;login:
FALL 2018 VOL. 43, NO. 3

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

SREcon18 Europe/Middle East/Africa

August 29–31, 2018, Dusseldorf, Germany
www.usenix.org/srecon18europe

OSDI '18: 13th USENIX Symposium on Operating Systems Design and Implementation

October 8–10, 2018, Carlsbad, CA, USA
Sponsored by USENIX in cooperation with ACM SIGOPS
www.usenix.org/osdi18

LISA18

October 29–31, 2018, Nashville, TN, USA
www.usenix.org/lisa18

Enigma 2019

January 28–30, 2019, Burlingame, CA, USA
www.usenix.org/enigma2019

FAST '19: 17th USENIX Conference on File and Storage Technologies

February 25–28, 2019, Boston, MA, USA
Co-located with NSDI '19
Submissions due September 26, 2018
www.usenix.org/fast19

NSDI '19: 16th USENIX Symposium on Networked Systems Design and Implementation

February 26–28, 2019, Boston, MA, USA
Co-located with FAST '19
Paper titles and abstracts due September 13, 2018 (Fall deadline)
www.usenix.org/nsdi19

SREcon19 Americas

March 25–27, 2019, Brooklyn, NY, USA

2019 USENIX Annual Technical Conference

July 10–12, 2019, Renton, WA, USA

Co-located with USENIX ATC '19

HotStorage '19: 11th USENIX Workshop on Hot Topics in Storage and File Systems
July 8–9, 2019

HotCloud '19: 11th USENIX Workshop on Hot Topics in Cloud Computing
July 8, 2019

HotEdge '19: 2nd USENIX Workshop on Hot Topics in Edge Computing
July 9, 2019

28th USENIX Security Symposium

August 14–16, 2019, Santa Clara, CA, USA

USENIX Open Access Policy

USENIX is the first computing association to offer free and open access to all of our conference proceedings and videos. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your membership fees play a major role in making this endeavor successful.

Please help us support open access. Renew your USENIX membership and ask your colleagues to join or renew today!

www.usenix.org/membership

;login:
FALL 2018 VOL. 43, NO. 3

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

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

www.usenix.org

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

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

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

©2018 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.

EDITORIAL

Musings

RIK FARROW

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

I have often mused about how the architecture of the CPUs we use influences the way our operating systems and applications are designed. A book I reviewed in this issue on the history of computing managed to cement those ideas in my head. Basically, we've been reprising time-sharing systems since the mid-60s, whereas most systems serve very different purposes today.

Along the way, I also encountered a couple of interesting data points, via pointers from friends who have been influencing me. One was Halvar Flake's CYCON 2018 talk [1], and another was about a new OS project at Google. The opinion piece by Jan Mühlberg and Jo Van Bulck that appears in this issue influenced me as well, as it describes a CPU feature that, among other things, could block the use of gadgets in return-oriented programming (ROP).

Flake explained that much of our current problems with security have to do with cheap complexity. Even though a device, like a microwave oven or intravenous drip-rate controller, only requires a PIC (Programmable Interrupt Controller), it may instead have a full-blown CPU running Windows or Linux inside it. CPUs are, by design, flexible enough to model any set of states, making them much more complex than what is needed inside a fairly simple device. Designers instead choose to use a full-blown CPU, usually with an OS not designed to be embedded, to model the states required. Vendors do this because many more people understand Windows or Linux programming than know how to program a PIC.

This isn't just a problem for ovens or routers. Let's not even discuss home routers, although the arguments for using a real OS are at least stronger in the case of routers. Dan Farmer published research, funded by DARPA, in 2013 about IPMI and BMC [2], the controllers found on most server-class motherboards. These controllers provide an over-the-network method of managing servers—e.g., rebooting them or installing updates. But the controllers are full-blown Linux systems, burned into ROM, using very old versions of Linux and exploitable software. The controllers can read or write any system memory, as well as use either a dedicated network interface or any network interface on the server, making them the obvious point for an undetectable attack using an embedded system that does no logging and cannot be patched. Ouch.

One of Flake's concluding slides had this bullet point, one I particularly liked:

- CPU-architecture and programming models are in flux for the first time since the 1980s.

I'd argue that the date is wrong, as we didn't get heavily into threaded programming until the noughts; other than that, the CPU architecture has remained very similar in rough outline to late '60s mainframes. But ignore the date and ponder Flake's implied suggestion: now is a good time for some serious changes in architecture and programming models.

Another data point is currently much more obscure. Google has a project, an OS named Fuchsia powered by the Zircon microkernel [3], based on another Google project, a microkernel named LK. Both appear to be focused for use in IoT and embedded systems. But Zircon has been designed to work on modern devices with more powerful CPUs and lots more memory than LK [4].

Zircon has a small kernel that manages processor time, memory, I/O, interrupts, and waiting/signaling. Everything else gets done in userspace, as processes. The enabling technologies that make this work well are IOMMUs and ARM SMMUs. Both the IOMMU and SMMU were designed to support virtual machines, allowing a VM to have access to, for example, a network interface queue. But these subsystems also mean that userspace programs can gain access to device memory and be able to copy data between the devices and other memory, something that has been a barrier to running system services in other microkernels.

While the Fuchsia project appears targeted at embedded systems, likely including support for Android where message passing is already used in the API, having a very small kernel reduces the immense attack space provided by modern operating systems. I've skimmed the source code for Zircon enough to see that it is a message passing system that does so by passing ownership of memory between processes and has support for both IOMMUs and SMMUs. Tinkering with the design of CPU paging systems, so that context switches don't require flushing page caches, would make this an even faster system.

I believe that Fuchsia is still in such an early phase that not a lot can be said about it, but I'm certainly excited by the concept. There are other microkernels in very wide use, such as seL4 [5], used on the radio side of hundreds of millions of cell phones. But with the potential to support Android, I think that Zircon may turn out to be something much more visible, and make devices much more secure than the usual OS used in devices like tablets and smartphones.

The Lineup

Jan Tobias Mühlberg and Jo Van Bulck sent me an opinion piece about the trouble with closed and complex CPUs. They have been working on hardware that will have an open design facilitating public verification as well as security features that will cut-off many exploit techniques. Bulck also had a paper about extracting keys from Intel SGX at ScAINet '18, part of the fallout from the exploits known as Meltdown.

While there was lots of interesting research at Annual Tech last summer, I asked two groups to write about their research since I thought both projects might have some interesting future impact, and both groups published their code.

Hu et al. write about their extension to ext4 that adds transactions, TxFS. By building upon journaling, a feature of other file systems types as well as ext4, they have added the ability to start, commit, or cancel transactions with the addition of kernel code (that is published) and a handful of function calls. I think that TxFS stands for Transaction File System, but might also be Texas File System, as the authors are at UT Austin.

Oakes et al. write about a lightweight container for use with Lambdas. AWS introduced Lambdas for serverless computing, but the problem with using these is the startup cost for loading a container complete with the necessary libraries for the servlet code. SOCK builds on previous work [6] and provides a much lighter-weight container than Docker, for example, and this article explains how and why that is done.

You can expect more articles about security in the Winter issue, as the security papers deadline came too late for me to ask authors to write for this issue. But we do have the sixth BeyondCorp article. Google's BeyondCorp focuses on securing the clients that access resources within Google, and this article reveals more about how the BC team has done this for their fleet of clients. While not everyone can expect to be able to do what Google has done, there are many useful pointers in the work they have made public in this article and the ones that have come before. For example, BC can whitelist software, something that anyone can do using policy in Windows or Macs or with a commercial product like Carbon Black (Bit9).

John Kristoff has written about his own project, a sensor net. Kristoff explains how he has set up instances that listen for probes and exploits on a handful of services, provides data via his website on the attacks he sees, and describes how you can set up your own sensor net. I found John's approach practical and interesting, and the cost is reasonable enough to be supported by small grants.

Aleatha Parker-Wood has written an excellent summary of the first Security and AI workshop, ScAINet. Applying AI techniques to security data, such as logs, is a growing area but one that is also fraught with issues that can make AI fail. The workshop examines both the benefits and the issues with using machine learning (ML).

I asked Rick Boone, an SRE at Uber, about the talk he gave during SREcon18 Americas. Boone explains how Uber does capacity prediction instead of capacity planning, using ML techniques that I recognized after my foray into ML in the Summer 2018 issue.

We have a new Python columnist. Peter Norton, co-author of several books and current SRE, takes us through his issues with how poorly Python packages that rely on shared objects work and how he'd like them to work. Norton crafts a new method for using packages that allows loading of shared objects without leaving a mess of files to clean up, relying on a relatively new Linux system call, `memfd_create()`.

David Blank-Edelman demonstrates GraphQL, an API query tool created by Facebook. GraphQL has nothing to do with graph databases, the topic of his Summer 2018 column, but instead provides an interface that is a step deeper than REST, and can return more results than REST with a single query.

Chris "Mac" McEniry demonstrates using GoLang with LDAP. Mac points out that while there are commandline tools for LDAP, having a GoLang app allows encapsulation of site-specific information.

Dave Josephsen waxes enthusiastic about data lakes. Data lakes imply large amounts of unstructured data that you don't want to spend money adding indices to, but do want to be able to query. Dave explains how this can be done using tools like Apache Parquet.

Dan Geer examines the numbers found in Mary Meeker's (of Kleiner Perkins) "Internet Trends 2018" presentation. Dan drills down and exposes the portions of the slides he found particularly interesting as representative of the types of data useful for security metrics, as well as pointing out the use of AI in content platforms.

Robert Ferrell explains that AI and bots are already in control of our online lives, so we might as well get used to it.

We have three book reviews: Mark Lamourine covers the fifth edition of the Nemeth classic and a book with proof that Agile techniques work, while I review a wonderful illustrated book covering the history of computing.

Changing CPU architecture is very hard, as companies have spent many billions of dollars tweaking their designs to produce the best performance. Flake points out that the same vendors are quite willing to trade off reliability for performance, as seen in Meltdown, an intersection between a trusted subsystem and branch prediction. We do have problems with security, ones that need to be dealt with, not only with changes to software toolchains but also to the underlying hardware. Let's hope that this is a direction that may prove fruitful soon, even if it's unlikely to prevent attacks on our critical infrastructure in the near term [7].

References

[1] T. Dullien, aka Halvar Flake, "Security, Moore's Law, and the Anomaly of Cheap Complexity," CYCON 2018: https://goo.gl/3HzQ1y.

[2] D. Farmer, "IPMI: Freight Train to Hell": http://fish2.com/ipmi/itrain.pdf.

[3] Zircon: https://fuchsia.googlesource.com/zircon/.

[4] S. De Simone, "An Early Look at Zircon, Google Fuchsia New Microkernel," *InfoQ,* April 15, 2018: https://www.infoq.com/news/2018/04/google-fuchsia-zircon-early-look; M. Bergan and M. Gurman, "Project 'Fuchsia': Google Is Quietly Working on a Successor to Android," *Bloomberg,* July 19, 2018: https://www.bloomberg.com/news/articles/2018-07-19/google-team-is-said-to-plot-android-successor-draw-skepticism.

[5] The seL4 Microkernel: https://sel4.systems/.

[6] S. Hendrickson, S. Sturdevant, E. Oakes, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," *;login:,* vol. 41, no. 4 (Winter 2016): https://www.usenix.org/publications/login/winter2016/hendrickson.

[7] D. Sanger, "Pentagon Puts Cyberwarriers on the Offensive, Increasing the Risk of Conflict," *New York Times,* June 17, 2018: https://www.nytimes.com/2018/06/17/us/politics/cyber-command-trump.html.

ENIGMA

A USENIX CONFERENCE

SECURITY AND PRIVACY IDEAS THAT MATTER

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

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

PROGRAM CO-CHAIRS

Ben Adida
Clever

Franziska Roesner,
University of Washington

The full program and registration will be available in November.

enigma.usenix.org

JAN 28–30, 2019

BURLINGAME, CA, USA

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

OPINION

Reflections on Post-Meltdown Trusted Computing

A Case for Open Security Processors

JAN TOBIAS MÜHLBERG AND JO VAN BULCK

Jan Tobias Mühlberg works as a Research Manager for embedded software security at imec-DistriNet, KU Leuven (BE). His research focuses on protected module architectures such as Sancus, software security, and formal verification and validation of software systems. Tobias is particularly interested in everything safety-critical, IoT security, embedded control systems, and low-level operating system components. He obtained a PhD from the University of York (UK) in 2009. jantobias.muehlberg@cs.kuleuven.be

Jo Van Bulck works as a PhD student at imec-DistriNet, KU Leuven (BE). His research explores hardware-based trusted computing from an integrated attack and defense perspective. He is currently the lead developer of the open-source Sancus architecture, where he is looking into processor design, compiler and operating system infrastructure, and case-study applications. More recently, his focus expanded to investigate architectural limitations and side-channel vulnerabilities in commodity Intel SGX x86 processors. Ultimately, both lines of work come together to establish a hardware-only root-of-trust. jo.vanbulck@cs.kuleuven.be

The recent wave of microarchitectural vulnerabilities in commodity hardware requires us to question our understanding of system security. We deplore that even for processor architectures and research prototypes with an explicit focus on security, open-source designs remain the exception. This article and call for action briefly surveys ongoing community efforts for developing a new generation of *open* security architectures, for which we collectively have a clear understanding of execution semantics and the resulting security implications. We advocate formal approaches to reason about the security guarantees that these architectures can provide, including the absence of microarchitectural bugs and side-channels. We consider such a principled approach essential in an age where society increasingly relies on interconnected and dependable control systems. Finally, we aim to inspire strong industrial and academic collaboration in such an engineering effort, which we believe is too monumental to be suitably addressed by a single enterprise or research community.

