This, of course, makes the incremental porting loop longer since type errors will only manifest at runtime. Luckily, many dynamically typed languages have typed extensions to precisely address this limitation. For example, TypeScript and Flow extend JavaScript with static type annotations.

Compiler plugins and toolkits. For languages not flexible enough to implement the RLBox tainted type system statically, we envision implementing the type system as part of language toolchains. For example, for C, we can implement RLBox as a Clang plugin (both to enforce the type system and to generate runtime checks). Alternatively, we can implement tainted types as part of interface description language (IDL) compilers. As mentioned above, for example, the Mozilla security team is integrating tainted types into the Firefox IPDL inter-process communication protocol IDL [7].

Beyond Software-Based Isolation

We designed RLBox to make it easy for developers to plug in different isolation mechanisms. This makes it easy to migrate code (e.g., by using the noop sandbox), as we have described. It also allows developers to use different isolation mechanisms that have different tradeoffs. For example, while in production we use Wasm for isolation, in [4] we evaluate two other isolation mechanisms: NaCl and traditional process-based isolation. These isolation mechanisms have different tradeoffs. Process isolation is simple but scales poorly—protection boundary crossing costs become prohibitive as the number of sandboxes exceed the number of available cores. Wasm and NaCl, on the other hand, scale to a large number of sandboxes and have cheap boundary crossings, but they impose an overhead on the sandboxed code.

At present, Wasm toolchains offer a practical and portable path to isolation. But this software-based isolation approach will inevitably be slower than running native code.

Hardware support for in-process isolation can offer solutions that are simple and more performant. Today, for example, Intel's Memory Protection Key (MPK) features incur roughly 1% overhead when used for in-process isolation [8], but this doesn't scale beyond 16 sandboxes. In the future, the CHERI capability-based system will similarly make in-process isolation—and memory safety more generally—cheap on ARM processors [6]. By making it easy to use these features transparently (e.g., for CHERI it can automatically adjust for ABI differences introduced by capabilities), RLBox could lower the barrier to adopting new hardware isolation features—and, we hope, this will encourage new hardware design for in-process isolation.

Bringing Sandboxing to the Developer Ecosystem

While RLBox has been a boon for our work in Firefox, it's just a starting point. Our hope is that library sandboxing will become a first-class activity in future development environments, and that RLBox's capabilities will ultimately be subsumed by standard parts of tomorrow's languages, toolchains, and package managers. We believe in many cases such support could allow the use of sandboxed libraries with a level of ease comparable to the use of unsandboxed libraries today.

FFIs and native code. Many popular safe languages such as Python, Ruby, and JavaScript make extensive use of native (typically C) code in their standard libraries and package ecosystems via foreign function interfaces (FFIs). Unfortunately, bugs in native code can completely break all high-level safety guarantees. Extending FFI interfaces and interface generation tools with first-class support for sandboxing native code is very natural—both because the FFI boundary is explicit and because developers are used to writing code that spans trust boundaries.

Package managers. In the ecology of package ecosystems there is constant competition between package authors to provide the best package for a given task. Security is among the ways that package authors have recently started differentiating their package from others. We have seen this clearly in the Rust ecosystem, where the presence (or absence) of unsafe code is one way that packages are compared.

Sandboxing is another way that package authors can provide differentiated value, by integrating sandboxing support into their library. This could look like authors distributing their packages with most or all of the work required to sandbox that package done upfront by the package author. Developers could then choose whether or not to enable sandboxing with minimal additional fanfare.

To facilitate this, the package author could specify a system level sandboxing policy (e.g., as a manifest file requesting access to parts of the file system or network), and developers could then choose if and how to grant these privileges when importing a package. Much of the work of writing validators for tainted types could also be mitigated by distributing validators as part of a sandboxed library. We even envision an ecosystem of sandbox interface declarations for existing packages, much like TypeScript type declarations for JavaScript packages, which will allow to developers to pull sandboxed interfaces much like they consume type declarations today.

Conclusion

Decades of attempts to detect and mitigate software vulnerabilities have yielded lackluster results. Even browsers, some of the most heavily targeted and scrutinized software, seem to provide an inexhaustible stream of exploitable vulnerabilities. In-process sandboxing can offer developers and security engineers another choice—moving code, especially legacy and third-party code, out of their trusted computing base by sandboxing it, thus mitigating the impact of a compromise.

We developed RLBox to make sandboxing practical. It is currently being used to sandbox third-party and in-house libraries in Firefox, and we hope that other C++ projects will choose to adopt it. Looking further, we hope to collaborate with developers of programming languages (and their toolchains and standard libraries), package managers, and processor architects to provide first-class support for in-process sandboxing. Small changes to make in-process sandboxing first-class can result in huge benefits for developers and security engineers.

References

[1] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, pp. 1741–1758: https://people.cs.kuleuven.be/~jo.vanbulck/ccs19-tale.pdf.

[2] F. Brown, D. Stefan, and D. Engler, "Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, pp. 199–216: https://www.usenix.org/conference/usenixsecurity20/presentation/brown.

[3] R. Johnson and D. Wagner, "Finding User/Kernel Pointer Bugs with Type Inference," in *Proceedings of the 13th USENIX Security Symposium (USENIX Security '04)*: https://www.usenix.org/event/sec04/tech/full_papers/johnson/johnson_html/.

[4] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting Fine Grain Isolation in the Firefox Renderer," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20),* pp. 699–716: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

[5] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: WebAssembly as a Practical Path to Library Sandboxing," arXiv, December 4, 2019: https://arxiv.org/abs/1912.02285.

[6] R. Grisenthwaite, "A Safer Digital Future, by Design," ARM Blueprint, October 18, 2019: https://www.arm.com/blogs/blueprint/digital-security-by-design.

[7] T. Ritter, "Support Tainting Data Received from IPC," Mozilla Bug 1610005, January 2020: https://bugzilla.mozilla.org/show_bug.cgi?id=1610005.

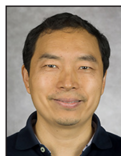
[8] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, Efficient In-Process Isolation with Protection Keys (MPK)," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pp. 1221–1238: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[9] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," in *Proceedings of the 15th USENIX Security Symposium (USENIX Security '06)*, pp. 121–136: https://www.usenix.org/legacy/event/sec06/tech/full_papers/xu/xu_html/.

[10] J. Yang, T. Kremenek, Y. Xie, and D. Engler, "MECA: An Extensible, Expressive System and Language for Statically Checking Security Properties," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03),* pp. 321–334: https://web.stanford.edu/~engler/ccs03-meca.pdf.

Using Safety Properties to Generate Vulnerability Patches

ZHEN HUANG, DAVID LIE, GANG TAN, AND TRENT JAEGER



Zhen Huang is an Assistant Professor in the School of Computing at DePaul University. He earned his BASc from Wuhan University and his MS and PhD from the University of Toronto. He works on computer systems with an emphasis on software security. zhen.huang@depaul.edu

David Lie received his BASc from the University of Toronto in 1998, and his MS and PhD from Stanford University in 2001 and 2004, respectively. He is currently a Professor in the Department of Electrical and Computer Engineering at the University of Toronto. He also holds appointments in the Department of Computer Science, the Faculty of Law, and is a research lead with the Schwartz Riesman Institute for Technology and Society. He was the recipient of a best paper award at SOSP for this work. David is also a recipient of the MRI Early Researcher Award and the Connaught Global Challenge Award. David has served on various program committees including OSDI, USENIX Security, IEEE Security & Privacy, NDSS, and CCS. Currently, his interests are focused on securing mobile platforms, cloud computing security, and bridging the divide between technology and policy. lie@eecg.toronto.edu

Automatic Program Repair (APR) methods attempt to fix vulnerabilities in programs comprehensively and without introducing new defects. Senx uses novel safety properties to generate patches, and it succeeds in generating patches for 32 of 42 real-world vulnerabilities. We explain how Senx works, compare it to other APR methods, and demonstrate why Senx is better at repairing source code.

Fixing security vulnerabilities in a timely manner is critical to protect users from attacks that exploit vulnerabilities. Unfortunately, a recent study shows that the average time to release software patches for vulnerabilities is 52 days, and the bottleneck lies in creating software patches [1].

Automatic Program Repair (APR) tools aim to automatically provide patches that fix vulnerabilities. Most of them rely on a set of positive/negative example inputs to produce a patch that makes the vulnerable program behave correctly according to these example inputs [4, 6, 7]. The patched program must pass the positive example inputs but raise errors on the negative example inputs. But obtaining a complete set of example inputs is often difficult, and the patched program may behave incorrectly on other inputs, or the vulnerability may still be exploited by other inputs [8]. We refer to this traditional method as "example-based."

We propose a different approach called "property-based" APR that relies on vulnerability-specific, program-independent, human-specified safety properties. A safety property specifies the condition on which a type of vulnerability cannot be triggered. For example, a safety property for buffer overflow vulnerabilities can be that a program should never have access beyond the bounds of a buffer.

Our property-based approach has three major advantages: 1) a small set of safety properties can be defined once and applied on numerous programs without the need to specify anything pertaining to each of the programs; 2) the properties are precise and complete by nature so they work for all possible inputs; 3) it leverages a specific vulnerability's context to generate a customized and efficient patch for the vulnerability, as opposed to the nonspecific and often inefficient patches generated by previous methods [5].

Property-based APR faces several outstanding challenges. First, it must identify the correct property to enforce for a given vulnerability because the properties are vulnerability-specific. Second, our goal is to generate source code patches that can be easily adopted by developers; as a result, the safety properties must be expressed using program entities such as variables. Third, the generated patches should affect program execution if and only if a safety property is violated. Finally, the generated patches should incur minimum performance overhead.

To address these challenges, we have designed Senx to automatically generate source code patches for security vulnerabilities using safety properties. We demonstrate the effectiveness of Senx using three important classes of vulnerabilities: buffer overflows, bad casts, and integer overflows. Our evaluation demonstrates that Senx is able to produce correct patches for over 76% of the vulnerabilities. And we believe that, in principle, Senx can generate patches for any class of vulnerabilities for which a safety property can be specified.

Dr. Tan is a Professor in the Computer Science and Engineering Department at Pennsylvania State University. He obtained his BE in computer science from Tsinghua University and his PhD in computer science from Princeton University. His research interests are computer security, formal methods, and programming languages. He currently serves on the DARPA ISAT study group. He has also received multiple awards, including a James F. Will Career Development Professorship from 2016 to 2019, an NSF CAREER Award, two Google Research Awards, a Distinguished Reviewer Award at the 2018 IEEE Symposium on Security and Privacy, a Ruth and Joel Spira Excellence in Teaching Award at Penn State, and some best paper awards at academic conferences.
gtan@cse.psu.edu

Trent Jaeger is a Professor in the Computer Science and Engineering Department at Pennsylvania State University. Trent's primary research interests are systems and software security. He has published over 150 research papers and the book *Operating Systems Security*, which has been taught in universities worldwide. Trent has made significant contributions to the Linux community, including mandatory access control, integrity measurement, process tracing, and namespace services. Trent currently serves the computer security research community on the Executive Committee of ACM SIGSAC as Past Chair, as Steering Committee Chair of NDSS, on editorial boards of *Communications of the ACM* and *IEEE Security & Privacy*, and on the Academic Advisory Board of the UK's Cyber Body of Knowledge project.
tjaeger@cse.psu.edu

Example-Based versus Property-Based

We now discuss the limitations of state-of-the-art APR tools. We use the program in Listing 1 as the target program, which is adopted from a real-world buffer overflow vulnerability CVE-2012-0947 in a popular media stream processing library. The program takes a string and its length as input, and outputs the reversed string. It outputs "" if an error occurs. Similar to the real vulnerability, two functions are used, one to allocate the output buffer, and the other to process the input string.

The buffer overflow happens when the size, specified from the command line, is smaller than the actual length of the input string. To fix the buffer overflow, a check can be added to ensure that the actual length of the string is smaller than the allocated size of the buffer into which it is copied. Note that the buffer size is only known to main; so the check should be added at line 19 to compare size against `strlen(argv[2])`. While a human developer can easily add this check, which indeed was in the official patch for the vulnerability, it presents challenges for state-of-the-art APR tools.

```
 1 char* rev(const char *inp, char *out) {
 2    // reverse a string
 3    //  inp is the input string
 4    //  out is an output buffer
 5    if (inp != NULL) {
 6        int i, len = strlen(inp);
 7        // Failed to check if (len + 1 <= size_of_out)
 8        for (i = 0; i < len; i ++)
 9            out[i] = inp[len - i];
10        out[i] = '\0';
11        return out;
12    }  else
13        return "###";
14 }
15
16 void main(int argc, char *argv[]) {
17    int size = atoi(argv[1]) + 1;
18    char *out = (char *)malloc(size);
19    // patch: if (strlen(argv[2]) + 1 > size) exit(1);
20    printf("%s\n", rev(argv[2], out));
21 }
```

Listing 1: A program that reverses an input string. It contains a buffer overflow in function `rev`.

Example-based approaches. Many APR tools rely on example inputs to fix vulnerabilities. For example, SemFix and Angelix use test inputs to find path constraints needed to generate fixes [4, 6]. Table 1 presents typical test inputs needed to use such tools to fix the buffer overflow for our example in Listing 1.

This approach has two drawbacks. First, the generated path constraints are often based on the concrete values used in the test inputs instead of the relationships between program variables. Given the test inputs in Table 1, SemFix and Angelix would wrongly infer that the value of `argv[1]` is not correlated with whether tests are positive or negative, based on the fact that it has the same values in both positive and negative test inputs.

Type	argv[1]	argv[2]	Output	Expected output
P	1	A	A	A
P	2	AB	BA	BA
N	1	ABC	CBA	###
N	2	ABC	CBA	###

Table 1: Test inputs and outputs for the program in Listing 1. Type "P" test inputs are positive test inputs, while type "N" test inputs are negative test inputs.

Second, the approach is highly sensitive to the completeness of test inputs. Because the length of the input string is smaller than 3 for positive tests whereas the length is not smaller than 3 for negative tests, SemFix and Angelix would incorrectly derive that `strlen(argv[2]) < 3` needs to be added to the program to fix the buffer overflow. The incorrect patch is generated due to the missing of a positive test input with `strlen(argv[2]) > 2)` in the test suite. This illustrates that example-based tools can easily fail when tests are missing in the test suite, which is notoriously hard to make complete.

Property-based approaches. AutoPaG creates patches using a predicate similar to a safety property [3]. But it handles only one vulnerability type, buffer overflows, so it cannot generate a correct patch if the vulnerability is of any other type. Moreover, it would fail to produce a patch if the safety property needs to be enforced in a location other than the function in which the vulnerability occurs. As in our example, the patch should be placed in the `main` function, but the buffer overflow occurs in the `rev` function. Lastly, the patch it generates can incur high performance overhead because it would add the patch to check the buffer size inside the `for` loop on line 8 due to the fact that the buffer overflow occurs within the loop.

Safety Properties

To generate a patch that fixes a vulnerability, Senx requires an input to trigger the vulnerability. The input can be a proof-of-concept exploit or an input generated by a fuzzer. With this input, Senx generates a patch that will enforce the safety property violated by the vulnerability.

A Senx patch can have one of two forms: 1) a check-and-error patch that inserts a check to detect if a safety property no longer holds and raises an error to direct program execution away from the path where the vulnerability resides; 2) a repair patch that modifies existing code to prevent a safety property from being violated.

Each safety property corresponds to a particular vulnerability class and is an abstract Boolean expression that will be mapped to concrete variables in a program. We describe below the three types of safety properties that Senx currently supports.

Sequential buffer overflows. A sequential buffer overflow occurs when a sequence of memory accesses traversing a buffer crosses from a memory location inside the buffer to a memory location outside of the buffer. The Senx safety property for buffer overflows defines two abstract objects: a memory access and a buffer. The term *buffer* refers to any bounded memory region, which may include structs, objects, or arrays. The term *memory access* corresponds to an array access or pointer dereference occurring inside a loop. This safety property covers both the case when the memory access exceeds the upper range of the buffer

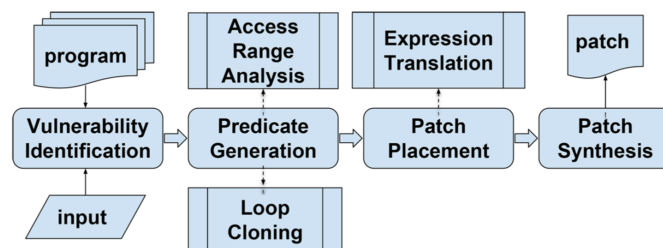


Figure 1: Workflow of Senx: each rounded rectangle represents a step in Senx's patch generation; each rectangle with vertical bars represents a component of Senx.

and the case when the memory access falls below the lower range (sometimes called a *buffer underflow*).

Bad casts. A harmful memory access can result from an offset from a base pointer beyond the upper bound of the buffer the base pointer is pointing to. This type of vulnerability may occur for several reasons, but it commonly occurs when a pointer is cast to a type that is incompatible with the object the pointer points to. The safety property for bad casts can prevent both bad casts for simple structs and objects, as well as nested structs and objects.

Integer overflows. An integer overflow takes place when a variable is assigned a value larger or smaller than what can be represented by the type of the variable. An integer overflow can lead to a vulnerability when the result of the overflow is then used in operations such as allocating a buffer, producing a buffer that is far smaller than expected. Consequently, the safety property for integer overflows checks that value used in certain operations is not the result of an integer overflow.

For our prototype, we have started with these three vulnerability classes. Nonetheless, they represent a good percentage of CVE vulnerabilities. Based on our informal analysis of the vulnerabilities published in `CVE Details` in 2018, the most popular vulnerability categories are denial of service, code execution, and overflow. By examining 100 randomly chosen CVE reports for each of the three vulnerability categories, we find that 25% of CVE vulnerabilities are buffer overflows, bad casts, or integer overflows. We believe the principles behind Senx can be extended to other vulnerability classes, and we plan to do so as our future work.

Senx

Senx aims to generate source code patches that can be easily verified and adopted by developers. As shown in Figure 1, Senx generates patches in four major steps: vulnerability identification, predicate generation, patch placement, and patch synthesis.

Vulnerability Identification

In vulnerability identification, Senx runs a program with an input that can trigger a vulnerability and outputs the violated

safety property, the *vulnerability point* (the program location where the safety property is violated), and the source code expressions for the execution trace. Senx runs the program using concolic execution to generate the execution trace corresponding to the vulnerability-triggering input. Senx records the execution trace as source code expressions, which conform to the syntax of the programming language of the target program, for synthesizing a source code patch. To support complex data types such as nested C/C++ structs, references to structs, and arrays with pointers, Senx records the relationships between data objects and the way data objects are referenced. This way Senx can recover the full expression for a data object such as `foo->f.bar[10]`.

Predicate Generation

During predicate generation, Senx takes the violated safety property, which also implies the type of the vulnerability, and the source code expressions generated by vulnerability identification, and outputs a predicate required to prevent the violation of the safety property. Senx maps the violated safety property to concrete expressions over variables, constants, and function calls in the form of the source code of the program.

For buffer overflows, Senx aims to insert the patch before the loop where a set of sequential memory accesses occurred; so it needs to extract expressions that represent the memory access range for the memory accesses. Senx uses two complementary loop analysis techniques: *access range analysis* and *loop cloning*. Both of them take a function *F* in the target program and an instruction *inst* that performs the faulty access in the buffer overflow, and output the symbolic memory access range of *inst*.

Access range analysis. Senx computes the access range of canonicalized loops. It relies on LLVM's built-in loop canonicalization functionality to convert the loop into a standard form. It starts with the innermost loop and iterates to the outermost loop, and accumulates increments and decrements on the loop induction variables.

For each loop, Senx retrieves the loop iterator variable and its bounds and the list of induction variables of the loop and their *update*, the fixed amount that an induction variable is increased or decreased by on each loop iteration. We use the loop in `bar` of Listing 2 to illustrate how access range analysis can be applied to nested loops.

```
 1 char *foo_malloc(x,y) {
 2   return (char *)malloc(x * y + 1);
 3 }
 4
 5 int foo(char *input) {
 6+  if ((double)(cols+1)*(size/cols)+1 >
 7+         rows * (cols+1) + 1)
 8+      return -1;
 9   char *output=foo_malloc(rows,cols+1);
10   if (!output)
11      return -1;
12   bar(p, size, cols, output);
13   return 0;
14 }
15
16 void bar(char *src,int size,int cols,char *dest) {
17   char *p=dest;char *q=src;
18   while (q < src+size)  {
19      for (unsigned j=0;j<cols;j++)
20         *(p++) = *(q++);
21      *(p++) = '\n';
22   }
23   *p = '\0';
24 }
```

Listing 2: A buffer overflow in CVE-2012-0947 with a patch, lines prefixed with "+"

In this example, Senx identifies `j` as the loop iterator variable, whose bounds are `0` and `cols`; it also identifies `j`, `p`, and `q` as induction variables, each of which has an update of `1` for the innermost `for` loop. Senx then symbolically accumulates the update to each induction variable based on the number of loop iterations, which is `cols`. Similarly, Senx finds `q` as the loop iterator variable, with `src` as its lower bound and `src+size` as its upper bound, and `q` and `p` as induction variables, whose accumulated update is `size` and `(cols+1)(size/cols)+1`, respectively, for the `while` loop enclosing the inner `for` loop.

Following the analysis of all the loops enclosing *inst*, Senx performs reaching definition dataflow analysis to find the definition that reaches the beginning of the outermost loop for the pointer *ptr* used by *inst*. In this example, we have *ptr*=`p` whose initial value is `dest` before the `while` loop. By adding the initial value `dest` to the accumulated update of `p`, we will have `dest+(cols+1)(size/cols)+1`. Therefore Senx decides the access range as `[dest,dest+(cols+1)(size/cols)+1]`.

Loop cloning. Senx cannot apply access range analysis to loops that LLVM cannot canonicalize. Instead it uses loop cloning for these loops. At a high level, loop cloning creates new code to compute the number of loop iterations. Senx produces the new code from a clone of the code of the loop in the target program, but removes the code that causes side effects. The new code is used by the generated patch to return the access range. Details on loop cloning can be found in [2].

Function calls. For certain cases, Senx can extract expressions containing function calls. Senx needs to ensure that the generated predicate does not call functions that have side effects. We define three types of side effect: 1) a change to the memory accessible outside of a function; 2) an invocation of a system call that has external impact; 3) an invocation of a function that has any side effect.

Senx uses a flow-sensitive, context-insensitive intraprocedural static analysis to identify the list of functions that do not have

any side effect. Senx initializes the list with functions on a whitelist and then adds each function that has no side effect to the list by analyzing every function of a target program.

Patch Placement

In patch placement, Senx uses the vulnerability point found in vulnerability identification and the predicate generated in predicate generation to find a program location to insert the patch. The patch location must be a point where all necessary variables in the predicate are in the scope. If variables in the predicate are from different scopes, Senx uses *expression translation* to translate the predicate into a new one formed from variables in a common scope. For check-and-error patches, Senx also requires the scope to have some error handling code to call. It uses Talos [1] to find a suitable error handling code.

Expression translation. Senx must produce a patch predicate that can be evaluated in a single function scope, because Senx generates source code patches. In some cases, a target program computes the buffer allocation size in one function scope but the memory access range in a different function scope. As a result, the expression representing the allocation size and the expression representing the memory access range are not valid in a single function scope.

To solve this problem, *expression translation* translates an expression from the scope of a source function to an equivalent expression in the scope of a destination function, without the need to add new function parameters and call arguments. This process is called *converging* the predicate. Expression translation exploits the equivalence between the arguments that are passed to a function by the caller and the function parameters that receive the values of the arguments.

We use the code in Listing 2 to illustrate how it works. To translate the buffer size involved in the buffer overflow, Senx starts with the buffer size expression `xy+1` in the scope of `foo_malloc` and for `x` substitutes `rows` and for `y` substitutes `cols+1` based on the call arguments at line 9. Hence `xy+1` becomes `rows(cols+1)+1` in the scope of `foo`.

Effectiveness of Senx

We evaluate the effectiveness of Senx and the quality of its generated patches using 42 real-world buffer overflow, bad cast, and integer overflow vulnerabilities that are from 11 mature and popular applications. For each vulnerability, we run the corresponding application under Senx with a vulnerability-triggering input. We manually examine the correctness of the generated patch if Senx generates a patch. Otherwise, we examine what caused Senx to abort patch generation. The list of the vulnerabilities and our detailed evaluation are presented in [2].

For the 42 vulnerabilities, Senx generates 32 patches, all of which are correct according to our criteria. Senx applies access range analysis and loop cloning roughly equally for the 13 patched buffer overflows. Senx is unable to apply loop cloning mainly because the loops involve calls to functions that have side effects that Senx cannot remove. Senx must use expression translation to generate 23.8% of the patches because the patches need to be placed in a function different from where the vulnerability occurs. The dominant cause for Senx to abort patch generation is that Senx cannot converge all variables in the patch predicate to a common function scope.

Comparison with other work. We compare the effectiveness of Senx against SemFix [6] and Angelix [4]. Due to the considerable effort required to run SemFix and Angelix, we made the comparison on only two vulnerabilities. Senx generates correct patches for both vulnerabilities, while SemFix and Angelix are unable to generate patches either because they cannot find an existing program construct to change in order to pass both positive test inputs and negative test inputs or because they cannot create a guard statement to prevent the vulnerabilities from being triggered.

Conclusion

Automatic patch generation is a promising solution to rapidly resolve software defects. However, the vast majority of these tools are not well-suited to address software vulnerabilities since they rely on test cases to generate correct patches, whereas it is difficult to have complete test cases for any moderately large target programs. To address software vulnerabilities, we built Senx, a system that uses human-specified safety properties to automatically generate patches. Senx uses three novel program analysis techniques: access range analysis, loop cloning, and expression translation. Evaluation shows that Senx generates patches correctly for 76% of the 42 real-world vulnerabilities.

Acknowledgments

This research was supported in part by an NSERC Discovery Grant (RGPIN 2018-05931) and a Canada Research Chair (950-228402).

References

[1] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie, "Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pp. 618–635.

[2] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using Safety Properties to Generate Vulnerability Patches," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pp. 539–554.

[3] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*, pp. 329–340.

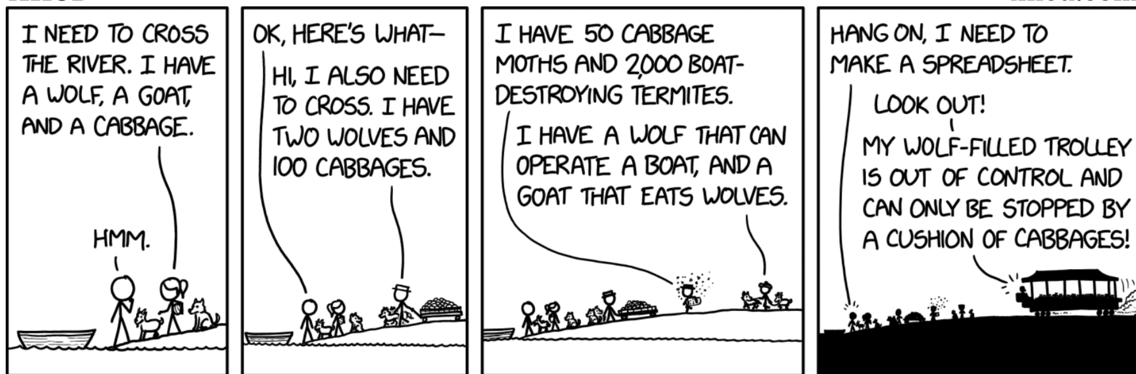
[4] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pp. 691–701.

[5] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: Highly Compatible and Complete Spatial Memory Safety for C," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pp. 245–258.

[6] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program Repair via Semantic Analysis," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*, pp. 772–781.

[7] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically Patching Errors in Deployed Software," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pp. 87–102.

[8] Z. Qi, F. Long, S. Achour, and M. Rinard, "An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*, pp. 24–36.



Interview with Sergey Bratus

RIK FARROW

Sergey Bratus is a Research Associate Professor of Computer Science at Dartmouth College. He helped co-found the LangSec movement and is interested in understanding and mitigating unintended computation. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. sergey@cs.dartmouth.edu

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

Disclaimer: The views presented in this interview are the author's personal views and do not necessarily represent the views of the U.S. Federal Government or its components, which partially funded some of the research presented at the LangSec workshop.

I first met Sergey Bratus during the USENIX Security Symposium in 2011. Sergey caught up to me in a stairwell at the Sir Francis Drake Hotel in San Francisco and started to make a pitch about something I had never heard of before. LangSec, short for Language Security, is a different way of thinking about both how to program more securely and why software gets exploited.

I found myself immediately intrigued, and Sergey has co-authored several articles and papers related to LangSec over the years. He also co-founded a LangSec workshop with Meredith Patterson, co-located with IEEE Security and Privacy ("Oakland") [1]. When I was studying papers at USENIX Security '20, I noticed several that appeared to have strong tie-ins to LangSec and decided to invite Sergey for an interview.

Rik Farrow: Software gets hacked when presented with input that manipulates the software in unexpected ways. I recall from early LangSec articles that any input parser that is more complex than a pushdown automaton will be vulnerable to this type of hacking. Do I have this right, and why are more complex parsers vulnerable?

Sergey Bratus: The programmer who sits down to write a parser faces a task quite unlike any other engineering task. All other kinds of engineers design for some well-defined operating environment conditions: this much wind speed for a bridge, this much current for an electric circuit, this expected temperature interval for a chip, etc. Within these conditions, the design must behave predictably: safety comes from predictability. By contrast, input-taking software, i.e., its parser, is supposed to withstand *any* inputs at all, an operating environment that cannot be easily searched or simulated. Yet, as with any other engineering, safety only comes from predictability.

Thus safety of a parser critically depends on the ability of the programmer to correctly predict the parser's behavior on all possible inputs. This is really hard, because reasoning about program behaviors *in general* is hard (or even algorithmically impossible) and is only feasible when the programmer walks a fairly narrow path, by correctly implementing automata that we *can* reason about and assuming no more about the inputs than these automata (if correctly implemented) can check.

Pushdown automata and their corresponding context-free languages are one particular sweet spot of predictability for which we have the mathematical and computing means of automated reasoning. This sweet spot is really something of a mathematical miracle, given how hard the general problem is.

In a word, every parser implemented on a general-purpose ISA wants to be a virtual machine on its inputs that matches the computing power of that ISA. Restraining it from being that machine for the attacker is what LangSec is about; it is surprising and fascinating that it is possible and practical to do so.

Various caveats apply, which LangSec aims to address in ways practical for a programmer who is not looking to be a mathematician or formal language theorist. However, the thing that makes it at all possible is the language-based approach, which gives us just the predictability, that is, safety, properties that we can check for and that aren't hard to express and understand.

Surprisingly, as the recent workshop's morning keynote [1] David Walker argued, this is also true for predicting behaviors of not just parsers but also networks. So the surprising effectiveness of using language-based models of computing system behaviors extends beyond what we normally think of as parsers.

RF: Programmers often build parsers according to their reading of a protocol specification. An infamous example of this going wrong was Heartbleed, where the TLS protocol included two different length values, and a popular implementation checked one while using the other. At USENIX Security, the "Composition Kills" paper [2] examines how the intersection of three email sender authentication protocols—SPF, DKIM, and DMARC—actually fail to authenticate the sender. Are protocols part of the problem that LangSec addresses?

SB: Yes. From its inception [17], LangSec has been calling attention not only to unintended behaviors of inputs on parsers, but also to security consequences of *parser differentials*, that is, divergent interpretations of the same messages by different parsers.

To have any predictability in a distributed system—which is really just a fancy name for a system with more than one component—it is natural to implicitly assume that all of its parsers interpret messages passed between the components in the same way. Whenever this assumption, made explicitly or implicitly, is violated, vulnerability likely ensues.

Vulnerabilities with the root cause in parser differentials have been in the news lately. The HTTP Desync vulnerabilities [3] such as the F5 Big-IP vulnerability [4], the "Psychic Paper" vulnerability in MacOS [5], and a vulnerability in GitLab [6] all involve parser differentials. Major past examples include several Android Master Key vulnerabilities [7], a timeless classic.

LangSec's perspective on the insecurity potential of parser differentials has been getting some notice. Another notable, recently published academic paper [8] discusses parsing of standard protocols and refers to LangSec. Dave Aitel drew attention to the LangSec nature of this growing vulnerability class on his DailyDave mailing list (https://seclists.org/dailydave/2020/q3/9). To quote Dave:

> Ten years ago a lot of the security community had a discussion about "LangSec"...which turns out to have been entirely correct in retrospect....
>
> Most people look at HTTP Desync as simply using Content-Length confusion—figuring out ways to make one request look like it's not the same length, and using that for SSRF or XSS or various other attacks. But *ANY DIFFERENCE IN THE PARSERS* leads to critical level attacks.

The surface of LangSec analysis in distributed systems has only been scratched, so there are likely many more major vulnerabilities waiting to be discovered.

RF: LangSec seems to be heading in the direction of language-based designs, that is, requiring language to provide security assurances. Java was supposed to do this, but there are many Java exploits. Some exist because there are extensions to Java written in unsafe languages, like C. But I believe that people have exploited Java via the bytecode itself.

SB: LangSec targets the root causes of insecurity on a different level than efforts aimed at general-purpose programming languages.

Java and other memory-safe languages target the ability of the programmer to unwittingly (or deliberately) create memory corruption or (non-corrupting) type confusion. For Java and JavaScript, this ability was largely taken away from the developer, which is a net positive, but not a panacea.

The problem of unexpected and unchecked input remains. Now these inputs are stored in memory-safe ways, but they are still not what the processing code expects, and they are still acted on. There is a lot of room in a general-purpose language to go wrong when acting on data that's not what the programmer expects. For programs such as web apps that produce outputs and issue commands, this problem will manifest as either the outputs or the commands not being as expected.

LangSec, by contrast, aims to offer general solutions that focus first and foremost on data languages, also called data formats.

Without a clear understanding of input and output data languages involved in a task, the programming language is only exchanging one bug class for another. For example, Java and JavaScript made memory corruption harder, although, as you note, not impossible. Still, regular programmers cannot accidentally corrupt memory with their code alone: it has to come from flaws in the language runtime implementation or, more typically, from their interactions. However, complexities of data languages and their transformations immediately manifested themselves in XSS, command execution bugs, parser differentials, etc., making notionally memory-safe web apps notoriously vulnerable to an array of attacks much less sophisticated than memory corruption exploits.

Note that outputs and the code that creates them ("unparsers") are as important as the inputs and their handling code: see, for example, [9] and the first workshop paper [1].

My understanding is that Google and Facebook had to integrate intricate type systems with their web development tool chains to just keep a lid on this problem, and their solutions are specialized to their respective processes.

LangSec absolutely takes to heart the dictum of functional programming: "Make illegal state unrepresentable." This dictum calls on a language designer or an API architect to construct the language or the API so as to make it impossible for the programmer to create illegal state—at least not without the compiler complaining very loudly. However, properly implementing this dictum wherever inputs or outputs are involved requires understanding what are the legal and illegal states of input, and the same for output. It requires LangSec.

RF: When I interviewed Natalie Silvanovich [10], she seemed to conflate the use of dynamic languages (those that handle memory allocation and freeing dynamically, like Rust and Go) as part of LangSec. What do you think?

SB: I'd like to start by saying that LangSec greatly benefited from interest and feedback from extraordinary vulnerability researchers, who were, in fact, among the first to grasp its practical value. For example, the closing keynote of the first LangSec workshop was by Felix "FX" Lindner, an early supporter of LangSec. This makes perfect sense, because leading vulnerability researchers see general patterns of software weaknesses, of input-driven exploitation, and of how its non-systematic mitigations fail. LangSec offered a unified and actionable way of explaining these patterns, and top vulnerability researchers were among the first to appreciate it.

In your interview, Natalie's take on the nature and scope of LangSec is spot-on:

> [LangSec] views the root cause of security issues to be that most protocols and other input formats are poorly defined and often have many undefined states, and the programming languages that process them also support a huge amount of undefined behavior. [LangSec] thinks all software should abstract out all input processing code, and design and implement it in a way that is verifiable, and has no undefined states or behavior.

As I mentioned earlier, and as Natalie notes, the common idea of managed-memory languages is to make illegal memory states impossible for the programmer to unwittingly create while writing regular code. Notably, LangSec aims further than basic memory corruption. Indeed, there are numerous examples of memory-safe software with deep flaws due to ad hoc handling of its input and output languages.

However, Natalie raised another important point in that interview: there are and will be bugs in programming languages and environments intended to be memory-safe or otherwise offer safety assurances. In this year's LangSec workshop's amazing invited talk, Natalie connected this insight with specific features of JavaScript that have been causing huge headaches worldwide, given how JavaScript has been "eating the Internet"—and pinpointed the ways out. See her slides at [1] for the discussion of these troublesome features. Natalie has a wonderful intuition here, which is entirely LangSec but takes us beyond file and message formats.

I would describe it as follows: Natalie sees data structures allocated in memory as data languages, with the runtime memory management code servicing these structures as parsers. Programming language feature choices made by JavaScript or Go about what kinds of objects and how their relationships are representable in the language force the implementations of these languages to handle ever more complex data languages of bytes in memory: for example, on the heap. Consequently, unnecessary complexity of these features causes the same devastating effects as unnecessary format complexity does on the software that processes the formats.

Any piece of the language's native runtime, including the memory manager and garbage collector, parses memory bytes all the time and often must decide if a chunk it parses is valid or not before it acts. Moreover, advanced memory management means that multiple actors read and write memory concurrently, and their parsing actions must all be synchronized, or else corruption occurs. There is a rich literature of hacker research here, including many nifty attacks on browsers and OS kernels. This area is waiting to be explored from the LangSec perspective, and Natalie's invited talk pointed out a very rich example.

RF: You've mentioned that language-based approaches could turn out to be amazingly productive in understanding routing. Can you explain how LangSec intersects with network routing?

SB: Routing and other network-processing tasks must process streams of packets or, at a higher level, events. These packets or events change the internal state of the receiving program. Essentially, just like a parser, a network stack or function performs input-driven computation. Many questions about routing come down to modeling and understanding this computation, and assuring that it is safe—that is, behaves predictably for all inputs it might receive.

With modern verification tools we can try to prove that a distributed system has some desired behavioral properties. But which properties and models are tractable to explore?

It turns out that thinking about sequences of networking events as a data language that drives language-processing tasks is surprisingly productive for reasoning about and verifying network router behaviors. Not only that, but understanding the routers' many configuration options as dialects of a common language was also an efficient way of organizing and searching the space of diverse configurations. The latter is arguably less surprising, because human designers of these spaces, as all humans, are creatures of language and tend to implicitly impose language-based ordering on complex spaces.

This was the subject of this year's workshop's morning keynote by Princeton's David Walker [1]. Of course, as the original LangSec paper [11] points out, treating observable system and network events as streams processed by input-driven automata predates LangSec. For example, Fred Schneider used this approach to characterize classes of enforceable security policies [12] and cited Lamport's prior work. However, it's still fascinating that formal language-based approaches are so productive far beyond parsing.

RF: Forms of distributed computing, such as cloud functions, are growing in popularity today. Cloud functions use RPCs and queues to communicate, and that seems to me to be an opportunity to either make things better by observing LangSec or much worse through the use of ambiguous protocols. Would you comment on that?

SB: This is very much the case: there is both the opportunity and the danger.

The danger is already manifesting itself in the surge of high-impact parser differential bugs. Recall Dave Aitel's quote above. Note that we don't yet have effective ways of fuzzing for parser differentials. So we are in a much worse position with respect to parser differential bugs than we are with regard to memory corruption bugs, where coverage-driven fuzzing in combination with various sanitizers have gotten really good.

There is also the opportunity. Exposing interfaces without the false comfort of keeping them "private" and only receiving well-formed data or only data from one particular writer applies evolutionary pressure towards properly defining these interfaces. LangSec is there as a natural match for this problem.

The story of the Amazon API Mandate as told by Steve Yegge [13] is the story of such evolutionary pressure creating a qualitatively better platform. From the LangSec perspective, this story is not surprising—it is an iconic story of the correct intuition.

RPC messages are explicitly data languages, and open cloud environments will exert pressure to validate RPC messages before acting on them. However, it is important to get the design of these data languages right, so that validating these inputs doesn't grow into intractable problems we encounter with legacy formats.

As cloud systems grow rapidly, so could their technical debt. For example, for many application protocols, their expressions in Protocol Buffers happen to be the closest they ever got to a mechanized specification. However, these specifications themselves may be ambiguous and vulnerable to parser differentials. Critiques such as [14] strongly urge caution.

These problems are going to be very important as we move to serverless styles of programming (AWS Lambda and Fargate, Azure Functions, etc.). They will take a while to explore and understand, just like understanding the significance of parser differentials took almost a decade, but to avoid accumulating insurmountable amounts of technical debt, we should start now.

RF: The Rust programming language claims to offer unprecedented security assurances in systems programming. Rust's secret weapon appears to be lightweight memory safety through compiler-imposed isolation, instead of having to rely on much more expensive safety solutions such as separating memory contexts with x86 hardware privilege rings or automatic memory management. Will LangSec remain relevant if Rust becomes the choice of systems programmers?

SB: The point of all programming language safety features, be it Java-like automatic memory management or Rust's type system that enforces a discipline on pointers, is to avoid unintended state and, as a result of that state, unintended execution from that state onward. The difference between the languages and approaches is what kind of unintended state is being prevented and how this is done.

Historically, it was very easy for a programmer to unwittingly create unintended state. Classic ISAs use contents of memory or registers as addresses to access memory "randomly," i.e., in arbitrary order and without checking what, if anything, was previously stored in that memory and when or how it got there. C/C++ exposed this indirect memory addressing through pointers, which could point practically anywhere and allowed nearly arbitrary arithmetic to be applied to them. Reasoning about code—for example, what the code would do on all inputs hitting a module's boundary—in the presence of arbitrary pointers is very hard (see Hind's 2001 survey [15]). The power of arbitrary indirect memory references is so great that it's possible to (re)compile any program into just x86 MOV instructions and a single JMP or an equivalent way of looping backwards [16], which is, of course, really bad news for program analysis.

Java approached this problem by abstracting away almost all indirect memory references, to heavily restrict what memory addresses the CPU might access on behalf of the program (notionally mediated by the JVM, but also observed by JIT-ed code). To do so, it took memory management away from the programmer, which made it less desirable for OS programming,

where managing memory is a significant part of the task, and a single automated way of doing it just does not fit all needs. Rust, via its type system, controls pointers in a different way, but for the same purpose: restrict where and when indirect memory references can point so that they become tractable, unlike C's pointers or assembly's indirect MOVs [16].

In each case, the language makes memory-corrupting references hard or impossible for the programmer to create in ordinary code. However, as we've seen with web programming, memory safety alone does not preclude abuse of complex interfaces, and can actually make exploiting these interfaces easier, because the attacker doesn't need to worry about crashing the system with a poorly crafted input. We often forget that memory safety without a clear understanding of what inputs and outputs are legal works both ways and can easily favor the attacker.

There is definitely a LangSec perspective on this: IPCs are data languages, and whatever Rust or any other compiler can do is all done for the purpose of consuming these languages safely and not letting them drive unintended computation in a module or microservice.

So the question is, once again: regardless of whatever kinds of checks can be done, what constitutes expected and valid IPC messages that, once validated, will cause only predictable system behaviors and no other "weird" behaviors? Can these expectations be precisely and unambiguously formulated and checked with tractable code, which could itself be checked for correctness?

Without a clear LangSec model of the inputs, validating IPC messages becomes an ill-defined game of guessing which kinds of memory corruption or command injection to mitigate, for example, by making the hardware explicitly protect some address ranges from access by all code except specially designated code parts (e.g., via x86 ring contexts). But what happens in other ranges and contexts? How can one guarantee that corruption spreading there would not trick a legitimately placed privileged ("ringed") deputy into corrupting the protected region by passing it some unexpected inputs? This is a really hard question to answer, and it needs higher-level models of intended input-driven behaviors.

So compilers and build environments in general should absolutely be doing more work to make sure only intended state occurs, and it's a great thing that they do.

LangSec, for its part, helps formulate what is and can be the intended, tractably checkable state when dealing with inputs, and helps system, protocol, and application designers avoid situations where ensuring predictability of input-handling code becomes unsolvable. So LangSec has a lot of work to do and many programming fields to help secure.

References

[1] The Sixth Workshop on Language-Theoretic Security (LangSec), at IEEE Security & Privacy (May 2020): http://spw20.langsec.org/workshop-program.html.

[2] J. Chen, V. Paxson, J. Jiang, "Composition Kills: A Case Study of Email Send Authentication," 29th USENIX Security Symposium (Security '20):https://www.usenix.org/conference/usenixsecurity20/presentation/chen-jianjun.

[3] HTTP Desync attacks: https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn.

[4] F5 vulnerability: https://research.nccgroup.com/2020/07/12/understanding-the-root-cause-of-f5-networks-k52145254-tmui-rce-vulnerability-cve-2020-5902/.

[5] "Psychic Paper" vulnerability: https://siguza.github.io/psychicpaper/.

[6] GitLab vulnerability: https://about.gitlab.com/blog/2020/03/30/how-to-exploit-parser-differentials/.

[7] Android Master Key vulnerabilities: http://www.saurik.com/id/17, http://www.saurik.com/id/18, and http://www.saurik.com/id/19.

[8] S. McQuistin, V. Band, D. Jacob, and C. Perkins, "Parsing Protocol Standards to Parse Standard Protocols," in *Proceedings of the Applied Networking Research Workshop (ANRW '20)*, pp. 25–31.

[9] L. Hermerschmidt, S. Kugelmann, and B. Rumpe, "Towards More Security in Data Exchange: Defining Unparsers with Context-Sensitive Encoders for Context-Free Grammars," in *2015 IEEE CS Security and Privacy Workshops*: pp. 134–141: http://spw15.langsec.org/papers.html#unparse.

[10] N. Silvanovich and R. Farrow, "Interview with Natalie Silvanovich," *;login:*, vol. 43. no. 2 (Summer 2020): https://www.usenix.org/publications/login/summer2020/farrow-0.

[11] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security Applications of Formal Language Theory," *IEEE Systems Journal,* vol. 7, no. 3 (September 2013), pp. 489–500.

[12] F. B. Schneider, "Enforceable Security Policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1 (February 2000), pp. 30–50: https://www.cs.cornell.edu/fbs/publications/EnfSecPols.pdf.

[13] S. Yegge, "Stevey's Google Platforms Rant," October 2011: https://gist.github.com/chitchcock/1281611.

[14] S. Maguire, "Protobuffers Are Wrong," Reasonably Polymorphic blog, October 10, 2018: https://reasonablypolymorphic.com/blog/protos-are-wrong/index.html.

[15] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*, pp. 54–61: https://courses.cs.washington.edu/courses/cse501/15sp/papers/hind.pdf.

[16] C. Domas, MOVfuscator: https://github.com/xoreaxeaxeax/movfuscator.

[17] D. Kaminsky, M.L. Patterson, and L. Sassaman, "PKI Layer Cake: New Collision Attacks Against the Global X.509 Onfrastructure," in *Proceedings of the 14th International Conference on Financial Cryptography and Data Security (FC 2010),* pp. 289–303: https://www.esat.kuleuven.be/cosic/publications/article-1432.pdf.

SYSTEMS

Characterization and Optimization of the Serverless Workload at a Large Cloud Provider

MOHAMMAD SHAHRAD, RODRIGO FONSECA, ÍÑIGO GOIRI, GOHAR CHAUDHRY, AND RICARDO BIANCHINI

Mohammad Shahrad is a Computer Science Lecturer at Princeton University and an incoming Assistant Professor of Electrical and Computer Engineering at the University of British Columbia. He received his PhD from Princeton University and his BSc from Sharif University of Technology. Dr. Shahrad's research aims to improve the efficiency of cloud computing systems through better resource management and enhanced system/architecture integration. mshahrad@ece.ubc.ca

Rodrigo Fonseca is a Principal Researcher at Microsoft Research and an Associate Professor at Brown University's CS Department. He is broadly interested in distributed systems, networking, and operating systems, and his current research involves ways to make cloud computing easier and more applicable for users and more efficient for providers. He holds a PhD from UC Berkeley, a MSc and BSc from Federal University of Minas Gerais, and is the recipient of an NSF CAREER award, an NSDI Test of Time Award, and a 2015 SOSP Best Paper Award. rfonseca@cs.brown.edu

Íñigo Goiri is a Research Software Developer at Microsoft Research. His current research focuses on the efficiency of large scale distributed systems. He holds a PhD from the University Politecnica de Catalunya (UPC). inigog@microsoft.com

Function as a Service (FaaS) has gained tremendous popularity as a way to deploy computations to serverless back ends in the cloud. We performed the first characterization of an entire production FaaS environment (Azure Functions) [1]. Our characterization revealed many unique aspects of serverless workloads compared to traditional cloud applications. Using this deep understanding, we designed a new dynamic resource management policy to improve the performance and reduce the memory footprint of serverless workloads. This new policy is now deployed in production, and our characterization data traces are publicly released for researchers.

Serverless characterization studies before our work can be classified into two main categories: those probing public serverless offerings externally and those looking at ways developers use FaaS offerings by investigating public repositories. These two classes of studies provide valuable information; external probing allows comparing the performance and availability of various FaaS providers using a set of benchmarks, and looking at public FaaS repositories allows finding popular programming trends. However, neither of them can offer insights on the aggregate workload seen by a provider. Only when the entire workload is known can one answer questions such as *"How often do functions get invoked?" "How long do functions execute for?"* or *"How much memory do serverless functions require?"* Answers to such basic questions have major implications for designing various components of serverless systems—from schedulers to virtualization environments to underlying hardware architectures.

We conducted the first detailed characterization of an entire production FaaS workload at a large cloud provider. To do so, we collected data on all function invocations across Microsoft Azure's entire infrastructure between July 15 and July 28, 2019. We invite the reader to read our recent USENIX ATC paper for methodology details and full characterization data [1]. The sanitized traces from a subset of our characterization data are also available publicly at https://github.com/Azure/AzurePublicDataset. In what follows, we summarize some of our characterization insights.

Composition of Applications

In Azure Functions, functions are grouped into applications. The application concept helps organize the software, and the application is the unit of scheduling and resource allocation. As shown in Figure 1, 54% of the applications have only one function, and 95% of the applications have at most 10 functions. The other two curves show the fraction of invocations and functions corresponding to applications with up to a certain number of functions. For example, we see that 50% of the invocations come from applications with at most three functions, and 50% of the functions are part of applications with at most six functions.

Composition of Triggers

Functions can be invoked in response to several event types, called triggers. Figure 2 shows the fraction of all functions and invocations per type of trigger. HTTP is the most popular in both dimensions. Event triggers correspond to only 2.2% of the functions, but they correspond to 24.7% of the invocations due to their automated, and very high, invocation rates. Queue triggers also have proportionally more invocations than functions (33.5% vs. 15.2%).

Gohar Irfan Chaudhry is a Research Software Engineer at Microsoft Research. He is part of the Systems Research Group and is currently working on improving efficiency of serverless infrastructure. Gohar.Irfan@microsoft.com

Ricardo Bianchini received his PhD in computer science from the University of Rochester. He is currently a Distinguished Engineer at Microsoft, where he leads efforts to improve the efficiency of the company's online services and datacenters. He also manages the Systems Research Group at Microsoft Research in Redmond. His main research interests include cloud computing, datacenter efficiency, and leveraging machine learning to improve systems. He is an ACM Fellow and an IEEE Fellow. ricardob@microsoft.com

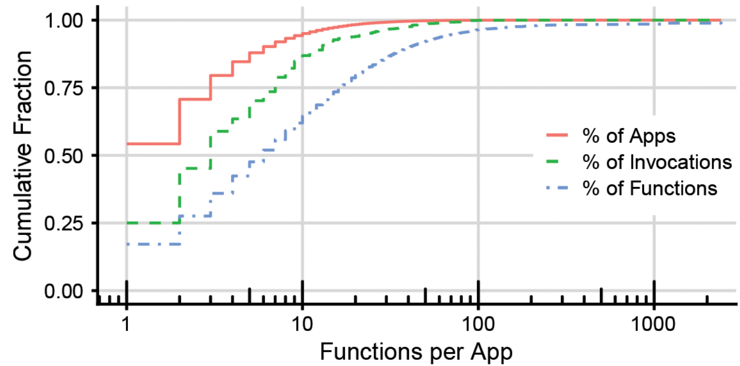


Figure 1: Distribution of function counts per application

The opposite happens with timer triggers. There are many functions triggered by timers (15.6%), but they correspond to only 2% of the invocations, due to their relatively low firing rate: 95% of the timer-triggered functions in our data set were triggered at most once per minute, on average.

Invocation Patterns

We observed that applications are invoked very differently. The number of invocations per day varies by over eight orders of magnitude for different applications. Another observation with strong implications for resource allocation is that the vast majority of applications and functions are invoked, on average, very infrequently: on average, 45% of the applications are invoked once per hour or less frequently, and 81% of the applications are invoked once per minute or less. The other side of this skewness was revealed to us by finding that the top 18.6% most popular applications represent 99.6% of all function invocations. Thus, keeping the applications that receive infrequent invocations resident in memory at all times is expensive.

Function Execution Times

An advantage of the serverless model is that users pay only for their execution time. Figure 3 shows the distribution of average, minimum, and maximum execution times of all function executions on July 15, 2019, which is similar to other days. We observed that 50% of the functions execute for less than 1 sec on average, and 96% of functions take less than 60 sec on average. These short executions in FaaS are unlike virtual machines (VMs). For example, a prior study reported that 63% of all VM allocations on Azure last longer than 15 minutes [2].

Trigger	%Functions	%Invocations
HTTP	55.0	35.9
Queue	15.2	33.5
Event	2.2	24.7
Orchestration	6.9	2.3
Timer	15.6	2.0
Storage	2.8	0.7
Others	2.2	1.0

Figure 2: Functions and invocations per trigger type

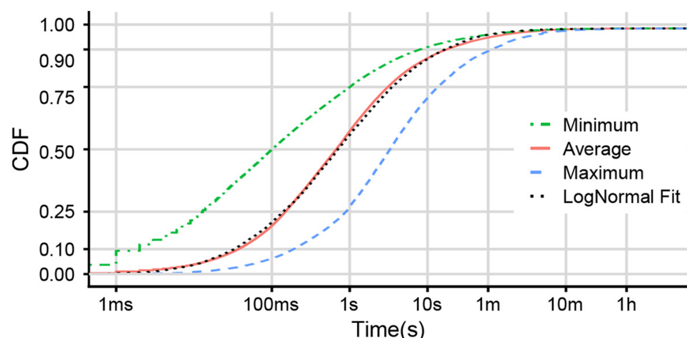


Figure 3: Distribution of function execution times

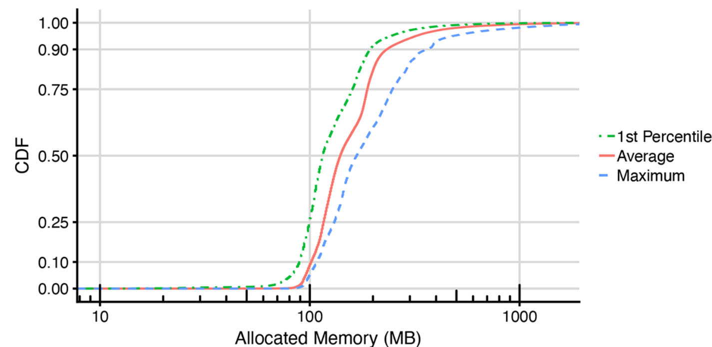


Figure 4: Distribution of allocated memory per application

FaaS applications experience cold starts. A cold start invocation occurs when a function is triggered, but its application is not yet loaded in memory. When this happens, the platform instantiates a worker for the application, loads all the required runtime and libraries, and calls the function. While Figure 3 does not include cold starts, we observed that the execution times from our characterization are the same order of magnitude as the cold start times reported for major providers [3]. Therefore, optimizing cold starts becomes extremely important for the overall performance of a FaaS offering. This can be done either by reducing the cold start latency [4, 5] or by eliminating cold starts. We took the second approach in designing our policy, which we describe later in the article.

Memory Usage

The memory demand of applications on the same day (July 15, 2019) is shown in Figure 4. Looking at the distribution of the maximum allocated memory, 90% of the applications never consume more than 400 MB, and 50% of the applications allocate at most 170 MB. We found no strong correlation between invocation frequency and memory allocation or between memory allocation and function execution times.

Designing a New Adaptive Resource Management Policy

One of our primary goals in understanding workload characteristics was to design better resource management policies. This is because the state-of-the-art in serverless resource management was too simplistic, where each application was kept in memory after function execution for a fixed amount of time. This keep-alive window is 10 minutes for AWS Lambda and IBM Cloud Functions, and was 20 minutes for Azure Functions. Such a policy is too rigid for the wide range of serverless applications. Developers usually circumvent this by creating regular artificial invocations to make sure their applications remain warm in memory. A smart dynamic policy can eliminate such a burden. Additionally, adapting to applications' invocation patterns would mean resources are not kept unused just to keep function images warm without executing them.

There are a few challenges in designing such a policy. As we showed earlier in this article, invocation frequency and pattern vary substantially across applications. A one-size-fits-all fixed policy is certain to be a poor choice for many applications. Adapting the policy to each application means tracking each application individually, and thus the cost to track the information for each application should be small. Finally, since function executions can be very short (i.e., more than 50% of executions take less than one second), running the policy and updating its state need to be fast. This is especially critical considering providers charge users only during their function execution times (e.g., based on CPU, memory). For instance, we cannot take 100 ms to update a policy prediction model for each 10 ms-long execution.

We propose a *hybrid histogram policy* that addresses all the above challenges. It identifies each application's invocation pattern, removes/unloads the application right after each function execution ends, reloads/pre-warms the application right before a potential next invocation, and keeps it alive for a period. The policy does so by capturing the history and predicting next idle times (ITs), defined as the time between the end of a function's execution and its next invocation. Three main components of the *hybrid histogram policy* include: (1) a range-limited histogram for capturing each application's ITs; (2) a standard keep-alive approach for when the histogram is not representative, i.e., there are too few ITs or the IT behavior is changing (again, note that this differs from a fixed keep-alive policy); and (3) a time-series forecast component for when the histogram does not capture most ITs.

Compared to fixed keep-alive policies, *hybrid histogram policies* are closer to optimal. As seen in Figure 5, hybrid policies deliver a significant reduction of unused memory time, while considerably improving the cold start percentage for applications. For instance, a hybrid policy with a four-hour histogram can deliver a 2.5× lower 3rd-quartile cold start percentage and 1.5× less memory time wastage compared to a fixed 10-minute keep-alive policy. Note that there is a tradeoff between cold starts and wasted memory time for both policy families, but hybrid substantially dominates all fixed policies.

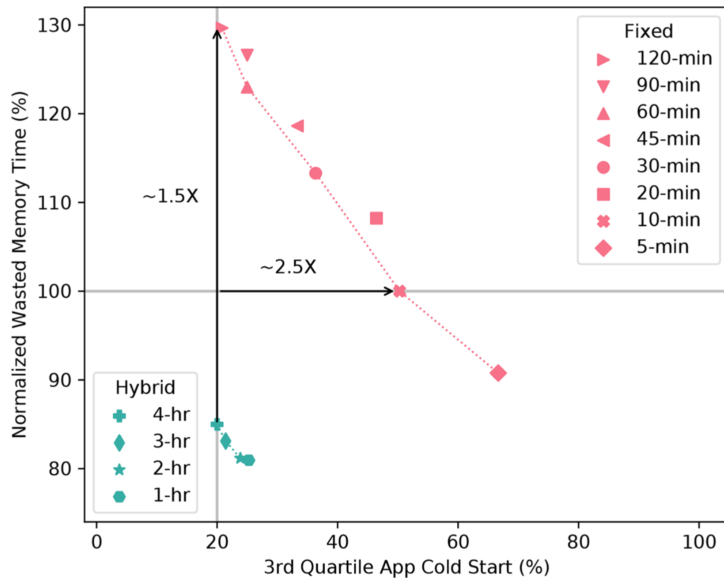


Figure 5: Tradeoff between cold starts and wasted memory time for the fixed keep-alive policy and our hybrid policy

The range-limited histogram at the core of the *hybrid histogram policy* is a lightweight data structure. We use it with a minute-long resolution, which means capturing a four-hour histogram requires an array of length 240. The other two components of the *hybrid histogram policy* complement it to boost performance while maintaining low overhead. Here, we describe some of our design choices and their implications for the policy:

- **Pre-warming to curtail keep-alive values while maintaining low cold starts**: One can eliminate cold starts by just setting the right keep-alive values, but this approach is too costly. Pre-warming allowed us to reduce memory wastage by about 34% compared to using just keep-alives, with a minor cold start increase.
- **Ignoring outlier ITs to deflate keep-alive values**: To exclude outliers of the IT distribution captured by the histogram, we use the 5th- and 99th-percentiles as head and tail cutoffs, respectively. This approach avoided the inflation of keep-alive values and resulted in a ~15% reduction in memory time wastage with a negligible impact of cold start performance of applications.
- **Checking the histogram representativeness to not use it prematurely**: The histogram might not be representative of an application's behavior when it has not observed enough ITs for the application or when the application is transitioning to a different IT regime. We decide whether a histogram is representative by computing the coefficient of variation (CV) of its bin counts and comparing it to a threshold (CV=2). This simple approach improved the 3rd-quartile application cold starts by nearly 49% with only a 3% increase in memory time wastage.
- **Using time-series forecast to eliminate cold starts of infrequent applications**: Using time-series forecast for infrequent applications reduced the percentage of applications that experience 100% cold starts by about 50%, i.e., from 10.5% to 5.2% of all applications. A significant portion of these applications have only one invocation during the entire week, and no predictive model can help them. Excluding these applications, the same reduction becomes 75%, i.e., from 6.9% to 1.7% of all applications.

We implemented our policy in Apache OpenWhisk [6], which is the open-source FaaS platform powering IBM's Cloud Functions. We refer the reader to our paper for implementation details [1]. We ran two experiments with 68 randomly selected mid-range popularity applications from our workload on our 19-VM OpenWhisk deployment: one experiment with the default 10-minute fixed keep-alive policy of OpenWhisk and another with our hybrid policy and a four-hour histogram range. Each experiment ran for eight hours with a total of 12,383 function invocations. We used FaaSProfiler [7] to automate trace replay and result analysis.

Figure 6 compares the cold start distribution of keep-alive and hybrid policies from the simulations (left) and the OpenWhisk prototype (right). As seen, the significant cold start reductions follow similar trends. On average and across the 18 invoker VMs, the hybrid policy reduced memory consumption of worker containers by 15.6%, which was also consistent with our simulation results. Moreover, hybrid policy reduced the average and 99-percentile function execution time 32.5% and 82.4%, respectively, due to a secondary effect in OpenWhisk, where the language runtime bootstrap time is eliminated for warm containers. The price for all of these is an additional 835.7μs latency on average, which is negligible compared to the existing latency of OpenWhisk components: the (in-memory) language runtime initiation takes O(10 ms) and the container initiation takes O(100 ms) for cold containers [7].

After getting promising results from simulations as well as the prototype implementation, we implemented our policy in Azure Functions for HTTP-triggered applications. Its main elements have rolled out to production. We used asynchronous updates from the workers to the Azure Functions controller to populate histograms. We keep the histogram in memory and do hourly backups to the database. We start a new histogram per day in the database so that we can track changes in an application's invocation pattern and remove histograms older than two weeks. When an application changes state from executing to idle, we use the aggregated histogram to compute its pre-warm interval and schedule an event for that time (minus 90 seconds). Pre-warming loads function dependencies and performs JIT where applicable. Each worker maintains the keep-alive duration separately, depending on how long it has been idle. We make all policy decisions asynchronously, off the critical path, to minimize the latency impact on the invocation.

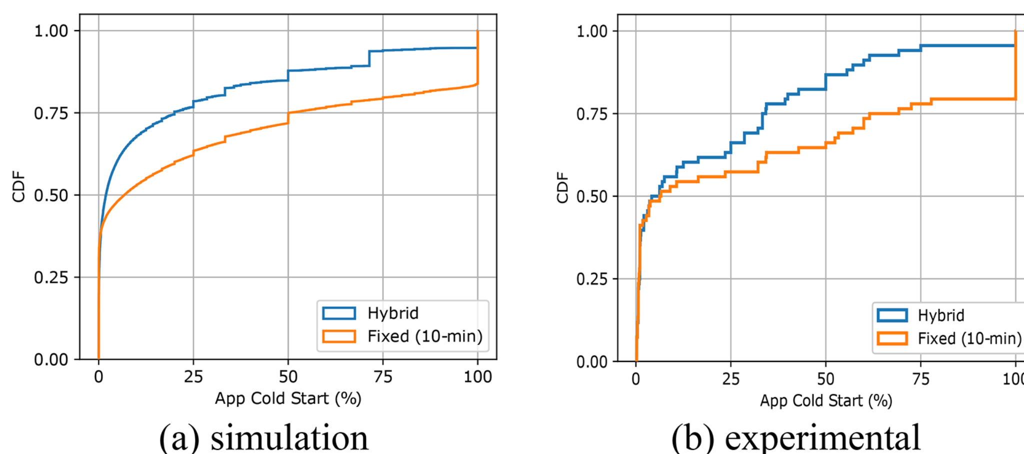


Figure 6: Cold start behavior of fixed keep-alive and hybrid policies in (a) simulation results and (b) experimental results from our OpenWhisk implementation

Conclusion

We characterized the entire production FaaS workload of Azure Functions, which unearthed several key observations for cold start and resource management. Based on them, we proposed a practical policy for reducing the number of cold starts at a low resource cost. The main elements of this policy have rolled out to production. We also released sanitized traces from a subset of our characterization data that is first of its kind. These traces will help researchers design future serverless systems based on realistic workloads and enable new research angles.

References

[1] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pp. 205–218: https://www.usenix.org/system/files/atc20-shahrad.pdf.

[2] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 153–167.

[3] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pp. 133–145: https://www.usenix.org/system/files/conference/atc18/atc18-wang-liang.pdf.

[4] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pp. 55–70: https://www.usenix.org/system/files/conference/atc18/atc18-oakes.pdf.

[5] K. Wang, R. Ho, and P. Wu, "Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment," in *Proceedings of the 14th EuroSys Conference 2019*, pp. 1–16.

[6] Apache OpenWhisk, Open Source Serverless Cloud Platform: https://openwhisk.apache.org/.

[7] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*, pp. 1063–1075.

Posh: A Data-Aware Shell

DEEPTI RAGHAVAN, SADJAD FOULADI, PHILIP LEVIS, AND MATEI ZAHARIA

Deepti Raghavan is a PhD candidate in computer science at Stanford University, advised by Phil Levis and Matei Zaharia. She focuses on topics in networking and distributed systems. She is interested in optimizing data processing for networked applications. deeptir@cs.stanford.edu

Sadjad Fouladi is a PhD candidate in computer science at Stanford University, working with Keith Winstein on topics in networking, video systems, and distributed computing. His current projects include general-purpose lambda computing and massively parallel ray-tracing systems. sadjad@cs.stanford.edu

Philip Alexander Levis is an Associate Professor of Computer Science and Electrical Engineering at Stanford University, where he heads the Stanford Information Networks Group (SING) and co-directs Lab64, Stanford's electrical engineering maker space. He has a self-destructive aversion to low-hanging fruit and a deep appreciation for excellent engineering. pal@cs.stanford.edu

Running I/O-intensive shell pipelines over the network requires transferring huge amounts of data but relatively little computation. We present Posh, a shell framework that accelerates unmodified shell workflows over networked storage by offloading computation to proxy servers closer to the data. Posh provides speedups ranging from 1.6× to 15× compared to bash over NFS for a wide range of applications.

The UNIX shell is a linchpin in computing systems and workflows. Developers use shell tools not only for data processing with core utilities such as `sort`, `head`, `cat`, and `grep`, but also for programs such as Git, ImageMagick, and FFmpeg. The UNIX shell was designed in a time dominated by local and then LAN storage when file access was limited by disk access times, so networked storage was an acceptable tradeoff. Today, solid-state disks have reduced access times by orders of magnitudes, while networked attached storage remains popular.

Running I/O-intensive shell pipelines over networked storage incurs high overheads. Consider generating a `tar` archive on NFS. The `tar` utility copies the source files and adds a small amount of metadata: the server reads blocks and sends them over a network to a client, which shifts their offsets and sends them back. NFS mitigates this problem by offering compound operations and server-side support for primitive commands such as `cp`, but even something as simple as `tar` requires large network transfers.

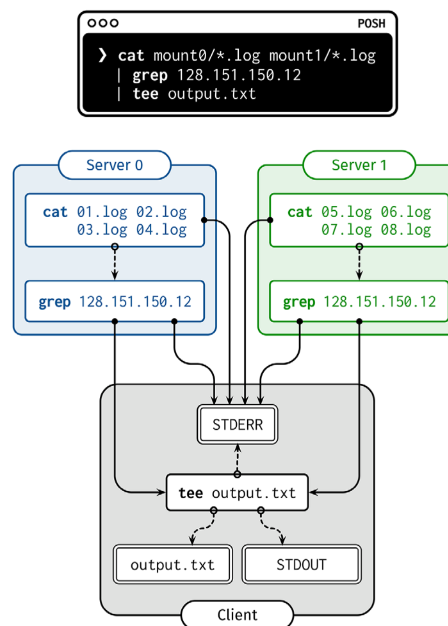
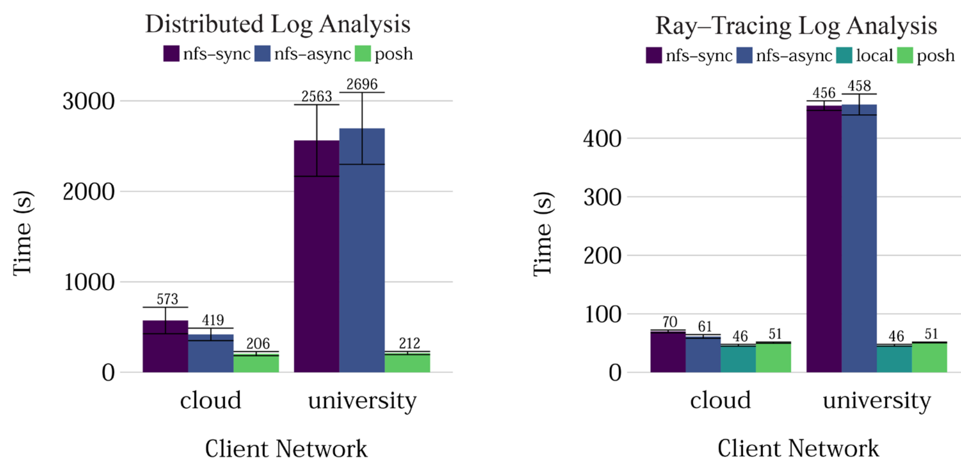


Figure 1: Users can type in unmodified shell workflows to Posh's shell prompt. Posh will transparently schedule and execute individual commands on remote proxy servers closer to the remote data but ensure the entire workflow retains local execution semantics.

Matei Zaharia is an Assistant Professor of Computer Science at Stanford and Chief Technologist at Databricks. He started the Apache Spark project during his PhD at UC Berkeley and has worked on other widely used data analytics and AI software, including MLflow and Delta Lake. At Stanford, he is co-PI of the DAWN lab working on infrastructure for machine learning. Matei's research was recognized through the 2014 ACM Doctoral Dissertation Award, an NSF CAREER Award, and the US PECASE award. matei@cs.stanford.edu



Figures 2 and 3: End-to-end latency of Posh on two applications, compared to NFS sync, NFS async, and local execution time for two networks, one where the client is in a university network and one where the client is in the same GCP region as the storage server. The Posh proxy runs directly on the NFS server. Posh provides between 1.6–12.7× speedups in the university-to-cloud network compared to NFS.

The underlying performance problem of using the shell with remote data is locality: because the shell executes locally, it must move large amounts of data to and from remote servers. Data movement is usually the most expensive (time and energy) part of a computation, and shell workloads are no exception. Near-data processing [1] is not a new paradigm: systems such as Spark [2], Active-Disks [3], and stored procedures in databases all move computation closer to the data. However, these systems require applications to use their APIs: they can supplement but not replace shell pipelines.

To address the shell performance problem of data locality, this article presents Posh, the "Process Offload Shell," a system that offloads portions of *unmodified* shell workflows to proxy servers closer to the data. A proxy server can run on the actual remote file server storing the data, or on a different node that is much closer to the data (e.g., within the same datacenter) than the client. Posh identifies parts of shell pipelines that can be safely offloaded to a proxy server and selects which candidates run on a proxy in order to minimize data movement. It then distributes computation across an underlying runtime while maintaining the exact output semantics expected by a local program. Figure 1 shows running a workflow via Posh. The user enters the unmodified workflow at the shell prompt and the output appears at the client's shell as normal, but Posh offloads some of the commands.

Posh is available at https://github.com/deeptir18/posh. This article will cover examples of shell workflows where Posh can be useful, a brief overview of the core ideas behind Posh, and how to get started with the system. For a detailed discussion of the research ideas behind Posh, we refer the reader to our USENIX ATC '20 paper [4].