The security community has traditionally assessed the trustworthiness of applications at the software level by reasoning about source code as if it were executed on an idealized abstract computing platform. With the advance of hardware-level trusted computing solutions that embed a root-of-trust directly in the hardware, it even becomes possible to abstract away the underlying operating system and supporting software. However, a recent line of microarchitectural attack research, with Rowhammer, Meltdown, and Spectre being prominent examples, revealed fundamental flaws in commodity hardware. These findings range from plain design errors to intricate side-channels and triggered an array of follow-up research, effectively rendering the search for exploitable bugs in commodity processors a playground for researchers who "may have, either directly or indirectly, an economic interest in the performance of the securities of the [affected] companies" (https://amdflaws.com/), and who may or may not act in the public interest with respect to responsible disclosure guidelines. The key lesson to be learned from this wave of microarchitectural vulnerabilities and the tiresome patching process is that current processors exceed our levels of understanding and need to be subjected to independent review and assessment.

Now, having security vulnerabilities in components that are in virtually everyone's computer or phone, and components that are commonly relied upon to build critical infrastructure—think of communications networks, data centers, and cloud systems up to the power grid and hospital equipment—is certainly worrisome. Yet, considering that computing platforms are designed by humans, we have to face that security vulnerabilities are to some extent inevitable. As a community, we must therefore welcome research efforts that enhance our understanding of the attack surface and the limitations of today's commodity computing infrastructure, and that responsibly handle security-related findings to swiftly patch existing systems and avoid introducing similar errors in the future.

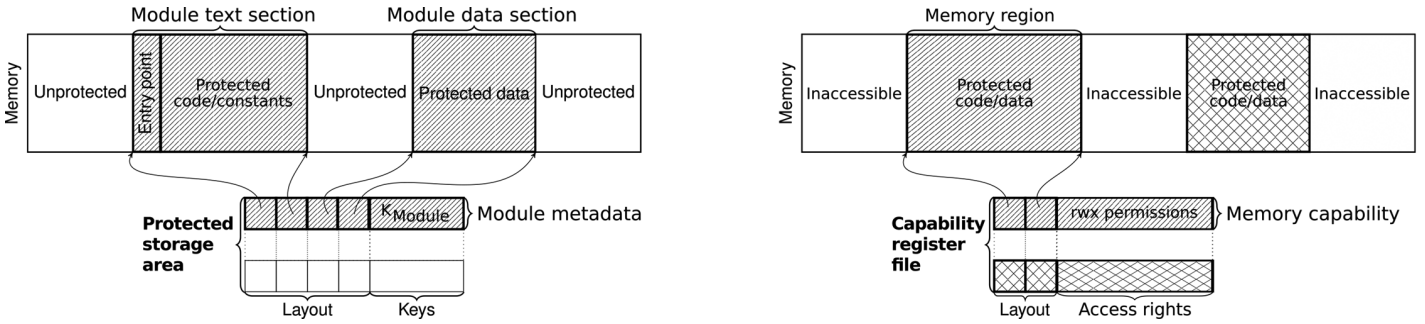


Figure 1: Fine-grained intra-address space isolation paradigms. **Left**: Sancus [5] uses the current value of the CPU's program counter to distinguish a protected module (hatched) from untrusted code. The module's *data* memory can only be accessed when executing in the corresponding *text section*, which can only be entered from a single predefined *entry point*. Software attestation is realized through a protected hardware storage area for metadata and cryptographic keys. **Right**: CHERI [6] relies on a dedicated CPU register file for unforgeable *memory capabilities* that provide read/write/execute permissions for individual memory regions (hatched). Flexible application protection domains are defined by deriving more restrictive capabilities at runtime.

Reverse Engineering Is Insufficient

Conducting this kind of research is far from easy, however, as prevalent business models of the industry hamper such efforts. That is, today's computing platforms are not designed to be analyzed, and intellectual property concerns commonly restrict the freedom of end users (i.e., companies, governments, researchers, the general public) to access hardware design internals, let alone source code. We deplore that even for processor architectures and research prototypes with an explicit focus on security, open-source designs remain the exception [1]. This situation leaves researchers at publicly funded institutions with no choice but to invest enormous reverse-engineering efforts before being able to fully understand the advertised security features, identify limitations and vulnerabilities, or formally prove security properties.

Great examples of such efforts in third-party reverse engineering include the Cambridge formal models [2] of the ARM instruction set architecture, or the fact that the most insightful security analysis of Intel's SGX trusted computing platform comes from MIT researchers [3]. Yet, much of these efforts need to be repeated for every academic publication that models, investigates, or reports on vulnerabilities in closed-source commercial products.

Of course, we acknowledge the importance of intellectual property protection for market shares and revenues in the commercial sector. We also acknowledge the contributions of industry initiatives that integrate strong security features in commodity hardware. Important achievements include secure virtualization extensions, TPM co-processors, and enclaved execution environments such as Intel SGX, ARM TrustZone, and AMD SEV. However, we strongly believe that it is close to impossible for vendors and producers to guarantee the absence of certain classes of critical vulnerabilities in their highly complex products [4].

Bridging the Trust Gap

We therefore argue that processors in a post-Meltdown world can no longer be considered opaque black boxes that implement an instruction set abstraction. Hardware vendors must not attempt to hide microarchitectural execution semantics but instead allow these details to become part of the specification, so that compilers and operating systems can fully take them into account. When looking at the development of open processors, we welcome a number of such initiatives. For example, a range of free and open-source CPU cores are listed on opencores.org. The RISC-V ISA (https://riscv.org/) enables processor innovation through open standard collaboration, with fully open and industry-competitive RISC-V implementations available.

What we need beyond openness, however, are CPUs with real support for security. We have not fundamentally reconsidered the concepts of hierarchical protection rings and virtual memory since the introduction of the Multics mainframe operating system in 1969. Only very recently have industry and academia developed alternative trusted computing solutions to isolate small software components without relying on privileged system software. As a constructive next step to bridge the trust gap between hardware and software, we envisage enhanced processor designs that allow applications to communicate fine-grained security constraints into the underlying CPU architecture. This would allow microarchitects to apply suitable optimizations while preventing unintended side-channel leakage across protection domains.

Two state-of-the-art secure processor prototypes with an explicit focus on openness are CHERI and Sancus. The CHERI [6] research project explores MIPS extensions for a fine-grained memory capability model. Our own Sancus [5] processor implements open-source (https://distrinet.cs.kuleuven.be/software/sancus/) trusted computing primitives for lightweight embedded applications, such as automotive control systems [7]. Figure

1 compares the CHERI and Sancus approaches to intra-address space isolation. Compared to the legacy Multics virtual memory paradigm, both offer a richer architectural expression of protection domain boundaries. Regarding Spectre- and Meltdown-type speculative execution vulnerabilities, we follow the argument of the CHERI authors [8]. A more explicit architectural notion of protection domains that can be propagated into the microarchitecture has the potential to enable true hardware-software co-design, where the security requirements of the application constrain microarchitectural optimizations.

Importantly, with open security architectures as a prerequisite, dependable hardware-software co-designs can be vetted from a formal perspective. Promising research results include machine-checkable proofs for both functional correctness and high-level integrity and confidentiality security properties [9], or the application of proven-correct analysis to verify the absence of digital side-channels in low-level assembly code. Enhanced hardware description languages such as SecVerilog [10] enable static information flow analysis at hardware design time, which leads to a notion of contractual execution semantics that compilers and applications can rely upon. Using this approach, performant processors can be built, for which the absence of timing side-channels and other undesired information leakage is statically proven. With such trustworthy CPUs as a basis, an especially promising avenue is to apply established techniques in the field of software engineering to develop dependable and highly secure trusted execution environments.

A Call for Action

Overall, we observe that vulnerabilities in software persist, but the research community has a good understanding of how to address these with established software engineering methods, modern programming languages, and advanced security features in modern processors. However, we also observe that there is a new class of widespread vulnerabilities in commodity hardware ranging from plain design errors to intricate side-channels. These vulnerabilities hamper efforts to improve security on all layers of a system's hardware and software stack. In today's world, where advanced societies increasingly rely on the security and reliability of critical infrastructure in domains such as the power grid, communication, transportation, and medical infrastructure, these vulnerabilities may have disastrous consequences for a great many people, whether exploited through malicious intent or triggered by accident.

We outlined one way to address these threats by relying on open designs and formal methods to develop a new class of secure and dependable processors. As a security community, we will benefit from such an effort by obtaining a shared and clear understanding of the protection mechanisms provided by these processors and of how software systems can be built to make proper use of hardware-level security primitives. It would then become unnecessary for researchers to painstakingly reverse-engineer microarchitectural design details as a prerequisite for exploring new attack techniques or alternative modeling approaches. By reaching the required level of performance while also emphasizing maintainability and rigorous availability guarantees, the envisaged class of processors would form an ideal basis for the design of the networked safety-critical control systems of the future. We believe that architectures such as RISC-V, CHERI, and Sancus present promising starting points for this highly necessary work, and we would like to inspire and invite collaboration in this field.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven. Jo Van Bulck is supported by a doctoral grant of the Research Foundation—Flanders (FWO).

References

[1] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-Based Trusted Computing Architectures for Isolation and Attestation," *IEEE Transactions on Computers*, vol. 67, no. 3 (March 2018), pp. 361–374: https://www.esat.kuleuven.be/cosic/publications/article-2750.pdf.

[2] A. Fox and M. O. Myreen, "A Trustworthy Monadic Formalization of the Armv7 Instruction Set Architecture," in *International Conference on Interactive Theorem Proving* (Springer, 2010), pp. 243–258: https://www.cl.cam.ac.uk/~mom22/itp10-armv7.pdf.

[3] V. Costan and S. Devadas, *Intel SGX Explained* (IACR, 2016), p. 86: https://eprint.iacr.org/2016/086.pdf.

[4] A. Baumann, "Hardware Is the New Software," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (ACM, 2017), pp. 132–137: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/05/baumann-hotos17.pdf.

[5] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20 (August 2017), pp. 1–33: http://www.beetzsee.de/leuven/2016-acmtops-sancus/paper.pdf.

[6] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3 (June 2014), pp. 457–468: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201406-isca2014-cheri.pdf.

[7] J. Van Bulck, J. T. Mühlberg, and F. Piessens, "VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17),* ACM, 2017, pp. 225–237: https://distrinet.cs.kuleuven.be/software/sancus/publications/acsac17.pdf.

[8] R. N. Watson, J. Woodruff, M. Roe, S. W. Moore, and P. G. Neumann, "Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks," University of Cambridge, Computer Laboratory, Technical Report no. 916, 2018: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf.

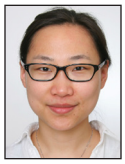
[9] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17),* ACM, 2017, pp. 287–305: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/10/komodo.pdf.

[10] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," *ACM SIGPLAN Notices*, vol. 50, no. 4 (May 2015), pp. 503–516: http://www.cse.psu.edu/~dbz5017/pub/asplos15.pdf.

TxFS

Leveraging File-System Crash Consistency to Provide ACID Transactions

YIGE HU, ZHITING ZHU, IAN NEAL, YOUNGJIN KWON, TIANYU CHENG, VIJAY CHIDAMBARAM, AND EMMETT WITCHEL



Yige Hu is a PhD student at the University of Texas at Austin, under the supervision of Professor Emmett Witchel. Her research interests include operating systems, storage, and heterogeneous architecture. yige@cs.utexas.edu

Zhiting Zhu is a PhD student at the University of Texas at Austin, working with Emmett Witchel. He is interested in operating systems and security. zhitingz@cs.utexas.edu

Ian Neal received his computer science and electrical engineering degrees from the University of Texas at Austin in 2018. His undergraduate honors thesis was on transaction file systems, and he has also worked on other storage systems in non-volatile RAM. He will be starting his PhD program in the fall of 2018 at the University of Michigan at Ann Arbor. ian.glen.neal@utexas.edu

Youngjin Kwon is a PhD candidate at the University of Texas at Austin under the supervision of Professors Emmett Witchel and Simon Peter. His research interests lie in operating systems, including file systems, emerging storage and memory technologies, system support for security, and virtualization. His research has been recognized by VMware, and he contributed an initial version of his research work to VMware commercial hypervisor. yjkwon@cs.utexas.edu

We introduce TxFS, a novel transactional file system that builds upon a file system's atomic-update mechanism such as journaling. Although prior work has explored a number of transactional file systems, TxFS has a unique set of properties: a simple API, portability across different hardware, high performance, low complexity (by building on the journal), and full ACID transactions. We port SQLite and Git to use TxFS, and experimentally show that TxFS provides strong crash consistency while providing equal or better performance.

Modern applications store persistent state across multiple files. Some applications split their state among embedded databases, key-value stores, and file systems. Such applications need to ensure that their data is not corrupted or lost in the event of a crash. Unfortunately, existing techniques for crash consistency, such as logging or using atomic rename, result in complex protocols and subtle bugs.

Transactions present an intuitive way to atomically update persistent state. Unfortunately, building transactional systems is complex and error-prone, leading us to develop a novel approach to building a transactional file system. We take advantage of a mature, well-tested piece of functionality in the operating system: the file-system journal, which is used to ensure atomic updates to the internal state of the file system. We use the atomicity and durability provided by journal transactions and leverage it to build ACID transactions available to userspace transactions. Our approach greatly reduces the development effort and complexity for building a transactional file system.

We introduce TxFS [4], a transactional file system that builds on the ext4 file system's journaling mechanism. We designed TxFS to be practical to implement and easy to use. TxFS has a unique set of properties. It has a small implementation (5200 lines of code) by building on the journal. It provides high performance, unlike various solutions that built a transactional file system over a userspace database [3, 12]. It has a simple API (just wrap code in `fs_tx_begin()` and `fs_tx_commit()`) compared to solutions like Valor [10] or TxF [8], which require multiple system calls per transaction and can require the developer to understand implementation details like logging. It provides all ACID guarantees, unlike solutions such as CFS [5] and AdvFS [11], which only offer some of the guarantees, and it also provides transactions at the file level instead of at the block level, unlike Isotope [9], making several optimizations easier to implement. Finally, TxFS does not depend on specific properties of the underlying storage, unlike solutions such as MARS [2] and TxFlash [7].

We find that file system transactions lead naturally to a number of seemingly unrelated file-system optimizations. For example, one of the core techniques from our earlier work, separating ordering from durability [1], is easily accomplished in TxFS. Similarly, we find TxFS transactions allow us to identify and eliminate redundant application I/O where temporary files or logs are used to atomically update a file; when the sequence is simply enclosed in a transaction and without any other changes, TxFS atomically updates the file, maintaining functionality while eliminating the I/O to logs or temporary files, provided that the temporary files and logs are deleted inside the transaction. As a result, TxFS improves

Tianyu Cheng received an MS in computer science with high honors from the University of Texas at Austin in 2017. He is interested in a wide range of topics, including computer architecture and graphics. He is currently working on GPU architecture validation with Apple Inc. tianyu.cheng@utexas.edu

Vijay Chidambaram is an Assistant Professor in the Computer Science Department at the University of Texas Austin. He works on distributed systems, operating systems, and storage. His work has resulted in patent applications by VMware, Samsung, and Microsoft. His research has won the SIGOPS Dennis M. Ritchie Dissertation Award in 2016, Best Paper Awards at FAST 2017 and 2018, and a Best Poster at ApSys 2017. He was awarded the Microsoft Research Fellowship in 2014 and the University of Wisconsin-Madison Alumni Scholarship in 2009. vijay@cs.utexas.edu

Emmett Witchel is a Professor in Computer Science at the University of Texas at Austin. He received his doctorate from MIT in 2004. He and his group are interested in operating systems, security, performance, and concurrency. witchel@cs.utexas.edu

performance while simultaneously providing better crash-consistency semantics: a crash does not leave messy temporary files or logs that need to be cleaned up.

To demonstrate the power and ease of use of TxFS transactions, we modify SQLite and Git to incorporate TxFS transactions. We show that when using TxFS transactions, SQLite performance on the TPC-C benchmark improves by 1.6x, and a microbenchmark that mimics Android Mail obtains 2.3x better throughput. Using TxFS transactions greatly simplifies Git's code while providing crash consistency without performance overhead. Thus, TxFS transactions increase performance, reduce complexity, and provide crash consistency.

We make the following contributions:

- We present the design and implementation of TxFS, a transactional file system for modern applications built by leveraging the file-system journal (see "TxFS Design and Implementation," below). We have made TxFS publicly available at https://github.com/ut-osa/txfs.
- We show that existing file system optimizations, such as separating ordering from durability, can be effectively implemented for TxFS transactions (see "Accelerating Programming Idioms with TxFS," below).
- We show that real applications can be easily modified to use TxFS, resulting in better crash semantics and significantly increased performance (see "Evaluation," below).

Why Use File-System Transactions?

We describe the complexity of current protocols used by applications to update persistent state and discuss a few case studies. We then describe the optimizations enabled by file-system transactions.

How Applications Update State Today

Given that applications today do not have access to transactions, how do they consistently update state to multiple storage locations? Even if the system crashes or power fails, applications need to maintain invariants across state in different files (e.g., an image file should match the thumbnail in a picture gallery). Applications achieve this by using ad hoc protocols that are complex and error-prone [6].

```
open(/dir/tmp)
write(/dir/tmp)
fsync(/dir/tmp)
fsync(/dir)
rename(/dir/tmp, /dir/orig)
fsync(/dir/)
```

(a) Atomic Update via Rename

```
open(/dir/log)
write(/dir/log)
fsync(/dir/log)
fsync(/dir/)
write(/dir/orig)
fsync(/dir/orig)
unlink(/dir/log)
fsync(/dir/)
```

(b) Atomic Update via Logging

```
// Write attachment
open(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)

// Writing SQLite Database
open(/dir/journal)
write(/dir/journal)
fsync(/dir/journal)
fsync(/dir/)
write(/dir/db)
fsync(/dir/db)
unlink(/dir/journal)
fsync(/dir/)
```

(c) Atomically adding a email message with attachments in Android Mail

Figure 1: Different protocols used by applications to make consistent updates to persistent data

In this section, we show how difficult it is to implement seemingly simple protocols for consistent updates to storage. There are many details that are often overlooked, like the persistence of directory contents. With current storage technologies, these protocols must sacrifice performance to be correct because there is no efficient way to order storage updates. Currently, applications use the `fsync()` system call to order updates to storage [1]; since `fsync()` forces data to be durable, the latency of a `fsync()` call varies from a few milliseconds to several seconds. As a result, applications do not call `fsync()` at all the places in the update protocol where it is necessary, leading to severe data loss and corruption [6].

We now describe two common techniques used by applications to consistently update storage, illustrated in Figure 1.

Atomic rename. The atomic rename approach is widely used by editors, such as Emacs and Vim, and by GNOME applications that need to atomically update dot configuration files. Protocol (a) illustrates the approach: the application writes new data to a temporary file, persists it with an `fsync()` call, updates the parent directory with another `fsync()` call, and then renames the temporary file over the original file, effectively causing the directory entry of the original file to point to the temporary file instead. Finally, to ensure that the original file has been unlinked and deleted properly, the application calls `fsync()` on the parent directory.

Logging. Protocol (b) shows another popular technique for atomic updates, logging. In the write-ahead version of logging, the log file is written with new contents, and both the log file and the parent directory (with the new pointer to the log file) are persisted. The application then updates and persists the original file; the parent directory does not change during this step. Finally, the log is unlinked, and the parent directory is persisted.

The situation becomes more complex when applications store state across multiple files. Protocol (c) illustrates how the Android Mail application adds a new email with an attachment. The attachment is stored on the file system, while the email message (along with metadata) is stored in the database (which for SQLite, also resides on the file system). Since the database has a pointer to the attachment (i.e., a file name), the attachment must be persisted first. Persisting the attachment requires two `fsync()` calls (to the file and its containing directory) [6]. It then follows a protocol similar to protocol (b). Android mail uses six `fsync()` calls to persist a single email with an attachment.

Removing `fsync()` calls in any of the presented protocols will lead to data loss or corruption. For instance, in protocol (b), if the parent directory is not persisted with an `fsync()` call, the log file may disappear after a crash. If the application crashes in the middle of updating the original file, it will not be able to recover using the log. Many application developers avoid `fsync()` calls due to the resulting decrease in performance, leading to severe bugs that cause loss of data.

In summary, safe update protocols for stable storage are complex and low performance. System support for file-system transactions will enable high performance for these applications.

Application Case Studies

We present two examples of applications (in addition to the previously described Android Mail) that struggle to obtain crash consistency using primitives available today. Several applications store data across the file system, key-value stores, and embedded databases such as SQLite. While all of this data ultimately resides in the file system, their APIs and performance constraints are different, and consistently updating state across these systems is complex and error-prone.

Apple iWork and iLife. Analysis of the storage behavior of Apple's home-user desktop applications finds that applications use a combination of the file system, key-value stores, and SQLite to store data. iTunes uses SQLite to store metadata separately from songs similar to the Android Mail application. Apple's Pages application uses a combination of SQLite and key-value stores for user preferences and other metadata (two SQLite databases and 128 .plist key-value store files). Similar to Android Mail, these applications use `fsync()` to order updates correctly.

Version control systems. Git is a widely used version control system. The `git commit` command requires two file-system operations to be atomic: a file append (logs/HEAD) and a file rename (to a lock file). Failure to achieve atomicity results in data loss and a corrupted repository [6].

For these applications, transactional support would lead directly to more understandable and more efficient idioms (rather than approaches like atomic rename used today). It is difficult for a user-level program to efficiently provide crash-consistent transactional updates using the POSIX file-system interface.

Optimizations Enabled by File-System Transactions

A transactional file-system interface enables a number of interesting file-system optimizations:

Eliminate temporary durable files. A number of applications such as Vim, Emacs, Git, and LevelDB provide reasonable crash semantics using the atomic rename approach. But these applications can simply enclose writes inside a transaction and avoid making an entire copy of the file. For large files, the difference in performance can be significant. Additionally, transactions eliminate the clutter of temporary files orphaned by a crash.

Group commit. Transactions can buffer file-system updates in memory and submit updates to storage as a batch. Batching

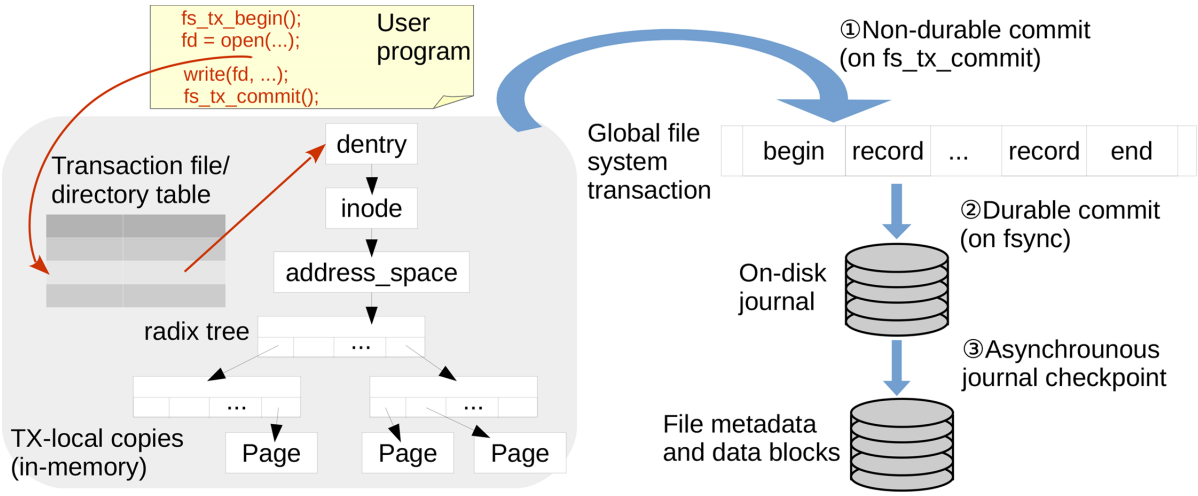


Figure 2: TxFS relies on ext4's own journal for atomic updates and maintains local copies of in-memory data structures, such as inodes, directory entries, and pages, to provide isolation guarantees. At commit time, the local operations are made global and durable.

updates enables efficient allocation of file-system data structures and better device-level scheduling. Without user-provided transaction boundaries, the file system provides uniform, best-effort persistence for all updates.

Eliminate redundant I/O *within* transactions. Workloads often contain redundancy; for example, files are often updated several times at the same offset, or a file is created, written, read, and unlinked. Because the entire transaction is visible to the file system at commit time, it can eliminate redundant work.

Consolidate I/O *across* transactions. Transactions often update data written by prior transactions. When a workload anticipates data in its transaction will be updated by another transaction shortly, it can prioritize throughput over latency. Committing a transaction with a special flag allows the system to delay a transaction commit, anticipating that the data will be overwritten, and then it can be persisted once instead of twice. Optimizing multiple transactions, especially from different applications, is best done by the operating system, not by an individual application.

Separate ordering from durability. When ending a transaction, the programmer can specify whether the transaction should commit durably. If so, the call blocks until all updates specified by the transaction have been written to a persistent journal. If we commit non-durable transaction A and then start non-durable transaction B, then A is ordered before B, but neither is durable. A subsequent transaction (e.g., C) can specify that it and all previous transactions should be made durable. Thus, we can use transactions to gain the benefit of splitting sync into ordering sync (osync) and durability sync (dsync) [1].

TxFS Design and Implementation

TxFS avoids the pitfalls from earlier transactional file systems. It has a simple API, provides complete ACID guarantees, does not depend on specific hardware, and takes advantage of the file-system journal and how the kernel is implemented to achieve a small implementation.

API

A simple API was one of the key goals of TxFS. Thus, TxFS provides developers with only three system calls: `fs_tx_begin()`, which begins a transaction; `fs_tx_commit()`, which ends a transaction and attempts to commit it; and `fs_tx_abort()`, which discards all file-system updates contained in the current transaction. On commit, all file-system updates in the TxFS transaction are persisted in an atomic fashion—after a crash, users see all of the transaction updates or none of them. This API significantly simplifies application code and provides clean crash semantics, since temporary files or partially written logs will not need to be cleaned up after a crash.

`fs_tx_commit()` returns a value indicating whether the transaction was committed successfully, or if it failed, why it failed. A transaction can fail for several reasons, including a conflict with another transaction or not enough storage resources. Depending on the error code, the application can choose to retry the transaction.

A user can surround any sequence of file-system-related system calls with `fs_tx_begin()` and `fs_tx_commit()`, and the system will execute those system calls in a single transaction. This interface is easy for programmers to use and makes it simple to incrementally deploy file-system transactions into existing applications. In contrast, some transactional file systems, such as Window's TxF and Valor, have far more complex, difficult-to-use interfaces.

TxFS isolates file-system updates only. The application is still responsible for synchronizing access to its own user-level data structures. A transactional file system is not intended to be an application's sole concurrency control mechanism; it only coordinates file-system updates that are difficult to coordinate without transactions.

Atomicity and Durability

Most modern Linux file systems have an internal mechanism for atomically updating multiple blocks on storage. These mechanisms are crucial for maintaining file-system crash consistency, and thus have well-tested and mature implementations. TxFS takes advantage of these mechanisms to obtain three of the ACID properties: atomicity, consistency, and durability.

TxFS builds upon the ext4 file system's journal. The journal provides the guarantee that each journal transaction is applied to the file system in an atomic fashion. TxFS can be built upon any file system with a mechanism for atomic updates such as copy-on-write. TxFS guarantees atomicity by ensuring that all operations in a user transaction are added to a single local journal transaction, and it persists the journal transaction to ensure durability.

Isolation and Conflict Detection

Although the ext4 journal provides atomicity and durability, it does not provide isolation. To provide isolation, TxFS has to ensure that all operations performed inside a transaction are not visible to other transactions or the rest of the system until commit time. Adding isolation for file-system data structures in the Linux kernel is challenging because a large number of functions all over the kernel modify file-system data structures without using a common interface. In TxFS, we tailor our approach to isolation for each data structure to simplify the implementation.