Examples of Posh

Posh is useful for shell workflows that are I/O bound, have smaller output than input size, are metadata heavy (make many file-system `stat()` requests), or are parallelizable. In this section, we will discuss examples of shell workflows that incur large overheads over networked storage and show that Posh accelerates them to achieve near-local execution time. Figures 2–4 illustrate the execution time of running each of these applications with an NFS mount configured with either `sync` and `async`, and with Posh, over two network settings: one where the client is in the same GCP region as the storage server ("cloud") and one where the client is in a university network outside the datacenter ("university"). Posh can offload computation

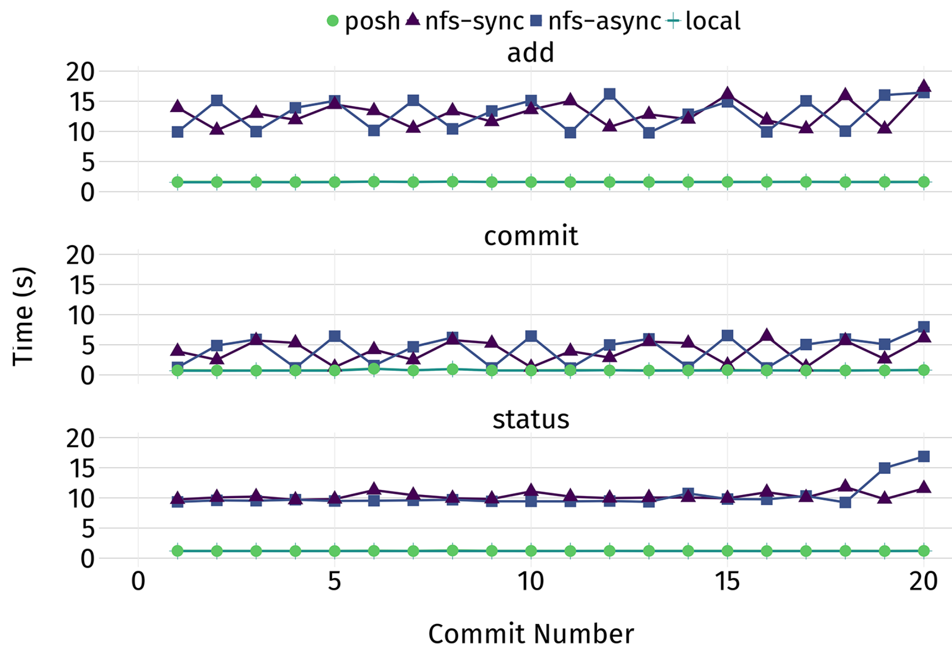


Figure 4: Average latency of 20 `git status`, `git add`, and `git commit` commands run on Chromium repo, of Posh compared to NFS and local execution, for a client in the same cloud datacenter as the storage server. Posh provides up to 10–15× speedups by preventing round trips for file system metadata calls.

to a proxy server directly running at the NFS servers. Figures 3 and 4 additionally include a baseline that demonstrates local execution time, where the data is stored on a local SSD. Compared to bash over NFS, Posh sees a 1.6–12.7× speedup in the execution time of these applications.

For each of these applications, the shell workflow (the bash script) itself is *completely unmodified;* the workload is just run within a Posh shell environment. Posh can accelerate these workflows because the shell knows metadata about the commonly used shell commands within these workflows, which we will discuss in the next section. We describe each workflow in turn.

Distributed Log Analysis (Figure 2)

This application is based on a workflow where system administrators run analysis on 80 GB of input logs split across five *different storage servers*, to search for an IP address within these logs. The workflow runs `cat` over all of the files and filters for a particular IP with `grep` and then writes the final results, only about 0.8 KB of data, back to a file stored locally at the client. Posh splits the computation across the five machines and aggregates the output in the correct order. By offloading and parallelizing, Posh improves the runtime by 12.7× in the university-to-cloud setting and by 2× in the cloud-to-cloud setting.

Ray-Tracing Log Analysis (Figure 3)

This workflow analyzes the logs of a massively distributed research ray-tracing (computer graphics) system [5] to track a ray (a simulated ray of light) through the workers it traversed. The analysis first cleans and aggregates each worker's log, 6 GB in total, into one 4 GB file. It then runs `sed` to search for the path of a single ray (e.g., a straggler) across all the workers and stores the output on a file at the client:

```
cat logs/1.INFO | grep "\[RAY\]" | head -n1 | cut -c 7- > \
   logs/rays.csv
cat logs/*.INFO | grep "\[RAY\]" | grep -v pathID | \
   cut -c 7- >> logs/rays.csv
cat logs/rays.csv | sed -n '/^590432,/p' > local/output.log
```

The output of `sed` is much smaller than the 10 GB of data processed. This application is a best-case workload for Posh: it is I/O bound and can be parallelized, and the output is a tiny fraction of the data it reads. Posh achieves an 8× improvement on the university-to-cloud network and no improvement on the cloud-to-cloud network: Posh offloads all the computation and only needs to stream the output of `sed` back to the client. However, the data movement overhead only matters in the university-to-cloud setting, where the network connection is slower.

Git Workflow (Figure 4)

This application imitates a developer's `git` workflow over the Chromium repository. After rolling back the repository by 20 commits and saving each commit's patch, the workload successively applies each patch and runs three `git` commands: `git status`, `git add` and `git commit -m`. Figure 4 shows the latency of each command for each of the 20 commits. These commands are extremely metadata-heavy: commands like `status` and `add` check the status of every file in the repository to see if it has been

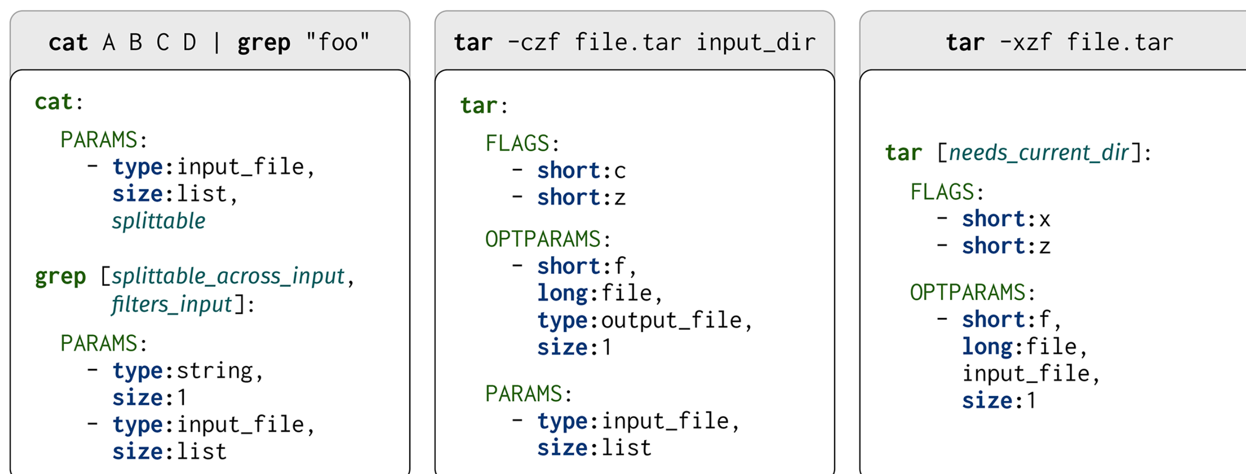


Figure 5: Example annotations for `cat`, `grep`, and `tar`. Most of the information in the annotations tells Posh information about the possible arguments for each command and their syntax. They contain type assignments for each argument, which tell Posh how the argument will be used as well as other information used for scheduling and automatic parallelization. `tar` requires more than one annotation because `tar -x` and `tar -c` invocations have conflicting type semantics: `-f` is an `input_file` in one case and an `output_file` in the other.

modified. When run over a networked file system, this incurs many round trips. In the cloud-to-cloud setting, this causes Posh to achieve 10–15× improvement over NFS. Running `git status` took up to two hours in the university-to-cloud setting, so we omitted this network for this application.

To enable Posh's acceleration of a shell workload, the user must provide metadata about the individual shell commands the workflow uses. This metadata, called *annotations*, allows Posh to determine which files these commands access, so it can further schedule the workflow across the underlying runtime. The next section will discuss annotations in more detail.

Transparently Offloading Shell Computation: Annotations

Annotations summarize information to Posh about individual shell commands, such as `tar`, `cat`, or `grep`. Posh's key insight is that many shell workflows only read and write to files specified in their command-line invocation, so Posh can deduce which files a workflow accesses by understanding which arguments correspond to files. Annotations contain a list of possible arguments and whether they correspond to files, so Posh can understand which files an arbitrary invocation of a command would access. Additionally, annotations contain information relevant to scheduling the workflow.

Consider a simple pipeline:

```
cat A B C D | grep "foo" | tee local_file.txt
```

Posh could try to offload any of the three commands: `cat`, `grep`, or `tee`. Posh must understand which files (if any) each command accesses and where these files live, so Posh must determine which arguments to the three commands represent file paths.

However, outside of the program, all of these arguments are seen as generic strings. For example, consider the following four commands:

```
cat A B C D | grep "foo"
tar -cvf output.tar.gz input/
tar -xvf input.tar.gz
git status
```

The `cat` command takes in four input files, while the argument to `grep` is a string. The second command, `tar -cvf`, takes an output file argument preceded by `-f`, followed by an input file argument not preceded by a short option. The third command, also `tar`, takes an input file argument preceded by `-f` and implicitly takes its output argument as the current directory. Finally, `git` also implicitly relies on the current directory as a dependency.

Secondly, in order to produce an execution schedule that reduces data movement, Posh must understand the relationship between the inputs and outputs of a command. In the `cat | grep` example, if the argument to `cat` is a remote file, to minimize data movement, Posh can offload both `cat` and `grep` since `grep` filters its input. Finally, for applications like the distributed log analysis application discussed previously, where the input files for a command live on different mounts, Posh needs to know how to safely parallelize the command in order to be able to offload it at all. However, parallelization is not safe for all commands: `wc`, for example, "reduces" the input, as opposed to commands like `cat` or `grep`, which merely map over the input. Posh's annotations summarize file dependencies, data movement semantics, and parallelization semantics for commonly used commands.

Figure 5 shows examples of annotations, for `cat`, `grep`, and `tar`. Most of the information in the annotations summarize the semantics for the arguments for each command, or information

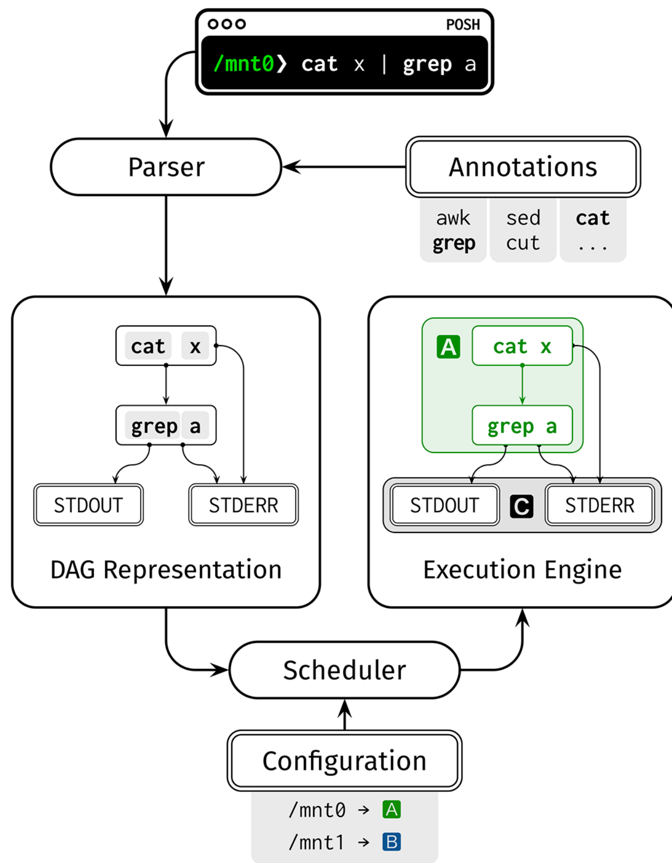


Figure 6: In Posh's main workflow, a shell command is passed to the parser, which uses the annotations to generate and schedule a DAG representation of the command. The DAG includes which machine—A, B, or C (client) here—to run each command on. The execution engine finally runs the resulting DAG.

that is summarized in the documentation pages for these commands. Moreover, they contain a type assignment for each argument: `input_file`, `output_file`, or `string`. For `cat`, the `splittable` keyword indicates to Posh that `cat` can be split in a data parallel way across its arguments, as long as the outputs are stitched together in the correct order. For `grep`, the `splittable_across_input` keyword indicates that `grep` can be parallelized across its standard input. As mentioned before, the `-f` argument indicates an `input_file` for a `tar -x` invocation but an `output_file` for a `tar -c` invocation. To resolve this, Posh allows multiple annotations per command, per type of invocation, and tries each until it finds an annotation that matches the current command invocation.

We envision that developers can share annotations for popular commands, so users do not necessarily need to write their own annotations. These annotations are inspired by recent proposals to annotate library function calls for automatic pipelining and parallelization [6]. Please see our research paper [4] for a more detailed overview of the Posh annotation interface.

Distributed Scheduling and Execution

This section briefly explains how Posh uses the annotations to schedule and execute shell workflows, summarized in Figure 6. The Posh parser turns each pipeline (each line of a shell workflow, potentially consisting of several commands combined by pipes and redirects) into a directed acyclic graph (DAG). This graph represents the input-output relationship between commands, the standard I/O streams (stdin, stdout, and stderr), and redirection targets. Posh then parses each individual command and its arguments using the corresponding annotation and completes the DAG by including additional input and output dependencies of the pipeline. The parser finally runs a greedy scheduling algorithm on the DAG and assigns an execution location to each command in the pipeline. In order to do this, the parser requires extra configuration information that specifies a mapping between each mounted client directory and the address for a machine running a proxy server for the corresponding directory. Our research paper [4] contains more details on the scheduling algorithm.

Getting Started with Posh

This section details the steps to running and using Posh.

0. Running the Posh servers

The administrator who controls the proxy server must run the Posh server binary, which allows it to receive requests to offload computation on behalf of a single remote file-system mount. The proxy server just needs read and write access to this folder; it need not run at the storage server itself. Invoking the server, shown below, requires specifying the absolute path for the mount being accessed and a temporary directory for writing the output of intermediate computation.

```
admin@~$ $POSH_SRC/target/release/server --folder /mnt/logs \
  --tmpfile /tmp/posh
```

1. Posh client configuration

The client needs to provide a file that contains annotations for any commands the client wants to accelerate. It must also have a list of proxy servers associated with client file-system mounts. The configuration file, shown below, maps IP addresses to the corresponding mount, written as an absolute path.

```
mounts:
      "255.255.255.0": "/home/user/remote_mount1"
      "255.255.255.1": "/home/user/remote_mount2"
```

2. Running the client shell

Posh provides two client binaries: one that provides a shell prompt and one that runs scripts by running each line in the script. To run the binary that provides a shell prompt, the client can run:

```
deeptir@~$ $POSH_SRC/target/release/shell-client \
  --annotations_file <annotations_file> --mount_file \
  <config_file>
posh>>>$ <ENTER COMMANDS>
```

3. Running applications
After running the shell, users can run unmodified shell workflows as normal. For example, the user could type in the following workflow from the distributed log analysis example discussed previously:

```
posh>>> $ cat mount0/logs/*.csv mount1/logs/*.csv \
    mount2/logs/*.csv mount3/logs/*.csv mount4/logs/*.csv \
    | grep '128.151.150' > $LOCAL_FILE
```

Conclusion and Next Steps

We have described Posh, a framework that transparently distributes I/O-heavy shell computation that operates on remote data, by pushing computation to run closer to the data. Posh uses annotations, a model of shell programs, to automatically infer what files an arbitrary command line will read and write to in order to schedule computation across proxy servers. Posh and its annotations provide a model of commands that enable rewiring their dependencies to direct output over the network rather than to a UNIX pipe while retaining local execution semantics. While Posh currently uses this model to transparently schedule and offload commands across proxy servers to push code closer to the data, it could in the future provide more optimal scheduling or even failure recovery. Consider programs that access files from two different locations that cannot be parallelized, such as `comm`. Instead of running them at the client, Posh could run them on one of the servers but stream or transfer the necessary inputs beforehand. To provide failure recovery semantics, Posh could rewrite workflows to write to temporary locations and only write to the final location when the entire operation is successful. For more information on this project, including our research paper, the code, and quick-start guides, please visit our GitHub page, https://github.com/deeptir18/posh.

Acknowledgments
We thank our ATC shepherd, Mahadev Satyanarayanan, and the anonymous ATC reviewers for their invaluable feedback. We are grateful to Shoumik Palkar, Deepak Narayanan, Riad Wahby, Keith Winstein, Liz Izhikevich, Akshay Narayan, and members of the Stanford Future Data and SING Research groups for their comments on various versions of this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, NEC, and VMware, as well as the NSF under CAREER grant CNS-1651570 and Graduate Research Fellowship grant DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Barbalace, A. Iliopoulos, H. Rauchfuss, and G. Brasche, "It's Time to Think about an Operating System for Near Data Processing Architectures" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*, pp. 56–61.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, pp. 15–28.

[3] A. Acharya, M. Uysal, and J. Saltz, "Active Disks: Programming Model, Algorithms, and Evaluation," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, pp. 81–91.

[4] D. Raghavan, S. Fouladi, P. Levis, and M. Zaharia, "POSH: A Data-Aware Shell," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pp. 617–631.

[5] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "Outsourcing Everyday Jobs to Thousands of Cloud Functions with gg," *;login:*, vol. 44, no. 3 (Fall 2019), pp. 5–11.

[6] S. Palkar and M. Zaharia, "Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, pp. 291–305.

Interview with Margo Seltzer

RIK FARROW

Margo Seltzer is Canada 150 Research Chair in Computer Systems and the Cheriton Family Chair in Computer Science at the University of British Columbia. Dr. Seltzer was also a co-founder and CTO of Sleepycat Software, the makers of Berkeley DB. Her research interests are in systems, construed broadly: provenance systems, file systems, databases, transaction processing systems, storage and analysis of graph-structured data, synthesizing system software, discrete optimization, and applying technology to problems in healthcare. She serves on the Computer Science and Telecommunications Board (CSTB) of the (US) National Academies and the Advisory Council for the Canadian COVID-19 contact tracing app. She is a past President of the USENIX Association and served as the USENIX representative to the Computing Research Association Board of Directors. She is a member of the National Academy of Engineering, a Sloan Foundation Fellow in Computer Science, an ACM Fellow, and a Bunting Fellow. She is recognized as an outstanding teacher and mentor, having received the Phi Beta Kappa teaching award in 1996, the Abramson Teaching Award in 1999, the Capers and Marion McDonald Award for Excellence in Mentoring and Advising in 2010, and the CRA-E Undergraduate Research Mentoring Award in 2017. Professor Seltzer received an AB degree in applied mathematics from Harvard/Radcliffe College and a PhD in computer science from the University of California, Berkeley.

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

I first noticed Margo Seltzer because she had brought her baby to a USENIX conference in the late ’90s. That was unusual, as I had seen few parents with their children at conferences. Later on, I got to know Margo better when she was on the USENIX Board and I was routinely attending board meetings.

What prompted me to ask someone as busy as Margo for an interview was her keynote address at the 2020 USENIX Annual Technical Conference entitled “The Fine Line between Bold and Fringe Lunatic” [1]. I recommend watching Margo’s talk, but the gist is simply this: you are likely to have a more interesting career if you are willing to take risks. That’s not what Margo actually says, just my own interpretation. She wants researchers to broaden the subject areas they keep abreast of as well as to consider researching at the frontier of knowledge.

Rik Farrow: You became a faculty member at Harvard, working in CS. Was that at all unusual?

Margo Seltzer: I think what was unusual was that I turned down a position at MIT (arguably ranked #1) for Harvard (pretty much unranked except in theory where we had Turing award winners).

RF: I don’t understand why ranking is important. Could you explain what the ranking means to someone taking an academic position for those of us who won’t have that experience?

MS: Ranking’s importance varies by who you ask.

There has been a lot of data analysis about the network formed by studying the migration of PhD students to faculty positions. New faculty typically have degrees from institutions from rankings higher than the ranking of the school in which they are teaching. So if you want to teach at a top N school; you’d better get a degree from a top (N-1) school. And if you want your students to get jobs at a top N school, then you want to be teaching at a top-1 school (or at least one of the “big 5”).

Unfortunately, rankings are a fuzzy metric—I advise undergrads going to grad school to place far more emphasis on the person/group with whom they will work than the ranking of the school, but students don’t always listen. And students from undergraduate institutions without a lot of advising don’t have much to go on other than the rankings: they don’t know the faculty.

So the ranking of the university at which you take a faculty position is directly correlated with the quality of students you get and the likelihood of placing them at other top institutions.

Thus—turning down MIT (arguably #1) for Harvard (top N > 20 and probably closer to 30–40 then) was shocking to most. I was definitely called an idiot by some.

Harvard, in particular, had a dismal reputation for granting tenure. There had been a famous case in 1983 where a person widely regarded as a superstar in his community was denied tenure by Harvard. So when I got there (1993), Harvard had not tenured anyone in computer science since 1981. My colleague, Stuart Shieber, broke that curse by getting tenure in 1996. Then Mike Smith and I both got tenure in 2000. Since then, Harvard has done very well by hiring strong people and making sure they get tenure.

RF: You begin your talk [1] by demonstrating how computer science has been partitioned over the years, beginning with the split between hardware and software, then software splitting into operating systems and programming languages, and so on. You encourage people to cross the many boundaries that exist today, and I do sometimes see that happening, for example, with file systems using key-value stores for metadata. Do you have other examples?

MS: The crossover between file systems and databases has grown a fair bit over the past 30 years, but it takes a lot of pushing:

Journaling (logging) was developed in the database community to provide transaction support in the '70s. It wasn't fully embraced by the file-system community until the '90s or later. At this point, it's fairly standard.

Transactions are another concept with a history in databases—we now see transactions in hardware (i.e., transactional memory) and every once in a while in file systems.

Program analysis (e.g., static analysis, symbolic execution) grew out of the programming languages community, but it has been and is being adopted in systems for bug finding.

The emergence of persistent memory (e.g., Intel Optane) brings together work from systems (virtual memory and single level store), databases (persistent objects), and file systems (persistent files).

So these things happen, but a lot of the time the researchers themselves don't think to look at work of other communities and will re-invent the wheel instead of borrowing it and making it work better.

RF: I agree that not looking at what has been done in other communities really slows down research and innovation in CS. But isn't there an issue with the amount of research, just in the small niches that we have today, being too overwhelming for most graduate students to cover?

MS: It is impossible to keep up with all the work being done in a single field, so how can one hope to know what's happening in other fields? In machine learning alone, something like 100 new papers show up on arXiv every day. So what is an overworked graduate student to do?

It's not necessary to read every paper published to know what's happening in a field. The key is really an openness to what's happening in other areas, a curiosity, and a willingness to do the hard work of trying to understand work from a different community when it's appropriate. One of the first things I do is encourage new graduate students to subscribe to The Morning Paper—https://blog.acolyer.org/. My understanding is that Adrian Colyer, the author, is not really a computer scientist, but every week he sits down and reads about three computer science papers and writes up great blog posts about them. And he moves from area to area, reading whatever is recent or what is particularly interesting to him. I love his posts—I have a mailbox full of ones I've not yet had time to read.

Just reading Adrian's blog posts will give a student a broad introduction to a lot of areas. But even that isn't enough.

You have to be willing to talk to other people—not just the people in your lab but people in other labs. Go to weekly grad student social events and really try to understand what people are working on. Here is the secret: you are going to have to be willing to ask naive questions. I call them stupid questions, but they aren't really stupid, they are mostly just the questions that someone unfamiliar with an area will ask. And even more important (and possibly scarier), you have to be willing to say, "Um, I didn't really understand that, can we go even more slowly?" I collaborate with many folks who are way more mathematically sophisticated than I am, and I tend to ask (a lot), "Could you explain that to me in small words?" To be honest, it took me a long time to get over the knee-jerk reaction of just nodding and pretending that I understood what was going on when I was lost, but I learn a ton more when I'm willing to take that risk. And who better to take that risk with than your peers? And you never know, you might find an area that intrigues you, a topic of mutual interest, or just something new and interesting.

The key is not to be an expert in everything but to have a vague sense of what people are working on in other fields, so that when the opportunity arises, you can draw ideas from disparate areas and know what the areas are and perhaps even with whom to consult (that fellow student you were chatting with just the other day).

Super secret #2: being able to talk to people in other areas will be your single greatest superpower on the interview trail, where you're expected to be able to have intelligent conversations with people from different areas.

Fun story: In my interview that wasn't really an interview (or perhaps it was the non-interview that really was an interview) at Harvard, I was taken to lunch by two theoreticians—one Turing Award winner and one future Turing Award winner. They peppered me with questions to the point that I was still working on my salad when they got to dessert! But clearly something worked—shortly after I arrived, one of them dropped by my office to ask for my "expertise" on a topic…I was floored. What on earth did I have to offer a world-renowned theoretician? Well, he had some interesting ideas about applying his latest work to storage, and well, he figured that perhaps there might be people who knew more about storage than he did. It was a good lesson for me—no one is above asking questions, and no one should limit themselves to a small box, even if they are the absolute best in that box!

RF: Provenance is the first of the fringe lunatic ideas you cover in your USENIX ATC '20 keynote. The quest for provenance began early in this century, largely as a way to be able to recreate data based on its provenance: what had happened to that data since it was created. I recall thinking at the time that this seemed like a reasonable thing to do, but later wondered if having to maintain orders of magnitude more data as provenance really made sense. The story you told about provenance includes different groups taking different approaches, along with attempts to unify some elements. Were you surprised at where researchers had taken the original idea after almost 15 years had passed?

MS: Yes and no.

Around 2014, I "gave up" on provenance—I felt that the community was focused so much on provenance collection that they were not giving ample thought to motivating users to collect provenance. I was frustrated and basically went in other directions—then, almost immediately, I got two provenance proposals funded with collaborators. In one, we focused on use from the beginning—in the other, I forgot my own lesson for several years and only rediscovered it a few years later, fortunately with enough time to change course.

That said, all those are in the higher levels of the stack.

I am actually thrilled that the systems community has embraced provenance and is thinking hard about how to use it: security, information flow, reproducibility, etc. I always felt that system level provenance was the glue that could hold lots of things together, and these folks are making it work.

So am I surprised: 1) No—I don't think any of the things people are doing would have surprised me in 2006. 2) Yes—it seemed like the field wasn't going anywhere, but it still is!

RF: The other example in your keynote had to do with program synthesis, although to me it sounded much more ambitious than merely being able to generate a program. The DARPA BRASS [Building Resource Adaptive Software Systems] program was really about extracting intent from systems so that when circumstances changed, the system could adapt to the change and still succeed in accomplishing the intent of the system. You were among the "fringe lunatics" (your words) who took that to mean making the operating system adapt to new hardware by synthesizing operating systems from machine descriptions. That sounds like a ridiculously tall feat to accomplish, but a very good example for your theme. Could you tell us how that worked out?

MS: We didn't synthesize a complete system, but we've synthesized several parts of the Barrelfish operating system [2] and nearly an entire port of our OS/161 educational operating system [3]—and we've done this for about four different processors!

References

[1] M. Seltzer, "The Fine Line between Bold and Fringe Lunatic," 2020 USENIX Annual Technical Conference (USENIX ATC '20): https://www.usenix.org/conference/atc20/presentation/keynote-seltzer.

[2] The Barrelfish Operating System: http://www.barrelfish.org/index.html.

[3] D. A. Holland, A. T. Lim, M. I. Seltzer, "A New Instructional Operating System," in *Proceedings of the 2002 SIGCSE Conference* (February 2002), pp. 111–115: https://www.seltzer.com/assets/publications/A-New-Instructional-Operating-System.pdf.

SRE Best Practices for Capacity Management

LUIS QUESADA TORRES AND DOUG COLISH

Luis Quesada Torres is a Site Reliability Engineer and Manager at Google, where he is responsible for keeping Google Cloud's Artificial Intelligence products running reliably and efficiently. In his spare time, Luis jumps from hobby to hobby: he composes and produces music across multiple genres, he skateboards, and he speaks Spanish, English, German, Swiss German, and Esperanto. Soon Japanese as well. luis@google.com

Doug Colish is a Technical Writer at Google in NYC supporting Site Reliability Engineering (SRE) teams. He contributed to several chapters of Google's "Building Secure and Reliable Systems" book. Doug has over three decades of system engineering experience specializing in UNIX and security. His hobbies include detailing and modifying cars, attending concerts, and watching and discussing great movies. dcolish@google.com

As an SRE, you're responsible for determining the initial resource requirements of your service and ensuring your service behaves reasonably even in the face of unexpected demand. *Capacity management* is the process of ensuring you have the appropriate amount of resources for your service to be scalable, efficient, and reliable. User-facing and company internal services must accommodate both expected and unexpected growth. We define utilization as the percentage of a resource that is being used. It's difficult to determine initial resource utilization and predict future needs. We present ways to estimate utilization and identify blind spots, and we discuss the benefits of building in redundancy to avoid failures. You'll use this information to design your architecture such that increasing the resource allocation for each component of the service effectively increases the capacity of the entire service linearly.

Principles of Capacity Management

A *service*, in the context of this article, is defined as the set of all of the binaries (service stack) that provides a set of functions.

Successful capacity management entails allocating resources from two complex points of view: *resource provisioning*, which provides the initial capacity to run the service now, and *capacity planning*, which safeguards the reliability of the service into the future.

At its core, capacity management must follow three basic principles in order to keep a service scalable, usable, and manageable:

- **Services must use their resources efficiently**. Large services that require a considerable amount of resources are expensive to deploy and maintain.
- **Services must run reliably**. Limiting resource capacity to improve service efficiency can put the service at risk of malfunctioning and suffering user-facing outages. There is a tradeoff between service efficiency and reliability.
- **Service growth must be anticipated**. Adding resources to a service can take a long time and has real world limitations around deployment. This may involve buying and deploying new equipment or building new datacenters. It may also require increasing capacity for other software systems and infrastructure that are dependencies of the service.

Complexities of Capacity Management

A large service is a complex living organism whose behavior is unexpected at times. You need to consider several areas when making engineering decisions that could potentially alter the service's scope:

Service performance. Understand how different components of the service perform under load.

Service failure modes. Consider the known failure modes and how the service behaves when subjected to them. Also, consider how the service might behave when subjected to unknown failure modes. Be prepared by generating a list of possible bottlenecks and service dependencies you may encounter.

Demand. Determine the expected user count and traffic, where the user base is located, and the usage patterns.

Organic growth. Estimate how demand may grow over time.

Inorganic growth. Keep in mind the long-term resource impact of adding new features or of the service becoming more successful than expected.

Scaling. Understand how the service scales when increasing resource allocations.

Market analysis. Estimate how market changes affect your ability to acquire additional resources. Research new technologies that can improve the performance, reliability, or efficiency of the service and the cost of implementing them. Investigate how quickly you can adopt new technologies, such as replacing HDDs with SSDs.

The goal of capacity management is controlling uncertainty. In the midst of the unknown, the service must be available now and continue to run in the future. A challenging but rewarding and delicate balance of tradeoffs is in play: efficiency vs. reliability, accuracy vs. complexity, and effort vs. benefit.

Use data to drive capacity decisions. You'll still make unavoidable mistakes, and you'll have fires to put out, often in creative ways. But the end result is a reliable business-critical service.

Resource provisioning addresses the tactical question, "How do I keep the service running right now?" while *capacity planning* addresses the strategic question, "How do I keep the service running for the foreseeable future?"

The following sections discuss these topics in detail.

Resource Provisioning

Our discussions focus on a serving system, that is, a service that responds to user requests by looking up some data. However, you can apply these principles equally to a data storage service, data transformation service, and most other things you can do with a computer.

Resource provisioning involves figuring out the *target utilization* of resources a service needs and allocating those resources. Target utilization is defined as the highest possible utilization for a specific *resource class* that allows the service to function reliably. A resource class refers to a specific type of computing asset. CPU, RAM, storage, etc. are resource classes.

To provision resources for your service, use demand signals as inputs and create the production layout with concrete resource allocations as output, as shown in Figure 1. Services often use several resource classes.

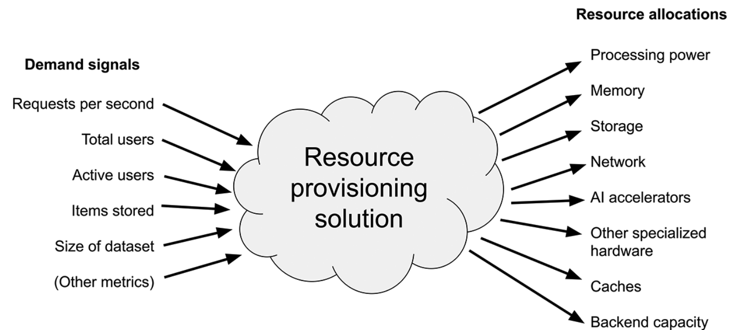


Figure 1: Demand signals and resource allocations of a resource provisioning solution

The Impact of Resource Shortages

A shortage of resources can make the service fail differently, depending on the resource class.

When resources become a bottleneck in the service's *critical path*, users experience increased latency. In a worst-case scenario, the bottleneck causes requests to backlog, resulting in ever-increasing latency and, eventually, the timeout of queued requests. Without a mitigation plan in place, the service fails to process requests and suffers an outage. The outage continues until the incoming traffic drops off, allowing the service to catch up, or until the service is restarted.

Resources that are often in the critical path include:

- Processing power
- Network
- Storage throughput

When resources become a bottleneck in the *non-critical path*, the service suffers delays in some of its non-time critical functions, such as maintenance or asynchronous processing. If these tasks are delayed long enough, they could impact service performance, features, data integrity, and even cause an outage in extreme cases.

When a service runs out of storage, writes fail. Even certain reads may fail if they are dependent on writes: for example, if the service or storage solution stores Paxos state to do consistent reads, or if the storage solution keeps track of all accessed data and the time it was accessed.

When other resources such as memory or network sockets are low, a service may crash, restart, or hang. A service with low resources may start to thrash from garbage-collection or misbehave in other ways. These failures decrease the service's capacity and can trigger cascading failure scenarios requiring human interaction to resolve.

For mitigation strategies, see the Decrease the Impact of Outages section below.

Estimating Utilization

Because of their different nature, resource usage and target utilization are different for every service and for each resource class. In order to estimate the target utilization for a specific service, each of the following aspects need to be considered.

Peak Usage

A service's peak usage is simply the highest usage rate over a given time period and depends on the nature of the service and the user base. The early hours of a business-related service may drive the weekday peaks. Social-related services peak late in the afternoon, at night, during weekends, or coinciding with social events such as concerts, sporting events, etc. When an unexpected event happens, usage can drop or soar. A global service's user base is spread across different countries and time zones, forming a more complex daily traffic pattern.

Assuming non-constant load, resource utilization shouldn't surpass 100% of the service's allocated resources during peak traffic. By not using all of its resources, the service has sufficient capacity to serve the peak and is not overprovisioned in any wasteful way.

Maximum Peak Utilization

Even at peak, it's a bad idea to run the service at 100% utilization. Some software, languages, or platforms will misbehave or garbage-collection thrash before CPU use even reaches 100%. A service will crash with an *out of memory* (OOM) error if a component reaches 100% memory utilization.

Fine-tuning your monitoring sufficiently to capture the precise resource utilization in small enough time frames (microseconds or even seconds) is tedious. Thus, it's difficult to determine the resource usage peak for low-latency applications.

Redundancy

Issues with rollouts, hardware, software, or even planned maintenance can cause the components of a service to fail or restart. This can result in a failure as small as a single binary instance crashing or as large as the whole service going offline.

Redundancy is a system design principle that includes duplicated components that are active only when they replace other components that failed. The degree of redundancy is denoted by $N+x$, where N is the total number of active components, and x is the number of backup components. Thus, $N+3$ indicates that three system components can fail because there are three duplicated components to replace them. Meanwhile, the service remains completely functional, regardless of the total number of components (N).

Redundancy can be applied within regions or across regions. A region is an independent failure domain located in a physical site different from other regions so that network issues or natural disasters do not impact more than one region at the same time.

Redundancy within Regions

Redundancy within a region is fairly trivial to achieve.

Within a region, you want to provide protection against failed binaries or physical machines. Typically, you can simply add extra instances of the service binaries running per region, with a load-balancing solution to redirect traffic if binaries or machines are down. The required extent of redundancy is tied to the infrastructure's service level agreement (SLA). Specifically, the SLA accounts for the total number of machines that can be in a failed state simultaneously and the speed in which new instances of binaries can be restarted on new machines.

Understand that redundancy within the region won't protect your service at all from failures that take out the whole region (power, network, natural disaster, etc.).

Redundancy across Regions

Redundancy across regions is far more complex.

Across regions, you'll need protection from total region outages. By deploying replicas, or full copies of the service stack in several regions, you can implement redundancy across regions to accommodate your service's load at peak. Note, each replica must have enough capacity to serve all of the expected load when any number of replicas are down based on your declared redundancy. As stated above, regardless of the number of replicas (N), the degree of regional redundancy of the service is defined as follows:

- **N+0**: when the service is up and running, but cannot tolerate any region going down
- **N+1**: when the service can withstand a single region going down
- **N+2**: when it can still serve with two regions down
- etc.

While some of this redundancy involves capacity, it's also about the service architecture itself. For example, consistent storage services often require that a majority of replicas are up and running to ensure that writes aren't rolled back.

Provisioning a service for N+2 has a positive effect on reliability: maintenance can be planned for an entire region at once, but lowers redundancy to N+1 during the maintenance. The service can still tolerate an unplanned incident in another region. This lowers the redundancy to N+0, but does not cause an outage. Note that failing over to another region may have effects on visible latency.

With N+0 redundancy and no tolerance for further failure, your priority is to mitigate or resolve the unplanned incident as fast as possible. One option is to complete or revert the planned maintenance work to bring the service back to N+1. Otherwise, any other region suffering an incident could cause a user-facing outage.