Split file-system functions. System calls such as `write()` and `open()` execute file-system functions that often result in allocation of file-system resources such as data blocks and inodes. TxFS splits such functions into two parts: file-system allocation and in-memory structures. TxFS moves file-system allocation to the commit point. In-memory changes execute as part of the system call, and they are kept private to the transaction.

Transaction-private copies. TxFS makes transaction-private copies of all kernel data structures modified during the transaction. File-system-related system calls inside a transaction operate on these private copies, allowing transactions to read their own writes. For example, directory entries updated by the transaction are modified to point to a local inode that maintains a local radix tree with locally modified pages. In case of abort, these private copies are discarded; in case of commit, these private copies are carefully applied to the global state of the file system in an atomic fashion.

Workload	FS	TX
Create/unlink/sync	37.35s	0.28s (133x)
Logging	5.09s	4.23s (1.20x)
Ordering work	2.86it/s	3.96it/s (1.38x)

Table 1: Programming idioms sped up by TxFS transactions. Performance is measured in seconds (s) and iterations per second (it/s). Speedups for the transaction case are reported in parentheses.

Two-phase commit. TxFS transactions are committed using a two-phase commit protocol. TxFS first obtains a lock on all relevant file-system data structures using a total order that follows the existing file-system conventions, so that deadlocks are avoided.

Conflict detection. Conflict detection is a key part of providing isolation. Since allocation-related structures such as bitmaps are not modified until commit time, they cannot be modified by multiple transactions at the same time and do not give rise to conflicts; as a result, TxFS avoids false conflicts involving global allocation structures.

Conflict detection is challenging because many file-system data structures are modified all over the Linux kernel without a standard interface. TxFS eagerly detects conflicts on data pages, taking advantage of the structured kernel API for page management. It lazily detects conflicts on directory entries and file metadata structures, quickly detecting at commit time whether these structures have been updated.

Summary. Figure 2 shows how TxFS uses ext4's journal for atomically updating operations inside a transaction and maintaining local state to provide isolation guarantees. File operations inside a TxFS transaction are redirected to the transaction's locally copied data structures, hence they do not affect the file system's global state, while being observable by subsequent operations in the same transaction. Only after a TxFS transaction finishes its commit (by calling `fs_tx_commit()`) will its modifications be globally visible.

Limitations

TxFS has two main limitations. First, the maximum size of a TxFS transaction is limited to one-fourth the size of the journal (the maximum journal transaction size allowed by ext4). We note that the journal can be configured to be as large as required. Multi-gigabyte journals are common today. Second, although parallel transactions can proceed with ACID guarantees, each transaction can only contain operations from a single process. Transactions spanning multiple processes are future work.

Accelerating Programming Idioms with TxFS

We explore a number of programming idioms where a transactional API can improve performance because transactions

Experiment	TxFS Benefit	Speed
Single-threaded SQLite	Faster I/O path, less sync	1.31x
TPC-C	Faster I/O path, less sync	1.61x
Android Mail	Cross abstraction tx	2.31x
Git	Better crash semantics	1.00x

Table 2: The table summarizes the micro- and macro-benchmarks used to evaluate TxFS and the speedup obtained in each experiment.

provide the file system a sequence of operations that can be optimized as a group. Whole transaction optimization can result in dramatic performance gains because the file system can eliminate temporary durable writes (such as the creation, use, and deletion of a log file). In some cases, we show that benefits previously obtained by new interfaces (such as osync [1]) can be obtained easily with transactions.

Eliminating File Creation

When an application creates a temporary file, syncs it, uses it, and then unlinks it (e.g., logging shown in Figure 1b), enclosing the entire sequence in a transaction allows the file system to optimize out the file creation and all writes while maintaining crash consistency.

The create/unlink/sync workload spawns six threads (one per core) where each thread repeatedly creates a file, unlinks it, and syncs the parent directory. Table 1 shows that placing the operation within a transaction increases performance by 133x because the transaction completely eliminates the workload's I/O. While this test is an extreme case, we next look at using transactions to automatically convert a logging protocol into a more efficient update protocol.

Eliminating Logging I/O

Figure 1b shows the logging idiom used by modern applications to achieve crash consistency. Enclosing the entire protocol within a transaction allows the file system to transparently optimize this protocol into a more efficient direct modification. During a TxFS transaction, all sync-family calls are functional NOPs. Because the log file is created and deleted within the transaction, it does not need to be made persistent on transaction commit. Eliminating the persistence of the log file greatly reduces the amount of user data but also file system metadata (e.g., block and inode bitmaps) that must be persisted.

Table 1 shows execution time for a microbenchmark that writes and syncs a log, and a version that encloses the entire protocol in a single TxFS transaction. Enclosing the logging protocol within a transaction increases performance by 20% and cuts the amount of I/O performed in half because the log file is never persisted. Rewriting the code increases performance by 55% (3.28 seconds, not shown in the table). In this case, getting the most performance out of transactions requires rewriting the code to eliminate work that transactions make redundant. But even without a programmer rewrite, just adding two lines of code to wrap a protocol in a transaction achieves 47% of the performance of doing a complete rewrite.

Optimizing SQLite logging with TxFS. Just enclosing the logging activity of SQLite in its default mode (Rollback) within a transaction increases performance for updates by 14%. Modifying the code to eliminate the logging work that transactions make redundant increases the performance for updates to 31%, in part by reducing the number of system calls 2.5x.

Separating Ordering and Durability

Table 1 shows throughput for a workload that creates three 10 MB files and then updates 10 MB of a separate 40 MB file. The user would like to create the files first, then update the data file. This type of ordering constraint often occurs in systems like Git that create log files and other files that hold intermediate state.

The first version uses `fsync()` to order the operations, while the second uses transactions that allow the first three file create operations to execute in any order, but they are all serialized behind the final data update transaction using flags to `fs_tx_begin()` and `fs_tx_commit()`. The transactional approach has 38% higher throughput because the ordering constraints are decoupled from the persistence constraints. Our previous work that first distinguished ordering from persistence required adding modified sync system calls [1], but TxFS can achieve the same result with transactions.

Evaluation

We evaluate the performance and durability guarantees of TxFS on a variety of microbenchmarks and real workloads. The microbenchmarks help point out how TxFS achieves specific design goals. The larger benchmarks validate that transactions provide stronger crash semantics and improved performance for a variety of large applications with minimal porting effort. For example, we modified SQLite to use TxFS transactions and measured its performance improvement. Table 2 presents a summary of the different experiments used to evaluate TxFS and the speedup obtained in each experiment. In the Git experiment, TxFS provides strong crash-consistency guarantees (no need for post-crash manual Git recovery) without degrading performance. Note that if not explicitly mentioned, all our baselines run on ext4 in its default ordered journaling mode. For more details please refer to the original publication [4].

Conclusion

We present TxFS, a transactional file system built with lower development effort than previous systems by leveraging the file-system journal. TxFS is easy to develop, is easy to use, and does

not have significant overhead for transactions. We show that using TxFS transactions increases performance significantly for a number of different workloads.

Transactional file systems have not been successful for a variety of reasons. TxFS shows that it is possible to avoid the mistakes of the past and build a transactional file system with low complexity. We believe that file-system transactions, given their power and flexibility, should be examined again by file-system researchers and developers. Adopting a transactional interface would allow us to borrow decades of research on optimizations from the database community while greatly simplifying the development of crash-consistent applications.

References

[1] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic Crash Consistency," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pp. 228–243: http://research.cs.wisc.edu/adsl/Publications/optfs-sosp13.pdf.

[2] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From ARIES to MARS: Transaction Support for Next-Generation, Solid-State Drives," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pp. 197–212: https://cseweb.ucsd.edu/~swanson/papers/SOSP2013-MARS.pdf.

[3] N. H. Gehani, H. V. Jagadish, and W. D. Roome, "OdeFS: A File System Interface to an Object-Oriented Database," in *Proceedings of the 20th Very Large Databases Conference (VLDB 1994)*, pp. 249–260: http://www.vldb.org/conf/1994/P249.pdf.

[4] Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel, "TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions," 2018 USENIX Annual Technical Conference (USENIX ATC '18).

[5] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, "Lightweight Application-Level Crash Consistency on Transactional Flash Storage," in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pp. 221–234: https://www.usenix.org/system/files/conference/atc15/atc15-paper-min.pdf.

[6] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, pp. 433–448: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-pillai.pdf.

[7] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional Flash," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 147–160: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/prabhakaran/prabhakaran.pdf.

[8] M. E. Russinovich, D. A. Solomon, and J. Allchin, *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, 4th edition (Microsoft Press, 2005).

[9] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon, "Isotope: Transactional Isolation for Block Storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pp. 23–37: https://www.usenix.org/system/files/conference/fast16/fast16-papers-shin.pdf.

[10] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, "Enabling Transactional File Access via Lightweight Kernel Extensions," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)*, pp. 29–42: https://www.usenix.org/legacy/event/fast09/tech/full_papers/spillane/spillane.pdf.

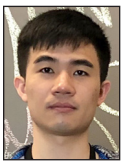
[11] R. Verma, A. A. Mendez, S. Park, S. S. Mannarswamy, T. Kelly, and C. B. Morrey III, "Failure-Atomic Updates of Application Data in a Linux File System," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pp. 203–211: https://www.usenix.org/system/files/conference/fast15/fast15-paper-verma.pdf.

[12] C. P. Wright, R. Spillane, G. Sivathanu, E. Zadok, "Extending ACID Semantics to the File System," *ACM Transactions on Storage (TOS)*, vol. 3, no. 2 (May 2007), pp. 1–40: http://www.fsl.cs.stonybrook.edu/docs/amino-tos06/amino.pdf.

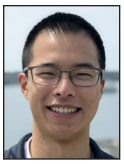
SOCK: Serverless-Optimized Containers

EDWARD OAKES, LEON YANG, DENNIS ZHOU, KEVIN HOUCK, TYLER CARAZA-HARTER, ANDREA C. ARPACI-DUSSEAU, AND REMZI H. ARPACI-DUSSEAU

Edward Oakes holds a BS from the University of Wisconsin-Madison in computer science and is an incoming PhD student at the University of California-Berkeley. As an undergraduate, he was advised by professors Andrea and Remzi Arpaci-Dusseau and is a primary contributor to the OpenLambda project. oakes@cs.wisc.edu



Leon Yang received his bachelor's degree from the University of Wisconsin-Madison, where he is currently pursuing a master's degree in computer science. Professor Remzi Arpaci-Dusseau is his adviser. He will be working as a software engineering intern at Facebook this summer and is a contributor to the OpenLambda project. gyang48@wisc.edu



Dennis Zhou is a recent MS graduate from the University of Wisconsin-Madison. He was advised by Andrea and Remzi Arpaci-Dusseau and worked on OpenLambda at the Microsoft Gray Systems Lab. This summer, he is joining Facebook as a Software Engineer working on Linux. dennisszhou@gmail.com

Kevin Houck is a recent Bachelor of Science graduate in computer science at the University of Wisconsin-Madison. He was advised by Professor Aditya Akella and has previously contributed to OpenLambda. This summer he will be continuing ongoing research in serverless computing and this fall will join Amazon as a Software Engineer. houck@cs.wisc.edu

Serverless computing is becoming increasingly popular as a way to avoid paying for idle periods and gracefully handle load spikes. Serverless platforms typically use containers to isolate lambda instances. General-purpose container systems such as Docker, however, are not well suited to serverless sandboxing and introduce unnecessary startup costs. In this work, we analyze the tradeoffs offered by alternative containerization primitives and use our findings to build a lean container system, SOCK, optimized for serverless workloads. Replacing Docker with SOCK in the OpenLambda serverless platform results in an 18x speedup.

The effort to maximize developer velocity has greatly influenced the way programmers write and run their code. Developers are writing code in higher-level languages, such as JavaScript and Python, and reusing libraries when possible in order to avoid memory management details and the re-implementation of common logic. Developers are also decomposing their applications into cooperating microservices, easing maintenance burdens and making incremental development simpler.

Containers are an increasingly popular way to deploy these microservices. Instead of virtualizing low-level resources (e.g., network interfaces), containers virtualize high-level resources (e.g., port numbers). Containers thus serve as a lightweight alternative to virtual machines, providing each microservice with a virtualized environment and eliminating the need to provision a different operating system for each microservice.

Recently, *serverless computation* has emerged as a new style of cloud platform that integrates a common development approach (application decomposition) with a popular deployment strategy (auto-scaling containers). In various serverless offerings, such as AWS Lambda [3], developers decompose their applications into handlers, called *lambdas*, that execute in response to web requests or other events. Lambda instances execute inside sandboxes (typically containers) and automatically scale up or down based on load. Leaving both the runtime and autoscaling to the platform, developers no longer need to manage servers themselves, hence the name "serverless." New instances are provisioned quickly (often in less than a second), and tenants are only billed during the handling of events, making serverless ideal for load bursts as well as cost savings during application idleness.

The Problem. While high-level languages, reusable libraries, containers, and serverless platforms all improve developer velocity, these approaches also create new infrastructure problems by making process cold-start more frequent and expensive. Languages like Python and JavaScript require heavy runtimes, making startup over 10x slower than launching an equivalent C program [1]. Reusing code introduces further startup latency from library loading and initialization [4]. Running microservices in separate containers, rather than just separate processes, introduces a variety of additional initialization overheads [7]. Serverless computing multiplies these costs: if a monolithic application is decomposed to N lambda handlers, startup frequency is similarly amplified.

Tyler Caraza-Harter completed his PhD at the University of Wisconsin-Madison in 2016, where he was advised by professors Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau and did research on containers and serverless computing. After graduation, he worked on Azure SQL at Microsoft Gray Systems Lab, and is returning this fall to UW-Madison as an instructor. Tyler is actively involved in two open-source projects, the Pivot Libre project for preferential voting (https://github.com/pivot-libre) and the OpenLambda project (https://github.com/open-lambda). tylerharter@gmail.com

Andrea Arpaci-Dusseau is a Full Professor of Computer Sciences at the University of Wisconsin-Madison. She is an expert in file and storage systems, having published more than 80 papers in this area, co-advised 24 PhD students, and received 11 best paper awards; for her research contributions, she was recently recognized with a UW-Madison Vilas Mid-Career Investigator award. She also created a service-learning course in which UW-Madison students teach CS to more than 200 elementary-school children each semester. dusseau@cs.wisc.edu

Remzi Arpaci-Dusseau is a Full Professor in the Computer Sciences Department at the University of Wisconsin-Madison. He co-leads a group with his wife, Professor Andrea Arpaci-Dusseau. They have graduated 24 PhD students in their time at Wisconsin, won 11 best-paper awards, and some of their innovations now ship in commercial systems and are used daily by millions of people. Remzi has won the SACM Student Choice Professor of the Year award four times, the Carolyn Rosner "Excellent Educator" award, and the UW-Madison Chancellor's Distinguished Teaching Award. Chapters from a freely available OS book he and Andrea co-wrote, found at http://www.ostep.org, have been downloaded millions of times in the past few years. remzi@cs.wisc.edu

Why, exactly, is it so slow to start containerized Python programs that have dependencies?

In order to answer that question, we embark on two performance analysis studies. First, we take a look at Linux containers, which are typically based on Linux namespaces and other abstractions. By instrumenting the kernel and isolating specific aspects of containerization (e.g., container storage), we identify several bottlenecks. For example, network namespaces are not scalable due to a single large lock in the kernel, leading to long latencies when many containers are created concurrently. Second, we study how Python programs use libraries in an analysis of 876K Python projects scraped from GitHub and 101K unique packages downloaded from the popular PyPI repository. We find that many popular packages take 100 ms to import, and installing them can take seconds.

We leverage the findings from these two studies to build a new special-purpose container system, SOCK (roughly for serverless optimized containers), that streamlines cold-start initialization for Python code that has library dependencies. We integrate SOCK with the OpenLambda serverless platform [5] to support modern development patterns, without incurring excessive startup latencies. SOCK uses lightweight isolation primitives, avoiding the performance bottlenecks identified in our Linux primitive study, to achieve an 18x speedup over the general-purpose Docker container system. SOCK also provisions new containers using a new approach that generalizes zygote initialization, a strategy introduced by Android for Java processes.

In an image-resizing case study, these strategies help SOCK reduce cold-start platform overheads by 2.8x and 5.3x relative to the AWS Lambda and OpenWhisk serverless platforms, respectively.

More results from our two performance studies and details about SOCK can be found in [9].

Breaking Down Container Performance

Namespaces are the key abstraction in Linux for logically isolating resources. Namespaces virtualize resources by allowing different containers to use the same virtual name, mapped to distinct physical resources on the host. For example, *network namespaces* allow different containers to use the same virtual port number (e.g., 80), backed by different physical ports on the host (e.g., 8080 and 8081). Similarly, *mount namespaces* give containers access to their own virtual file system roots, backed by different physical directories in the host. Linux also provides namespaces for UTS, IPC, PID, and other resources.

An `unshare` system call allows a process to create and switch to a new set of namespaces. Arguments to `unshare` allow careful selection of which resources need new namespaces. Namespaces are automatically reaped when the last process using them exits.

The flexibility of `unshare` allows us to study the performance and scalability of the various namespaces, used independently or in conjunction. Combining the performance numbers with measurements from kernel instrumentation revealed two scalability bottlenecks, in the network and mount namespaces.

During creation of a network namespace, Linux iterates over all existing namespaces while holding a global lock, searching for namespaces that should be notified of the configuration change. Thus, costs increase proportionally as more namespaces are created. Network namespaces are the primary bottleneck preventing high throughput of concurrent calls to `unshare`.

Figure 1 shows the impact of network namespaces on overall creation/deletion throughput (i.e., with five namespaces). With unmodified network namespaces, throughput peaks at about 200 c/s (containers/second). With minor optimizations (disabling IPv6 and eliminating

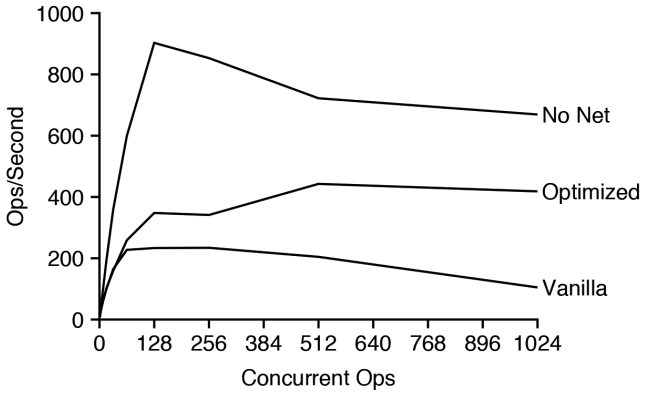


Figure 1: Network namespaces

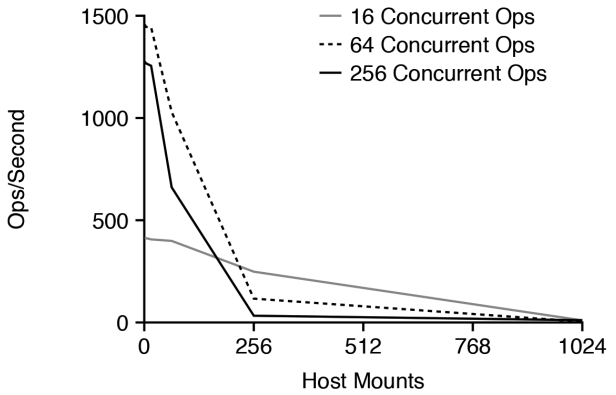


Figure 2: Mount namespaces

the costly broadcast code), it is possible to churn over 400 c/s. However, eliminating network namespaces entirely provides throughput of 900 c/s.

In contrast to network namespaces, it is possible to concurrently create many mount namespaces. However, mount namespaces scale poorly with the number of preexisting mount points on the host, as each new mount namespace starts as a copy of the host's mount points. Figure 2 illustrates this problem: if there are few mount points on the host, we can create nearly 1500 mount namespaces per second. However, as the number of host mounts grows large, the rate at which namespaces can be cloned approaches zero.

Implications. The `unshare` system call provides significant flexibility over which namespaces are used for containers. Depending on the use case, not every namespace may be necessary, so costly namespaces (e.g., those for the network and mount points) should be avoided when possible. Network namespaces are useful for servers that listen on a port, but are less applicable for serverless lambdas that take input from the framework and typically run behind a Network Address Translation layer. Mount namespaces provide a flexible mechanism for exposing specific host mount points inside a container, but in simpler scenarios, the older `chroot` Linux system call may be a better option for isolating the file system. Using `chroot` is essentially free, with calls taking less than one microsecond.

The Cost of Reusing Code

Even if lambdas are executed in lightweight sandboxes, reusing code by relying on various packages can make cold start slow, because the libraries must be re-imported and initialized every time lambda instances are rebalanced or scale up [10].

In order to understand these library-related costs, we scrape and analyze 876K Python projects from GitHub. We expect that few of these applications currently run as lambdas; however, our goal is to identify potential obstacles that may prevent them from being ported to lambdas in the future. We extract likely dependencies in the projects on packages in the popular Python Package Index (PyPI) repository, resolving naming ambiguity in favor of more popular packages. We find that 36% of imports are to just 20 popular packages, shown along the x-axis in Figure 3.

If one of these package is being used for the first time (by a lambda instance or in some other scenario), it will be necessary to *download* the package over the network (possibly from a nearby mirror), *install* it to local storage, and *import* the library to Python bytecode. Some of these steps may be skipped upon subsequent execution, depending on the platform. Figure 3 shows these costs for each of the packages. Fully initializing a package takes 1 to 13 seconds. Every part of the initialization is expensive on average: downloading takes 1.6 seconds, installing takes 2.3 seconds, and importing takes 107 ms.

Implications. Many modern applications, such as Gmail, regularly experience request latency in the tens of milliseconds (including Internet RTT) [5]. If such applications are ported to serverless platforms, even the smallest library-initialization cost (i.e., importing) will dominate, to say nothing of download and install costs that could be necessary. Circumventing these overheads will be key to making serverless a viable option to such latency-sensitive applications.

Serverless Containers

We now describe our design and implementation of SOCK, a container system optimized for use in serverless platforms. SOCK carefully avoids the bottlenecks identified in our analysis of container performance. We integrate SOCK with the OpenLambda serverless platform, replacing general-purpose Docker containers as the primary sandboxing mechanism for OpenLambda. We use additional pools of SOCK containers to construct a caching system that helps lambda instances avoid the startup latencies identified in our study of Python libraries.

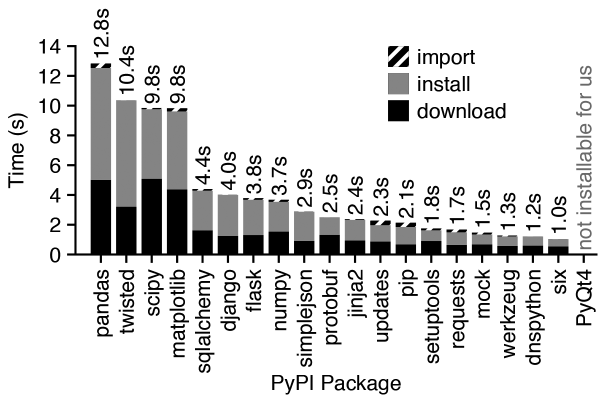


Figure 3: Package initialization costs

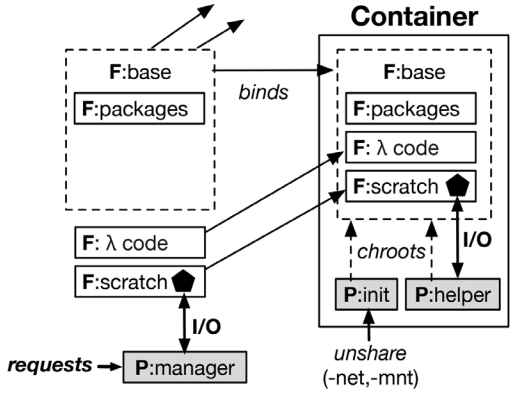


Figure 4: Lean containers

Lean Containers

Figure 4 shows how SOCK efficiently and securely creates containers without requiring costly mount or network namespaces. An init process ("P:init") calls `unshare` to create the necessary namespaces. A second helper process ("P:helper") joins the namespaces later and is responsible for forwarding events and requests to the lambda handler.

Provisioning container storage involves first populating a directory on the host to use as a container root. SOCK stitches together a root directory using several bind mounts. A bind mount efficiently makes a directory at one location in the host file system appear at a second location. SOCK first bind mounts a base directory ("F:base") containing an Ubuntu installation as read-only to serve as a container root; we can afford to back this by a RAM disk as every handler is required to use the same base. A directory used for package caching ("F:packages") is then mounted over the base, as described later. The same base and packages are read-only shared in every container. SOCK finally binds handler code ("F: λ code") as read-only and a scratch directory ("F:scratch") as writable in every container.

The initial processes running in the container (i.e., "P:init" and "P:helper" in Figure 4) call `chroot` to use the populated directory as the container's root file system. We do not require other host mounts in the container, so SOCK avoids the costly creation of a new mount namespace for the container.

The scratch-space mount of every SOCK container contains a UNIX domain socket (the black pentagon in Figure 4) that is used for communication between the OpenLambda manager and processes inside the container. Event and request payloads received by OpenLambda are forwarded over this channel. Thus, lambda instances do not need to listen for input on network ports, so we avoid using poor-scaling network namespaces.

Generalized Zygotes

Zygote provisioning is a technique where new processes are started as forks of an initial process, the zygote, that has already pre-imported various libraries likely to be needed by applications. Linux's copy-on-write sharing reduces the memory consumption of the forked child processes and saves them from all needing to perform the same library initialization work. Zygotes were first introduced on Android systems for Java applications [4].

We implement a more general zygote-provisioning strategy for SOCK. Specifically, SOCK zygotes differ as follows: (1) the set of pre-imported packages is determined at runtime based on usage; (2) SOCK scales to very large package sets by maintaining multiple zygotes with different pre-imported packages; (3) provisioning is fully integrated with containers; and (4) processes are not vulnerable to malicious packages they did not import.

The key challenge to using zygotes for SOCK is integration with containers. We do not trust either lambda handler code, or the package code that handlers may import, so both zygote processes and handlers are containerized. Landing a forked child process in a new container, distinct from the container housing the zygote process, requires a non-trivial relocation protocol described in detail in [9].

Serverless Caching

We use SOCK to build a three-tier caching system for OpenLambda, shown in Figure 5. First, a *handler cache* maintains idle handler containers in a paused state; the same approach is taken by AWS Lambda [3]. Paused containers cannot consume CPU, and unpausing is faster than creating a new container; however, paused containers consume memory, so SOCK limits total consumption by evicting paused containers from the handler cache on an LRU basis.