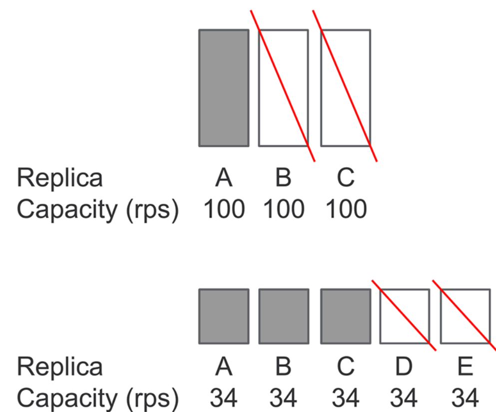
Expected load: 100 requests per second (rps)

Running N+2 on 3 replicas

2 replicas can go down (N+2)
3 - 2 = 1 replicas stay up to serve 100 rps
Each replica is provisioned to serve 100 rps/ 1 replica = 100 rps/replica
Total capacity provisioned for is 100 rps/replica x 3 replicas = 300 rps
At level flight, the maximum utilization for N+2 is 100 rps / 300 rps = 33%

Running N+2 on 5 replicas

2 replicas can go down (N+2)
5 - 2 = 3 replicas stay up to serve 100 rps
Each replica is provisioned to serve 100 rps / 3 replicas = 34 rps/replica
Total capacity provisioned for is 34 rps/replica x 5 replicas = 170 rps
At level flight, the maximum utilization for N+2 is 100 rps / 170 rps = 59%



The cost of running this service on 5 replicas is 170 / 300 = 56.6% of the cost of running it on 3 replicas

Figure 2: Example comparison of the cost of resource provisioning a service with three and five replicas

The Cost of Redundancy

The more regions a service operates in, the lower the cost of running any level of redundancy. Consider the service described in Figure 2. It needs to run with N+2 redundancy. In the first setup, it is running three replicas (N=3), and in the second setup, it is running five (N=5). Both configurations have two spare replicas (+2) and thus can withstand two replicas failing.

Next, examine the five-replica setup. Its replicas are smaller in size, and even when two replicas fail and both spare replicas are in use, there are still three active replicas to share the load. This results in the five-replica N+2 setup costing 56.6% of the three-replica service using the same degree of redundancy. See the calculations provided in Figure 2.

Homogeneous and Heterogeneous Services

It's easier to implement redundancy for services with homogeneously sized replicas than those services with heterogeneously sized replicas.

Your service must be provisioned to handle failures in the largest region. If regions have different capacities (i.e., heterogeneous), the capacities needed to withstand the unavailability of the other largest regions are different in each region. The result is that your smaller regions require more resources, and your overall required resources to serve the same load are higher.

Replicated and Distributed Traffic

Provisioning for redundancy also depends on the characteristics of the service's traffic.

Stateless services, such as web servers that handle user requests, receive traffic that is distributed among replicas. Requests that read from storage services can also be distributed across replicas in different regions. Provisioning these for N+1 or N+2 is trivial and follows the logic from the previous example.

Services that handle requests replicated across regions, such as writes, behave differently. Each write to an entity needs to be eventually written to every single replica to keep your service's data consistent across replicas.

When a replica becomes unavailable, replicated write requests do not cause additional load to the replicas that remain up. However, there is a cost incurred when the unavailable replica comes back online. This replica needs to catch up with outstanding writes that were missed during its downtime. This operation increases its load. The replicas that remain running provide the data needed to sync the recovering replica, increasing the load on all replicas during recovery. Ideally, this is capped to avoid hurting low-latency traffic across the entire set of replicas.

Each service and each component can receive a different proportion of replicated and distributed traffic, which need to be factored in when resource provisioning.

Latency-Insensitive Processes

A service typically has latency-insensitive processes such as batch jobs, asynchronous requests, maintenance, and experiments.

However, these processes put additional strain on the service while it handles the production load, which is latency-sensitive. The service thus requires additional resources to accommodate a higher peak, increasing its cost.

You can minimize the extra cost of latency-insensitive requests by assigning them lower priorities or by scheduling them during low-load periods in order to decrease the overall peak. Note, both of these solutions need to be properly tested and carefully deployed to prevent service interruptions.

Additional Resources for the Unknown

The last aspect to consider is the unknown factor. There are many good reasons to throw in additional resources when provisioning a service: for example, the performance regression of an underlying library supported by another team or when implementing a team-external requirement such as encrypting all RPCs.

Spare capacity can keep the service performing as expected, in regards to latency and errors, if anything goes wrong. However, keep in mind that this decision can be expensive, so make sure that the tradeoff in reliability, predictability, and scaling is worth the cost.

Capacity Planning

While *resource provisioning* refers to the process of determining the correct amount of resources to keep your service running right now, *capacity planning* entails forecasting demand to guarantee resource supply in the future.

Overview of Capacity Planning

Like resource provisioning, capacity planning is an attempt to determine the amount of each computing asset (resource class) you need to sustain the service. However, it involves making those determinations at multiple points in time: for example, your resource needs in three months, six months, or a year.

For an existing service, capacity planning uses historic demand to forecast growth to build on top of resource provisioning for your service's maximum peak utilization, redundancy requirements, latency-insensitive processes, and the unknown factor. Generally, you'll want to add to this forecast any planned new consumers of your resources, including new services, marketing campaigns, new features, etc.

You'll need different amounts of each individual resource class for each component in your service. Take RAM, for example. A web server may need a lot of RAM, whereas a proxy server may need very little. To determine the various values of a single resource when you are planning for future capacity, take into account the following:

- The number of different components (database, proxy, application) in your service
- The number of instances of each component (1 database, 2 proxy, 2 application)
- The regions your service runs in (i.e., across-region N+1 or N+2)
- The number of data points you need for your forecast

While this is a simple example of a complex formula, a single resource class like RAM may require you to think in terms of the following:

(# of different components) × (# of instances of each component) × (# of regions) × (# of datapoints) × (other contributing factors)

As you can see, when you consider all resource classes for all server types in all regions and add in redundancy, the number of capacity values that you must determine grows exponentially.

Forecasting Resources

Capacity planning is an extremely complex process as there are myriad factors at play, and each can change independently. Expanding on the high level overview above, consider the following when forecasting:

Resource Classes by Component

In addition to determining the total number of components, you must also consider the various resource classes that each one utilizes: RAM, CPU, storage, network, etc. One component may use one set of resource classes, and others may have a very different set. If your service consists of many components, the set of resource classes that you must track quickly increases.

Multiple Regions

If you are required to run in many regions around the world, you can imagine how forecasting a single resource class such as CPU for various machines (web, database server, application, proxy, etc.) is made even more difficult. Add in all of the other resources classes for all machines, redundancy across all regions over a given period of time (six months or a year from now) to start your planning.

Service Demand

Demand depends on the success and adoption rate of the new service and is only known after the service is launched. You must update forecasts over time and correct long-term predictions. Understand you are preparing for a sudden unplanned load increase that can cause an outage if ignored.

Other unexpected events like natural disasters, network interrupts, or power outages can drastically alter your traffic patterns. Even planned situations such as social events or the beginning or end of holidays can affect your service in unexpected ways. It's challenging to extrapolate the changing impact of such events year to year as new features are launched or the user base varies.

Changes in user distribution in different time zones also have service implications. Traffic may appear more or less spread out across the day, unexpectedly raising and lowering peak demand.

Growth

Growth depends on the success of your service. It may take some time (and marketing campaigns) for users to learn about your service and take interest, and the interest may grow slowly or sharply over time. Other services on the Internet can have a dependency on yours, and their success or failure can directly affect your service. A successful external service can increase traffic to you, and vice versa.

There may be social, economic, political, or other factors that may increase or decrease your user traffic. You have to determine your growth rate and take this into account for your capacity planning sessions.

Forecasting Example

To illustrate the multitude of potential separate resource class values you, as the service owner, must try to predict correctly, let's use a very simple example:

Resource Classes for a Two-Component Service

Suppose you have a small service such as a social media application. It consists of two machines, a web server and a database. The web server uses CPU and RAM, and the database uses CPU, RAM, HDD storage, HDD throughput, and SSD storage. This is a total of six unique resource class values to define. Note, this is far short of a complete set of values in a real-world application.

By having three replicas, you now have 18 values to define. If you are forecasting quarterly for 12 months, that number jumps to 72 (four quarters per year × 18).

Trends That Impact Your Service

You've learned that your social media service is affected by seasonal trends. You have an increase in traffic at the beginning of the holiday season (Nov–Dec), another during spring break, and one more at the start of summer. Your forecasting cannot be just a linear increase in resources, you must account for the spikes during peak times of the year.

You may also experience similar trends with peaks during the month for batch-processing tasks such as data cleanup or database compaction. The load may be different each month, or even each week, further complicating your ability to estimate resource utilization accurately.

Best Practices

We present several best practices for capacity management to help you anticipate common problems and pitfalls.

Load Testing

Run a small replica of the service at target utilization and above, and exercise failover, cache failures, rollouts, etc. Assess how the service reacts to and recovers from overload, and empirically validate that the resource allocation is adequate to serve a defined load. Be careful when extrapolating estimates from your data. If a binary instance allocated with one CPU can serve 100 requests per second, it's generally OK to assume that two binary instances, each with one CPU, can serve 200 requests per second in total. It is not OK to assume that a binary instance with two CPUs allocated can serve 200 requests per second. There may be bottlenecks other than processing power.

Holistically Evaluate the Capacity

While you should add extra capacity for the unknown, avoid stacking too many resources and inadvertently overprovisioning the service. However, provide enough spare resources so the service can withstand issues. This can buy some extra time to secure resources in case the service is more successful than was expected and was provisioned for.

Decrease the Impact of Outages

It's possible to prepare the service so that outages have a lower impact when it runs out of resources. Suggested preventative measures include:

- **Graceful degradation**. The service disables some non-critical features to relieve resource usage when it's overwhelmed.
- **Denial-of-Service (DoS) attack protection**. Provided in case the increased traffic comes from an ill-intentioned party.
- **Effective timeouts**. Requests eventually time out, and the service drops the requests without wasting further resources on them.
- **Load shedding**. The service quickly rejects requests when it's overwhelmed, allowing a routing layer above to retry the requests or make them fail fast. This avoids the issues of a service falling behind and wasting efforts on requests that are going to time out anyway.

Quota Management and Throttling

Deploying a quota system helps limit the throughput between your service and the back end, providing isolation from other services using that same back end. Whenever a service sends more requests than expected and reaches the quota limit, the back end throttles the services rather than overloading itself and impacting other services using that back end.

Monitoring

The relevant metrics gathered from monitoring your service provide data to guide resource provisioning and capacity planning decisions. Using our sample service above as a model, the following are very useful:

Load metrics

- Incoming requests per second
- Latency-insensitive load
- Number of active users
- Number of total users

Resource metrics
- Resource allocations
- Actual resource usage
- Quota usage
- How many requests are throttled

Performance metrics
- Latency
- Errors

High-level health metrics (that can help filter out other tainted metrics data)
- When the service was impacted by an outage
- When the service was undergoing maintenance

Alerting

Use alerts for resource provisioning and capacity planning to prevent outages. Some examples of useful alerts are those that trigger when the service is not at the intended redundancy level and is therefore underprovisioned, alerts that indicate the service lacks future resources according to forecasts, current performance issues, etc.

Resource Pooling

Pooling is the grouping of resources so that several services share them rather than providing separate allocations per service. Pooling is often used to decrease planning complexity and to reduce resource fragmentation, hence, improving the efficiency of a service. When you implement this strategy, planning for large services is still detailed and precise. However, small services use a pool of resources that is provisioned for as a single entity, approximately and conservatively. This decreases the effort on capacity planning at the expense of isolation.

General SRE Best Practices

Follow the basic SRE principles that you would for any service. For example, store the capacity state as a configuration in a version control system and require peer reviews for any changes. Automate enforcement, roll out all changes gradually, constantly monitor your service, and be ready to roll back if needed.

In the event of a failure or other issue, exercise blameless postmortems to honestly learn from the mistakes, and commit to improving the system to avoid repeating them.

Evaluating a Service for Capacity

When evaluating capacity for a new or existing service, we recommend determining its resource requirements by following these steps:

Hardware	Specs
Processors	CPU type and count (cores)
Graphics Processing Units	GPU type and count
Storage	HDD (hard drives) and SSD (solid state disk): • Amount of storage (TB) • Bandwidth • IOPS
Network	Intra datacenter, inter datacenter, ISP access: • Latencies • Bandwidth
Back Ends	Services and capacity needed
Other	AI accelerators, other special hardware

Table 1: Resource assessment template

1. Estimate the resources needed to serve the expected load. Use the template in Table 1 and fill it in with the expected service demand for the different resource classes.
2. Calculate and factor in the target utilization of the different components of the service. You may need to perform load testing to assess:
   - Peak usage
   - Maximum peak utilization
   - Redundancy
   - Latency-insensitive processes
   - Spare resources for the unknowns
3. Consider aspects such as:
   - Priority
   - Region
   - Service components
   - Specific points in time and time into the future (monthly, quarterly, for six months, a year, etc.)
4. Perform forecasting, considering whether you need to plan for capacity per:
   - Priority
   - Region
   - Service components
   - Number of points in time per year

5. Continue to learn about capacity management:
   - Watch the video "Complexities of Capacity Management for Distributed Services" for an extended tech talk on the topic [1].
   - Read the *;login:* article "Capacity Planning" [2].
   - Review the "Software Engineering in SRE," "Managing Critical State," and "Reliable Product Launches at Scale" chapters of Google's *Site Reliability Engineering* [3].

Conclusion

In this article we discussed the components and complexities of capacity management. We separated the topic into two parts: *resource provisioning*, which addresses the tactical question, "How do I keep the service running right now?" and *capacity planning*, which addresses the strategic question, "How do I keep the service running for the foreseeable future?" Answering these questions is not a trivial task, and each requires reviewing different aspects of your service.

When provisioning resources, examine the various demand signals (input) and their effect on the resource allocations (output). It helps to understand the expected peak demands the service may face and the amount of redundancy you're required to build into it. Have you considered the impact of resource shortages and vendor supply?

Capacity planning forces you to attempt to predict what the service and, more importantly, its load look like in the ever-changing future. You have to fully understand your service to do this—for example, you need to identify the peak cycles and when they occur, determine the number of locations you must run in and the varying capabilities of each, and anticipate the natural, social, and even legal events that might impact your service. When it's time to add more capacity, do you have the approvals or funds to accommodate the growth?

While the many best practices we presented are all important, following solid SRE tenets helps simplify capacity management: perform proper load testing, implement extensive monitoring and alerting, use source control systems, understand the strengths and weaknesses of your service, develop a capacity plan, and be prepared to anticipate growth and scale when needed.

Acknowledgments

The authors are grateful for the suggestions from JC van Winkel, Michal Kottman, Grant Bachman, Todd Underwood, Betsy Beyer, and Salim Virji.

References

[1] L. Quesada Torres, "Complexities of Capacity Management for Distributed Services," Google Tech Talk: https://www.youtube.com/watch?v=pOo0oKNM9I8.

[2] D. Hixson and K. Guliani, "Capacity Planning," *;login:*, vol. 40, no. 1 (February 2015): https://www.usenix.org/system/files/login/articles/login_feb15_07_hixson.pdf.

[3] B. Beyer, C. Jones, N. R. Murphy, and J. Petoff, eds., *Site Reliability Engineering*, Chapters 18, 23, and 27: https://landing.google.com/sre/sre-book/toc/index.html.

The Case for CS Knowledge in SRE

ADAM MCKAIG

Adam McKaig is a staff Site Reliability Engineer at Datadog in New York, where he looks after a metrics system. Previously he has built things at Google, the *New York Times*, Bloomberg, and UNICEF. His favorite language is C++, which probably says it all. adam.mckaig@gmail.com

During my career as an SRE, I've become convinced that knowledge of traditional computer science topics like data structures and algorithms are, while not essential to hacking together something that kind of works, an essential part of building reliable and scalable systems. This wasn't always my position on the matter; as a self-taught programmer, I got a long way without a clue about the fundamentals, believing that my own empirical approach was superior and that the world would catch up soon enough. In this article, I'll share a few of the more interesting problems that changed my attitude, how they were diagnosed, and how they were solved with better data structures and/or algorithms.

Most systems start life as an idea and are hacked together at first. The priority is to get something into production as soon as possible and iterate on it without worrying about what comes next. There's nothing wrong with that, but it doesn't work for long, and the next phase—productionization, that is, scalability, reliability, and so on—necessitates an almost totally different approach and skill set. It's also the most interesting part.

The main difference between the pre- and post-productionization phase is that the implementation details don't matter during the former, so long as it works. Linear, log-linear, even quadratic algorithms are blazing fast on modern hardware while n is small, and RAM is as good as unlimited. But however much one is willing to spend on cloud bills, once n starts getting large in any dimension, consistent high performance can only be achieved by carefully choosing and implementing the appropriate data structures and algorithms to avoid having to compromise on features. Ideally, one would be able to predict the growth of every dimension of n and design accordingly in advance, but in practice it's usually done reactively, when some subsystem is approaching its performance limits.

It's highly instructive to implement every detail oneself, but rarely is it necessary in practice; even the most esoteric data structures and algorithms are readily available as packages for most languages. Much more important is to develop an intuition for their performance characteristics and to be able to spot those same characteristics in production workloads.

Practical Examples

These are real examples of things going wrong at scale. I've redacted sensitive details and condensed them for brevity, but these are issues encountered in production at large companies you've probably heard of.

Fixing an Assumption

My team was supporting an old C++ service, part of a messaging system, which was having trouble sustaining its required write throughput. The service was consuming create/update/delete events from a message bus, and providing an API to view the most recent messages sent or received by a given user. It had worked fine for a long time, but it couldn't keep up as the rate of events increased, and users were complaining that the API was serving stale data

during peak hours. This service was running on a single big machine, so the most obvious solution was to shard the service and run it on many machines. But that would take time, and we wanted to improve the situation sooner.

We improved things a bit by providing a lot more CPU and looked into the implementation. What we found was unremarkable: a big `std::map` (an ordered tree) holding the latest messages, keyed by the user ID and timestamp. Writes would either insert a record, or fetch a record, patch it, and replace it. Reads would find all messages with a matching user ID and return them, which was efficient because they were adjacent and already sorted. Old records were garbage collected in a background thread by periodically walking the entire tree.

Ordered trees are a great default for mixed workloads, that is, workloads which have a similar proportion of reads and writes. But when we looked at the data from production, we saw that the rate of reads was actually remarkably low compared to the writes, which accounted for the vast majority of work. These writes weren't slow, but they weren't fast enough to keep up with the desired volume. We also saw that our read latency was consistently far below the threshold at which we would be paged about it. So we investigated how we might speed up writes, knowing that we were able and willing to *sacrifice some read performance* to do so.

It was simple for us to swap out the map with an LSM (log-structured merge) tree, a data structure which resembles an ordered tree but offers far more scalable inserts at the cost of slower and less predictable reads, using an existing open source package. We dark-launched this change into production and observed, as we'd hoped, a tremendous improvement in throughput with only a modest regression in read latency. I don't recall anyone ever complaining about the latter.

This incident taught me that although most systems rightly expect mixed workloads and so optimize for that, that isn't always the reality in production, and making concessions on one side can yield big improvements on the other.

Consider Non-Requirements

Here's a totally different example. Much later, at a different company, I was supporting a distributed key-value datastore (of sorts) written in Go. The overall workload was fairly mixed: lots of writes and lots of reads. The system stored highly denormalized event data and was primarily used to answer arbitrary questions like, "What are the most-viewed widgets by users who looked at this widget this week?" in real-ish time.

One subsystem was causing trouble: the directory service, which basically kept track of which data were on which storage node, and how CPU-loaded each was, so that the query nodes could fan out incoming reads to the right places. This subsystem was read-heavy, and the load varied throughout the day as end users came and went. The rate of writes was more consistent, since it was simply proportional to the number of storage nodes, which periodically announced the ranges of keys they had and their overall CPU load. Both would change regularly as data was rebalanced by a separate subsystem.

The problem we were seeing here was that many directory reads were too slow during peak hours. Up to about the 90th percentile was fine, but above that, performance varied wildly. We were able to improve things by horizontally scaling (roughly doubling) the number of directory nodes, thereby reducing the rate of reads that each had to handle, but this caused two more problems: utilization of these nodes was now low enough that well-intentioned cost-saving alerts were going off, which needed silencing; and this increased load on the storage nodes, because they needed to send twice as many announcements! Clearly this was a temporary mitigation, so we looked into improving the read throughput.

The implementation was (roughly) an augmented interval tree, storing ranges of keys mapped back to the storage node they could be found on, and a map of nodes to their last-reported CPU load. Writes would update both of these: key ranges would be inserted into the tree, and the load would be updated. Reads would read from both: the tree would be queried for nodes containing matching keys or key ranges (of which there could be many), and the load of each node looked up from the map.

The bottleneck here was of course the tree, because there wasn't much else to the system. Profiling indicated that reads were too often being blocked by writes, which had to lock the tree while they were mutating it.

Given the requirements, and without fundamentally changing how the system worked, we couldn't think of an obviously better implementation. We started designing a sharded directory service, making it a nested distributed system of sorts, but so many tricky edge-cases came up that we shelved it until it was really necessary—which in the end it never was. The solution presented itself when we went back and *reconsidered the requirements*.

We needed to maintain an up-to-date map of keys to nodes, which was small enough to fit on one node, fast to query, and fast enough to write that it didn't interfere with the reads. But it didn't need to be completely up-to-date: this was an OLAP system, not OLTP, and the map was always a bit stale because storage nodes only reported periodically. Could we put a cache in front of the tree, to speed up some reads in exchange for making the data slightly more stale? We couldn't think of a cache key which would actually be effective, since the keyspace was so large, but someone suggested: how about we cache the whole tree? We have plenty of spare RAM.

The resulting implementation was simple and effective: rather than one tree, we stored three. One was used to serve reads; one was updated as writes arrived. To update the read tree, the write tree was locked and copied to a third location, and only then were incoming reads briefly blocked as a pointer was swapped to point to the new read tree. This frequency was tunable, and in practice even doing so once a second was enough to virtually eliminate the variance in read throughput.

I think about this incident often when considering requirements and am reminded to carefully consider non-requirements, too. Here, freshness and low memory usage were non-requirements. The older implementation was simple but much slower than necessary because it fulfilled requirements which were unnecessary.

Undoing Lock Contention

Here's another example. More recently, I was supporting a disk-based time-series database, written in C++. This system had a mixed read/write workload, which is typical for time-series systems. The writes were small, usually containing a single point for a lot of metrics, and there were a lot of them. The reads were far fewer, but far larger, often fetching data for a single metric across a wide range of time.

My team was being paged because the error budget of our query availability was being depleted—slowly, but fast enough that we would run out by the end of the month if we did nothing. We could correlate the start of the problem with an organic increase in traffic, so we assumed that the problem would remain until we solved it—or until our customers got fed up and the traffic went away. We mitigated the problem by throwing extra capacity at it, but decided to investigate further.

We determined that a small fraction of the synthetic queries issued by our probers were taking so long to complete that they were timing out. They seemed to occur randomly (in both time and space) but, curiously, appeared to be correlated with small spikes in the fraction of *all* queries timing out. The problem was rare enough that we didn't have any relevant traces available, so we increased the fraction of traces until we caught a few of them. The same pattern presented in all of them: the query appeared to be fanning out to a few storage nodes, as expected, and returning quickly from all but one of them, which timed out.

We examined various metrics emitted by the node where the timeout occurred, around the time it did. RPC server latency was typical at the 90th percentile, but it spiked around the 98th for less than a minute, then went back to normal. CPU load was normal. Memory usage was up by a small amount. IOPS was as expected. None of these things seemed to be the cause, so we looked into the implementation. What causes random latency spikes when not under any kind of load?

The nodes in question had two jobs: store incoming data and make it available for querying. The implementation was roughly as follows: each unique time series was stored as a buffer of (timestamp, value) pairs. To quickly look up these series, a central metadata object served as an index, holding nested maps of field names and values, which in turn held pointers to the buffers. This was a big object, and it was protected by one big lock.

Writes and queries were able to scan for matching series while holding a reader lock, meaning that many such scans could occur at once, and the object would not change under them. Upon finding the pointers to the relevant series, points were appended or fetched from the vectors, which were protected by another read/write lock. But there was a special behavior for writes containing new series. Those were not present in the metadata object, and the buffers didn't exist. So before inserting the points, the implementation took an exclusive (writer) lock, allocated the new buffers for each new series, and inserted the relevant elements to the metadata object.

Experts speculated that the cause of those read latency spikes was likely to be *lock contention* on this metadata object. This was confirmed with instrumentation and profiling.

Unlike in my previous example, these nodes were resource-constrained, and these metadata objects already accounted for a significant fraction of the total RAM usage. We couldn't trade space for speed. We needed to make the writers hold the locks for less time.

We accomplished this by replacing the global metadata lock with narrow locks on the individual nested objects within it. When a write included previously unseen series, it would lock only the relevant map while inserting. This went all three levels deep (metric names, field names, and field values), resulting in many small locks instead of one large one. Writes might need to acquire multiple nested locks, but each was brief, and blocked only a fraction of reads rather than all of them. The new implementation was far more complex and idiosyncratic than the original, and it was right that it was put off. But when the time came, it was very satisfying to see it replaced with something so much more performant.

This project taught me that as throughput increases, so too does the importance of careful locking. Even very brief pauses can have a large impact if they're blocking many requests.

Conclusion

These experiences, and others, have changed my approach to growing and maintaining software. I'm writing about them because I wish that I'd become convinced sooner that this fundamental knowledge was important and worth studying, and perhaps concrete examples would have helped.

Finally, some unsolicited advice: Next time you're faced with a persistent performance or reliability problem, by all means do what is necessary to mitigate the problem first, but consider, then, identifying the underlying bottleneck. Are the performance characteristics of your data structures misaligned with the shape of your actual workload? Has some value of n become too large to ignore? These problems can be solved, and we must not be afraid to do so.

Save the Dates!

USENIX ATC '21

JULY 14–16, 2021
SANTA CLARA, CA, USA

Co-located with OSDI '21

Paper submissions due:
Tuesday, January 12, 2021

The 2021 USENIX Annual Technical Conference brings together leading systems researchers for the presentation of cutting-edge systems research and the opportunity to gain insight into a wealth of must-know topics, including virtualization, system and network management and troubleshooting, cloud and edge computing, security, privacy, and trust, mobile and wireless, and more.

PROGRAM CO-CHAIRS

Irina Calciu
VMware Research

Geoff Kuenning
Harvey Mudd College

www.usenix.org/atc21

OSDI 21

JULY 14–16, 2021
SANTA CLARA, CA, USA

Co-located with USENIX ATC '21

The 15th USENIX Symposium on Operating Systems Design and Implementation brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software. The symposium emphasizes innovative research as well as quantified or insightful experiences in systems design and implementation.

PROGRAM CO-CHAIRS

Angela Demke Brown
University of Toronto

Jay Lorch
Microsoft Research

www.usenix.org/osdi21

COLUMNS

Book Review

Implementing Service Level Objectives by Alex Hidalgo

LAURA NOLAN

Laura Nolan's background is in Site Reliability Engineering, software engineering, distributed systems, and computer science, with a career split roughly evenly between software engineering and SRE-like roles. She has contributed to a number of SRE books, including *Site Reliability Engineering, Seeking SRE*, and *97 Things Every SRE Should Know.* Outside of work, Laura is a part-time student of technology ethics at Dublin City University and is an active campaigner against autonomous weapons. Laura currently serves on the board of USENIX.
laura.nolan@gmail.com

In the past two years, Service Level Objectives (SLOs) have become almost synonymous with Site Reliability Engineering (SRE). SLOs are a reliability target—a threshold of availability and correctness that the users of a service should be satisfied with and that the service ought to be able to meet under normal circumstances.

Site Reliability Engineering [1], published in 2016, set out SLOs as a foundational topic: "It's impossible to manage a service correctly, let alone well, without understanding which behaviors really matter for that service and how to measure and evaluate those behaviors." The *Site Reliability Workbook* [2] upped the ante in 2018, saying, "[SREs'] day-to-day tasks and projects are driven by SLOs: ensuring that SLOs are defended in the short term and that they can be maintained in the medium to long term. One could even claim that without SLOs, there is no need for SREs."

SLOs appear to be simple—we just need to choose how many nines we want—and SLO adoption has often been held up as the first step on any organization's path towards adopting SRE practices. There is a school of thought that sees SRE as a cookie-cutter approach that can be generically applied to any service: just define your SLOs, configure your error-budget-based alerting, build a release pipeline with canarying and rollback, automate away the bulk of your repetitive work, adopt the Incident Management System, and do blameless postmortems and voila—your service will be reliable. Now, these are all worthwhile practices for sure, but are they enough? I believe not. They will get you part of the way there, and it's a good roadmap for productionizing a greenfield project. However, any sizable real-world system will have its own challenges, rough edges, and sharp corners. You need depth and a lot of context to run systems well, not just a cookie-cutter shallow SRE process. If you want to be an SRE for a database tier, you will need to learn a lot about databases in general and your database in particular to do it well. If you are SRE for Java-based services, you need to understand the JVM as well as your services' design, and so on.

Because context is so important, I personally believe that setting up a structured weekly production meeting with comprehensive notes and solid tracking of action items is actually a better first starting point than SLOs with a team new to SRE—you use it immediately to build shared context on services and identify burning fires and pain points that can be mitigated quickly. This shared context becomes a useful foundation for defining SLOs. But deep service expertise is not generic—it's qualitative, not quantitative, and it takes time to build. It's not as easy to write an article or a book chapter about it. You can't create a platform to sell Context-as-a-Service, there are no clever-sounding acronyms, and there are no graphs for executives. In short, it isn't going to help you sell anything or get you promoted (not directly, anyway).

As SREs go, therefore, I'm something of an SLO skeptic. However, even I concede that though SLOs may not be a silver bullet, they are nonetheless useful, and stable SRE teams ought to ensure that their services have appropriate SLOs. SLOs do have a lot of benefits: they can provide an explicit "contract" of sorts between services provided by different teams, helping create clarity about expected reliability and customer needs. SLOs can help you set alerting thresholds and feed into decision making about priorities (without being the sole input to that process).

Nines Are Not Enough

It's been encouraging to see the conversation around SLOs gain nuance and depth in the past year, compared to the fairly basic treatments in the original 2016 *Site Reliability Engineering* [1] and the 2018 *Site Reliability Workbook* [2]. Mogul and Wilkes' HotOS 2019 "Nines Are Not Enough" paper [3] is required reading for anyone interested in the topic—it includes an analysis of some of the real-world complexities and tradeoffs of providing SLOs, including the role of customer behavior; no system can defend an SLO when arbitrary behaviors are allowed. Narayan Desai's talk on SLOs, "The Map Is Not the Territory" [4], discusses a different set of difficulties, particularly around how the aggregation process can mask significant customer pain, especially for low-QPS services or unevenly distributed errors that affect some customers much more than others. Finally, August 2020 saw the release of an entire book [5] dedicated to SLOs: Alex Hidalgo's *Implementing Service Level Objectives*.

Hidalgo introduces the core concepts—SLOs, service level indicators (SLIs), and error budgets—in a similar way to the SRE book [1] and SRE workbook [2] but spends time considering SLOs for significantly more "shapes" of services: not only simple RPC or HTTP services, but also datastores, compute platforms, pipelines, and batch jobs. Data reliability gets an entire chapter, written by Polina Giralt and Blake Bisset, which proposes 13 properties of data, such as freshness, accuracy, and completeness, along with discussions on how to measure these. This chapter is particularly welcome: many of us are running either datastores or pipelines or both. The properties of data-intensive systems are both more complicated than those of the simple request-processing systems normally used to illustrate SLOs, as well as usually more difficult to measure.

Chapter 4, "Choosing Good SLOs," is the foundation for the whole book. Again, it covers a lot of the same ground as the SRE book and SRE Workbook, but with some valuable additions. There is a particularly good discussion of the organization and operational problem that arises from having too many SLOs—resist the temptation to think that every important metric should have an SLO associated with it. The discussion on SLO composition, meaning how to think about your services' reliability in relation to that of the services you depend on, is valuable and doesn't shy away from detail. Toby Burress and Jaime Woo's chapter on probability and statistics develops this further, alternating between theory of probability and statistics and concrete applications to difficult SLI calculations (such as infrequent batch jobs, requests that can be retried), and latency in queueing systems.

Burress and Woo's chapter and the "Architecting for Reliability" chapter by Salim Virji are very useful treatments of the math involved in building (or modifying) services to meet a desired SLO. Hidalgo places a lot of emphasis on getting SLIs right, which is very worthwhile because this is often much more difficult in practice than the introductory examples from the SRE book suggests. There is significant material on the details of computing SLIs, including fairly well-known best practices such as use of percentiles rather than means and how to deal with time windows, as well as less well-known practical problems such as infrequent events and noisy or low-quality data. This is built on later by Ben Sigelman's chapter on measuring SLIs and SLOs, which discusses the tradeoffs involved in computing SLIs from time-series databases and structured event databases (or logs) and distributed traces. Sigelman's chapter usefully points out a number of traps for the unwary, such as relying on metrics reported by potentially malfunctioning services as opposed to other systems' view of those services.

Niall Murphy's chapter on SLO-based alerting rounds out the section of the book that is focused on the technical details of implementing SLOs. I particularly like that this section presents a progressive set of steps for improving your alerting in a brownfield situation. There is a valuable discussion of how to set up separate long-duration and short-duration alert thresholds to detect both major short term-problems and significant but slower-burning issues that are consuming your error budget at a higher than anticipated rate. This valuable alerting pattern is not used widely enough, but it is an excellent mechanism for catching serious but not immediately catastrophic problems without causing excessive pager noise.

The "Worked Example" chapter puts together a set of SLOs for several user-facing and internal systems with a variety of architectures and requirements. The author does a consistently good job of putting the end-user experience front and center here and relating it to the SLOs and SLIs proposed. However, most of the systems and SLOs proposed are fairly simple, and this chapter could do more to reinforce Giralt and Bisset's chapter on data systems, or Murphy's chapter on alerting.

The book closes with a series of less technical chapters on the theme of building an SLO culture, discussing topics like setting up SLOs in organizations new to the concept, how your SLOs may evolve as your service changes over time, how to make your SLOs discoverable to other teams, and how to advocate for SLOs. The final chapter (on SLO reporting) contains a fairly lengthy polemic on why SLO reporting is superior to reporting based on Mean Time To Recovery (and similar metrics)—Hidalgo is right to say that these kinds of measurements are subjective and not generally meaningful because incidents are so different from each other.

Any engineer who works day-to-day with reliability, metrics, monitoring, and alerting ought to have a copy of this book. Even those who don't necessarily want to see how deep the SLO culture-change rabbit hole goes will gain much from the technical chapters, which can inform your monitoring and alerting strategies and even the tradeoffs made in your system architecture.

References

[1] "Service Level Objectives," Chapter 4 in B. Beyer, J. Petoff, N. R. Murphy, and C. Jones, eds., *Site Reliability Engineering: How Google Runs Production Systems* (O'Reilly Media, 2016).

[2] "Implementing SLOs," Chapter 2 in B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, eds., *Site Reliability Workbook: Practical Ways to Implement SRE* (O'Reilly Media, 2018).

[3] J. C. Mogul and J. Wilkes, "Nines Are Not Enough: Meaningful Metrics for Clouds," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19),* pp. 136–141: https://dl.acm.org/doi/pdf/10.1145/3317550.3321432.

[4] N. Desai, "The Map Is Not the Territory," at SREcon19 EMEA 2019: https://www.usenix.org/conference/srecon19emea/presentation/desai.

[5] A. Hidalgo, *Implementing Service Level Objectives* (O'Reilly Media, 2020).

Systems Notebook
What's in That Container?

CORY LUENINGHOENER

Cory Lueninghoener makes big scientific computers do big scientific things, mainly looking at automation, scalability, and large-scale system design. If you don't see him hanging out with the LISA and SREcon crowd, he's probably out exploring the mountains of northern New Mexico.
cluening@gmail.com

Have you ever opened your refrigerator to get a tasty snack and caught sight of that one container in the back, the one that is unmarked but you know has been there since sometime last June? And as you close the door, you kind of wonder if it just moved a little? Have you ever felt the same way about Linux containers running on your servers? What exactly is in there? And how did they get there in the first place?

Containers on Linux have been the new hotness for some time now, which I suppose makes them pretty hot indeed. But despite the ubiquity of containers today, a lot of us still only interact with them by running `docker run`. Not one to take a whale for its word, I think it's worth looking more deeply at what's really going on. While there's only room to scratch the surface, over the next several pages I'm going to take a look at what Linux containers are made of, how to create a super-simple container using a few command line tools, and how to use those same tools to understand what Docker is doing under the covers.

But first, some caveats. Mentioning "Linux containers" can cause strong reactions among some people, so I want to state upfront this isn't going to be a column about the security implications of containers, how various operating systems provided the same functionality earlier (or better), what the exact definition of a "container" is, or anything else like that. This is just a quick look at (spoiler!) how Linux namespaces are used to provide container functionality. There are some simplifications in here for the sake of brevity and clarity, so forgive me if I leave out your favorite details about Linux containers. If you want a real deep dive into everything here, take a look at the references at the end of this column.

It's All Part of the Process

Back in the age of dinosaurs, when operating systems textbooks were written, you may recall learning that a *process* is the embodiment of a program running on a UNIX-like system. It contains the program code itself, as well as its active memory, a pointer to what instruction is currently running, and various other bookkeeping data structures. A booted system starts out with a single process running, process ID (PID) 1, and all other processes on the system can trace their lineage through a series of `fork()` and `exec()` system calls back to that initial process. All running processes are given a unique PID number, and, by default, all processes exist in a global shared *namespace* that lets them see information about all other processes currently running on the system. On a UNIX-like system, much of the information about running processes is presented to users in the `/proc` file system. There, the information is organized by directories named after processes' numeric IDs.

What if, instead of process information existing in a global namespace, processes could have their own independent views of what `/proc` looked like? In this scenario, after a process forks, the parent process would be told that its child got an incrementally higher PID, while the child process would be told that it is PID 1: the first process on a fresh system. Everything else would be shared between these processes—the kernel, file systems, users—but the new process would be in a new process namespace and have a new, empty view of what other processes exist on the system.

Can You Guess My Namespace?

This world exists, and it has existed since 2007 when kernel version 2.6.24 introduced *PID namespaces*. Similar to how variable namespacing in a programming language can keep variables in one function hidden from variables in another function, namespaces in the Linux kernel can create private views of kernel data for different processes. When a process creates and joins a new PID namespace, the kernel tells it that it is PID 1 and the only process running on the system. All descendants of this new PID 1 will be put in the same PID namespace, and their view of the running system will be limited to the contents of their namespace.

There are two important things to note about this functionality: one is that PID namespaces are created in a hierarchy, much like the way that processes are created. This means that processes higher up in the namespace hierarchy can see all of the processes in PID namespaces that exist below them, while processes in leaf namespaces can only see processes that are members of their own PID namespace. The other is that multiple PID namespaces can exist at the same time, meaning a system can have many processes running on it that all believe they are PID 1.

But That's Not All

PID namespaces aren't the only namespaces that can be created, and they weren't even the first ones to be included in the Linux kernel. That distinction belongs to mount namespaces, which appeared in Linux 2.4.19 in 2002. Today there are eight namespaces available, and they all have the same goal: give processes a private view of certain system resources. Along with the PID namespace that we've already seen, this includes hostname and network information (UTS and Network namespaces), file systems (Mount namespace), system users (User namespace), and time, resource, and IPC objects (Time, Cgroup, and IPC namespaces).

In its simplest form, a Linux container is nothing more than processes in one or more private namespaces. But looking at the list of available namespaces, you can start to imagine how you could use them to turn simple processes into something that looks like an entirely new computer without relying on starting up a virtual machine.

Where Does It Come From?

Like many kernel internals related to processes, the bookkeeping that makes namespaces work is exposed to userspace in `/proc`. Any process running on a modern Linux kernel has a `/proc/[PID]/ns` directory associated with it, and the namespaces that that process belongs to are presented as symbolic links within that directory. For example, to look at the namespaces that your current shell belong to, you can do the following:

```
[root@localhost ~]ls -l /proc/$$/ns
total 0
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 cgroup ->
  'cgroup:[4026531835]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 ipc ->
  'ipc:[4026531839]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 mnt ->
  'mnt:[4026531840]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 net ->
  'net:[4026531992]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 pid ->
  'pid:[4026531836]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 pid_for_children ->
  'pid:[4026531836]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 user ->
  'user:[4026531837]'
lrwxrwxrwx. 1 root root 0 Aug 24 03:40 uts ->
  'uts:[4026531838]'
```

The numbers that the links point to are unique identifiers for each of the namespaces on the system, and any processes that share those numbers also share that particular namespace.

A program can use the `clone()` system call function with appropriate flags to create a copy of itself in a new namespace, ready to be replaced with a call to `exec()`. Meanwhile, an existing process can use the `unshare()` call to create and join private, non-shared namespaces or `setns()` to join existing namespaces. Each of these functions accepts a set of flags that specify what new namespaces to create.

Alternatively, the `unshare` and `nsenter` command line tools, provided by the `util-linux` package, can be used to create processes that are in new namespaces or members of existing namespaces from the command line. These tools provide command-line options to control which namespaces are created or joined.

Let's Get Our Hands Dirty

Let's take a look at namespaces on a real system. Using the `unshare` command line tool, it is easy to create a new process with one or more namespaces that are unique from its parent. We'll start by creating a new shell that's in a new PID namespace, but shares its other namespaces with its parent. With that new shell created, we can look at what processes are visible both inside and outside this miniature "container" and how to add more processes to it.

You can follow along on your own system: all of these examples were run on a CentOS 8 virtual machine booted up using Vagrant and VirtualBox, and for clarity each line of the shell session has been prefixed with `[o]`, for outside of the new namespace, or `[i]`, for inside the new namespace.

First, we'll use `unshare` to create a new shell in a new PID namespace.

```
[o] [root@localhost ~]unshare --fork --pid --mount-proc
  /bin/bash
[i] [root@localhost ~]#
```

At this point, the unshare command has started a new copy of bash with a new PID namespace. Both the old shell and the new shell have the same prompt, so it's kind of anticlimactic. But recall that in the previous section we saw that the list of namespaces a process belongs to is exposed in /proc/<PID>/ns. If you compare the new shell's namespaces against the shell in the previous section, you can see that the new shell is indeed in a new PID namespace:

```
[i] [root@localhost ~]ls -l /proc/$$/ns
[i] total 0
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 cgroup ->
  'cgroup:[4026531835]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 ipc ->
  'ipc:[4026531839]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 mnt ->
  'mnt:[4026532155]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 net ->
  'net:[4026531992]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 pid ->
  'pid:[4026532156]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 pid_for_children
  -> 'pid:[4026532156]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 user ->
  'user:[4026531837]'
[i] lrwxrwxrwx. 1 root root 0 Aug 24 03:42 uts ->
  'uts:[4026531838]'
```

Looking closely, you'll notice that the new shell is also a member of a new Mount namespace. The unshare man page explains why in its PID namespace section:

"It also implies creating a new mount namespace since the /proc mount would otherwise mess up existing programs on the system."

So we gained two namespaces for the price of one.

Since this new shell is a member of a new PID namespace, the only processes it knows about are itself, which it sees as PID 1, and its descendants. We can see this by running the ps command:

```
[i] [root@localhost ~]ps -ef
[i] UID          PID    PPID  C STIME TTY          TIME
  CMD
[i] root           1       0  0 03:41 pts/0    00:00:00
  /  bin/bash
[i] root          18       1  0 03:44 pts/0    00:00:00
  ps -ef
```

Meanwhile, this same process is visible from the system's default namespaces with a different PID number. It just takes some sleuthing to find it. Starting up another login shell on the system, we can find the namespaced process by looking for the original unshare process and examining its only child. Here we find that the parent namespace identifies our namespaced shell as PID 33783:

```
[o] [root@localhost ~]ps -ef
[o] UID          PID    PPID  C STIME TTY          TIME
  CMD
   ...
[o] root       33782    5283  0 03:41 pts/0    00:00:00
  unshare --fork --pid --mount
[o] root       33783   33782  0 03:41 pts/0    00:00:00
  /bin/bash
   ...
```

After one process creates a new namespace, other processes can join it. Having found our new container process from outside of its PID namespace, we can also start a new shell within its new PID namespace. With the original namespaced bash process still running via unshare, we can use the nsenter command to join it by targeting its external process ID:

```
[o] [root@localhost ~]nsenter --all --target 33783 /bin/bash
[i] [root@localhost /]ps -ef
[i] UID          PID    PPID  C STIME TTY          TIME
  CMD
[i] root           1       0  0 03:41 pts/0    00:00:00
  /bin/bash
[i] root          19       0  0 03:47 pts/1    00:00:00
  /bin/bash
[i] root          34      19  0 03:47 pts/1    00:00:00
  ps -ef
```

Let's review what all we just did: starting with a fresh virtual machine, we created a new process in a new PID namespace, confirmed that it appeared as PID 1, and started another new process inside that same namespace. Now, let's take it a step further.

Let's Build a Simple Container

Let's get one step closer to a full Docker-style container by building a new operating system image and starting processes using it. CentOS includes the debootstrap package, which can be used to install a full Ubuntu system inside of a single directory tree on a CentOS system. We can use that tool to create an Ubuntu file-system image in /root/ubuntu-bionic, and then use unshare along with chroot to create a shell with new Mount and PID namespaces in use. Once that shell is running, it will look exactly like it is running on an Ubuntu system. This can all be done from within a clean CentOS 8 install in a virtual machine.

```
[o] [root@localhost ~]yum install epel-release
    <output trimmed>
[o] [root@localhost ~]yum install debootstrap
    <output trimmed>
[o] [root@localhost ~]mkdir ubuntu-bionic
[o] [root@localhost ~]debootstrap --arch=amd64 bionic
  /root/ubuntu-bionic/ http://mirrors.vcea.wsu.edu/ubuntu/
    <output trimmed>
[o] I: Base system installed successfully.
```

```
[o] [root@localhost ~]unshare --fork --pid --mount-proc
  --mount chroot /root/ubuntu-bionic /bin/bash
[i] root@localhost:/mount -t proc proc /proc
[i] root@localhost:/mount -t sysfs sysfs /sys
[i] root@localhost:/ps -ef
[i] UID          PID    PPID  C STIME TTY          TIME
  CMD
[i] root           1       0  0 03:54 ?        00:00:00
  /bin/bash
[i] root          13       1  0 03:54 ?        00:00:00
  ps -ef
[i] root@localhost:/head -2 /etc/os-release
[i] NAME="Ubuntu"
[i] VERSION="18.04 LTS (Bionic Beaver)"
```

This still doesn't fully replicate the full containerization provided by tools like Docker, but we've started to get close. In the last several sections, we've essentially run `docker build` (using `debootstrap`), `docker run` (using `unshare`), and `docker exec` (using `nsenter`). As homework, you can expand this work by combining this same set of commands with other namespaces, giving you the ability to change the hostname, assign private network interfaces, and more.

Now, let's try the same tricks with a real Docker container.

What Was That about Docker?

Way back in the first paragraph of this column, I asserted that many people's main interface to containers is `docker run`. Since then, we've learned that containers are just processes with unique namespace configurations that give them the ability to see different root file systems, different process trees, and the like. When Docker starts a container, it uses the exact same kernel mechanisms we just looked at to get the job done. That means that you can use these same tools to interact with Docker containers, but without the Docker commands. As an example, let's use `nsenter` to replicate the base functionality provided by `docker exec`. As with the earlier examples, everything here was done within a CentOS 8 virtual machine built using Vagrant and VirtualBox.

To start, we'll fire up a simple Docker container and start a sleep inside it so that the process is easy to find:

```
[o] [root@localhost ~]docker run -it ubuntu bash
[i] root@94802998616b:/sleep 300
```

We can find this same process from outside of the container, just like we did before:

```
[o] [root@localhost ~]ps -ef | grep sleep
[o] root       52039   51999  0 04:00 pts/0    00:00:00
  sleep 300
```

Using the `nsenter` command, we can start a new shell that joins all of the same processes that the `sleep` command is a member of:

```
[o] [root@localhost ~]nsenter --all --target 52039 /bin/bash
[o] root@94802998616b:/ps -ef
[o] UID          PID    PPID  C STIME TTY          TIME
  CMD
[o] root           1       0  0 03:59 pts/0    00:00:00
  bash
[o] root           8       1  0 04:00 pts/0    00:00:00
  sleep 300
[o] root           9       0  0 04:01 ?        00:00:00
  /bin/bash
[o] root          12       9  0 04:01 ?        00:00:00
  ps -ef
[o] root@94802998616b:/
```

The new shell is now a member of the Docker container, complete with the container's hostname (`94802998616b`) and the only four processes it knows about (two instances of `bash`, plus `sleep` and `ps` processes). We've just replicated the base functionality of `docker exec` with standard Linux utilities.

Conclusion

Building containers by hand is more of an interesting trick than something that's useful in production, but knowing what's going on underneath Docker, Buildah, Podman, and other container tools gives you greater insight into how to tune, debug, and work with those tools. By understanding the underlying technology and how to access it with lower-level tools, you have a better overall view of how your system works and how to keep it running optimally.

References

If you want to dig deeper into Linux namespaces, here are two great places to start:

Namespaces(7) Linux manual page: https://man7.org/linux/man-pages/man7/namespaces.7.html.

M. Kerrisk, "Namespaces in Operation" series, *The Linux Weekly News*, January 4, 2013: https://lwn.net/Articles/531114/.

iVoyeur
BPF and Histograms

DAVE JOSEPHSEN

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

Argus Panoptes was the all-seeing primordial giant and slayer of the mother of all monsters, Echidna, in Greek mythology. In some tellings he has 100 eyes, some combination of which are always open, though the Renaissance painters (perhaps to save on paint?) always depict him with just the two.

Whether he was a many-eyed giant or merely a very astute, tall man, his powers of observation were so legendary that Hera herself entrusted to Argus the task of guarding the promiscuous nymph Io (inexplicably disguised as a cow) in order to keep her away from Zeus, Hera's unfaithful husband and king of the gods.

Setting aside for a moment the questionable rational of hiring a P.I. to track your cheating husband's *lover* rather than the man himself, Argus proved to be so effective at this task, watching Io day and night, and never once letting her out of his sight for a microsecond, that Zeus eventually had him murdered in order to reunite with his mistress.

I guess the all-seeing Argus *didn't see that comin'*.

That it's often possible to see everything and yet still fail to comprehend is, I think, one of several invaluable lesson Argus Panoptes gave his life to teach us. Every bit as true today as it was in the golden age.

Let's say for example you have a spreadsheet of latency times and other metrics from a front-end web server. With five minutes' worth of samples, you can scroll around and probably tease out some patterns. But with a full day of data, things become vastly more opaque. Ironically, the more you see, the less you begin to comprehend.

In my last column, we talked about the various data structures eBPF uses to pass data from protected kernel space into userspace where we can get our hands on it, and we discovered that our sample BCC tool, `biolatency`, was using a built-in histogram data-structure to summarize data in kernel space. In this issue, as promised, we're going to talk a little about histogram theory and how histograms enable us to make sense of massive amounts of data, thereby achieving comprehension rather than mere observation.

As I'm sure you probably already know, a histogram is a visual representation of a pile of data. Instead of plotting each value in the set, we depict a series of buckets or "bins" which are sized to indicate how many measurements in the sample set fell within the bounds of each bucket. Figure 1 is your obligatory example histogram of 500 measurements, ranging in value from 0 to 100. As you see, it looks like a bar graph, except rather than representing a single metric, each bar represents *the magnitude* of measurements whose value fell between the bar before it, and the bar after it.

Histograms are pretty great because we can depict what the overall data set *looks* like independent of its size. It doesn't matter if the set contains five minutes of data, or five days, we can use the same amount of pixel-space to represent it. Further, histograms are far more representative of the distribution than any combination of statistical reference metrics like

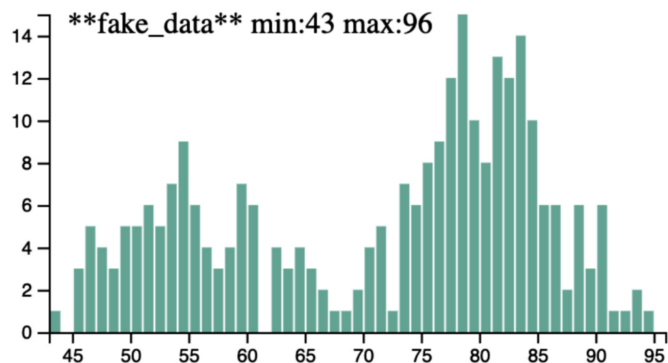


Figure 1: Obligatory example histogram of 500 measurements, ranging from 0 to 100

average, min/max, sum, and p-values, and because they amount to a struct of counters, histograms are super cheap to compute and store.

A histogram's "resolution" is said to vary with its bin-width. You can't accurately represent the value 2.63 if you have bins spaced at integer intervals, for example. That 2.63 would *resolve* to a "3" if our bucket-widths were integer-spaced. So it's important to put some thought into how best to situate the width and total number of bins for a given data set. Obviously, the bin-width decreases (resolution improves) as the total number of bins increases. There are, as you can probably imagine, quite a few strategies to find the optimal total value of bins, or *k value*, for a given data set.

One of the most basic and popular ways of computing the optimal number of bins for a given distribution is the "powers of two" rule, which says the optimal number of bins is the square root of the total number of samples in the set, or for a group of samples s:

$$k = \sqrt{s}$$

This is the formula used internally by Excel histograms and many other simple implementations when we want to provide a cheap, hands-off way of choosing a bin-number. With the powers of two formula, you'd wind up with 16 buckets for a data set with 256 samples in it, for example. There are various takes on this, like Sturges' formula, which uses the base-2 logarithm of the max value in the data set.

$$k = \lceil \log_2 n \rceil + 1$$

There is no universally correct number of bins, and every algorithm carries tradeoffs. Sturges, for example, gives poor results for data sets where n<30 but works well on sets with a large range. It's no accident, however, that we've immediately wandered into the land of squares and base-2 logarithms. As it turns out, base-2 logarithms and histograms share something of a magic relationship in computerland, where the underlying representation of basically everything is a binary number.

So far, we've been talking about histograms whose bins are all the same width, aka "Linear Histograms." But there's no particular reason that this should be true. It's absolutely possible to vary the bin-widths within a set number of bins.

In fact, if we were to vary our histogram bin-widths along $\log_2$ boundaries instead of making them all the same width, each boundary between our bins would represent an order of magnitude increase in the sample set. Another way to state that is: every bucket would represent a consecutive bit in a binary number. Therefore, a base-2 "log linear histogram" can use 64 bins to represent a 64-bit int, which is a very large set [0 - 18,446,744,073,709,551,615].

Again, these buckets are not the same size. Instead, they become exponentially fatter at each boundary. The first bin represents the numbers between 0 and 1. The second, 2 and 4, the next 8 and 16 and so on. Now consider this structure in the context of kernel-based metrics like the disk I/O latency we are trying to measure with `biolatency`, and I think you'll realize that this sort of structure is kind of ideal for our problem domain. Our disk I/O is going to *usually* be great, somewhere on the order of tens of milliseconds, where a powers-of-two histogram's resolution is going to be very good. Then we're going to have a small number of outliers on the order of seconds, or possibly tens-of-seconds.

Most in-kernel latency metrics distribute like this: a vast number of very small-value measurements and then a rare smattering of exponentially larger outliers. Many network metrics fit this pattern even more, with normal measurements near 0 and outliers in the billions. The pattern is so pervasive, that BPF has a helper function, called `bpf_log2l()`, which returns the base-2 log of a given measurement, so you can convert any measured value to $\log_2$ scale in-kernel before passing it into the histogram.

Wait what? Convert the value? I thought we were talking about bin boundaries, not modifying the value of our measurements, Dave.

Well, we are. But you'll remember that both the probe code itself as well as the histogram storage `struct` are in-kernel. The histogram implementation [1] is a bare-bones linear histogram, with a statically configured number of same-sized bins. There is no way to create variable-sized bins. But we can simulate the same effect by creating a 64-bucket in-kernel linear histogram and converting (compressing) our measurements to $\log_2$ scale values before storing them.

Remember, we're not storing the actual values, we're merely incrementing *counters* within buckets that roughly align to our values. So if we compress the scale of our values to $\log_2$, we are effectively creating $\log_2$ indexes; we can come back at print-time in userspace and recompute the *indexes* using the in-kernel

bucket values as a base-2 exponent, and all the counts will neatly line up with the correct magnitude.

Taking a look at `biolatency`'s storage code [2], we see that every time biolatency commits a value to the data-structure, it uses the `bpf_log2l()` helper function. This is somewhat obfuscated by the find/replace pattern in the Python BCC tools, but the invocation to create the `HISTOGRAM` looks like this:

```
BPF_HISTOGRAM(dist);
```

It's possible to pass in a bin number and a data-structure to represent the index values, but the log-linear use case is so pervasive in BCC that the defaults are aligned to our use case, and the above invocation will create a 64-bin, int-indexed histogram called "dist". We write to it like so:

```
dist.increment(bpf_log2l(delta));
```

Where "delta" is a u64 representing the difference in nanoseconds between the `blk_account_io_start` and `blk_account_io_done` kprobes firing. The kernel will use the delta value to find or create the appropriate bucket and add a +1 to it.

At the end of script-execution, when the userspace side of `biolatency` catches a `Ctl-C`, it grabs the histogram from kernelspace [3] using `get_table()`, the same function we used to grab `structs` from userspace in my previous article. Python BCC has a print-function that knows how to re-compute the indexes of log-linear histograms for us, so all we need to do is pass the dict [4] into the `print_log2_hist()` function, passing along the appropriate labels to make the resulting graph more human readable.

I sometimes wonder what Argus Panoptes would make of the modern world. Thinking about him as a sort of spherical-cow of observation, the hypothetically perfect monitoring machine. Would he be blinded by the abundance of spread-spectrum data being flung in every direction about our heads? Would he be mesmerized to the point of paralysis at the sight of a computer, with its unending infinitesimal internal machinations?

Personally, I vastly prefer to work with instrumentation systems like eBPF, which reward system-knowledge and rely on an interrogative question/answer relationship between operator and machine, over the packaged *measure everything* monitoring systems of the world. I think this is probably true of most engineers. Certainly the ancient Greeks agree, who valued as priceless the oracles, while relegating the spherical-cow of observation to, well, cow-watching. I think that puts us in pretty good company.

Take it easy.

References

[1] https://github.com/torvalds/linux/blob/master/Documentation/trace/histogram.rst.

[2] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py#L127.

[3] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py#L198.

[4] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py#L210.

SIGINFO

The Tricky Cryptographic Hash Function

SIMSON L. GARFINKEL

Simson L. Garfinkel is a Senior Computer Scientist at the US Census Bureau and a researcher in digital forensics and usability. He recently published *The Computer Book*, a coffee table book about the history of computing. sigmail@simson.net

Cryptographic hash functions are one of the building blocks of modern computing systems. Although they were originally developed for signing digital signatures with public key cryptography, they have found uses in digital forensics, digital timestamping, and cryptocurrency schemes like Bitcoin.

Cryptographic hash functions like MD5, SHA-1, and BLAKE3 are widely used and appreciated by programmers, end users, and even lawyers! Nevertheless, I'll start off this column with a basic description of what hash functions are and the hash functions that are used today. Then I'll delve back to the first references to them that I've been able to find and give a bit of their history. I'll briefly touch on their uses in cryptography and then discuss how they also found use in digital forensics. I'll end with a puzzle from Stuart Haber, one of the co-inventors of the blockchain concept. Unless otherwise noted, all of the timing runs were performed on my Mac mini (vintage 2018) with a six-core Intel Core i5 processor running at 3 GHz. The hashing was done with OpenSSL 1.1.1d, compiled September 10, 2019, that ships with the Anaconda Python distribution.

Hash Functions

Hash functions take a sequence of bytes of any length and crunch it down to a block of seemingly random bits and a constant length. This block is typically called the *hash*, taken from the popular dish that involves chopping up food and then cooking it together.

Hash functions are widely used in computer science—they are the basis of the Python dictionary, for example. The basic idea of hashing was invented by IBM scientist Peter Luhn back in the 1950s as a technique to help speed up searching for words in text [1].

Cryptographic hash functions are fundamentally different from the hash functions that Luhn developed. For starters, their output is much larger. Python's hash function takes a string and returns an `int`—that is, 32 or 64 bits—which then becomes an index into an array (modulo the size of the array). Cryptographic hash functions return more than a hundred bits, each (ideally) with an equal and independent probability of being a 0 or a 1, which is then used as a kind of placeholder for the object itself. Writing in RFC 1186 back in 1990 about his MD4 algorithm, Ron Rivest stated: "The algorithm takes as input an input message of arbitrary length and produces as output a 128-bit 'fingerprint' or 'message digest' of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest."

The field of cryptographic hash functions has evolved considerably since 1990. Today we say that these functions should have several properties. First, it should be computationally infeasible to find a sequence of bytes that has a specific hash, called *pre-image resistance*. It should also be infeasible to find a second sequence m_2 that has the same hash as a first sequence m_1, called *second pre-image resistance*, or to find any two objects that have the same hash, called *collision resistance*. Finally, the output of the hash function should be indistinguishable from a random number generator. That is, there should be no way to predict the output of the hash from its input other than by running the actual hash function. This is called *pseudo-randomness*.

Cryptographic hash functions were first described in detail by Ralph Merkle in his 1979 PhD thesis [2], published just a few years after Diffie and Hellman introduced the world to public key cryptography and Rivest, Shamir, and Adleman disclosed the public key system that has memorialized their initials. Merkle called the functions "one-way hash functions," because it was easy to take a message and find its corresponding hash, but "effectively impossible"—or at least "computationally infeasible"— to take a hash and find a corresponding message. The RSA cryptosystem can't sign a number larger than the product of the prime numbers *p* and *q*—which today is typically a few thousand bits. Given a one-way hash, Merkle wrote, the newfangled digital signature schemes could be used to sign a message of any length: simply hash the message first, then sign the hash.

The idea of hashing a message and then signing the hash is standard operating procedure, but back in 1979 this was brand new stuff. What I find so enchanting about Merkle's PhD thesis is the combination of wonder, excitement, and amazement it conveys. Merkle's words help me to understand what it was like to live in a world where public key cryptography was new and nobody really knew how to use it or even quite what to do with it.

Today we know lots of things that you can do with hash functions—even without public key technology. In his PhD thesis, Merkle shows how it's possible to create digital signatures with just a one-way hash function. We now call these *Merkle signatures*. The critical insight is that you can take a secret message (call it M0) and hash it (call that H0). If you hand-deliver H0 to a friend today, you can send an authenticated message to your friend at some point in the future by sending M0. Your friend can verify the authenticity of M0 by hashing it and producing H0. In his thesis, Merkle credits this original idea to Leslie Lamport, as described in Diffie and Hellman's original "New Directions" [3] article, although Merkle notes that the scheme is much more efficient using cryptographic hash functions.

Of course, just being able to send a 0 by itself is not useful. So instead of giving your friend just H0, you give the friend H0 and H1 (which is the hash of M1). Now you can send one bit of authenticated information—either a zero or a one—by choosing to reveal either M0 or M1. Give your friend 256 different H0s and 256 different H1s, and you can now send 256 bits of digitally signed data. The disadvantage of this scheme is that each signature block can only be used once, so it's not tremendously efficient (although there are ways around this problem as well). The advantage of Merkle signatures is that they are very fast to compute, and it is widely thought that they are resistant to cracking by quantum computers, should such machines ever become practical.

If you want to sign 10 documents at the same time, you can compute the hash of each document (call that DH0 through DH9), then concatenate all of these hashes together, hash the resulting block (call that DHH), and sign that. You can prove the signature of any document by giving someone that document, the hashes for the other nine documents, and the signature for DHH: the verifier recomputes the missing document hash, uses DH0 through DH9 to compute DHH, and verifies that. This approach and the corresponding data structure, when extended to multiple levels, is now called a Merkle Tree.

The Rise and Fall of Many Hash Functions

The first widely used cryptographic hash function was MD2, developed by Rivest for use in an early secure email system. The source code for MD2, dated October 1, 1988, appears in RFC 1115, one of the early RFCs describing a system for sending encrypted messages over the Internet. This system was never widely adopted, but its ideas and data formats were quite influential.

Although no practical attack on MD2 was ever published, researchers started publishing theoretical attacks against it in 2004. Support for MD2 was removed from the popular OpenSSL cryptographic toolkit in 2009. But the real problem with MD2 wasn't its security but its speed: MD2 is an extraordinarily slow algorithm. Even on my 2018 Mac mini, Rivest's 1988 code takes 43 seconds to hash 256 MiB of data. Imagine how slowly it ran back in 1988!

Rivest went back to the drawing board. MD3 didn't make it out the door, but MD4 was released and appears in RFC 1186 (October 1990). Flaws were soon discovered in MD4 and it was not widely used. In 1991, Rivest invented MD5; the algorithm was published by the Internet Engineering Task Force (IETF) in April 1992 as RFC 1321.

MD5 is more than a hundred times faster than MD2; on my Mac mini, OpenSSL's MD5 implementation hashes that same 256 MiB file in just 0.37 seconds. Like MD2, MD5 also produces a 128-bit hash.

MD5 is still in use today, but it should no longer be used because it is now relatively straightforward to generate two blocks of data that have the same MD5 hash. That is, MD5 no longer has collision resistance. The first MD5 collision was demonstrated back in 2004; the Wikipedia article on MD5 has a nice write-up about how to create two documents that have an MD5 collision.

On the other hand, there is still no publicly known attack on MD5 that will let you find a block of data with a specific MD5 hash—that is, it still is publicly considered to have *pre-image resistance*. Nevertheless, MD5 is not to be trusted. For example, Amazon's Simple Storage Service (S3) still uses the MD5 algorithm for the "ETag" value that lets users check the integrity of uploaded files. The idea is that your software can compute the MD5 of a file, upload the file to S3, and then check the file's ETag to see if the value is the same. Although this works in practice, if you happen

to upload two files that have the same MD5, Amazon will happily give them both the same ETag.

In digital forensics, it's common to use file hashes to search a computer for *files of interest*, a broad term that includes stolen corporate documents, child sexual abuse materials, and other kinds of documents sought by investigators. Typically, an organization looking for materials will distribute a list of hashes for such files to investigators in the field. The investigators then run a program that hashes every file on a suspect's laptop and sees if any of those files has a hash that matches the list. If there's a match, then the suspect presumably has the file of interest. MD5 is still used in this application: after a collision is found, the investigator then looks at the matching file to see if it is in fact the file being sought.

Nevertheless, even in these applications, I try to avoid using MD5. That's because there are many articles on the Internet telling people not to use MD5 because it is not secure. I just don't think that it's a good use of one's time to argue that it's acceptable to use MD5 for some applications but not others.

Another hash function that is in wide use is SHA-1, the Secure Hash Algorithm, adopted by the National Institute of Standards and Technology in 1995. SHA-1 produces a 160-bit hash. Even though concerns about SHA-1 were raised within a few years of its being published, the National Institute of Standards and Technology (NIST) didn't formally recommend that we stop using SHA-1 until 2006. Eleven years later, Google published two PDFs that had identical SHA-1 hash values but render differently [4]. The two files are each 422,435 bytes long and differ in 62 bytes. They also look visually similar, except that one has a blue banner across the top while the other has a red banner.

As Andrew Tannenbaum once said, the nice thing about standards is that there are so many of them to choose from. Realizing that SHA-1 was likely to be compromised, in 2001 NIST revised the Secure Hash standard to allow for more rounds of computation and longer hash values, also called *residues*. Collectively called SHA-2, these revised algorithms include SHA-256, SHA-384, and SHA-512. In 2006 NIST initiated a competition for a new Secure Hash Algorithm. Nine years later NIST declared that an algorithm called Keccak would be adopted as SHA-3. This new algorithm is based on a fundamentally new mathematical approach called a sponge construction, in which input data are absorbed and then the hashed value is squeezed out. For details about these algorithms, as well as the multiple controversies surrounding their adoption, I refer you to the Wikipedia pages for SHA-1, SHA-2 and SHA-3.

It used to be the case that MD5 was dramatically faster than SHA-1, which was faster than SHA-256 (the 256-bit version of SHA-1), which was faster than SHA-512. That's no longer the case, in part due to better implementations and in part due to the fact that we're now running on 64-bit processors. In Table 1, I present the times to hash a 1 GiB file with several of the algorithms I mentioned above.

Bits	128	160	256	384	512
Family					
MD5	1.45				
SHA-1		1.03			
SHA-2			2.18	1.48	1.48
SHA-3			2.65	3.45	4.90

Table 1: Time in seconds to hash 1 GiB using OpenSSL 1.1.1d on the author's 2018 Mac mini

Hashing in Digital Forensics

Beyond searching for contraband, over the past three decades, digital forensics researchers have developed approaches to use cryptographic hashes for authenticating evidence, searching for file fragments, and even gauging file similarity. We can now even search a hard drive for contraband data in less time than it takes to read the hard drive's contents! These more sophisticated approaches have yet to be widely adopted, showing the difficulty of moving techniques from the lab to the field.

There are many definitions of digital forensics, but most of them link it to the recovery and analysis of digital information. Digital forensics techniques have many uses, including data recovery, event reconstruction, malware analysis, and even analyzing systems for the leakage of personal information. One of the best-known uses of digital forensics, though, is taking data from devices that were used by criminals and using that data as evidence in a court of law.

One of the early uses of cryptographic hash functions in digital forensics was to certify that the copy of a hard drive made by an investigator had not been changed after it was acquired. Forensics software would make a copy of the hard drive, called a *disk image*, and then compute the cryptographic hash of the disk image. The investigator would then write this hash in ink into their investigator's notebook. Although the computer scientist in me wishes that the early programs would have then signed this hash with a private key, this pen-and-paper record provided sufficient validation for US courts.

The fact that you could make two, five, or even 50 disk images of the same hard drive and they would all have the same hash engendered a lot of confidence in this basic digital forensics technique: a hashed disk image became the gold standard of digital evidence preservation and created the assumption that the data in the disk image was unchanged since the disk was seized from the suspect. Of course, this assumption was wrong: a crooked officer could easily have planted the incriminating evidence on

the hard drive *before it was imaged*. Such malfeasance is rare, fortunately, and there are other forensic techniques that can both detect and defend against such behavior.

These days, hashes are still used to establish that data taken from a digital device hasn't been altered since it was originally captured. However, the ability to repeatedly reimage a device and consistently get the same hash is quickly fading. When a modern operating system deletes a file, it can tell a solid state drive (SSD) to proactively erase the associated flash storage pages using the ATA TRIM command (called UNMAP in the SCSI command set). The drive doesn't immediately erase the page, but it may do so in the future. If the disk is imaged before the pages are erased, the disk image will contain the blocks' now-deleted data. But if the disk is left turned on, it may eventually erase these blocks. If you image the SSD a second time, then the blocks that were erased will now read as zeros, and the second image will have a different hash than the first. It is also increasingly difficult to get a "disk image" of a cell phone, as the data on many cell phones is accessed through an API, rather than by mounting the cell phone's internal storage. Such file collections are sometimes called "logical images."

If you use a hash that is 160 bits long, you can split it into six numbers of 25 bits each (throw out the remaining 10). If you have an array of 2^{25} bits, you can store information relating to that hash by setting the six indices to a 1. Although this is not an effective way to uniquely store the original 160 bits, it has several advantages, especially for digital forensics. If you assemble a list of file hashes for a million stolen documents and store them all in a single 4 MiB Bloom filter, only six million bits (at max) in that Bloom filter will be set. Not only will the Bloom filter be much smaller than the list of a million hashes (which would take up 20MiB, instead of 4MiB) and is much more compressible, it's also significantly faster to search. Of course, when searching a Bloom filter there is always the risk of a false positive—some other document might have a hash that, when chopped into four parts, just happens to match four other partial hashes. This kind of false positive can be an advantage, though, if the files that you are hashing are highly confidential: if the criminal who stole some of your confidential documents manages to steal your Bloom filter, that person won't be able to reverse engineer the Bloom filter and have it spill the hashes of all the documents that you consider sensitive. In either event, the Bloom filter's false positive rate can be tuned as needed for the specific application.

My colleague Vassil Roussev spent several years working with hashes and Bloom filters and developed a metric for determining how similar two files are. The metric works by scanning files for what Roussev called "statistically improbable features" and then hashing a window of 64-bytes and storing the hash in a Bloom filter. When a certain fraction of bits in the Bloom filter fill up, Roussev's algorithm starts on the next filter. With this system, the similarity of two files is proportional to the number of bits that are set in common in the filters. One of the neat things about this system is that you can compare Bloom filters for a small file and a very large file and find out if the small file resides *inside* the larger file. This even works if the larger "file" is an image from a multi-terabyte-sized disk array [6].

Roussev's *similarity digest* overcomes a fundamental problem of using cryptographic hashes to find files of interest. By design, if you change just one bit of a file, the file ends up with a completely different cryptographic hash. Such changes can be made intentionally to thwart investigators. The similarity digest doesn't suffer from this problem.

In my own work, I found that a 4 KiB of data extracted from most video files and JPEGs tends to be highly identifying, even possibly unique. So my system chopped sensitive files into 4 KiB chunks, hashed them, and stored the hashes in a high-performance database we built called *hashdb*. You can then search a TB-sized drive to see if it holds any of the videos in your collection by randomly sampling a small fraction of the drive's sectors, hashing them, and looking up the hashes in the database. In theory, this would let us probabilistically search a TB-sized drive for the presence of a sensitive video in just a few minutes [7]. In practice, we found it too difficult to obtain sector hashes of sensitive files due to organizational and administrative issues, so we never deployed this technology.

Digital Timestamping

One use of cryptographic hashes that was pioneered in the 1990s and is coming back into vogue is to use them as the basis of a digital timestamping service.

The roots of using hashes for timestamping go back to 1989, when a researcher at MIT accused Thereza Imanishi-Kari of scientific fraud and misconduct. One of the key allegations was the data in laboratory notebooks had been fraudulently altered. Both the US Congress and the US Department of Health and Human Services (HHS) opened investigations. The US Secret Service raided Dr. Imanishi-Kari's lab and seized her notebooks. Although the HHS Office of Research Integrity (ORI) concluded that fraud had taken place, that finding was overturned on June 21, 1996, by the HHS Research Integrity Adjudications Panel, which found that ORI "did not prove its charges by a preponderance of the evidence" (a relatively low legal standard).

Stuart Haber and Scott Stornetta were cryptographers at Bellcore (Bell Communications Research). They wanted some way that cryptography could protect organizations from the allegations that were flying around MIT of notebook alterations.