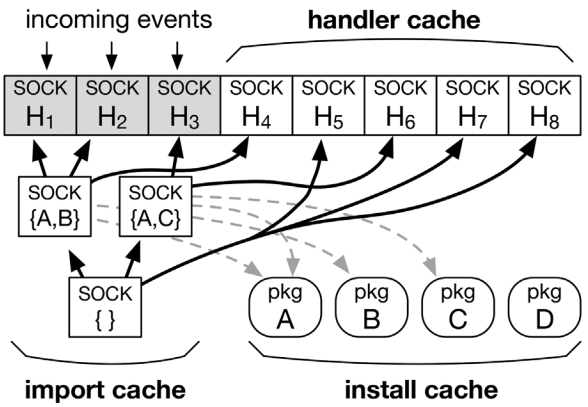


Figure 5: Serverless caching

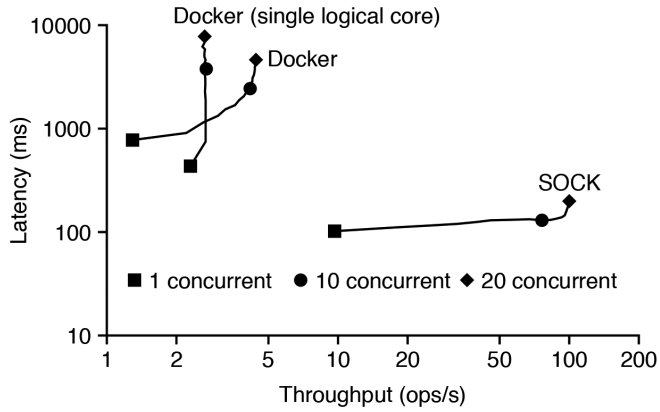


Figure 6: Docker vs. SOCK. Request throughput (x-axis) and latency (y-axis) are shown for SOCK (without zygotes) and Docker for varying concurrency.

Second, an *install cache* contains a large, static set of pre-installed packages on disk. This installation is mapped read-only into every container for safety. Some of the packages may be malicious, but they do no harm unless a handler chooses to import them.

Third, an *import cache* is used to manage zygotes. We have already described a general mechanism for creating many zygote containers, with varying sets of packages pre-imported. However, zygotes consume memory, and package popularity may shift over time, so SOCK decides the set of zygotes available based on the import-cache policy.

In addition to deciding when to add or remove entries from the cache, the import-cache policy needs to decide which zygote to use as the parent from which to fork a child process to serve as the lambda instance. In this regard, the SOCK cache is fundamentally different from traditional caches. Lookup in a traditional cache returns in a hit or miss. SOCK caches never miss and always return one or more hits. Even in the worst case, SOCK can provision a new process by forking a simple Python interpreter with no libraries pre-imported. Or, in the more useful case, there may be multiple zygotes, with varying subsets of the necessary packages pre-imported.

In general, SOCK attempts to choose a zygote that pre-imports a larger subset of the required libraries. This minimizes the number of libraries that a child must import after it is forked from the parent zygote.

One tempting policy to improve performance when possible is to choose zygotes that import a superset of the packages needed by a handler. The child process would then need to import nothing after it is forked. However, we assume the packages may be malicious; pre-importing a library that a handler does not want would expose the handler to a new threat. Thus, for safety, SOCK only chooses zygotes that have imported subsets of the required packages.

Performance Comparisons

We now evaluate the performance of SOCK's lean containers relative to Docker-based OpenLambda and other platforms.

SOCK avoids many of the expensive operations, such as network namespaces, necessary to construct a general-purpose container. In order to evaluate the benefit of lean containerization, we concurrently invoke no-op lambdas on OpenLambda, using either Docker or SOCK as the container engine. We disable all SOCK caches and zygote preinitialization. We run this experiment on two machines, a package mirror and an OpenLambda worker. The machines have 8-core 2.0 GHz Xeon D-1548 processors and 64 GB of RAM. We allocate 5 GB of memory for the handler cache and 25 GB for the import cache.

Figure 6 shows the request throughput and average latency as we vary the number of concurrent outstanding requests. SOCK is strictly faster on both metrics, regardless of concurrency. For 10 concurrent requests, SOCK has a throughput of 76 requests/second (18x faster than Docker) with an average latency of 130 milliseconds (19x faster).

Some of the namespaces used by Docker rely heavily on RCU synchronization, which provides a read-optimized locking mechanism. RCU usage scales poorly with the number of cores [8]. Figure 6 also shows Docker performance with only one logical core enabled: relative to using all cores, this reduces latency by 44% for concurrency = 1, but throughput no longer scales with concurrency.

In addition to streamlining the container-creation protocol, SOCK provisions Python interpreters from zygotes with pre-imported packages. We evaluate these mechanisms with a real-world case study: on-demand image resizing [6]. We write a lambda that reads an image from AWS S3, uses the Pillow package to resize it, and writes the output back to S3. For this experiment, we compare SOCK to AWS Lambda [3] and OpenWhisk [2],

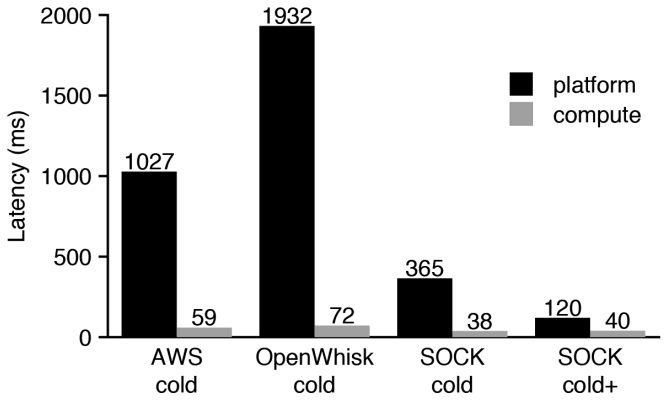


Figure 7: AWS Lambda and OpenWhisk. Platform and compute costs are shown for cold requests to an image-resizing lambda. S3 latencies are excluded to minimize noise.

using 1 GB lambdas (for AWS Lambda) and a pair of m4.xlarge AWS EC2 instances (for SOCK and OpenWhisk); one instance services requests and the other hosts handler code.

For SOCK, we preinstall Pillow and the AWS SDK (for S3 access) to the install cache and specify these as handler dependencies. For AWS Lambda and OpenWhisk, we bundle these dependencies with the handler itself, inflating the handler size from 4 KB to 8.3 MB. For each platform, we exercise cold-start performance by measuring request latency after re-uploading our code as a new handler. We instrument handler code to separate compute and S3 latencies from platform latency.

The first three bars of Figure 7 show compute and platform results for each platform. "SOCK cold" has a platform latency of 365 ms, 2.8x faster than AWS Lambda and 5.3x faster than OpenWhisk. "SOCK cold" compute time is also shorter than the other compute times because all package initialization happens after the handler starts running for the other platforms, but SOCK performs package initialization work as part of the platform. The "SOCK cold+" represents a scenario similar to "SOCK cold," where the handler is being run for the first time but a different handler that also uses the Pillow package has recently run. This scenario further reduces SOCK platform latency by 3x to 120 ms.

Conclusion

Serverless platforms promise cost savings and extreme elasticity to developers. Unfortunately, these platforms also make initialization slower and more frequent, so many applications and microservices may experience slowdowns if ported to the lambda model. In this work, we identified container initialization and package dependencies as common causes of slow lambda startup. Based on our analysis, we built SOCK, a streamlined container system optimized for serverless workloads that avoids major kernel bottlenecks. We further generalized zygote provisioning and built a package-aware caching system. Our hope is that this work, alongside other efforts to minimize startup costs, will make serverless deployment viable for an ever-growing class of applications.

Acknowledgments

Feedback from anonymous reviewers has significantly improved this work. We also thank the members of ADSL and our colleagues at GSL for their valuable input.

This material was supported by funding from NSF grant CNS-1421033, DOE grant DESC0014935, and student funding from Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, Microsoft, or other institutions.

References

[1] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in 2018 USENIX Annual Technical Conference (USENIX ATC '18).

[2] Apache OpenWhisk: https://openwhisk.apache.org/.

[3] AWS Lambda: https://aws.amazon.com/lambda/.

[4] D. Bornstein, "Dalvik Virtual Machine Internals," talk at Google I/O, 2008: https://www.youtube.com/watch?v=ptjedOZEXPM.

[5] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*: https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf.

[6] B. Liston, "Resize Images on the Fly with Amazon S3, AWS Lambda, and Amazon API Gateway," AWS Compute Blog: https://aws.amazon.com/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway, January 2017.

[7] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM Is Lighter (and Safer) than Your Container," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 218–233: http://cnp.neclab.eu/projects/lightvm/lightvm.pdf.

[8] P. E. McKenney, "Introduction to RCU Concepts: Liberal Application of Procrastination for Accommodation of the Laws of Physics for More Than Two Decades!" LinuxCon Europe 2013.

[9] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," 2018 USENIX Annual Technical Conference (USENIX ATC '18).

[10] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," 2018 USENIX Annual Technical Conference (USENIX ATC '18).

USENIX Supporters

USENIX Patrons
Facebook • Google • Microsoft • NetApp • Private Internet Access

USENIX Benefactors
Amazon • Bloomberg • Oracle • Squarespace • VMware

USENIX Partners
Booking.com • CanStockPhoto • Cisco Meraki
DealsLands • Fotosearch • thebestvpn.com

Open Access Publishing Partner
PeerJ

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SECURITY

BeyondCorp
Building a Healthy Fleet

HUNTER KING, MICHAEL JANOSKO, BETSY BEYER, AND MAX SALTONSTALL

Hunter King is an Engineer on the Security Operations team at Google. Currently, he focuses on endpoint integrity and identity. Hunter has also been a Lead Engineer in the BeyondCorp effort for the last seven years. Prior to Google, he was a Security Researcher at SecureWorks. He enjoys hiking, tinkering, and making lights blink. Hunter holds a bachelor's degree in computer science from Colgate University. hunterking@google.com

Michael Janosko is a Security Engineer Manager in Google's Enterprise Infrastructure Protection group, where he helps secure the way Google works. On weekends, he enjoys a good cup of coffee while building forts with his son. janosko@google.com

Betsy Beyer is a Technical Writer for Google Site Reliability Engineering in NYC, and the editor of *Site Reliability Engineering: How Google Runs Production Systems* and the forthcoming *Site Reliability Workbook*. She has previously written documentation for Google Datacenter and Hardware Operations teams. bbeyer@google.com

Max Saltonstall is a Technical Director in the Google Cloud Office of the CTO in New York. Since joining Google in 2011, he has worked on video products, internal change management, IT externalization, and coding puzzles. He has a degree in computer science and psychology from Yale. maxsaltonstall@google.com

Any security capability is inherently only as secure as the other systems it trusts. The BeyondCorp project helped Google clearly define and make access decisions around the platforms we trust, shifting our security strategy from protecting services to protecting trusted platforms. Previous BeyondCorp articles discussed the tooling Google uses to confidently ascertain the provenance of a device, but we have not yet covered the mechanics behind how we trust these devices.

Our focus on platform security is supported by a wealth of evidence [1] in the industry that end users are the number one target of a wide range of attacks that also vary in sophistication. Attackers can devise quite advanced social engineering attacks as mechanisms to deliver malicious code onto devices, where they can then exploit the large attack surface of modern operating systems. Advanced attackers aim to reuse trust inherent in the device, the credentials on the device, or the trust granted to the user to further exploit systems.

To successfully prevent compromise in environments with a constant mix of trusted (enterprise web apps, corporate credentials) and untrusted content (external software repos, social media, personal email, etc.), the platforms themselves must have a layered and consistent set of controls. As a result, the platforms that make up the fleet are the new perimeter.

Building upon Previous Work

The work we describe in this article builds upon the work described in the white paper "Fleet Management at Scale" [2] and the previous five BeyondCorp articles [3]. Building on this foundation, our team aimed to further strengthen the BeyondCorp model by:

1. Defining what a healthy fleet looks like from a common control perspective
2. Ensuring that these controls are consistently and comprehensively applied, measured, and enforced
3. Using these measurements to drive continuous improvement in our control set

Defining the Threats against Your Environment

As with any defensive security effort, it's important to first define the threats against the environment you're trying to protect. When creating this list of threats, it's helpful to think of classes of attacks instead of all the variants of a single attack. Attackers are constantly discovering new variants of attacks, which makes defining the entire tactical threat environment impossible. However, if you successfully mitigate a class of attacks, then variants within that class should be less concerning [4].

At a very high level, some classes of threats to consider against your platforms include:

1. **Unknown devices**: sensitive systems accessed by unknown or unmanaged devices
2. **Platform compromise**: exploitation of a misconfigured operating system or software on the platform
3. **Security control bypass**: system compromise through unused or misconfigured security policy

4. **Privilege escalation**: code execution resulting in privileged system controls takeover and persistence on the system
5. **Software compromise**: installation and persistence of malware
6. **Attack persistence**: prolonged persistence of attackers due to lack of inspection
7. **Authentication bypass**: compromise of the platform through password theft or authentication bypass
8. **Data compromise**: unauthorized access to sensitive data on disk, memory, or in transit
9. **Attack concealment**: prolonged persistence of attackers due to lack of logging and monitoring
10. **Attack repudiation**: hampered investigations due to attackers' ability to cover their tracks

Addressing These Threats through Improved Fleet Health

With these threats defined, you can better identify the classes of controls you need to mitigate these threats. Then you can measure the state of these controls (their effectiveness, and whether they are on or off) through device inspection at service access time. Table 1 maps each of the categories of threats outlined above to the qualities ("Control") one would expect to see in an ideal trusted platform.

Characteristics of a Healthy Device

A healthy fleet is composed of healthy devices supported by tooling, processes, and teams to maintain fleet health. We consider a device to be healthy if:

- It can withstand most attacks.
- It provides sufficient telemetry to contain a compromise when one occurs.

Let's take a deeper look into the reasons why each of the qualities of an ideal trusted platform we enumerated above are important.

Fleet Inventory and Asset Management

Hardware is the foundation on which the OS and applications run. Limiting hardware configuration variations allows you to more effectively reason about the capabilities and limitations of the devices in your fleet. An inventory system places an upper bound on the number of devices able to connect to sensitive systems through device access provisioning.

OS and Software Configuration Management

Software management is a key component to maintaining a healthy fleet. A centralized management infrastructure should drive a consistent platform configuration to ensure that instances of the trusted platform:

#	Threats	Control
1	Unknown devices	Fleet inventory and asset management
2	Platform compromise	OS & base software configuration management
3	Security control bypass	Security policy management & enforcement
4	Privilege escalation	Resilience against system takeover & persistence
5	Software compromise	Software control and anti-malware
6	Attack persistence	Remotely verifiable platform state
7	Authentication bypass	Robust authentication of platform and user
8	Data compromise	Data protection
9	Attack concealment	Logging and log collection for detection capability
10	Attack repudiation	Response capability on platform/ Detection & response

Table 1: Threat classes and potential mitigations

- Are secure by default, with minimal drift over time
- Continue to benefit from security improvements over time

The ability to patch the running OS, the sensitive software stack, and protective agents is paramount to a healthy security posture. It's equally important to manage configurations (e.g., software auto-update policy) in a central location.