For those who have never worked in the physical sciences, let me assure you that physical laboratory notebooks can be serious stuff. Research organizations might distribute individually

serialized notebooks to their scientists, who are expected to date and sign each page. Mistakes are supposed to be crossed out with a single line, so that the erroneous entry can still be read. Corrections must also be dated and signed. One reason for such procedures is to establish clear evidence regarding the date that something that is discovered or invented, which might one day be a key fact in patent litigation. Such procedures are also designed to protect against fraud.

Electronic laboratory notebooks would seem to offer none of the protections of physical notebooks, since digital data can be changed without a trace. One obvious approach is to hash a document with a timestamp and sign the result with a secret key. The problem with this approach is the holder of the secret key—call it the timestamping agency (TSA)—must be trusted not to write a fraudulent signature.

Haber and Stornetta came up with an approach that eliminated the need to trust the timestamping agency. In their first patent (US 5,136,634, filed August 4, 1992), the TSA maintains a special hash called the *catenate value*. When a new document is to be timestamped, the TSA creates a receipt by hashing the document's hash with the current date. The TSA then takes this receipt and hashes it with the previous catenate value to create the next concatenate value. All of the hashes, with all the timestamps, thus make up a hash chain. The system that they ultimately developed, described in US Patent 5,781,629 (filed February 21, 1997), arranges document hashes into a Merkle Tree. The two founded a company called Surety, which is still going strong today.

Satoshi Nakamoto, the pseudonymous author of the original Bitcoin paper [8], references Haber and Stornetta's article [9] as one of Bitcoin's inspirations. For a list of other academic contributions that ended up in Bitcoin, see Narayanan and Clark's article in *ACM Queue* [10].

Finally, here is the puzzle from Stuart Haber:

Let's say it is 2020 and you have a document D with a digital timestamp certificate from 1997. The certificate is based on MD5, a hash function that was secure then but today suffers from known collision attacks, although the algorithm is still pre-image resistant. What do you do? You could certainly timestamp the document today, but that doesn't prove that it was around back in 1997. How do you renew the timestamp in a way that's mathematically defensible?

The solution, says Haber, is to timestamp the concatenation of the 1997 document and its 1997 digital certificate. Because MD5 is still believed to be pre-image resistant, we can't make a document today that has the same MD5 of an arbitrary document from 1997. Timestamping the concatenation today proves to the future that both exist today, and the certificate from 1997 proves today that the document must have existed back in 1997.

References

[1] H. Stevens, "Hans Peter Luhn and the Birth of the Hashing Algorithm," *IEEE Spectrum*, January 30, 2018: https://spectrum.ieee.org/tech-history/silicon-revolution/hans-peter-luhn-and-the-birth-of-the-hashing-algorithm.

[2] R. S. Merkle, "Secrecy, Authentication and Public Key Systems," Technical Report No. 1979-1, Information Systems Laboratory, Stanford Electronics Laboratories, Department of Electric Engineering, Stanford University, June 1979: https://www.merkle.com/papers/Thesis1979.pdf.

[3] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory,* vol. 22, no. 6 (November 1976), pp. 644–654: https://doi.org/10.1109/TIT.1976.1055638.

[4] Google's announcement is at https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html. You can download the two PDFs from https://shattered.it, where you will also find a visualization of the file's internals and links to the program that produced the files.

[5] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM,* vol. 13, no. 7 (July 1970): pp. 422–426: https://doi.org/10.1145/362686.362692.

[6] V. Roussev, "Managing Terabyte-Scale Investigations with Similarity Digests," in G. Peterson and S. Shenoi, eds., *Research Advances in Digital Forensics VIII* (Springer, 2012), pp. 19–34: https://doi.org/10.1007/978-3-642-33962-2_2.

[7] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks, "Distinct Sector Hashes for Target File Detection," *IEEE Computer,* vol. 45, no. 12 (December 2012): pp. 28–35.

[8] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[9] S. Haber and W. S. Stornetta, "How to Time-stamp a Digital Document," *Journal of Cryptology,* vol. 3, no. 2 (1991), pp. 99–111.

[10] A. Narayanan and J. Clark, "Bitcoin's Academic Pedigree," *Communications of the ACM,* vol. 60, no. 12 (November 2017), pp. 36–45: https://doi.org/10.1145/3132259.

Programming Workbench

Compressed Sparse Row Format for Representing Graphs

TERENCE KELLY

Terence Kelly studied computer science at Princeton and the University of Michigan, followed by a long stint at Hewlett-Packard Laboratories. Kelly now writes code and documentation promoting persistent memory programming and other programming techniques. His past publications—some of tragicomic interest only—are listed at http://ai.eecs.umich.edu/~tpkelly/papers/. tpkelly@eecs.umich.edu.

Welcome to the second installment of Programming Workbench. Today's topic is *compressed sparse row* (CSR) format, a compact and efficient way to represent graphs in memory. As usual, all example code is available in machine-readable form [6].

Graphs provide a generic abstraction that finds numerous applications for modeling *connectedness* and *ordering* in computing systems. Undirected graphs, for example, can represent communications links among computers; directed graphs can encode dependencies or precedence constraints in software compilation, software package installation, and job scheduling problems. Top computer science textbooks emphasize two ways of representing graphs in memory: adjacency matrices and adjacency lists [1, 8]. Today we'll consider other options that offer different tradeoffs and sometimes provide significant advantages. In particular we'll see that *compressed sparse row* (CSR) format—a compact and memory-hierarchy-friendly graph representation—is sometimes the format of choice. Understanding CSR in detail rounds out a programmer's education and informs the buy-or-build decisions that routinely confront practitioners.

We'll begin in the next section by reviewing ways of representing graphs, including CSR. Then we'll walk through a working C11 program that converts an edge list representation of a graph into CSR format. Finally we'll conclude by suggesting extensions and exercises to help better understand the tradeoffs surrounding CSR. For brevity, we'll restrict attention to unweighted directed graphs, but we thereby lose little generality: an *undirected* edge can be represented by two *directed* edges in opposite directions, and adding edge weights to a CSR representation is easy.

Graph Representations

Figure 1(a) shows a directed graph that we'll use as a running example. We follow the convention that vertexIDs range from 1 to V inclusive, where V is the total number of vertices. The example graph contains V=9 vertices and E=9 directed edges. For example, there's a directed edge from vertex 2 to vertex 1, shown as an arrow near the top of Figure 1(a). Vertices 5 and 9 have in-degree zero and out-degree zero, i.e., they have neither incoming nor outgoing edges. Zero-degree vertices arise naturally in applications; for example, they may represent software packages with no dependencies or compute jobs with no precedence constraints.

Rather than treating zero-degree vertices as special cases, removing them and/or handling them "out of band," we'll take the simpler approach of representing them straightforwardly. *Self edges*, i.e., edges that point from a vertex to itself, do not appear in our example, but they pose no special difficulties for the graph representations discussed below. We omit self edges for brevity; they arise relatively infrequently in applications of practical interest.

In many practical applications, a graph is given as a file that essentially contains an *edge list* of "from"/"to" vertexID pairs, possibly mummified in a more elaborate format such as XML or JSON. Figure 1(b) shows an edge list representation of our example graph. The first line in the list, "`2 1`," represents the directed edge from vertex 2 to vertex 1. Zero-degree vertices, such as 5 and 9 in our example, don't appear in an edge list, so metadata accompanying the

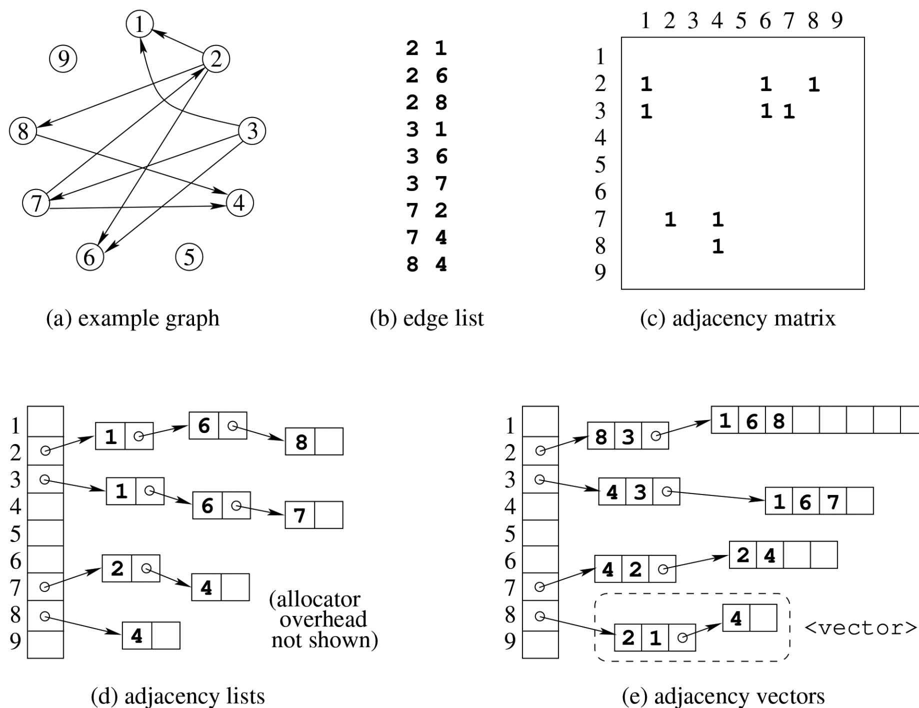


Figure 1: Textbook representations of running example, a directed graph with nine vertices and nine edges. The C++ STL `<vector>` depicted within the dashed oval in Figure 1(e) is a two-part data structure: a partially filled data array, on the right, located via the header on the left, which contains the *capacity* of the data array, the number of positions in the array occupied by user data (which may be less than the capacity, as shown here), and a pointer to the data array itself. The header of the `<vector>` enclosed by the dashed oval indicates that the data array can hold two integers but is currently holding only one. This `<vector>` represents the adjacencies of vertex 8, and the lone integer contained in the data array corresponds to the directed edge from vertex 8 to vertex 4, i.e., the last line of the edge list in Figure 1(b).

edge list must ensure that zero-degree vertices don't go missing: Thanks to our vertexID convention, simply knowing V ensures that we don't overlook zero-degree vertices. For clarity, Figure 1(b) shows a sorted edge list, but edge lists seldom arrive sorted in practical applications.

Figure 1(c) depicts a standard textbook *adjacency matrix* representation of our example graph. A directed edge from vertex i to vertex j appears as a "*1*" at row i, column j of the adjacency matrix; all other matrix entries are zero (not shown for clarity). An adjacency matrix is efficient for some operations, such as testing in constant time whether an edge connects a given pair of vertices. The major downside of adjacency matrix representation is that it requires $O(V^2)$ bits even for *sparse* graphs in which most vertex pairs are not connected by an edge. Sparse graphs arise frequently in practice, and for large sparse graphs an adjacency matrix wastes too much memory on zero entries.

The representation that most textbooks recommend for sparse graphs uses *adjacency lists*, shown in Figure 1(d). On the left is an array of pointers indexed by "from" vertexID; each pointer is the head of a singly linked list of "to" vertexIDs. Adjacency lists are flexible—it's easy to add or delete vertices—and they are indeed more compact than adjacency matrices for sparse graphs. However they entail unfortunate time and space overheads of their own: space overheads include the "`next`" pointer in every list node; list nodes will also carry allocator overheads if a general-purpose allocator like `malloc()` creates them. We suffer time overheads when we traverse an adjacency list because we must chase pointers across the address space, creating random memory accesses that today's computers penalize heavily compared with sequential accesses. If we transform an unsorted edge list representation into dynamically allocated adjacency lists in the straightforward way, the list nodes for each adjacency list will be scattered across the heap, exacerbating the pointer-chasing problem.

Using C++ Standard Template Library `<vector>`s instead of linked lists might seem like one way to reduce the overheads of adjacency lists. Figure 1(e) shows the resulting *adjacency vectors* representation. As with adjacency lists, an array indexed by "from" vertexID contains entry points to `<vector>`s of "to" vertexIDs. The dashed oval at the bottom of Figure 1(e) encloses the `<vector>` of vertexIDs adjacent to vertex 8. A `<vector>` is typically implemented as a two-part structure consisting of a

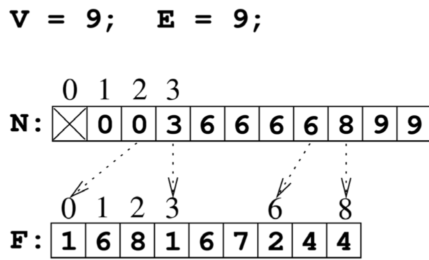


Figure 2: Compressed sparse row (CSR) representation of example graph. The vertices adjacent to vertex `a` are stored in positions `N[a]` through `N[a+1]-1` of array `F`. For example, consider vertex 2 from Figure 1(a): directed edges extend from vertex 2 to vertices 1, 6, and 8. `N[2]` is zero, so the adjacencies of vertex 2 start at `F[0]`; `N[2+1]-1` is 2, so they extend through `F[2]`. `F[0]` through `F[2]` contain the expected vertexIDs: 1, 6, and 8.

header containing the number of *allocated* entries, the number of *occupied* entries, and a pointer to an array of the entries themselves [10]. If we read a graph given as an edge list into adjacency `<vector>`s in the straightforward way, each `<vector>` grows as vertexIDs are added to it. Implementations typically *double* allocated capacity each time a `<vector>` fills up as it grows. The result is that up to roughly half of the allocated capacity of each vector can be unused; this waste may erode the benefits of reducing pointer and allocator overheads compared with adjacency lists. On the positive side, `<vector>`s reduce the time overhead of chasing pointers because they store adjacent vertexIDs in compact arrays.

The representations shown in Figure 1 don't exhaust all of the possibilities. For example, we sometimes need fast access to the *incoming* as well as the outgoing edges of a vertex, which is easy to arrange by associating a second adjacency list with each vertex. And nothing prevents us from using *both* an adjacency matrix and adjacency lists or `<vector>`s simultaneously, if we have sufficient memory. Using both representations yields the strengths of both: constant-time queries to test the existence of an edge between a given pair of vertices, and efficient access to the adjacent vertices of a given vertex.

Compressed Sparse Row Format

CSR originated in high-performance scientific computing as a way to represent sparse matrices, whose rows contain mostly zeros. The basic idea is to pack the column indices of non-zero entries into a dense array. CSR is more compact and is laid out more contiguously in memory than adjacency lists and adjacency `<vector>`s, eliminating nearly all space overheads and reducing random memory accesses compared with these other formats. The price we pay for CSR's advantages is reduced flexibility: adding new edges to a graph in CSR format is inefficient, so CSR is suitable for graphs whose structure is fixed and given all at once. CSR also carries a *cognitive* overhead: it's trickier than the other formats we've reviewed, and it uses arrays in FORTRANesque ways seldom seen in systems-y C/C++ code or in mainstream Java code. We'll walk through it slowly.

Figure 2 depicts the CSR representation of our example graph. First we'll consider the specifics of how CSR encodes a handful of the example graph's structural features, and then we'll describe CSR in more general terms. Like the textbook sparse-graph representations discussed earlier, CSR facilitates finding the adjacencies of a given vertex, i.e., the vertices at the "to" ends of edges emanating out of a given "from" vertex. CSR finds adjacencies using two layers of array indexing.

The CSR depicted in Figure 2 contains V, E, and two arrays of integers, `N` and `F`. Notice that `F` presents horizontally the same sequence of "to" vertexIDs that appear vertically in the right-hand column of the sorted edge list of Figure 1(b). Given a "from" vertexID, we find all corresponding "to" vertexIDs by indexing into `F` via `N`. We'll walk through the process of finding the adjacencies of the first three vertices in our example graph to gain intuition for how CSR encodes graph structure.

The out-degree of vertex 1 is encoded as the *difference* between `N[1]` and `N[2]`. Since `N[1]` *equals* `N[2]`—both are zero—the out-degree of vertex 1 is zero, so there are no adjacent vertices to be found. The out-degree of vertex 2 is `N[2]` subtracted from `N[3]`, which is 3. The IDs of the three vertices adjacent to vertex 2 are in array `F` starting at position `N[2]`, i.e., at `F[0]`, as indicated by the dotted arrow in Figure 2 from `N[2]` to `F[0]`. The out-degree of vertex 3 is the difference between `N[3]` and `N[4]`, which is three; the IDs of the three vertices adjacent to vertex 3 begin at position `N[3]` in `F`, i.e., at `F[3]`, as shown by a second dotted arrow in Figure 2. The figure contains a dotted arrow for every vertex with out-degree greater than zero; the arrowheads partition `F` into four sub-arrays of adjacencies.

In general, the out-degree of any vertex `a` is `N[a+1]` minus `N[a]`. The IDs of the vertices adjacent to `a` are located in array `F` starting at `F[N[a]]` and continuing through `F[N[a+1]-1]` inclusive. In other words, the entries of `N`, indexed by "from" vertexID, "point to" contiguous regions of `F` containing the adjacent "to" vertexIDs. Array `F` contains E entries, one for each edge. Array `N` contains V+2 entries: `N[0]` is unused and `N[V+1]` contains E. The total amount of memory required for CSR format is almost exactly equal to `sizeof(int)` multiplied by (V+E), so it's easy to determine if available memory is adequate based on a graph's size parameters.

If E exceeds `INT_MAX`, a larger integer type, e.g., `int64_t`, must be used for the elements of `N`, because the entries of `N` refer to positions in E-long array `F` and the last entry of `N` contains E. Similarly, `F` must use a sufficiently large type to represent vertexIDs up to V. Moreover it's actually best to choose a type for vertexIDs

such that V is strictly *less* than the `MAX` of the type, because V+1 is used as an index into `N`. Unsigned integer types may be used for arrays `N` and `F`, though signed integer types might be preferable, e.g., if we want to catch signed overflow errors at runtime with a compiler flag like GCC's `-ftrapv`.

CSR offers different tradeoffs than alternative formats. On the positive side, it eliminates memory allocation overheads almost completely. Furthermore, while array `N` contains the moral equivalent of pointers, they can be smaller than conventional pointers (32 vs. 64 bits), depending on the size of E and the relative sizes of `ints` and ordinary C pointers. The IDs of vertices adjacent to a given vertex are contiguous in `F`, so visiting all adjacent vertices involves zooming through an array, which is much faster on modern computers than chasing pointers down an adjacency list. While *adding* a new edge to a CSR representation isn't efficient—it would require insertion into the middle of `F`, which would take $O(E)$ time on conventional memory [3]—*deleting* an edge is quick and easy: to delete an edge, simply set its entry in `F` to zero, which is not a valid vertexID, and ignore zero entries in `F`.

CSR isn't magic. When applied to the kinds of graphs that arise naturally in practical applications, many important graph algorithms, including traversal algorithms such as breadth-first search and depth-first search, must inevitably perform random memory accesses. CSR can't eliminate random memory accesses that are inherent to the computational task at hand; it can merely avoid introducing additional random accesses that arise as side effects of the format.

The Code

The C11 program listed in this section, "`el2csr.c`," converts an edge list representation of an unweighted directed graph to CSR format; the source code is available at [6]. We'll discuss everything substantive, skipping boilerplate like `#includes`. The purpose of the example code is to illustrate CSR format, so it avoids niceties for brevity and clarity.

The macros below handle error checking. `BAIL()` prints an error message prefixed by the file name and line number where it is called then `exit()`s. `CAL()` calls `calloc()` and bails if allocation fails.

```
#define ERRSTR strerror(errno)
#define                          S1(s) #s
#define                    S2(s) S1(s)
#define COORDS __FILE__ ":" S2(__LINE__) ": "
#define BAIL(...)                              \
  do { fprintf(stderr, COORDS __VA_ARGS__); \
       exit(EXIT_FAILURE); } while (0)
#define CAL(p, n, s)                                      \
  do { if (NULL == ((p) = (int *)calloc((n), (s))))       \
         BAIL("calloc(%lu, %lu): %s\n", (n), (s), ERRSTR);\
     } while (0)
```

Readers may recall from the previous Programming Workbench column a function-like macro called "`DIE()`" that differs from `BAIL()` above but serves a similar purpose. The contrast between the two stems from differences in how they are used and from differences in the programs they inhabit. `DIE()` is used exclusively to handle failed library calls, and thus it is adequate for `DIE()` to report only the name of the failed call via `perror()`. By contrast, `BAIL()` is sometimes used to check user input, so it supports flexible `printf()`-like formatting of more informative diagnostics, such as the input line number where a parse error occurs. `DIE()` is used in multithreaded code where failed library calls may arise from Heisenbugs, so it aborts with a core dump to facilitate debugging. `BAIL()` serves a simple single-threaded program and is used in situations where a core dump would not be very helpful, so it merely calls `exit()`. `DIE()` expands to an expression because it is used in contexts that demand expressions, but `BAIL()`'s simpler role allows it to expand into a statement block, which is easier to understand.

The following `struct` will eventually contain a CSR representation of a graph. The roles of `V`, `E`, `N`, and `F` are as described in the previous section.

```
static struct {
  int V,  // max vertexID; valid vertexIDs are [1..V]
      E,  // total number of edges
     *N,  // indexed by "from" ID; outdeg(v) == N[1+v]-N[v]
     *F;  // "to" vertexIDs accessed via N[]
} CSR;
```

For brevity we consider only unweighted graphs, but it's easy to handle edge weights: add to the `struct CSR` above an E-long dynamically allocated array of weights—one weight for every edge in array `F`. Note that such edge weights can be updated efficiently; they need not be completely static.

One way to understand CSR format is to study the function below, which prints a text representation of the graph in the `struct` above. The outer `for` loop iterates over all vertexIDs `a`. The inner `for` loop iterates over all vertexIDs `b` such that `a` directed edge exists from `a` to `b`. Pointers `begin` and `end` delimit the part of array `F` containing `a`'s adjacent vertexIDs.

```
static void print_adjacencies(void) {
  printf("per-vertex adjacencies:\n");
  for (int a = 1; a <= CSR.V; a++) {
    int *begin = CSR.F + CSR.N[  a],
        *end   = CSR.F + CSR.N[1+a];
    printf("%d:", a);
    for (int *b = begin; b < end; b++)
      printf(" %d", *b);
    printf("\n");
  }
}
```

Our `struct CSR` contains ordinary `int` variables, which are 32 bits long on many computers. In practice we may encounter graphs with many billions of vertices and edges, so when the user enters graph size parameters V and E on our program's command line, we verify that they both fit in an `int`—with room to spare, because we index into array `N` using values up to V+1. Function `s2i()` below performs string-to-`int` conversions carefully and gripes if it encounters weirdness of any kind. The C11 `static_assert` feature confirms at compile time our assumption that the largest integer type is larger than an `int`.

```
static_assert(sizeof(intmax_t) > sizeof(int), "int sizes");
static int s2i(const char *s) {
  char *p;  intmax_t r;
  errno = 0;
  r = strtoimax(s, &p, 10);
  if (0 != errno || '\0' != *p || 0 >= r || INT_MAX <= r)
    BAIL("s2i(\"%s\") -> %" PRIdMAX ", errno => %s\n",
         s, r, ERRSTR);
  return (int)r;
}
```

The `main()` function begins by declaring a few variables and checking user-supplied command-line arguments, then opening the file containing the edge list representation of the input graph. Reading V from the command line, as opposed to inferring it from the largest vertexID on the edge list, accommodates zero-degree vertices with IDs greater than any on the edge list, like vertex 9 in our example graph.

```
int main(int argc, char *argv[]) {
  int a, b, line = 0, t = 0;
  FILE *fp;

  if (4 != argc)
    BAIL("usage:  %s  V  E  edgelistfile\n", argv[0]);
  CSR.V = s2i(argv[1]);
  CSR.E = s2i(argv[2]);
  if (NULL == (fp = fopen(argv[3], "r")))
    BAIL("fopen(\"%s\"): %s\n", argv[3], ERRSTR);
```

Next, we allocate memory for the `N` and `F` arrays using the `CAL()` macro, which calls `calloc()`. As explained above, array `N` is of size V+2 because it is indexed with integers up to V+1.

```
  CAL(CSR.N, 2 + (size_t)CSR.V, sizeof *CSR.N);
  CAL(CSR.F,     (size_t)CSR.E, sizeof *CSR.F);
```

We make *two* passes over the input file to construct CSR format. The first pass, below, verifies the sanity of each vertexID pair and stores the out-degree of each vertex in array `N`; later `N` will be altered to play its role in CSR format as described in the previous section. We check for flagrant parse errors and verify that the E given on the command line matches the length of the input file.

```
  while (2 == fscanf(fp, "%d %d\n", &a, &b)) {
    line++;
    if (0 >= a || a > CSR.V || 0 >= b || b > CSR.V)
      BAIL("%d:  bad vertexID:  %d %d\n", line, a, b);
    if (a == b)
      fprintf(stderr, "%d:  warning:  self edge\n", line);
    CSR.N[a]++;
  }
  if (! feof(fp))
    BAIL("parse error after %d lines: %s\n", line, ERRSTR);
  if (line != CSR.E)
    BAIL("%d input lines != %d edges\n", line, CSR.E);
```

The standard `fscanf()` function used above silently performs incorrect conversions if the input vertexIDs are too large. For example, on my system `fscanf()` happily converts 4,294,967,299 to 3 without complaint. Performing conversions more carefully, e.g., with the `s2i()` function that we saw earlier, would substantially increase the overhead of parsing the input. Instead we warn users that it's their responsibility to ensure that vertexIDs on the input edge list must not exceed the V argument supplied on the command line, which is checked carefully by `s2i()`.

This next bit of code updates the contents of array `N` to contain *cumulative* out-degrees. After the code below executes, `N[a]` contains the *sum* of the out-degrees of vertices 1 through `a` inclusive. `N[V+1]` contains the sum over all vertices of their out-degrees, i.e., the number of edges E.

```
  for (a = 1; a <= CSR.V; a++) {
    t += CSR.N[a];
    CSR.N[a] = t;
  }
  CSR.N[a] = t;
  assert(CSR.N[1 + CSR.V] == CSR.E);
```

We're still not done with array `N`, because at this point each entry `N[a]` is too large by the out-degree of vertex `a`. Our second and final pass over the input fixes the problem. The second pass adds edges to `F` while walking the moral-equivalent-of-pointers in `N` back to their final correct CSR values.

```
  rewind(fp);
  while (2 == fscanf(fp, "%d %d\n", &a, &b))
    CSR.F[--CSR.N[a]] = b;  // add directed edge a -> b

  if (0 != fclose(fp))
    BAIL("fclose(): %s\n", ERRSTR);
```

Sorting the outgoing edges of each vertex isn't strictly necessary, but we'll do it anyway because it makes it easy to detect duplicate edges. Furthermore it allows us to perform a binary search on each vertex's adjacencies in $O(\log D)$ time, where D is the average out-degree. Would it be easier to sort the input edge list rather than sorting the adjacencies of each vertex? That might be conceptually simpler and easier to implement, but it would be asymptotically less efficient: sorting the edge list with a general method such as `qsort()` would would require $O(E \log E)$ time, whereas sorting the adjacencies of each vertex requires $O(V D \log D)$ time; the latter is typically smaller. The integer comparison function below, `icmp()`, seems prone to overflow in the subtraction operation—consider `INT_MAX` minus negative one—but overflow can't

happen in our program because all of the integers being sorted are non-negative.

```
static int icmp(const void *a, const void *b) {
  return *(const int *)a - *(const int *)b;
}
...
  for (a = 1; a <= CSR.V; a++)
    qsort(CSR.F + CSR.N[a],
          (size_t)(CSR.N[1+a] - CSR.N[a]),
          sizeof *CSR.F, icmp);
```

Now that we've constructed CSR format, we dump it for inspection and then print per-vertex adjacencies using the function we defined earlier:

```
  printf("dump CSR format:\n"
         "V = %d    E = %d\n"
         "N: ", CSR.V, CSR.E);
  for (a = 0; a <= 1 + CSR.V; a++)
    printf(" %d", CSR.N[a]);
  printf("\n"
         "F: ");
  for (a = 0; a < CSR.E; a++)
    printf(" %d", CSR.F[a]);
  printf("\n");

  print_adjacencies();
```

Our final chore before terminating is to deallocate arrays `N` and `F`:

```
  free(CSR.N);
  free(CSR.F);

  return 0;
}
```

Running `el2csr` on our example graph yields the expected results:

```
% ./el2csr 9 9 example_graph.txt
dump CSR format:
V = 9    E = 9
N:  0 0 0 3 6 6 6 6 8 9 9
F:  1 6 8 1 6 7 2 4 4
per-vertex adjacencies:
1:
2: 1 6 8
3: 1 6 7
4:
5:
6:
7: 2 4
8: 4
9:
```

The example code tarball contains a random graph generator and a test script in addition to `el2csr.c`. The test script compiles the random graph generator and compiles `el2csr` in a special test mode that dumps an *edge list* representation of the input graph to a file. The test script then feeds many random graphs to `el2csr` and verifies that in each case the edge list regurgitated by `el2csr` is byte-for-byte identical to the sorted input file.

Persistence

Converting an edge list to CSR format takes time—parsing textual input can be orders of magnitude slower than running a graph analysis algorithm—and it would be wasteful to perform the conversion more often than necessary. It's usually best to store the binary CSR representation of the graph in a file for future use. One way would be to `write()` variables `V` and `E` and arrays `N` and `F` to a file. An easier and more elegant approach is to employ "the persistent memory style of programming" [4, 5]: lay out the data structures in a file-backed memory mapping using `msync()` to persist the data after constructing a CSR representation and later using `mmap()` to load the file containing CSR back into memory as needed. This is convenient and is often the most efficient way to handle graphs in practical applications, because after the initial conversion to CSR format no further parsing or serializing is ever needed. The CSR file is in the compact in-memory format used by subsequent analyses, which access the data via LOAD instructions after `mmap()`-ing the file into memory.

Other Implementations

The Boost Graph Library [9] offers C++ implementations of many graph algorithms, and it supports several graph formats including adjacency lists and CSR. BGL emphasizes generic programming and is written in a different style from my example code; comparing the two may lead the reader to additional insights. Galois is a platform for parallel computation that includes substantial support for graphs [2]. Distributed/scale-out graph analysis platforms were blooming like mushrooms in the research community several years ago; many were so grotesquely inefficient that they are of tragicomic interest only [7].

Going Further

Extending my example code can be an informative exercise. You can avoid the time overhead of parsing the input edge list on the second pass by converting it to a temporary binary edge list on the first pass. Adding support for weighted edges is easy. To appreciate the benefits of CSR format over adjacency lists or adjacency `<vector>`s, compare their memory footprints on real or randomly generated graphs. Similarly, compare the runtimes of standard graph algorithms on the different formats.

Random graph generators are often used for testing and performance benchmarking because they make it easy to sweep key graph parameters such as size, average degree, and density. Storing large random graphs in short-lived files can be slow, awkward, and cluttery, but an easy trick avoids the need to create files, even when their consumer must make multiple passes over each: run multiple instances of the random graph generator as background jobs that spit identical byte streams into named pipes, one pipe for every pass needed by the consumer. For

example, if the `el2csr` program listed above is the consumer, it would be modified to read two identical byte streams from two named pipes supplied on the command line—an easy exercise. This approach preserves a clean separation of responsibilities between graph generator and graph consumer while avoiding the fuss of large temporary files.

Conclusion

Compressed sparse row is typically the best format for sparse graphs, provided that new edges aren't added and relatively few edges are deleted. CSR is compact, avoiding the memory waste of adjacency lists and `<vector>`s, and its memory footprint can be calculated directly from V and E. CSR is furthermore contiguous in memory, eliminating the time overhead of pointer chasing. It's easy to persist CSR in memory-mapped files, and CSR is convenient once you become accustomed to it. The two-pass construction approach implemented above is asymptotically faster than sorting an edge list.

Graphs are essentially simple, and coding graph algorithms can be positively pleasant. The next time you're faced with a problem involving graphs, consider solving it by writing your own code instead of using someone else's software; the result might well be superior overall. Please share your experiences and feedback with me!

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd edition (MIT Press, 2009).

[2] "Galois": https://iss.oden.utexas.edu/?p=projects/galois.

[3] T. Kelly, H. Kuno, M. Pickett, H. Boehm, A. Davis, W. Golab, G. Graefe, S. Harizopoulos, P. Joisha, A. Karp, N. Muralimanohar, F. Perner, G. Medeiros-Ribeiro, G. Seroussi, A. Simitsis, R. Tarjan, and S. Williams, "Sidestep: Co-Designed Shiftable Memory and Software," HP Labs Tech Report HPL-2012-235, November 2012: https://www.labs.hpe.com/techreports/2012/HPL-2012-235.pdf.

[4] T. Kelly, "Persistent Memory Programming on Conventional Hardware," *ACM Queue*, vol. 17, no. 4 (July/August 2019): https://queue.acm.org/detail.cfm?id=3358957.

[5] T. Kelly, "Good Old-Fashioned Persistent Memory," *;login:*, vol. 44, no. 4 (Winter 2019): https://www.usenix.org/publications/login/winter2019/kelly.

[6] T. Kelly, Example code to accompany this article: https://www.usenix.org/sites/default/files/kelly_csr_code.tar.gz.

[7] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST?" 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS '15): https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-mcsherry.pdf.

[8] R. Sedgewick and K. Wayne, *Algorithms*, 4th edition (Addison-Wesley, 2011).

[9] J. G. Siek, L. Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual* (Addison-Wesley, 2002).

[10] B. Stroustrup, *The C++ Programming Language*, 4th edition (Addison-Wesley, 2013). See p. 888 for the representation of `<vector>`s.

For Good Measure

Counting Broken Links: A Quant's View of Software Supply Chain Security

DAN GEER, BENTZ TOZER, AND JOHN SPEED MEYERS

Dan Geer is a Senior Fellow at 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

Bentz Tozer is a Senior Member of Technical Staff in In-Q-Tel's Cyber Practice, where he identifies and works with startups with the potential for high impact on national security. In previous roles, he has performed security research and software development with a focus on IoT devices and embedded systems. He has a PhD in systems engineering from George Washington University. btozer@iqt.org

John Speed Meyers is a Data Scientist in IQT Labs and a researcher who focuses on cybersecurity, especially network traffic analysis and software supply chain security. He holds a PhD in policy analysis from the Pardee RAND Graduate School. He's ambivalent about computers. jmeyers@iqt.org

"Without data, you're just another person with an opinion."

—*W. Edwards Deming*

It is tempting to tune out the cyberattack news cycle, dismissing the seemingly random assortment of reported attacks as nothing more than chance encounters of lucky defenders with unlucky attackers. It is easy to see the noise. It takes more effort—what amounts to digital wading—to find the signal, especially when dealing with public reporting on cyberattacks, but wade we did to assess the extent of software supply chain attacks. These attacks prey on the trust that makes code reuse possible and that produces the modern software cornucopia enjoyed by software developers and consumers alike.

We read of the event-stream attack [1] where an individual with malicious intent took over a popular JavaScript library and slipped code that steals cryptocurrency wallet credentials into a dependency of the associated npm package; ShadowHammer [2] in which a back-doored update utility with a legitimate certificate was distributed through official channels; and of barrages of typosquatting attacks on package registries [3] such as npm, RubyGems, and PyPI. Learning about these incidents led us to collect and review reports of software supply chain attacks in order to better understand the characteristics of these incidents and trends. While doing so, we also noticed the emergence of more systematic research. There's been measurement of the susceptibility of package manager users to typosquatting [4], the creation of a sophisticated malware detection pipeline for package managers [5], the building of a package manager download client that protects users from malware [6], and other efforts to gather and classify reports of software supply chain compromises [7–9].

We collected our data set in order to answer basic questions about software supply chain attacks such as: How frequent are known instances of attacks? What is the relative occurrence of different attack types? What is the length of time from initial deployment of such attacks to public discovery? However, while attempting to obtain these quantitative metrics, we were also faced with more fundamental, qualitative questions, like: What is (and is not) considered a software supply chain attack? What are the definitions of different attack types? How should attack impact be defined and measured? We report on how we built this data set, answer the quantitative questions that we set out to understand, and then, based on these findings, offer some thoughts on how to use data to combat software supply chain attacks.