Security Policy Enforcement

Trusted platforms should enforce security policies consistently, and report and log any deviations from expected policy. Security policy is often intertwined with the general OS management and configuration policies mentioned above. However, security policy is unique because it's a *mandatory access control policy that users cannot subvert*. For example, consider minimally inclusive login policies: this strategy lessens the threat of lateral movement, and removing root privileges by default helps mitigate the damage a rogue process can inflict.

Resilience against System Takeover and Persistence

The goal here is to layer defenses so that malware execution doesn't necessarily compromise the security of the system. Ensure that hosts can report abnormal behavior before advanced malware can silence a host's logging subsystem.

Software Integrity and Control
You should be able to restrict unauthorized code execution on the platform. Common strategies include either only allowing known good software and explicitly blocking suspected bad software. We generally prefer an allowed list strategy: it's possible to define the applications you need to accomplish your work, but the potentially bad actors or software you need to block are infinite.

Remotely Verifiable Platform State
The platform should have a cryptographically verifiable integrity mechanism that provides guarantees on the underlying platform—from the firmware up to and including the running OS. Some examples include first-command-execution control [5], secure boot, and remote attestation.

Robust Authentication of Platform and User
Wherever possible, credentials should be hardware-backed or hardware-isolated on a system. Windows Defender Credential Guard [6] is one example of this capability.

Data Protection
We assume that any user's system has some sensitive data; therefore, sensitive data should be encrypted both at rest and in transit. To handle lost or stolen devices, devices should support remote wipes that destroy any data stored on the system and any long-term credentials.

Logging and Log Collection for Detecting Threats
To provide defense in depth, the platform threat model should assume that attackers will bypass preventative controls and that machines will be compromised. To mitigate this risk, your platforms should be able to log such incidents. Logging should include user- and device-attributable audit records for all sensitive data accesses or modifications, including changes to the platform's security controls, state, and behavior. This information should be streamed to a centralized logging facility. The ideal logging strategy prevents unauthorized processes from tampering with the logs.

Response Capability on Platform / Detection and Response
If a threat is detected, platform capabilities should facilitate remote incident response by authorized intrusion analysts. Tools like GRR can provide remote accessibility for performing this analysis [7]. We prefer to keep device-in-hand forensics to a minimum, as this strategy can't scale to respond to a widespread breach. Ideally, authorized analysts should be able to create a forensically sound timeline of an incident and augment the investigation with one-off pulls from the affected systems. By re-creating an event, the Detection and Response team can obtain a thorough picture of what happened and respond accordingly.

Maintaining a Healthy Fleet

A group of client devices with the controls detailed above make for a generally healthy and secure fleet. To reach that state, we first needed to figure out how to bootstrap our platform trust.

Building Up Trust

Sensitive services should only be accessed by trusted devices. We divide system trust into tiers. Devices can earn different levels of trust based on their characteristics and behavior [8].

Unfortunately, this approach results in a chicken and egg problem: transitioning a device into a trustworthy state requires access to a client software repository, yet a client software repository *is* a sensitive system. To resolve this issue, we introduce an *Identified* state in the journey from untrusted to trusted. An identified device is one our inventory system believes to be in good standing but is not trusted for some reason. These devices can access a subset of our client software repository in order to install remediation software. This software enables a machine to report device state, download and apply required patches, and take all necessary steps to fulfill the requirements of a trusted platform.

As you work towards building a healthy fleet, you achieve a better understanding of your environment. As a result, you're in a stronger position to grant access confidently. The next challenge is maintaining that state as technology and your business continue to change. The following section discusses how to keep the fleet in a good state of health as you evolve, and how to correct quickly when health degrades.

Combating Device Entropy

Once in the hands of users, devices are prone to becoming less secure as security guarantees atrophy over time. We've found a few strategies useful in our fight against entropy.

The first and most powerful strategy is to integrate access decisions with an inventory system. All machines should be known and trusted before they're granted access to internal resources. At Google, we add every machine in our fleet to our corporate inventory during the receiving and imaging process. We promptly remove access from any devices reported as missing, stolen, or lost. To encourage timely reporting of lost or stolen devices, we require users to self-report before they can receive a replacement device.

It's also important to have strong telemetry around the state of any machine that accesses your environment. Facebook's OS Query [9] is an excellent open source telemetry tool for Linux, OS X, and Windows: it allows you to measure device properties such

as a machine's OS version, patch level of critical software, and encryption status.

Finally, patch and configuration management tools [10] enable you to change the security state of a machine—transitioning an untrusted machine into a trustworthy one. BeyondCorp uses access restriction to help drive user actions such as rebooting or accepting updates.

Detecting Unhealthy Hosts

Throughout the lifecycle of a host, certain actions or inactions might cause a device to transition to an unhealthy state. Our trust inference system [11] detects state changes by performing continuous trust evaluations. When a device fails to meet our trust criteria, we downgrade its trust level to *Identified*. We notify the machine's owner and provide instructions for remediating their device.

Our Detection and Response Team acts as an additional datasource for trust decisions. This team can remove trust from any machine that's acting maliciously.

Providing Flexible Policies

At a quick glance, defining fleet healthiness is a straightforward task. However, like most IT environments, the devil is in the details (and the exceptions). When dealing with a plethora of different OSes and a wide variety of use cases, you encounter many of these details.

As we roll out controls to the fleet, we always attempt to introduce thresholds of policy compliance rather than institute absolute requirements. This strategy allows users greater flexibility to operate within a good state and avoids draconian rule sets that break many of our users (causing them to seek out workarounds or overrides). For example, if a user needs to apply a non-critical patch, we give them a grace period before downgrading their access.

We also believe it's important to design preventative controls to provide signal to your incident detection and response capabilities. To that end, we work to integrate these controls into our security information and event management pipeline so that they can report and log relevant policy-related data. Capturing data about when we allow access and when we block access according to policy can aid in future forensics and incident detection.

Rolling Out and Scaling These Principles

A typical development process and rollout by the Security Team and its partners starts with the design and prototype phases, followed by a period to gather feedback across the fleet and from our users. Over time, we've arrived at a strategy of first rolling out controls in monitor mode and crafting our dogfood [12] populations to facilitate debugging. For instance, we might push a new USB auditing agent to a subset of a hardware engineering organization, as this population often interacts with custom USB components. As a result, we'll uncover edge cases that will likely crop up in a less concentrated form across a broader sample size. Alternately, we might slice the dogfood geographically and prepare local support staff in advance of the change.

When rolling out new controls, clear communication helps build understanding of the new policies and why they exist. Mapping each control to the threats it addresses helps everyone understand why the Security team has chosen a particular action. High transparency and explicit explanations of our criteria have increased understanding among our users and helped us build consensus among stakeholders. When they saw we had no concealed objectives or motives, we could bring them fully on board with our vision of the future and our timeline to get there. Often, teams tasked with making security-driven changes can benefit from seeing the big picture goal, which increases the credibility of the request and therefore also increases buy-in from partner teams. This buy-in often leads to a virtuous cycle of feedback about how you can make the fleet even more secure.

Platform Measurement and Control Parity

Once you define your baseline expected qualities, you'll find you can't apply controls universally—capabilities vary (sometimes widely) among platforms, both in terms of the device itself and in the management/policy layer. For example, Chome OS's Secure Access provides robust software control, but Linux has no out-of-the-box capabilities that prevent malware. To ensure consistency in security across our fleet, we needed to normalize security evaluations. While it's probably not appropriate to expect 100% parity across different platforms (as capabilities and threat models differ), we aim to be consistent when classifying a control as sufficient versus a security risk that requires action.

To accomplish normalized evaluations, we analyzed the current state of all relevant platforms with respect to how well they met our control ideal state. We then evaluated the gaps from ideal in totality. We created an overall fleet health report for each platform managed at Google—not a report card, but a shared understanding of capabilities. For each platform, we evaluated the following:

- Can the platform support the control?
- Is the control turned on by default?
- Can we measure the state of the control?
- Is the fleet in compliance?

To drive objective measurement and equivalencies, you might consider:

- Anchoring these strategies in a shared measurement unit: time since patch released, geo-location, count

- Driving your measurements from a relative reference point: versions from current, features supported vs. implemented

Setting these standard measurements is the hard part. Once you have equivalency, your ability to discuss fleet health will greatly improve.

Where preventative controls are lacking or only partially effective, you can look for other ways to mitigate risk—for instance, higher monitoring/detection signal confidence or a compensating control that is more effective on a platform. You may find that you're relying on a subjective overall sense of robustness of the platform against attack. Modern operating systems have very complex attack surfaces, capabilities, and threat models; the best way we've found to aggregate all this information still boils down to manually comparing the desired characteristics of the device versus its actual characteristics. This comparison allows us to make high-level recommendations around projects to fill gaps and to prioritize those projects. No matter the source of the data driving these conclusions, it's important to document the rationale for the conclusion or at least the process that generated it. Doing so allows people beyond the immediate security engineers to understand the fleet state.

Deviations from Ideal

Despite all the best efforts to define, roll out, measure, and enforce controls, you may inevitably face the harsh reality that 100% uniform control deployment is a mythical state where unicorns frolic unconcerned about malware and state-sponsored attackers. You need to have a plan for deviations from the ideal state, root cause analysis, and exception handling.

Many deviations are naturally occurring, resulting from broken processes, faulty management tooling, flaky releases, and other root causes. For instance, there are often delays in applying patches on a system. It's important to understand when it makes sense to grandfather in exceptions fleetwide, and preventing the growth of the exception group versus when you should instigate hard corrections in control states. If you're clear about the threat model and user impact tradeoffs, you can drive good decisions here.

Exceptions should be measurable and time-based. We recommend you classify root causes in a consistent fashion across the fleet so that you can drive understanding around any gaps and identify places where controls are not suited to the fleet or certain classes of users. If an exception is perpetually renewed (or otherwise never expires), the control is not working. You should redesign the control or revisit your assumptions about its role in the fleet.

Getting Started

How do you start putting the BeyondCorp principles discussed in this article into practice on your own fleet? A general approach involves four main steps:

1. Define the security controls you care about.
2. Find a way to measure those controls.
3. Determine where your fleet isn't in compliance.
4. Fix workflows that don't work with your defined security stance or define exceptions.

The first essential step is defining the goals you want to achieve. You shouldn't create a set of desired security controls in a vacuum–these controls should be specific responses to threats you need to defend against. Explicitly enumerating threats provides you a heuristic to measure effectiveness and a framework to reason about the priority of individual properties. Consult partner teams (see "Lessons Learned," below) when defining and ranking desired qualities. As you clarify your threats and the controls that will mitigate them, build in tests such as unit tests or end-to-end red team assessments to evaluate how effective those controls are. Then you can determine whether they actually meet your security goals in practice.

In order to ascertain a device's security posture, you must be able to measure its current state versus the ideal state. If you haven't already, you'll need to roll out instrumentation software to your fleet to collect relevant data. However, raw data is only half of the story: you also need to define the ideal state your devices will be measured against. As a large fleet guarantees variation, you need to define multiple ideal states in order to cover all potential valid use cases.

Once you can measure the security stance of your fleet, you can start examining devices with deviations from the ideal. Some deviations might pose no security risk (as they're mitigated by compensating controls), but other deviations will uncover gaps. We focused our initial efforts on ensuring that new machines are in compliance with a control from the first moment employees use them. Once we knew that all new devices began their lives in a known good state, we could turn our attention to the rest of the machines in our fleet to improve overall fleet health.

Establishing an exception framework so you can create exceptions for the existing fleet when enforcing a new control is equally important. The deviation in the fleet will thus remain static, allowing you to remediate existing machines while keeping new machines in compliance. Once you isolate the problem to a grandfathered portion of the fleet, you can cluster failure reasons. These clusters will uncover problems shared by entire classes of devices or workflows. Tackling the largest and most risky of these clusters first will provide the largest security win for the smallest amount of effort. Repeat this clustering and

remediation process until you have resolved the main issues in the fleet. One-off issues may need explicit exceptions if a user's workflow is explicitly not compatible with a desired security property.

While this system requires a lot of collaboration and hard work from many different teams, completing the effort gives you and your organization a more resilient position in the face of constant attack.

Lessons Learned

Instituting a coherent program for measuring and evaluating trust and fleet health is not a short-term project. Fully achieving the goals outlined in this paper (and the more general goals of BeyondCorp) requires significant resources. That being said, some lessons we've learned over the past couple of years can save you some time and headaches.

Set Milestones Early

Set key milestones sooner rather than later. Determine which properties you care about and rank them (at least roughly). This exercise helps you allocate resources efficiently and provides the motivation to implement large-scale projects. Incorporating data from a fleet management system into your authorization decisions is an excellent initial milestone. This alone will keep unknown devices from reaching your services and has the side benefit of providing a known good device inventory.

Decide How to Handle Exceptions

Define your approach to exceptions early in the project. Every fleet contains devices that cannot fully comply with the ideal security stance. Determining the procedural and technical implementation of exception management is key to a successful rollout. Define the reasons an exception can be granted, how to document those reasons, the maximum length of time an exception can exist before it must be reexamined, and the review process for existing exceptions.

Engage with Partner and Impacted Teams Early

A successful implementation of BeyondCorp requires work from the entire IT organization. Engaging with partner and impacted teams early in the process will dramatically streamline the enforcement portion of a rollout. For example:

- The device procurement and onboarding teams will need to ensure they keep the fleet management system up to date as devices are added or retired from the fleet.
- Other security teams will provide valuable input while defining machine security properties and potential inputs into the overall system.
- Traditional IT support teams will field the vast majority of user escalations. It is essential they understand the goals of the project and are able to help troubleshoot user issues.