Software Supply Chain Compromises: Data Set and Analysis

We built a data set based on public reporting of software supply chain security compromises, which is available at https://github.com/IQTLabs/software-supply-chain-compromises. This data set defines software supply chain attacks as attacks that intentionally insert malicious functionality into build, source, or publishing infrastructure or into software components with the goal of propagating that malicious functionality through existing distribution methods. Exploiting a vulnerability found within a software's supply chain was insufficient

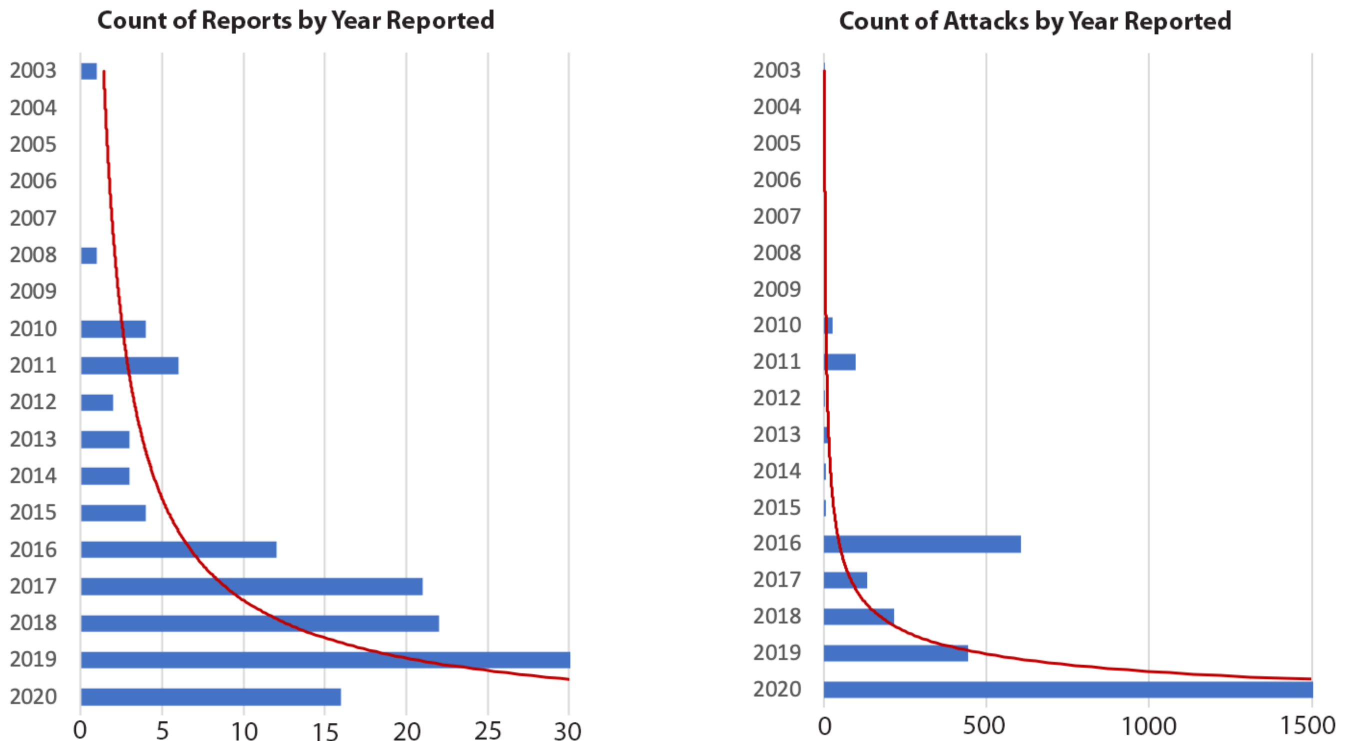


Figure 1: The number of reports and attacks by year reported

to merit inclusion. We attempted to count three different units of software supply chain security compromise: attacks, reports, and incidents. An "attack" is a distinct action to compromise a software supply chain, e.g., deliberate introduction of a vulnerability into source code. A "report" is a public disclosure of one or more software supply chain attacks, e.g., a blog post from a security researcher who has identified the existence of an attack in an open source library. An "incident" is a single instance of an attack reaching a target, e.g., the download of a compromised application from a download server. In effect, we hoped to use "incident" as a measurement of the impact of an "attack."

Figure 1 describes the trend of reports and attacks by the year in which the report or attack was announced, a decision that reflects the limited data available about the starting date of many of these attacks. The number of reports and attacks has been increasing over time; though included here, years before 2010 include only a count of one in 2003 and one in 2008. Because reporting can be delayed, 2020 and, perhaps, 2019 may be undercounted as yet. (The lines are power-law fits; the exponent is 1.2 for count of reports and 2.5 for count of attacks.)

Table 1 groups these reports and attacks into major and minor categories based on the actions of the attacker, not the perspective of the victim. These categories were influenced by the work of the "in-toto" project [7] but were adapted and extended organically while collecting this data set and do not represent any established classification scheme. The development of a standard taxonomy in the future would be beneficial.

Table 1 tentatively suggests that there is an inverse relationship between the estimated level of effort required to execute an attack type and the frequency of reported attacks of that type. For example, 41 percent of attacks in our data set are classified as typosquatting, which merely requires the attacker to create an account on a package registry, identify unclaimed package names that are plausible misspellings of legitimate packages, and publish the malicious package under those names. On the other end of the spectrum, a build system compromise is one of the least common attack types in our data set, perhaps because it involves several challenging steps, including obtaining access to a target's build environment and introducing a compromised component into the build process without being detected. As we discuss below, while the success of an attack is difficult to objectively define and measure, it seems possible that the effort required to successfully deploy an attack is directly proportional to the number of incidents, with typosquatting attacks affecting fewer individuals than a build system compromise like ShadowHammer. This indicates that increasing the level of effort required to successfully deploy attacks on software registries could significantly reduce the quantity of reported software supply chain attacks.

Major Type	Build, Source, and Publishing Infrastructure					Software Registry			
Minor Type	Build System Compromise	Firmware Implant	Source Code System Compromise	Publishing: Certificate Attack	Publishing: Delivery System Compromise	Account Takeover	Dependency Compromise	Malicious Package	Typosquatting
Count	11:13	7:32	9:39	6:18	29:35	11:14	12:333	51:1,373	15:1,247

Table 1: Count of Reports:Attacks by major and minor categories. (Note: Both reports and attacks can be assigned to multiple categories.)

Data on incidents is not consistently available, and metrics are not consistent across attack types, making quantitative analysis across this data set infeasible. For example, download metrics were sometimes available for a malicious package accessible on a software registry, but the number of victims and number of times the package was executed after download are generally unknown. In other cases, a lower bound on the number of compromised applications has been reported, but the extent of propagation is unknown. However, the limited data that is available indicates the potential for widespread impact. In the case of the event-stream attack, there were over 7 million package downloads reported for the 53 days it was available on npm, and some unknown number of those downloads were of the compromised version, rather than older, non-malicious versions. For ShadowHammer, Kaspersky, which identified and reported the attack, stated that the attack affected over 57,000 of their users and estimated that the attack was distributed to over 1 million people. In the case of the typosquatting attacks identified by Perica and Zekić, the one package where information is reported was downloaded over 1700 times over nearly two years. While details are limited, it is clear that the potential force multiplication caused by the propagation of an attack through existing software delivery methods is highly appealing to attackers.

Another way to measure the success of a software supply chain compromise is the length of time it is active. Known as "dwell time," it is the number of days a threat remains undetected within a given environment; if the detection date is not available, we use the public announcement date. Figure 2 displays the distribution of dwell time for all reports with available data (n=59).

Defending the Supply Chain

Our analysis of known software supply chain attacks indicates that weaknesses in the software supply chain are numerous and are being appropriated by cybercriminals with increasing frequency. Per the usual, attacks as yet unknown are surely present, so you should assume that we are undercounting. Counting is hard. The spike in attacks in 2016 includes two special cases: a research project where a student intentionally uploaded 214 typosquatting packages to various software registries to measure download frequency, and the intentional deletion of 273 npm packages by a developer who was angry that npm took a package namespace away from him and wanted to wreak havoc. Earlier this year, ReversingLabs found 700+ malicious packages in RubyGems, while Duo found 500+ malicious Chrome extensions, both evidently the first time anyone had looked into such unknown unknowns. Counting is hard.

We believe there exist at least three major obstacles that prevent software developers, security teams, and software users from adequately protecting the software supply chain and from shielding themselves or their organization from such attacks.

First, there is a striking absence of data collection and analysis that would help identify and assess risks associated with these attacks, especially those involving open source software. This absence is surprising given the inherently public nature of open source software development and the ubiquity of open source dependencies within modern software applications. In other words, much of the data needed to identify potential and actual risks associated with a software component is hosted on publicly accessible development platforms like GitHub and is thus available to any interested party. Unfortunately, much of this information is not analyzed, allowing attackers to hide in plain sight.

To identify attacks, defenders will need to cull and analyze software development-related data. To start, a software bill of material that describes all dependencies of an application provides transparency and allows for investigation of direct and indirect dependencies. Other rich data sources are open source code repositories and package registries, which contain information about developer turnover, code commits, and version releases stored in these repositories. Defenders can also expose inconsistencies between independent data sources, verifying, for instance, the relationships between source code stored in

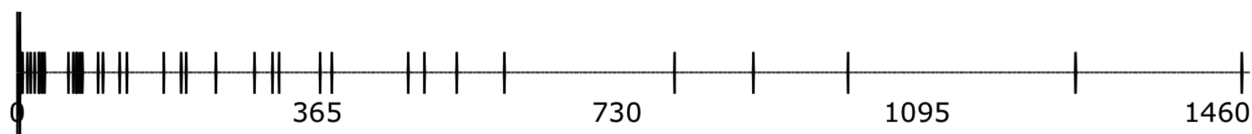


Figure 2: Distribution of dwell time in days for reports; dwell time for 12 reports was zero days; the median=34.

a repository and a released library or executable stored in a package registry. Countermeasures will also likely require an understanding of the individuals and organizations that directly or indirectly contribute to the development and distribution of software, especially the individuals or organizations that can publish changes. Importantly, defenders will need this information for all dependencies of a distributed software application, whether those dependencies are part of the build process, release process, or are included at runtime. As always, the wellspring of risk is dependence, and risk, unlike benefit, is transitive.

Second, existing application security products are unable to identify the distinctive characteristics of software supply chain attacks. Moreover, there has been limited adoption of what tools and processes do exist in order to prevent instances of supply chain attacks within released software. These issues force software developers and users to trust—but not verify—vendors and their products, rendering judgments about product software supply chain quality impossible and compelling acceptance of unknown risks within critical software.

These deficiencies should be a rallying cry for those who want to develop and build a new class of application security products, tools designed to uncover instances of software supply chain attacks. Existing application security tools are designed to identify defects in source code or executables and determine the conditions under which those defects are exploitable. These tools will not, however, identify a well-written software supply chain compromise. These attacks arise from the existence of undesired functionality with respect to the intended purpose of the software. This new breed of application security products will need context and an understanding of the expected use case of the application, concepts lacking in current application security products. This will not be easy.

Third, reducing the software supply chain attack surface also requires adopting existing technologies and processes that provide the information needed to verify the origin and content of source code and binaries, eliminating or mitigating many of the risks of compromise. One practical step is using digital signatures and certificates to verify file integrity. Employing reproducible builds and publishing relevant metadata to an immutable distributed ledger can also allow consumers to independently verify the integrity of a software component. Best of breed tools for network and endpoint protection should also be deployed within the development, publication, and operational environment to limit opportunities for compromise pre-commit.

Ultimately, securing the software supply chain of any product requires continuous assessment of components, vendors, and operational environments in addition to orchestration and analysis of relevant data. These processes, to be successful, require significant investment in automation and collaboration between all participants in the software supply chain. Nothing less is needed if sharing common software dependencies is to be a strength, the topic of this column two issues ago [10], rather than the liability it appears to be today.

References

[1] T. Hunter II, "Compromised npm Package: Event-stream," Intrinsic/VMware, November 26, 2018: https://medium.com/intrinsic/compromised-npm-package-event-stream-d47d08605502.

[2] "ShadowHammer: Malicious Updates for ASUS Laptops," Kaspersky Daily, March 25, 2019: https://www.kaspersky.com/blog/shadow-hammer-teaser/26149/.

[3] R. Perica and A. Zekić, "Mining for Malicious Ruby Gems," ReversingLabs, July 17, 2019: https://blog.reversinglabs.com/blog/suppy-chain-malware-detecting-malware-in-package-manager-repositories.

[4] N. P. Tschacher, "Typosquatting in Programming Language Package Managers," University of Hamburg, Bachelor Thesis, 2016: https://incolumitas.com/data/thesis.pdf.

[5] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Measuring and Preventing Supply Chain Attacks on Package Managers," arXiv, February 4, 2020: https://arxiv.org/abs/2002.01139.

[6] M. Taylor, R. K. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "SpellBound: Defending against Package Typosquatting," arXiv, March 6, 2020: https://arxiv.org/pdf/2003.03471v1.pdf.

[7] S. Torres, H. Afzali, A. Sirish, and L. Puehringer, "Software Supply Chain Compromises," November 11, 2019: https://github.com/in-toto/supply-chain-compromises.

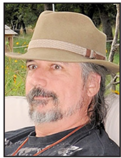
[8] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *Proceedings of the 17th Conference on Detection of Intrusions and Malware,* and *Vulnerability Assessment (DIMVA 2020)*: https://link.springer.com/chapter/10.1007/978-3-030-52683-2_2.

[9] T. Herr, J. Lee, W. Loomis, and S. Scott, "Breaking Trust: Shades of Crisis Across an Insecure Software Supply Chain," Atlantic Council, July 6, 2020: https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/.

[10] D. Geer and G. Sieniawski, "Who Will Pay the Piper for Open Source Software Maintenance?" *;login:*, vol. 45, no. 2 (Summer 2020): https://www.usenix.org/publications/login/summer2020/geer.

/dev/random
Discontent Creator

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

Some enterprising individual on a business-oriented social media site recently tried to flatter me (at least, I'm presuming he meant it that way) into accepting him as a connection by labeling me "a fellow influencer and content creator." The naturally curious sort that I am, I decided I should probably try to understand what it was he was calling me by conducting a little online research. The Internet being a fractal rabbit hole that leads to an infinity of equally fractal rabbit holes, I got a little distracted. After six or seven hours I eventually ran across this definition for influencer: "a person with the ability to influence potential buyers of a product or service by promoting or recommending the items on social media."

Now, were I indeed any variety of influencer, my novels would doubtless occupy positions much higher on the bestseller list than they currently enjoy. My sales rankings are so abysmally low, in fact, that they very nearly wrap back around to the top like a pinball score. Regular readers of this column will also have a pretty clear idea what I think of social media. Associating me with a commercial product there would be a disastrous mistake on anyone's part, as I am at best a "dissuader" and at worst, "anathema."

I'm trying to remember the last time I influenced anyone to purchase something. My wife buys things at the store because I ask her to, but I don't think that really counts. This paucity of persuasive acumen is partly due to the fact that the majority of my friends are too old to be influenced by me or much of anyone else. By the time you get to my age, you like what you like and don't what you don't, regardless of what other people say. Besides, my idea of a ringing endorsement goes something like this: "I bought this three-horsepower slip-clutched double overhead cam citrus peeler yesterday and it hasn't fallen apart yet. Sweet." I don't habitually rate purchases, but if I did it would be with little crescent wrenches, not stars.

It's been my observation that facts and even basic grammatical awareness are largely regarded as irrelevant in the headlong rush to online influence. The medium is no longer merely the message; it now constitutes the whole enchilada. What is being said is far less crucial to modern audiences than *how* it is being said. Presentation has superseded rhetoric, form obliterated function. Communication itself has been wholly subsumed by advertising. Clarity and meaning are outmoded concepts.

Even the label "content creator" is spurious. We're *all* content creators, although most of us create content that doesn't need to exist in this or any other universe. There is nothing inherently salutary in creating content unless that content has value in and of itself. I, for example, allow every new kitten who comes to live in my house to flounce across my keyboard and thus construct a "short story." I suppose that makes my cats content creators, too. Content creators, I might add, who haven't the slightest interest in generating followers or accumulating likes, unless by "likes" one means petting and/or treats.

Most of the content I see created in the IT realm is commendably utilitarian, which means someone, somewhere, probably has an actual use for it, even if it's only to give the folks in the C suite something to chew on while they're packing their golden parachutes. This stands in stark contrast to the bulk of what "influencers" produce, which resembles transcripts of conversations overheard in a high school hallway accompanied by *way* too many photos of the originator and is more closely related to secretion than creation. There has never before been a generation so fascinated with their own visages. I'm not really a fan of reflective surfaces around my house in general, much less selfies. Most pictures I've seen of myself are disturbing in good light, terrifying in bad.

Before the age of social media, writers often wrote stories. Some of these stories were factual, some flowed from a practiced imagination. While not every story rose to the level of high art, referring to them simply as "content" is akin to calling works of portraiture "pigment." Highways have no intrinsic worth until they enable vehicles containing people or goods to travel from place to place, just as content means nothing unless it conveys something significant to the reader other than self-referential metaphor. "Yo dawgs, check out my new cowboy boot b-ball kicks!!!!" is not what I'm talking about here.

The chief problem I see with content creation for its own sake is that it muddies an already densely opaque pool of verbiage. These people seem to be paid by the word, as well, which means they often take two paragraphs to say what could have been expressed in a single sentence. That really does no one any favors, especially in our era of breathtakingly short attention spans. The more fluff there is to wade through, the less likely the waders are to chance upon something they actually would benefit from reading. It's no wonder misinformation is rampant. The signal-to-noise ratio of the Internet has never been very high, but lately it seems to have plummeted off a precipice. Searching for reliable information online is like trying to find one specific pebble in a gravel parking lot, while a hundred "helpful" people crowd around, all pointing in different directions, shouting at you that they know exactly where it is.

My conclusions, then, are that "content creator" seems to be another name for "one who writes filler," and an "influencer" is what we of my generation called a "corporate shill." You may argue that this column demonstrates that I myself thereby meet the content creator definition, but I must respectfully dispute this assertion. What I create is quite clearly *dis*content, although with this being a pandemic-bedeviled election year, that market is already seemingly saturated. I have faith in the near-bottomless hunger of the public for toxic disillusionment, though. It certainly keeps Hollywood humming along.

Save the Dates!

nsdi '21

18th USENIX Symposium on Networked Systems Design and Implementation

APRIL 12–14, 2021 | VIRTUAL EVENT

The 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21) focuses on the design principles, implementation, and practical evaluation of networked and distributed systems. Our goal is to bring together researchers from across the networking and systems community to foster a broad approach to addressing overlapping research challenges.

PROGRAM CO-CHAIRS

James Mickens
Harvard University

Renata Teixeira
Netflix

www.usenix.org/nsdi21

BOOKS

Book Reviews

MARK LAMOURINE AND RIK FARROW

Effective Python: 90 Specific Ways to Write Better Python, 2nd Edition
Brett Slatkin
Pearson Education Inc., 2020, 444 pages
ISBN 978-0-13-485398-7

Reviewed by Mark Lamourine

In *Effective Python*, Slatkin offers a rather different twist on the cookbook format for programming references. In the conventional form, each chapter opens with a problem or a question. The body of the chapter then consists of a solution with some exposition. The premise is that the reader is learning the language features and capabilities. The recipes provide language-specific ways to achieve what are normally common goals.

Slatkin's approach is more of a catalog of best practices for the Python coder. The book is subtitled "90 Specific Ways to Write Better Python." Each of the 90 small "items" referred to in the subtitle opens with a recommendation. For example, Item 9: "Avoid `else` Blocks After `for` and `while` Loops." The main body of the item is a presentation of an argument for the recommendation. The arguments range from readability and performance to avoidance of common coding errors. Slatkin's arguments tend to follow a pattern. First he shows how the feature or construct is used commonly. He talks about why the typical usage makes sense at first and then how it can lead to problems. Only then does he offer his alternative, using new code fragments and explaining how the new code's behavior addresses the problems cited. Each item ends with a summary bullet list of things to remember.

The items are grouped into chapters thematically. Most are related to language features like lists, functions, or classes. The opening and closing chapters are more about idiom, style, and human behavior and are entitled, respectively, "Pythonic Thinking" and "Collaboration."

These two chapters directly express a thematic undercurrent that runs throughout the book: coding is a human endeavor and a craft and in every instance involves the judgment of the developer. Despite his recommendations favoring specific behaviors and constructs over others, Slatkin always appreciates why the common usage *is* common. His recommendations are always presented in a way that is meant to be persuasive rather than strident or proscriptive.

I started using Python with version 1.5, and version 2.x has been a staple for me since it was released in 2000. For me, version 3 was always "someday." I'm embarrassed to realize it's been 12 years. With the sunsetting of version 2 in January 2020 [1], it has become important not just to learn the differences, but to commit to version 3. The second edition of *Effective Python* treats only version 3, with none of the back references or porting comments that have been common for a decade.

There have been a couple of changes that I didn't really internalize. One was the introduction of the `bytes` and `str` types for representing strings. I understand the difference between ASCII byte strings and UTF-8, but the treatment in Python and the idiomatic usage have never become second nature. Slatkin addresses this as Item 3 in the first section. He shows how to recognize them and how to convert between them. More importantly he indicates *when* to convert between them and when to leave them alone.

I had been in the habit of using the `print()` function in Python 2 from the `__future__` module and the `str.format()` method instead of the formatting operator (%) for a long time. I was surprised to learn that there is a new string formatting method introduced in version 3.6 called *Literal String Interpolation*, or *f-strings*, for the prefix used to indicate one in the code. These work more like Jinja2 templating, where you use the name of the variable inline to resolve and replace the value in the string. What I really like about this item is the way Slatkin demonstrates the earlier methods, showing how it is easy to make errors with them. Finally, he demonstrates f-strings in a way that highlights how they resolve the problems.

Don't be fooled by my initial examples. *Effective Python* addresses some rather deep theoretical constructs as well. It has a fantastic treatment of generators, not just what they are and how to use them, but how they work and *why* to use them. Hint: avoid large in-memory arrays of computable values. Slatkin's treatments of metaclasses and concurrency also brought me some "aha!" moments. The references in these cases are provided for anyone who is learning about these topics for the first time and that's a good thing.

The only real quirk I noted with Slatkin's style is that it really requires you to properly read the text. In many reference works, once you're familiar with the topic you can skim to find just the bit of code you need. While none of the items here are particularly long or deep, the teaching style requires the reader to follow the narrative. I don't think that this is a problem, but it was an adjustment I had to make to get the most out of what I was reading.

Normally, I would encourage new coders to set something like this book aside until they had some experience and context to bring to it. In the case of *Effective Python,* I would consider suggesting they get it, skim the first few sections, and then set it aside. It will be there when they begin to ask the questions it tries to answer. It's a book I expect to return to.

Dependency Injection Principles, Practices, and Patterns

Steven van Deursen and Mark Seemann
Manning Publications, 2019, 522 pages
ISBN 978-1-61729-473-0

Reviewed by Mark Lamourine

I remember my confusion the first time I heard the term *dependency injection* (DI). I'd seen it used in some Ruby code with unit and functional tests, but I didn't know it had a name and didn't understand the formal basis for it. Since then I've spent significant time failing to create testable or flexible code using DI. Understanding DI has been on my back burner, and when I saw this book I had to see what DI is about.

"Principles, Practices and Patterns" is actually a pretty good description of the book. The authors are clearly fans of Martin Fowler, Robert Martin, and the *Design Patterns* [2] "Gang of Four." They make explicit reference to several design patterns that are extensively used to implement DI constructs. They also make good use of proper UML diagrams to illustrate object dependency relationships and life cycle. While prior knowledge of design patterns and UML isn't required, it will definitely help a reader understand the theory and the assertions the authors make about the structure of software and the effect that has on testability and maintainability.

At the end of Chapter 1 I found a paragraph that is easy to miss but critical to understanding this book. The goal of the authors is to help readers implement code designed with *loose coupling*. That is, code that depends on interface and API definitions of the code it uses rather than on the specific implementation. The authors' core assertion is that loose coupling is a generally desirable trait of well-designed systems. Dependency injection is just the technique they are offering to enable loose coupling. It is easy to lose track of that emphasis when trying to absorb the somewhat dense concepts that follow.

One thing becomes evident during the first three chapters: loosely coupled code looks more complex than tightly coupled code, at least at first look. In Chapters two and three, the authors show a simple three-layer application with a database, a user interface, and some business logic sandwiched in between. They do a good job of showing the options and decisions that lead to tightly coupled code. The design decisions are primarily a function of the desire for initial simplicity. They are natural and straightforward, based on the intent of the application.

In the following chapter the authors re-implement the application with a design in which the components are carefully decoupled. Each of the interacting classes defines an interface rather than just providing a function or method for callers. The design is significantly larger, increasing from four classes to nine and with three interfaces. The chapter is also twice as long. That extra text is used to explain the different considerations that are needed to design decoupled code. It takes a deliberate approach and the development of a set of habits to view a problem this way.

The second section is where the theory gets deep. These chapters present the DI design patterns and set of anti-patterns, concluding with a chapter on DI code smells. The final two sections show how to implement applications with DI, first directly and then using a kind of DI factory called a DI Container. These take existing classes and reflect them to create a new class that allows DI. The examples given are specific to .Net, though the illustration is useful.

I was a little concerned when I realized that the examples and code samples are written in C# and make use of .Net libraries. My worries were unfounded. The C# code will be clearly understood by anyone familiar with C++ or Java, and the .Net library references are reminiscent of Java APIs. The examples have the camelcase verbosity one would expect from those languages as well, but it doesn't interfere with clarity. Very few of the code fragments are longer than a single page, and the typeset annotations are well placed and clearly associated with the lines they describe.

Dependency Injection Principles, Practices, and Patterns provides a lot to chew on, and it's going to take me a while to properly consume and digest it. I have several web projects going where I hope to make use of it.

Building Secure and Reliable Systems

Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea, and Adam Stubblefield
Google LLC and O'Reilly Media Inc., 2020, 557 pages
ISBN 978-1-492-08312-2

Reviewed by Mark Lamourine

When Google publishes a guide for infrastructure, you can be sure that it's worth reading. The real question is: is it something you can use? Google works on a scale that few other companies can. As a purely practical matter, few companies have the resources or the strictest requirements for efficiency that characterize the handful of truly colossal Internet companies. I picked up *Building Secure and Reliable Systems* with a bit of skepticism.

I was also concerned with the size of the book. At over 500 pages it's still not the largest infrastructure book I've read. I wanted to see how the authors managed the challenge of providing a useful level of information in a manageable amount of space.

This is the third book in a series Google has published on the topic of site reliability engineering (SRE) [3]. This is Google's refinement of the system administration model that has been called DevOps. The first defines and describes the philosophy of the SRE model and the role that the SRE plays in an operational organization. The second is a "workbook" for SREs, describing how they go about their job. This third volume provides a set of best practices both for the enterprise and for the service groups. It puts the SRE into the context of a complete organization in a way that can be appreciated both by the SREs and by their management and business peers.

Where the earlier volumes focused on workers and their tasks, this one illuminates the factors to consider in design and implementation of computer systems. The chapters alternate between discussion of a single desirable aspect of a system and case studies to give concrete examples.

Those design aspects are not things that are usually put high on the system requirements list: understandability, resilience, and recovery. The only element specifically for security is least privilege. The authors recognize that encryption and user authentication get a lot of attention. Defense in depth requires more care and thought. Security vulnerabilities will always be present, but exploits can often be neutralized by limiting what an attacker can access.

Each of these chapters really just provides additional incentive to follow ordinary design best practices. These discussions provide weight to arguments against cutting corners in design and implementation, and they provide rationale for better decisions than are often made.

The implementation section addresses considerations for reliability during the realization of the design, with chapters on writing, testing, and deploying the code and on surveying and debugging systems. In these chapters, the real nature of the writing comes through. This is a volume of collected wisdom: it's a series of thoughts and reminders—remembering to stop and think when things go wrong, for example, and to pair work where one is typing and the other is a scribe, both to avoid losing information and for mentoring.

The final couple of chapters talk about what I think has become the most critical aspect of software development and system administration: culture. There can be a lot of focus on technical stars in hiring and team formation. What experience has shown me is that people want to do good work and to learn and challenge themselves. The most common frustration is poor team empowerment and communication. All of the preceding chapters are nullified if the developers and admins aren't given the freedom and incentives to collectively evaluate and then act on their decisions.

Each of the chapters is nicely self-contained. The writing is clean, almost sanitary. This reflects the Google aesthetic of minimal bling, flash, and distraction. The authors provide frequent cross references, and each chapter concludes with a summary and a list of references. The spare nature of the writing style makes for a surprisingly readable text.

Except for a few details and discovery of a small set of obscure but useful tools, I didn't learn a lot that was new. For the developer or sysadmin, this book is a good complete compendium of the highest level considerations for system design. For project management, it is a window into the kinds of things the team should be discussing and resolving throughout a project. I didn't see anything here that made me think "you can only do that if you're Google." I'll keep *Building Secure and Reliable Systems* handy for when I need to champion more thoughtful, purposeful design and operational behaviors. "See, this is how Google does it."

Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats

Alex Matrosov, Eugene Rodionov, and Sergey Bratus
NoStarch Press, 2019, 448 Pages
ISBN-13: 978-1-59327-716-1

Reviewed by Rik Farrow

I chose this book to review after listening to an invited talk at WOOT '20 by the main author, Alex Matrosov, and because Sergey Bratus is also an author. I would ordinarily have steered clear of books primarily about Windows, but once you get past Part 1, about rootkits, you find yourself in territory relevant to Linux systems. The authors cover information relevant to anyone running software on Intel or AMD chipsets.

The book is well written and organized, and it includes example code (all Windows) and dumps from malware and firmware samples. I had little trouble reading the book, although I did want a glossary of abbreviations handy after a while, as there are loads of obscure TLAs.

The authors start out by describing TDL3 and Festi rootkits. These are "old," designed for 32-bit versions of Windows that have long been out-of-date but likely still run. Most of the techniques used—plugins to extend the malware, using a rolling XOR as "encryption," changing registry keys—seem familiar. What makes Festi interesting is the malware designers' knowledge of kernel internals. They hook both file and network drives very deep in the software stack, making them difficult to discover through Host Intrusion Prevention System (HIPS) products, as these tools also install hooks at the same layer.

As security in Windows improves, malware writers have shifted their focus to bootkits, methods of infecting the kernel during the boot process. Here again you will find information relevant to any operating system that relies on x86 chipsets. The authors cover the boot process and provide analyses of bootkit samples, as well as the arms race in bootkits that leads to UEFI Boot. UEFI is supposed to provide secure boot, with checks of the authenticity of code, but in many cases vendors have not properly implemented the standard.

Chapters 15 and 17 demonstrate the use of a tool, called Chipsec, that allows you to probe your firmware settings. The tool works for Windows, Linux, and macOS, and you can find the tool on GitHub at https://github.com/chipsec/chipsec. With the tool, you could see if your firmware is write-protected and whether SPI flash memory protections also have been enabled. The authors have tested a number of motherboards, and many of them have either not enabled or included firmware protections, making the system more susceptible to bootkit malware.

I really hadn't been paying much attention to Windows malware over the last decade, and the focus of this book is on two specific areas that cover some of the most sophisticated attacks possible. I enjoyed reading the book and learning about the malware, even if it was not particularly relevant to me, as "I don't do Windows." Still, there's more than enough here that's relevant to Linux users, as malware writers are now turning their attention to Linux servers.

References

[1] https://www.python.org/doc/sunset-python-2.

[2] E. Gemma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, 1994).

[3] https://landing.google.com/sre/books/.

Statement of Ownership, Management, and Circulation, 09/30/2020

Title: USENIX Association/ *;login:*
Pub. No. 1044-6397
Frequency: Quarterly
Number of issues published annually: 4
Subscription price: $90.

Office of publication: 2560 9th St., Suite 215, Berkeley, CA 94710-2565
Contact Person: Sara Hernandez. Telephone: 510-528-8649 x100

Headquarters or General Business Office of Publisher: USENIX Association, 2560 9th St, Suite 215, Berkeley, CA 94710-2573

Publisher: USENIX Association, 2560 9th St, Suite 215, Berkeley, CA 94710-2573

Editor: Rik Farrow; Managing Editor: Michele Nelson, located at office of publication.

Owner: USENIX Association. Mailing address: As above.

Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: None.

The purpose, function, and nonprofit status of this organization and the exempt status for federal income tax purposes have not changed during the preceding 12 months.

Publication Title USENIX ASSOCIATION/ ;login:			**Issue Date for Circulation Data Below** 09/01/2020 — Fall ’20 Issue	
Extent and Nature of Circulation			**Average No. Copies Each Issue During Preceding 12 Months**	**No. Copies of Single Issue (Fall 2020) Published Nearest to Filing Date**
a. Total Number of Copies *(Net press run)*			1863	1875
b. Paid Circulation *(By Mail and Outside the Mail)*	(1)	Mailed Outside-County Paid Subscriptions	805	818
	(2)	Mailed In-County Paid Subscriptions	0	0
	(3)	Paid Distribution Outside the Mails	513	529
	(4)	Paid Distribution by Other Classes of Mail	0	0
c. Total Paid Distribution			**1318**	**1347**
d. Free or Nominal Rate Distribution *(By Mail and Outside the Mail)*	(1)	Free or Nominal Rate Outside-County Copies	79	79
	(2)	Free or Nominal Rate In-County Copies	0	0
	(3)	Free or Nominal Rate Copies Mailed at Other Classes	20	25
	(4)	Free or Nominal Rate Distribution Outside the Mail	73	30
e. Total Free or Nominal Rate Distribution			**172**	**134**
f. Total Distribution			**1490**	**1481**
g. Copies Not Distributed			374	394
h. Total			**1863**	**1875**
i. Percent Paid			88.46%	90.95%
Electronic Copy Circulation				
a. Paid Electronic Copies			410	444
b. Total Paid Print Copies			**1729**	**1791**
c. Total Print Distribution			**1899**	**1925**
Percent Paid (Both Print and Electronic Copies)			91%	93%

Submit Your Work!

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

30TH USENIX SECURITY SYMPOSIUM

AUGUST 11–13, 2021 | VANCOUVER, B.C., CANADA

The 30th USENIX Security Symposium will bring together researchers, practitioners, system administrators, system programmers, and others to share and explore the latest advances in the security and privacy of computer systems and networks.

Winter paper submission deadline: Thursday, February 4, 2021

PROGRAM CO-CHAIRS

Michael Bailey
University of Illinois at Urbana–Champaign

Rachel Greenstadt
New York University

www.usenix.org/sec21

Seventeenth Symposium on Usable Privacy and Security

Co-located with USENIX Security '21
August 8–10, 2021 | Vancouver, B.C., Canada

The Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021) will bring together an interdisciplinary group of researchers and practitioners in human computer interaction, security, and privacy. The program will feature technical papers, including replication papers and systematization of knowledge papers, workshops and tutorials, a poster session, and lightning talks.

Mandatory Paper Registration Deadline:
Thursday, February 18, 2021

Symposium Organizers

General Chair
Sonia Chiasson, *Carleton University*

Technical Papers Co-Chairs
Joe Calandrino, *Federal Trade Commission*
Manya Sleeper, *Google*

www.usenix.org/soups2021

NOTES

USENIX Board of Directors

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

PRESIDENT
Amy Rich, *Redox*
arr@usenix.org

VICE PRESIDENT
Arvind Krishnamurthy, *University of Washington*
arvind@usenix.org

SECRETARY
Kurt Andersen, *LinkedIn*
kurta@usenix.org

TREASURER
Kurt Opsahl, *Electronic Frontier Foundation*
kurt@usenix.org

DIRECTORS
Cat Allman, *Google*
cat@usenix.org

William Enck, *North Carolina State University*
will@usenix.org

Laura Nolan, *Slack Technologies*
laura@usenix.org

Hakim Weatherspoon, *Cornell University*
hakim@usenix.org

EXECUTIVE DIRECTOR
Casey Henderson
casey@usenix.org

;login: Enters a New Phase of Its Evolution

Cat Allman, Rik Farrow, Casey Henderson Arvind Krishnamurthy, and Laura Nolan

For over 20 years, *;login:* has been a print magazine with a digital version; in the two decades previous, it was USENIX's newsletter, *UNIX News*. Since its inception 45 years ago, it has served as a medium through which the USENIX community learns about useful tools, research, and events from one another. Beginning in 2021, *;login:* will no longer be the formally published print magazine as we've known it most recently, but rather reimagined as a digital publication with increased opportunities for interactivity among authors and readers.

Since USENIX became an open access publisher of papers in 2008, *;login:* has remained our only content behind a membership paywall. In keeping with our commitment to open access, all *;login:* content will be open to everyone when we make this change. However, only USENIX members at the sustainer level or higher, as well as student members, will have exclusive access to the interactivity options. Rik Farrow, the current editor of the magazine, will continue to provide leadership for the overall content offered in *;login:*, which will be released via our website on a regular basis throughout the year.

As we plan to launch this new format, we are forming an editorial committee of volunteers from throughout the USENIX community to curate content. This new model will increase opportunities for the community to contribute to *;login:* and engage with its content. In addition to written articles, we are open to other ideas of what you might want to experience. We welcome your comments and suggestions: login-comm@usenix.org.

Interview with Clem Cole

Rik Farrow

Clem Cole is an old school hacker and "Open Sourcerer" with more than 45 years of free and open source system development experience. Clem has held practically every position in the computer field from operator, programmer, and designer to VP of Engineering, CTO, and startup founder. He first encountered the early editions of UNIX in the 1970s while at Carnegie Mellon University, later doing his graduate work at the University of California, Berkeley. He has been designing and developing operating systems and technical computing systems ever since, currently leading an international team of engineers. He helped to write one of the original TCP/IP implementations in the late 1970s, and is known as one the authors of the precursor to IM, the UNIX talk program, as well as other more humorous and notorious hacks. He is honored to be a past President of the USENIX Association and the 2016 winner of the Linus Pauling Prize for Science. clem@ccc.com

I first met Clem Cole at a USENIX conference, probably in the 90s, but I had encountered him via a paper he helped with in 1985. Ted Kowalski had written `fsck`, bringing together ideas from three previously existing UNIX programs, `ncheck`, `icheck`, and `dcheck`, and experience using earlier IBM programs, Scavenger and Vulture [1], for recovering after disk crashes.

I had also heard that Clem had a long history with USENIX, and decided to interview him for this, the final print issue. I had learned by reading early issues of *UNIX Notes* and *;login:* that USENIX conferences were how UNIX users exchanged information in the early days, and it occurred to me that Clem was a participant I could ask about this.

Rik Farrow: When did you begin working with UNIX? I encountered UNIX in the early 80s, while working for companies in the Bay Area.

Clem Cole: By the 80s UNIX was even cooler than when I first encountered it. I started using UNIX with Fifth Edition version in 1976 while at Carnegie Mellon (CMU).

Truth is, my first experience with UNIX and C in particular, coming from the IBM 360 and DEC PDP-10, found me skeptical. But I had been schooled in the CMU gospel of using systems program languages (BLISS in this case) so I had already started to transition from 360 assembler.

What made UNIX/C really cool was that as much as I liked the stuff we had on the PDP-10s (like the XGP—the predecessor to the laser printer) BLISS on the PDP-11 required cross compiling. C was self-contained. The documentation for C was almost non-existent, with the exception of Dennis Ritchie's paper in V5 and V6 in the `c` directory in `/usr/doc`. The code from the compiler was not great compared to BLISS, but it was "good enough."

And we had our own PDP-11 in the CMU Electrical Engineering Digital Lab and we did not have to share it with many other folks.

Ted had a xeroxographic copy of the Lions book [2] and I made my own copy. Then Ted came back with proofs for Kernighan and Ritchie [3] in a binder and I read those two documents that spring and things about UNIX started to click. Pretty soon I started to see that I could get most anything I had been able to do on the PDP-10s and the 360 on the PDP-11 and I only shared it with a few other people. That was way cool.

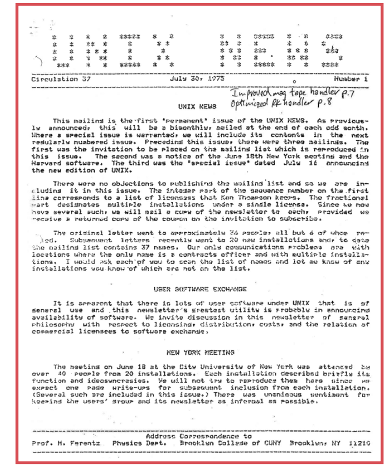
And then one day we had a disk crash on a machine in CMU's BioMed Department. I got a call from the guys that ran it, and they wanted to try to use the EE system to try to fix the disk. Ted and I both had used a disk reconstruction program on the IBM and Ted had been an IBM MTS hacker at the University of Michigan before he came to CMU. I remember spending a number of hours with `ncheck/icheck/dcheck` and grumbling to Ted as we were working with their disk.

It turns that out Ted had started a new program but it was not complete. Now he had a mission. By the way, the original name of the `fsck` program used a different second letter.

The other thing I saw around then was a copy of some of the original issues of *UNIX News* that Columbia University was printing up. I don't remember who had them, but I think it was someone else with a connection to Harvard or maybe Columbia. I got on the mailing list somehow and started eating it up.

As an undergrad I could not travel, but when I first started to work for Tektronix in 1979, I went to my first USENIX conference (I want to say Toronto, but I could be wrong). An early winter one was Boulder where USENIX had rented a movie theater, the same theater that was featuring the new movie, *Black Hole*. What I do remember the most of that conference is that's where Tom Truscott regaled us on his homemade autodialer they built so they could run UUCP.

Originally, we came to those meetings representing our orgs—universities or commerical entities. You were supposed to have the signature page of your AT&T UNIX license to join. I don't remember when the first personal memberships were offered, but I was the seventh person to join USENIX.


UNIX NEWS

Circulation 37 — July 30, 1975 — Number 1

Improved mag tape handler p.7
Optimized RK handler p.8

UNIX NEWS

This mailing is the first "permanent" issue of the UNIX NEWS. As previously announced, this will be a bimonthly, mailed at the end of each odd month. Where a special issue is warranted, we will include its contents in the next regularly numbered issue. Preceding this issue, there were three mailings. The first was the invitation to be placed on the mailing list which is reproduced in this issue. The second was a notice of the June 18th New York meeting and the Harvard software. The third was the "special issue" dated July 16 announcing the new edition of UNIX.

There were no objections to publishing the mailing list and so we are including it in this issue. The integer part of the sequence number on the first line corresponds to a list of licensees that Ken Thompson keeps. The fractional part designates multiple installations under a single license. Since we now have several such, we will mail a copy of the newsletter to each, provided we receive a returned copy of the coupon on the invitation to subscribe.

The original letter went to approximately 76 people, all but 6 of whom replied. Subsequent letters recently went to 20 new installations and, to date the mailing list contains 37 names. Our only communications problems are with locations where the only name is a contracts officer and with multiple installations. I would ask each of you to scan the list of names and let me know of any installations you know of which are not on the list.

USER SOFTWARE EXCHANGE

It is apparent that there is lots of user software under UNIX that is of general use and this newsletter's greatest utility is probably in announcing availability of software. We invite discussion in this newsletter of general philosophy with respect to licensing, distribution, costs, and the relation of commercial licensees to software exchange.

NEW YORK MEETING

The meeting on June 18 at the City University of New York was attended by over 40 people from 20 installations. Each installation described briefly its function and ideosyncrasies. We will not try to reproduce them here since we expect one page write-ups for subsequent inclusion from each installation. (Several such are included in this issue.) There was unanimous sentiment for keeping the users' group and its newsletter as informal as possible.

Address Correspondence to
Prof. M. Ferentz Physics Dept. Brooklyn College of CUNY Brooklyn, NY 11210

Vol. 1, No. 1 of *UNIX News*, July 30, 1975. The circulation was 37.

So back to your question. In those days DEC did not support UNIX, so we had a "we all are in this together" attitude. Everything was "open source" because we all had licenses. I think the thing that is lost today is that it was the cost of the hardware that was the limit to being part of the "UNIX club," not the cost of the UNIX software sources.

RF: That's interesting, since AT&T raising the license fee for the source for System V Release 4 (SVR4) was the main reason for the UNIX wars [4] that began in the late 80s.

CC: Actually, that's not quite true. The UNIX wars had started long before then. The 1988 SVR4 release and the raising of the license redistribution fee in particular was the source of the "fair and stable license terms" of Open Software Foundation (OSF) verses UNIX International (UI). You have to understand that each time AT&T had released a commercial redistribution license (starting with V7) the fees had gone up. The vendors had been having a knock-down, drag-out war for 5–10 years by 1988. UI vs. OSF was just the final battle.

The problem for the vendors was they treated UNIX like they owned their OSs and made them private with lots of local hacks to create vendor lock-in for their customers. The UNIX wars were really based on who got to decide what the definition of UNIX was going to be.

AT&T thought they got to say it because they owned the intellectual property. But the Berkeley Software Distribution (BSD) version had the greatest mind share as it included TCP/IP support. DEC, Apollo, HP, Masscomp, Sun, IBM and others had their customers running some version of UNIX on their hardware. And independent software vendors were annoyed because life had not gotten better—hence the 1985 /usr/group UNIX Standard that would later beget the IEEE POSIX work.

RF: How large were those early USENIX conferences? Dozens of people, hundreds of people? I've heard that by the late 80s there could be thousands of people attending.

CC: When they were at Harvard, Columbia, etc.—that is, the time of *UNIX News*—a conference fit into a classroom. By the time of Boulder it was probably about 100–150—about half a movie theatre full.

By the late 80s (when the final phase of the UNIX wars started) the San Francisco conference and I think the 10th USENIX conference in Portland were over a thousand. I think the peak was probably two or three thousand.

After Portland (summer 1985), USENIX started to fork into smaller dedicated conferences targeting subtopics and trying to keep it to be about 150–200 per conference.

RF: So what was it like at Boulder? Most of us know what modern USENIX conferences are like, with most having a focus on paper delivery, and some on talks.

CC: Remember there were no papers or proceedings in those days. Just talks. No PowerPoint either. Just overhead slides. And people came with prepared talks and signed up to give them. Nothing was preplanned.

It was also the first time I met Bill Joy. He had already started to build a cult around him. He talked about UCB Pascal and was very interesting. I also met Dennis Ritchie and Steve Bourne for the first time and was awed at how down to earth they were. They asked me questions and wanted my opinion. That was so cool. It was really a collegial setting. We were all sharing our experiences.

I think that's also where I met Bruce Borden for the first time. He had written the first parts of `mh` and he gave a talk about it. I wanted to try it immediately after I got back. As I said, Truscott talked about his fake autodialer and I remember being so impressed with it.

Boulder was sort of the start of Usenet. Truscott described the three schools connected using UUCP in North Carolina in his talk. Brian E Redman—the "ber" of "Honey-Dan-Ber" UUCP—said he would like to be able to call him from the systems in BTL in Whippany, NJ, named after the Marx brothers. Dennis Ritchie offered up the Bell Labs `research` system in Murray Hill.

At Tektronix, we were working with the University of California, Berkeley (UCB) on things like Spice, so I had talked my boss into letting me buy an autodialer to call `ucbvax`. So when I got back we joined up, and UCB was talking to `research`. Within a year Armando Stettner added `decvax` and `ihnp4` was added during the same time frame.

The fact is Bell Telephone Labs (BTL) had a large internal UUCP-based network before Usenet (which was smaller), but once `research` and `ihnp4` joined that changed the dynamic. Also, BTL was not on the ARPANET which a couple of the Usenet sites sort of were. Funny thing, forwarding from the ARPANET was discouraged originally. But at some point, the DARPA folks realized "Metcalf's Law" [5], that joining two networks make each a lot more valuable.

RF: Gaining membership in the early Internet/ARPANET was extremely weird from my perspective.

CC: When the ARPANet and the early Internet were set up, the US government paid for everything. To join, you had to be a DoD contractor and had to be sponsored. I read that in 1975 dollars, the cost per host was $250K a year and that also explains why some of the choices were made. A site on the network, like MIT and CMU, were not going to put their connection at risk.

Part of the problem was that many universities wanted to be part of the ARPANET but could not be as they were not doing ARPA work. Sometime early in the Reagan administration, the US government wanted to get out of funding the networking experiment they had started. Originally, CS-NET was set up by the NSF and contracted to BBN, so any research group could get a connection, but you had to pay for it and your leased telco connections. In fact, to help keep costs down, CS-NET set up a UUCP-like system, called "Phone-Net."

Within a a year or so, CS-NET was allowed to connect commercial sites too. That's how Masscomp got its connection to BBN and became part of the emerging Internet for $50K a year. Rick Adams had forked out UUNET and they too joined CS-NET and became the "offical" (recognized) UUCP/ARPANET bridge.

Finally, we started to get regional networks competing with CS-NET because the telco costs could be kept in check better and the era of the ISP appeared. The other change was that telco costs had to drop. That $50K/year Masscomp connection was a 56Kbit/second leased line from Westford, Massachusetts to Cambridge—a distance of about 30 miles. A T1 (1.44Mbit/sec) connection was closer to $50K/month.

So for a long time, the bulk of the email and netnews traffic was UUCP over dial-up.

The good news is that the telcos did start to figure out how to build cheaper digital circuits. And, once "cheap enough" connections arose, the need for UUCP and dial-up started to fade.

RF: I remember using `ihnp4` as a mail forwarder, but not many people today are going to recognize that hop. Was `ihnp4` in the Chicago area?

CC: Indian Hill New Products System 4, or `ihnp4` was in suburban Chicago. The three big national sites for Usenet were `decvax`, `ihnp4` and `ucbvax`. There was a study done by someone at BTL that concluded that for every call `ihnp4` underwrote, it generated between 10 and 20 downstream calls and that was good for AT&T so they continued to underwrite it. At its peak, `decvax` had a half to three-quarters of a million dollar phone bill. That was the trigger for USENIX to start to look for an alternative. Rick Adams, working for the USGS and running the site named `seismo`, proposed the

UUNET site, which USENIX helped fund [6]. UUNET become a commercial entity the following year.

RF: UUCP mail forwarding over IP is a big topic. I interviewed Mary Ann Horton [7], who had worked for BTL as well as doing a lot with Usenet. She explained things like the maps people were distributing at USENIX conferences and a separate tool to help people forward UUCP mail. Peter Salus' article [6] also explains her role in the maps project, and more details about the founding of UUNET.

CC: The printed maps were given away at conferences, but the tool was used to try to shorten paths for email and net news traffic. Remember, until IP where you have flat address space, UUCP was purely store and forward and at the complete message level. IP is store and forward at the packet level and the other difference is that the "store" time was in minutes to hours for UUCP, as opposed to microseconds for IP.

RF: What role did you experience USENIX meetings playing in getting an effective email network started?

CC: Well, it really was a confluence of time and events. Because of USENIX we were meeting. Most people could not be part of the ARPANET for reasons I've already covered. Because Version 7 UNIX included UUCP, everyone now had a way to send intersite email if you had at least a Version 7 UNIX box, a modem and a friend with an auto-dialer. Remember that self-dialing modems didn't exist yet and to dial out to another site required a DN11 and a Bell model 801 ACU—automatic calling unit—Truscott's trick not withstanding.

References

[1] Clem Cole's post about the origins of `fsck`: https://minnie.tuhs.org/pipermail/tuhs/2017-May/011467.html.

[2] J. Lions, "Commentary on UNIX 6th Edition, with Source Code": https://www.amazon.com/Lions-Commentary-Unix-John/dp/1573980137.

[3] B. Kernighan and D. Ritchie, "The C Programming Language": https://en.wikipedia.org/wiki/The_C_Programming_Language#History.

[4] "The UNIX Wars": https://en.wikipedia.org/wiki/Unix_wars.

[5] Metcalf's Law: https://en.wikipedia.org/wiki/Metcalfe%27s_law.

[6] P. Salus, "Distributing the News: UUCP to UUNET," *;login:*, vol. 40, no. 4 (August 2015): https://www.usenix.org/system/files/login/articles/login_aug15_09_salus.pdf.

[7] R. Farrow, "Interview with Mary Ann Horton," *;login:*, vol. 45, no. 1 (Spring 2020): https://www.usenix.org/publications/login/mar20/horton

The USENIX meetings had been around for 8–10 years before Usenet comes into play. But it was already clear before what we now call the Internet replaced ARPANET, that people wanted/needed email—that Usenet was organically born.

Again, it was need and timing more than anything else that helped get UUNET started, plus the wild growth of the Internet we saw in the 90s.

RF: How did your involvement with USENIX change over the years?

CC: USENIX has a special place in my heart. Without a doubt it helped my career. When I first started coming I was in the audience soaking things in, then I transitioned to someone writing and presenting papers. Next I was asked to be on program committees and eventually chair a few conferences. I was nominated and elected to the Board and eventually became president. I still participate as I can and I would consider working on the Board again as well as other projects that folks consider.

Interview with Kirk McKusick

Rik Farrow

Dr. Marshall Kirk McKusick writes books and articles, teaches classes on UNIX- and BSD-related subjects, and provides expert-witness testimony on software patent, trade secret, and copyright issues particularly those related to operating systems and filesystems. He has been a developer and commiter to the FreeBSD Project since its founding in 1993. While at the University of California, Berkeley, he implemented the 4.2BSD fast filesystem and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD. He earned his undergraduate degree in electrical engineering from Cornell University and did his graduate work at the University of California, Berkeley, where he received master's degrees in computer science and business administration and a doctoral degree in computer science. He has twice been president of the board of the USENIX Association, is currently a board member and treasurer of the FreeBSD Foundation, a senior member of the IEEE, and a member of the USENIX Association, ACM, and AAAS.

In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar (accessible from the Web at http://www.mckusick.com/~mckusick/) in the basement of the house that he shares with Eric Allman, his partner of 40-and-some-odd years and husband since 2013. mckusick@mckusick.com

I first met Kirk McKusick at a USENIX conference in the 1990s. By that point I was working with Dan Klein and others on the tutorial committee and listened to portions of all tutorials given during LISA conferences, so I might have met Kirk that way. Later, Kirk and I would sometimes meet during FAST workshops.

Once when we were sitting together during paper presentations, someone presented a method of speeding up `fsck` on Linux `ext` filesystems by caching the results of intermediate phases. Feeling a bit mischievious, I mentioned to Kirk that this sounded like an improvement that belonged in the Fast File System (FFS), something Kirk had written, taught, and still supported in BSD. Kirk replied that this should be easy, as policy and implementation were kept separate in BSD, unlike in Linux. By the next morning, he had created a new version of `fsck`.

Rik Farrow: When did you first encounter UNIX?

Kirk McKusick: I encountered UNIX for the first time while at the University of Delaware in 1976. Later that year, I was a graduate student at the University of California, Berkeley (UCB), and started the month after Ken Thompson ended his sabbatical, in August of 1976. Thompson had helped install UNIX Version 6 on a PDP 11 there, working with Chuck Haley and Bill Joy, two other UCB graduate students. They also worked on a version of Pascal Thompson had written, and demand for that lead to the first Berkeley Software Distribution in 1977 [1].

RF: When and why was the Computer Science Research Group (CSRG) started?

KM: Professor Bob Fabry, with help from Bill Joy, had been working on getting a research grant from DARPA and needed a project name, so he decided to call the project the Computer Systems Research Group (CSRG). That was in June of 1980.

RF: You are best known as the author of the Fast File System, today known as the UNIX File System [2]. How did that come about?

KM: The filesystem developed for the early UNIX versions had terrible performance, getting throughput of only 2% of the bandwidth of current disks. Doubling of block size, to 1024 bytes, managed to raise the throughput to 4%, so this area seemed like fertile ground for research. I was working for the university, but part time, as full time work would have required the university to provide benefits as well—still an issue today. My advisor's research grant had ended during the summer, and I asked Bill Joy, who I used to share an office with, if he could give me a project to work on.

References

[1] S. Pate, *UNIX Filesystems: Evolution, Design, and Implementation*, (Wiley, 2003): https://www.oreilly.com/library/view/unix-filesystems-evolution/9780471456759/chap01-sec008.html

[2] K. McKusick, "A Brief History of the BSD Fast File System," *;login:*, vol. 32, no. 3 (June 2007): https://www.usenix.org/system/files/login/articles/584-mckusick.pdf

Joy had started work on a filesystem prototype, but had only written the superblock and cylinder group structures so far. He handed the project off to me, and I finished the rest as a userspace file system.

Joy convinced me to drop the prototype into the kernel, and that took me months, as there are concurrency and race conditions as well as other things, like cache invalidation, to handle. Then Joy convinced me to store my own home directory on the new filesystem, to show that I believed in my work. I realized that there was no way to do backups, so I wrote `dump` to backup and `restore` to recover from backups.

I also got tired of running `icheck`, `dcheck`, and `ncheck`, the first three passes you'd see with `fsck`, so I got `fsck` running. All this was a sidetrack on my way to getting my PhD.

Later on, Joy funded my trip to a Boston USENIX conference with DARPA money. I created a couple of hand-written slides, and Joy took them to get typed up. There were about 1200 attendees when I went to speak, but the slides Joy had provided were nothing like the ones I had written. When I told people that, they laughed, and the presentation went well.

The FFS could get around 40% of the bandwidth of disks, 10x the performance of the older filesystem. I learned from this experience that you should pick problems where there is a lot of fertile ground.

;login: and Open Access

Laura Nolan

As a USENIX volunteer for the past several years, one of the things I value most about USENIX is the association's unequivocal support for open access—the distribution of research online free of cost or other access barriers. Paywalled content restricts the spread of ideas and knowledge, and it reduces the impact of the work that researchers and writers do.

USENIX's support for open access has been a long-held stance. In our announcement of our move to open access in March 2008 we said that "we hope to set the standard for open access to information, an essential part of our mission." We do set that standard by making open access as straightforward as it can be: all USENIX authors and speakers retain copyright of their work, and all proceedings and recordings are freely published online—no money, or even a registration process, is needed to view any USENIX open access content.

USENIX became open access for all conference proceedings in 2008, followed by recordings of talks in 2010. Since 2010, *;login:* has been the sole exception to our open access philosophy, with access to each edition of the digital version of the magazine restricted to USENIX members for the first year after publication.

;login: is changing after this issue. From the start of 2021 it will no longer be a print magazine. But *;login:* will still be here, just in a different form—and this transition also brings the opportunity to resolve this final anomaly in USENIX's commitment to open access. All new *;login:* content will be available to all immediately after publication.

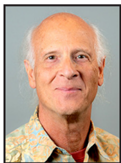
;login: is an old friend now and part of me is sad to see this chapter of its history end. However, as a regular contributor to *;login:*, I welcome the opportunity to share the content I create more widely. I believe that *;login:* is a very special part of USENIX—it spans the divide between research and industry, and it truly is a reflection of the best of all that we are. Making *;login:* freely accessible to all will make it even more meaningful.

Our Favorite *;login:* Articles, 2005–2019

Rik Farrow, Laura Nolan, and Arvind Krishnamurthy

When we learned that print *;login:* was to end with this, the Winter 2020 issue, several of us decided that we would pick out some of our favorite articles published during the previous 15 years. To help divide the work of skimming through the 90 issues published during the period, we each focused on a four- or six-year set of issues.

Of course, you probably have your own favorites. One problem with printed magazines is that feedback is rare, other than the occasional email or comment made to the editor during a conference. Some of Rik's favorites are based on data collected from Web server log files. The new, digital format will provide more opportunities for engagement, with the ability to add comments to articles. We hope you enjoy this walk through *;login:*'s rich recent past and look forward to introducing you to new articles that continue this tradition in its new medium.



Rik Farrow: 2005–2010

Although I had the ability to use the log analyses I had collected during these years, I've included articles that were not just popular over time, but also some that I particularly liked. In a way, this was a difficult task for me, because most articles were published because I liked the topic and felt that the authors involved could write well. Also, at least one article in each issue was designated as open, meaning that those articles, typically the first article after "Musings," had an advantage over other articles that wouldn't be available for download by non-members for another year. For these reasons, I didn't adhere to the rankings found in the log analysis obsessively, and included other articles.

2005

I started working as editor this year, and I will confess that as I worked through this six-year period, the articles did get better over time. Editing *;login:* really was a learning experience for me.

I had been surprised at some of the things that appeared high in the log analysis when I first started. *;login:* included a column on C# by Glen McCluskey, and his most popular column was about serialization in C# [1]. Another column, this one by Adam Turoff, explained date and time formatting in Perl [2]. Ceph was introduced this year as well, with an article by the authors [3].

2006

Steve Johnson, past USENIX Board President, and author of yacc, lint, and the Portable C compiler while at Bell Labs, wrote one of my favorites, an article about how hardware affects performance. Steve demonstrated how the stride affects performance when processing large arrays of 64 bit values [4].

Rob Thomas and Jerry Martin of Team Cymru wrote about their very practical research into the underground economy. They joined dozens of underground sites, learning about just how much a stolen credit card or bank account is worth on the black market [5].

2007

Simson Garfinkel needed to analyze vast quantities of data collected from hard drives, and working with Amazon saved him thousands of dollars in hardware costs. Simson explained how he used S3 and EC2, sharing his experiences with what were new services at the time [6]. There were also two articles about Xen, the hypervisor technology used at Amazon, in this issue.

Sam Stover, Dave Dittrich, John Hernandez, and Sven Dietrich installed malware on Windows systems so they could analyze the Storm and Nugache trojans. What made these trojans different was that they used P2P communication instead of IRC for command and control [7].

2008

Edward Walker wondered if Amazon's cloud could take the place of large clusters for scientific computing, and discovered that there are definitely differences between a cluster you control and configure and the cloud [8]. There was also an article about Solaris virtualization options in this issue.

David N. Blank-Edelman wrote one of his most popular Perl columns, where he used the open source Timeline tool and some Perl modules to convert `crontab` files into Gantt charts [9].

2009

Alva L. Couch wrote "Is It Easy Being Green?", an article about the two different types of green, ecology and money [10].

As a nice example of the variety of work found in *;login:*, Rudi Van Drunen had a popular article about hardware. Rudi wrote about digital and analog signals, how they work and how analog gets converted into digital signals: [11].

2010

Konstantin Shvachko, one of the authors of HDFS, penned an article about the limitations of HDFS, due to its design: [12]. The same issue had another Ceph article, pointing out Ceph's scalability.

Andrew Tanenbaum, Raja Appuswamy, Herbert Bos, Lorenzo Cavallaro, Cristiano Giuffrida, Tomáš Hrubý, Jorrit Herder, Erik van der Kouwe, and David van Moolenbroek published an update on Minix3 [13]. There had been two other Minix3 articles published in *;login:* during the decade. This article focused on the ability to restart portions of the kernel, a topic of the first paper at OSDI '20.

Laura Nolan: 2011–2015

The time period that I reviewed for this piece is also when I first started to attend USENIX events and to read *;login:*. Rereading the editions from these years made me incredibly nostalgic (and not only for in-person conferences!). It was very difficult to choose only one article for each year, to the extent that I gave up and quite frankly, just cheated.

2011

;login: has a very strong track record on security. The article I've chosen to represent 2011 is Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina on "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation" [14]. This article was dedicated to the memory of one of the authors, Len Sassaman, who had passed away earlier that year. It's a very thoughtful piece that characterises well-known security exploits (such as printf-family string format vulnerabilities) as a form of "weird instruction," and casts security as a problem of computability: what execution paths can our programs be trusted not to take, under any circumstances?

2012

My favourite article from 2012 is an example of cascading failure writ small: "Understanding TCP Incast and Its Implications for Big Data Workloads" by Yanpei Chan, Rean Griffith, David Zats, Anthony D. Joseph, and Randy H. Katz [15], which provides a systems model that explains pathological network throughput problems seen in early big-data systems. It's also a research and industry collaboration, which is apt for an association that spans industry and academia.

2013

;login: has had a variety of wonderful regular columnists, but none can top James Mickens for sheer entertainment value. His 2013 column "The Saddest Moment" [16] combines savage satire of papers and presentations about Byzantine fault tolerance with effortless education on the topic.

2014

2014 was a tough year to pick one favourite, because this year included both Brendan Gregg's debugging mystery "The Case of the Clumsy Kernel" [17], as well as "Analysis of HDFS under HBase: A Facebook Messages Case Study" by Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau [18], which demonstrates how "mechanical sympathy" between workloads and the infrastructure they run on is critical at scale, but can easily get lost with layered abstractions.

However, for sheer controversy, the article of the year has to be Todd Underwood's "The Death of System Administration" [19], based on his LISA keynote in 2013. Underwood proposes a future where operations engineers with software sensibilities (or vice-versa) working with better platforms will supersede manual systems administration work. We may not be sitting on the couch sipping bourbon and eating bon-bons quite yet, but I think Underwood is fundamentally correct about the direction we're traveling in.

2015

My 2015 pick (albeit with an off-by-one error) is a brace of articles about bugs. Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm's article "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems" [20] highlights how basic attention to error-handling code can increase the reliability of production systems.

"What Bugs Live in the Cloud?: A Study of Issues in Scalable Distributed Systems" by Haryadi S. Gunawi, Thanh Do, Agung Laksono, Mingzhe Hao, Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, and Riza O. Suminto [21] analyses three types of troublesome bugs found in distributed systems such as Hadoop, HDFS, HBase, Cassandra, ZooKeeper, and Flume. The analysis of the varieties of "SPoF bugs" that can crash entire systems that are intended to be redundant should be required reading for all software engineers and SREs.

Arvind Krishnamurthy: 2016–2019

I focused on articles published over the last few years and what struck me was the rich diversity of the articles. *;login:* has routinely included articles from both academia and industry, often provided tutorials on recent developments in software engineering, and discussed emerging trends in the computing industry.

2016

My favorite article from 2016 is “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems” by Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca [22]. Debugging distributed systems using logs is a difficult task, as what is recorded on logs is defined a priori and since it is hard to correlate log entries across a distributed system. Pivot tracing provides a novel approach that combines dynamic instrumentation with causal tracing and is thus suitable for production systems.

2017

My 2017 pick is an article describing an industry system that is in widespread use. Daniel Firestone describes a cloud-scale programmable virtual switch in “VFP: A Virtual Switch Platform for Host SDN in the Public Cloud” [23]. The article describes how Microsoft Azure enforces SDN policies across its large datacenters using the virtual switch. In addition to laying out the motivation for building the system, the article describes the design constraints that are unique to a public cloud.

2018

For 2018, I picked a practitioner’s guide to working with XDP, a new programmable layer in the kernel network stack. In the article, “XDP-Programmable Data Path in the Linux Kernel” [24], Diptanu Gon Choudhury provides background information on Berkeley Packet Filter, a core kernel technology introduced almost two decades ago, and how it has been recently extended to provide a power programmable layer inside the kernel that is intended to close the performance gap with respect to kernel-bypass solutions.

2019

My favorite article from 2019 is “Noria: A New Take on Fast Web Application Backends” by Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris [25]. This article describes a system that addresses performance problems faced by many web application backends. It outlines a system design that doesn’t neatly fit into traditional categories, such as databases or streaming engines, but rather creates a bridge across these technologies.

References

[1] "Working with C# Serialization," by Glen McCluskey, February, 2005: https://bit.ly/3e5IUfV.

[2] "Date and Time Formatting in Perl," by Adam Turoff, June, 2005: https://bit.ly/2TGKFqw.

[3] "Ceph as a Scalable Alternative to the Hadoop Distributed File System," by Carlos Maltzahn, Esteban Mol Ina-Estolano, Amandeep Khurana, Alex J. Nelson, Scot T A. Brandt, and Sage Weil, August, 2010: https://bit.ly/3jItHTg.

[4] "Algorithms for the 21st Century," by Stephen C. Johnson, October, 2006: https://bit.ly/3kMIvRS.

[5] "The Underground Economy: Priceless," by Rob Thomas and Jerry Martin, December, 2006: https://bit.ly/3jG0hF7.

[6] "Commodity Grid Computing with Amazon's S3 and EC2," by Simson L. Garfinkel, February, 2007: https://bit.ly/2HR2n87.

[7] "Analysis of the Storm and Nugache Trojans: P2P Is Here," by Sam Stover, Dave Dittrich, John Hernandez, and Sven Dietrich, December, 2007: https://bit.ly/35RrNuB.

[8] "Benchmarking Amazon Ec2 for High-Performance Scientific Computing," by Edward Walker, October, 2008: https://bit.ly/38hFTbh.

[9] "Practical Perl Tools: Back in Timeline," by David N. Blank-Edelman, April, 2008: https://bit.ly/3n1zjKd.

[10] "Is it Easy Being Green?" by Alva L. Crouch, June, 2009: https://bit.ly/32ltMGw.

[11] "Signals," by Rudi Van Drunen, June, 2009: https://bit.ly/36aMim1.

[12] "HDFS Scalability: The Limits to Growth," by Konstantin V. Shvachko, April, 2010: https://bit.ly/3exxeCE.

[13] "MINIX 3: Status Report and Current Research." by Andrew Tanenbaum, Raja Appuswamy, Herbert Bos, Lorenzo Cavallaro, Cristiano Giuffrida, Tomáš Hrubý, Jorrit Herder, Erik van der Kouwe, and David van Moolenbroek, June, 2010: https://bit.ly/2IvP2T4.

[14] "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation," by Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina, December, 2011: https://bit.ly/3kkSRr5

[15] "Understanding TCP Incast and Its Implications for Big Data Workloads," by Yanpei Chan, Rean Griffith, David Zats, Anthony D. Joseph, and Randy H. Katz, June, 2012: https://bit.ly/3pmmE6w.

[16] "The Saddest Moment ," by James Mickens, May, 2013: https://bit.ly/2Uft09H.

[17] "The Case of the Clumsy Kernel" by Brendan Gregg, April, 2014: https://bit.ly/3eQiGhD.

[18] "Analysis of HDFS under HBase: A Facebook Messages Case Study" by Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, June 2014: https://bit.ly/32B9Csc.

[19]"The Death of System Administration, " by Todd Underwood, April, 2014: https://bit.ly/3loTilN.

[20] "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems, by Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, February, 2015: https://bit.ly/3kpawy1.

[21] "What Bugs Live in the Cloud?: A Study of Issues in Scalable Distributed Systems" by Haryadi S. Gunawi, Thanh Do, Agung Laksono, Mingzhe Hao, Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, and Riza O. Suminto, August, 2015: https://bit.ly/38vGjuV

[22] "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems" by Jonathan Mace, Ryan Roelke, and Rodrigo Fonsec, Spring, 2016: https://bit.ly/3eONpf0.

[23] "VFP: A Virtual Switch Platform for Host SDN in the Public Cloud," by Daniel Firestone, Fall, 2017: https://bit.ly/2JUdHkB.

[24] "XDP-Programmable Data Path in the Linux Kernel," by Diptanu Gon Choudhury, Spring, 2018: https://bit.ly/32DxrPY.

[25] "Noria: A New Take on Fast Web Application Backends," "Noria: A New Take on Fast Web Application Backends" by Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris, Spring, 2019: https://bit.ly/38wzVna.

01.

"Amazing product, developed by some of the most seasoned pros in the industry."

02.

"Great products that work, easy and quick to install and provide real value."

03.

"We 🖤 our canaries."

04.

"The concept and use of Canarytokens has made me very hesitant to use credentials gained during an engagement. If the aim is to reduce the time taken for attackers, Canarytokens work well."

05.

"Their on-prem canary is one of the only things that caught me right away in post-exploitation without my knowing I was burned. Solid concept and product."

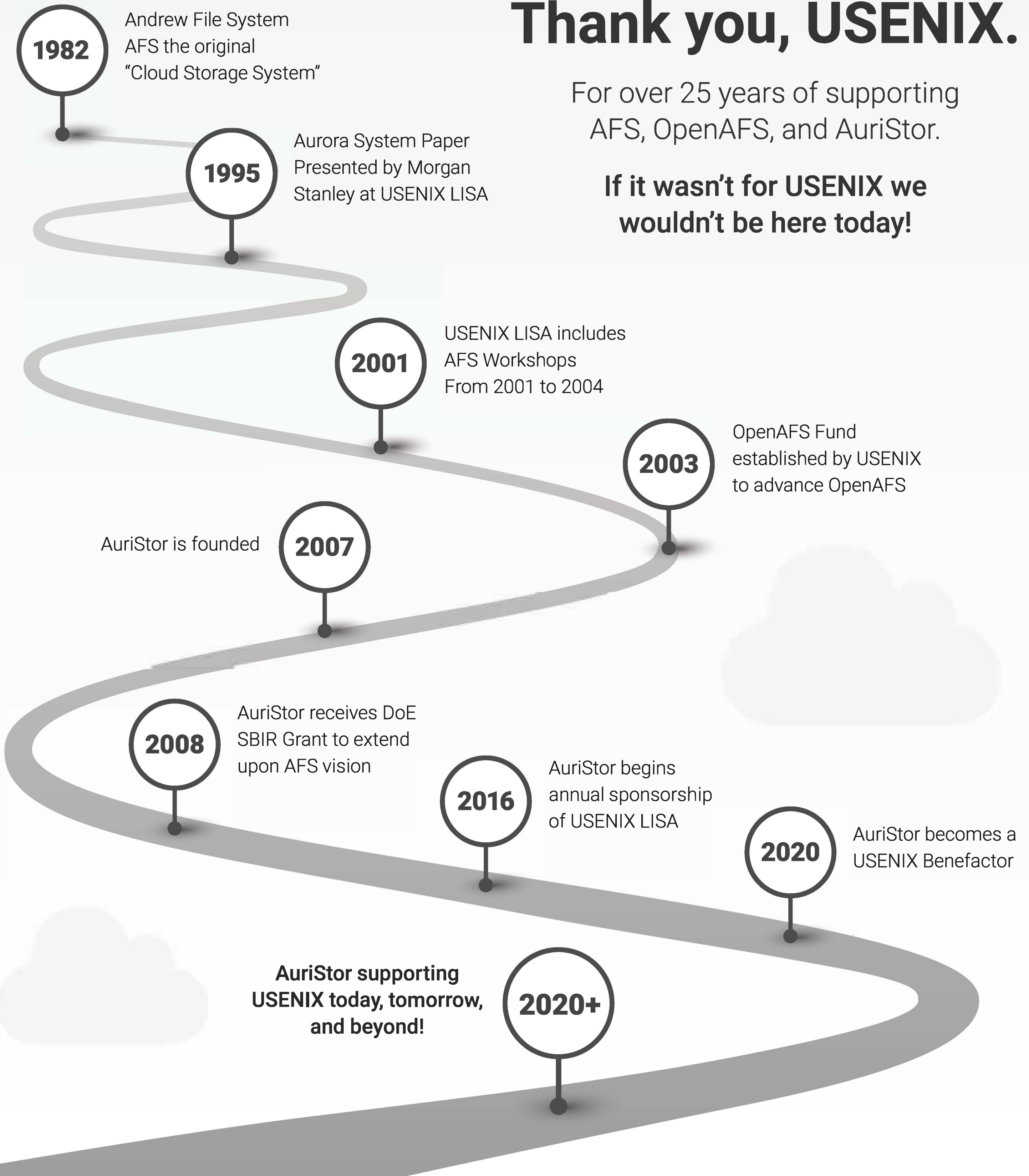
06.

"Don't think just get them."

Attackers and Defenders Finally Agree

https://canary.tools/love

THINKST CANARY



AURISTOR®

THE GLOBAL NAMESPACE FILE SYSTEM

Learn more about AuriStorFS at **auristor.com/filesystem**

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

ENIGMA

A USENIX CONFERENCE

Security and Privacy Ideas that Matter

FEATURED SPEAKERS

Kate Starbird,
University of Washington

Carmela Troncoso
Apple

Scott Shapiro,
Yale University

Jack Cable,
Security Researcher and
Student, Stanford University

The full program and registration are available now.

FEB 1–3, 2021 | **VIRTUAL EVENT**

usenix.org/enigma2021