You also need a way to communicate with the users who will be directly impacted by this change. Ensuring that the average user can actually follow and complete self-remediation steps reduces the load on IT and time wasted on troubleshooting.

Conclusion

Securing your employees' machines is a cornerstone to securing the crucial information your company handles. To this end, we thoroughly evaluate and regularly inspect all corporate devices to validate their health. Only known healthy devices can access critical internal systems and information.

Employees and their devices have already earned the attention of malicious actors, and it's up to you to defend employees while keeping them productive. To do that, you need a strong sense of fleet health, clear policies and measurements, and a process for handling deviations from the goal state. With consistent controls and enforcement, we believe every enterprise can simultaneously boost fleet health and security, improving resilience to an ever-increasing variety of attacks and threats.

Acknowledgments

While this continues to be a large cross-functional effort across Google and there are many contributors to this project, we want to acknowledge Cyrus Vesuna for his work on defining common trusted controls across our platforms.

References

[1] See Verizon, "2018 Data Breach Investigations Report: Executive Summary": https://www.verizonenterprise.com/resources/reports/rp_DBIR_2018_Report_execsummary_en_xg.pdf; Mandiant, *M-Trends 2018*: https://www.fireeye.com/content/dam/collateral/en/mtrends-2018.pdf.

[2] Google, "Fleet Management at Scale," November 2017: https://services.google.com/fh/files/misc/fleet_management_at_scale_white_paper.pdf.

[3] https://cloud.google.com/beyondcorp/#researchPapers.

[4] New variants often stretch the common understanding of classes of attacks, so you can't ignore variants completely. For instance, the industry thought we had a good grasp on microarchitecture security up until 2018—see Jann Horn, Project Zero (Google), "Reading Privileged Memory with a Side-Channel," January 3, 2018: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[5] Such as Intel's Boot Guard: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-family-mobile-brief.pdf.

[6] Microsoft's Defender Credential Guard: https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard.

[7] https://github.com/google/grr.

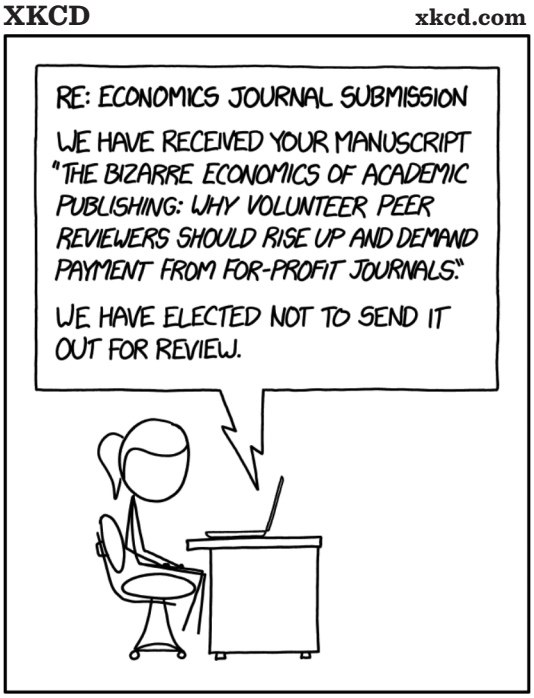
[8] For a description of trust levels and calculation, see B. Osborn, J. McWilliams, B. Beyer, M. Saltonstall, "BeyondCorp: Design to Deployment at Google": https://ai.google/research/pubs/pub44860.

[9] https://osquery.io/.

[10] For more on the tools we use at Google, see "Fleet Management at Scale: How Google Manages a Quarter Million Computers Securely and Efficiently": https://ai.google/research/pubs/pub46587.

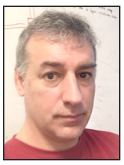
[11] For more on the trust inference system and the other moving parts of our BeyondCorp model, see B. Osborn, J. McWilliams, B. Beyer, M. Saltonstall, "BeyondCorp: Design to Deployment at Google": https://ai.google/research/pubs/pub44860.

[12] Dogfood: early release of products to employees to get feedback and catch bugs before a wider release.



Building an Internet Security Feeds Service

JOHN KRISTOFF



John is a Network Architect at DePaul University's Information Services division and an adjunct faculty member at DePaul's College of Computing and Digital Media. He is also enrolled as a PhD student in computer science at the University of Illinois Chicago. John's primary career interests, experience, and expertise are in Internet infrastructure, Internet measurement, and internetwork security. John is or has been associated with a number of organizations and projects in associated fields of research and technology, some of which include DNS-OARC, IETF, FIRST, Internet2, NANOG, and REN-ISAC. jtk@depaul.edu

I produce a set of threat intelligence security feeds compiled from unsolicited communications to a distributed network of Internet systems. The umbrella platform for the project has a home at DataPlane.org where pipe-delimited text-based data feeds are freely available for non-commercial use. Read on for a behind-the-scenes look at how a mix of open source software, leased Internet hosts, and a dash of system administration deliver security feed data to some well-known and widely relied upon security projects and organizations.

Not long ago I proposed an antivirus programming-related idea for a class research project as part of my graduate course work. My professor felt "virus checkers are [not] an effective mechanism, because they are backward looking (at past history)." Presumably other types of threat intelligence systems that construct lists from observed, malicious activity associated with IP addresses, URLs, and domain names would be summarily dismissed along a consistent line of thinking.

My operational friends might mock a sneer and mouth "ivory tower, sheesh" under their breath at the very suggestion of their ineffectiveness. While there is an appeal to the idea that these sorts of approaches to security protection are discouragingly insufficient and futile, the use of threat data learned from past events is relied upon by many as a part of their security strategy. Whatever you believe about historical data for mitigation, threat intelligence in the form of black lists is widely used and can fetch premium prices when the data is unique, comprehensive, and reliable.

System Overview

The core components of the DataPlane.org security feeds are made up of three distinct subsystems as depicted in Figure 1. A set of sensor nodes collect unsolicited communications and relay logs of activity back to a central collection and processing system. The central collector stores events in raw log files and extracts fields of interest for insertion into a master database. Periodically, the database is scanned for recent suspicious activity seen by sensor nodes, which is extracted and pushed to a website for public consumption.

Producing security feed data would be nothing without a source from which to derive insight. How does one go about compiling source data? There are essentially three ways. One way is to get it from someone else. This is surprisingly very common in the security community. People and organizations share, sell, barter, and trade raw data all the time. If you ever compare threat intelligence between providers, do not be surprised to see overlap. Sometimes vendors produce the same intel independently, but when you see redundancy they are just as likely if not more so to have obtained raw data from a common original source.

The second way to obtain threat intelligence data is to actively seek it out. This may come from active monitoring, probing, data capture, crawling, and so forth. Obtaining data this way is often how one threat intelligence provider differentiates itself from another, since

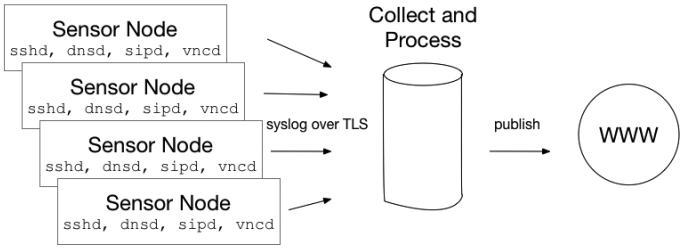


Figure 1: DataPlane.org security feeds system overview

these methods are often distinctly proprietary and unknown to others. This can also be the most costly and least robust approach. As targets of data collection activities change, move, react, or go away, data gathering processes must adapt else the end product may prove untrustworthy or absent of any insight at all.

The third way, a passive approach, is the easiest and cheapest, but it is not without limitations. Passive data collection is when you let the data come to you, from a honeypot or darknet monitor, for instance. The security feeds from the DataPlane.org project use a type of passive approach. DataPlane.org sensors mimic real applications, but they never allow access beyond simple unauthenticated application requests nor allow access to the system beyond an authentication phase.

I've had a fair amount of experience designing and compiling security feeds for nonprofit and commercial use. A few years ago I decided to run my own independent, free service for the community. Why do I do it? I can afford it, but most importantly because it pays dividends in subtle ways. For example, since I am also a PhD student, I can leverage the DataPlane.org platform for research ideas and data measurement experiments. Running DataPlane.org also gives me a platform with which to remain in the good graces of the security community. If nothing else, the security community is largely built upon reputation and trust. I've recently had offers of support and kudos from an array of benefactors. There is some non-zero amount of street cred that helps ingratiate myself with others I might not otherwise have had a chance to please.

Sensors

One drawback to a sensor network as used by DataPlane.org stems from what it does not or cannot see: targeted attacks, for example. It will fail to see threats that simply never cross its paths. My aim with the DataPlane.org project is to obtain a reasonably broad, sampled view of undirected Internet threats at diverse geographic locations (both from a physical location and an Internet routing perspective). Passive monitoring is of almost no value in IPv6 because of the sheer size of the address space. I focus on IPv4 networks with all the limitations this implies.

At recent count, the DataPlane.org project has approximately 100 sensor systems dispersed around the globe on six continents and at least one IP address in roughly 1/3 of all routable IPv4 /8 prefixes. While this isn't the world's biggest, most diverse, distributed network of systems, it might be one of the larger ones of this type run by a single individual.

This may lead to an obvious question. How much does this infrastructure cost? Before answering, let's just briefly consider how the network is not constructed.

I've been involved in similar projects in the past where people or organizations donate a sensor or threat intelligence data for the good of the project. While this can be a source of tremendous data, the reliability of the underlying source infrastructure is frequently a problem. Processes mysteriously stop, systems go down, or the friend at the organization who provided access to the raw data has left the organization and now no one left knows you or is motivated to fix a problem.

An approach used by many reasonably well-funded research groups such as CAIDA and RIPE is to send hosting volunteers a disposable system that can be plugged in, turned on, and then remotely managed with minimal additional supervision from host networks. These include the CAIDA Ark project (http://www.caida.org/projects/ark/) and the RIPE Atlas project (https://atlas.ripe.net/). These systems, too, can only gather data to which they are exposed, but at least in this scenario the only worry is the availability of power and connectivity. However, acquiring, provisioning, and delivering more than a handful of sensors to those who agree to host them may be cost-prohibitive for anyone operating on a tight budget.

For the DataPlane.org sensor network, I've opted to lease Internet nodes, usually from low-end virtual machine hosting providers. Two popular places to find low-cost hosting providers are https://www.webhostingtalk.com and https://www.lowendtalk.com. Prices vary but typically range from approximately $15 (US) to $60 per year for a minimally sized VM with one public IPv4 address.

I've built the network perhaps a little larger than it really needs to be with a little over 100 sensors, and my total cost is approximately $3000 per year. Luckily, the cost of running the DataPlane.org project is a luxury I can afford to fund myself. I plan to continue to do so as long as I'm gainfully employed and as long as it provides a value to myself and the community. More modest sensor networks could be set up for significantly less money.

One of the biggest challenges for the DataPlane.org project isn't so technical. Hosting providers come, go, get bought out, and change their infrastructure. Managing hosting provider dynamics accounts for most of the time I spend on the project. If you'd like to

build your own network of leased systems, I can offer you a handful of tips, summarized below, having dealt with dozens of providers:

- **Historicity**: Consider the history of the provider. Beware of fly-by-night operations.
- **Reputation**: Many low-cost providers have mixed reviews, but the handful that consistently receive low marks probably deserve them for a reason.
- **Payment option**: PayPal is generally the safest for the customer. Do you really want to entrust your credit card information to providers with such slim margins? On a related note, I recommend avoiding any provider who wants a scan of government-issued identification. They don't need it, and you don't want them to have it.
- **Support**: You might not expect platinum service, but you should expect to receive a response to an email within one or two business days. An easy way to evaluate the liveliness of a provider is to send them a low-priority inquiry and see how they respond, if they do.
- **Professionalism**: This attribute applies to both the provider and customer. Customers should want a provider who is courteous in public and when interacting with customers. Likewise, the customer should be mindful of low-cost provider limitations, adjust expectations accordingly, and interact appropriately.

Setting up a DataPlane.org sensor consists of three basic steps: installing the OS, deploying the sensor applications, and configuring logging. I standardize on a minimal Debian stable distro. It is lightweight for low-powered VMs, easy to maintain, and almost always an option with every provider. My sensors require very little disk, memory, or network bandwidth. I can get away with just 256 MB of RAM, and was running an older system with just 64 MB not long ago. The DataPlane.org sensor configuration places only modest demands on system resources.

A sensor build includes multiple common network application listeners with which to produce threat intelligence data. These include DNS, SIP, SSH, and VNC, for example. For some applications, such as DNS and SSH, I use slightly customized versions of well-known implementations (e.g., BIND and OpenSSH, respectively). The SIP and VNC listeners are custom daemons specifically written for the DataPlane.org project rather than full protocol implementations. The custom daemons support enough of the base protocol to interpret unsolicited requests and log application-specific detail. These daemons can be found in the DataPlane.org GitHub repository (https://github.com/dataplane).

The final core capability of the sensor is to log all the desired monitored applications with syslog. Sensor applications of interest must log sufficient detail to be useful for threat intelligence purposes. For sensor applications like DNS, SIP, SSH, and VNC, this should include not only the source IP address responsible for generating the event, but also an NTP-synchronized timestamp set to UTC and a source port when transport protocols like TCP or UDP are involved. A source port helps networks doing network address translation correlate a specific event to an internal IP address. The syslog daemon should forward events of interest to a central collector. How DataPlane.org does this is detailed in the next section.

Central Collector and Processor

Within many networks, syslog is used to send locally generated logs from a host, daemon, or application to a remote collector for safekeeping and later analysis. The DataPlane.org sensor network is little more than a distributed set of syslog clients and a syslog server. However, because sensors are distributed globally on various types of hosting networks, I wanted to ensure some amount of log message reliability and privacy. Therefore all logs sent from sensors to the central collector are over a TLS connection. The sensor is configured with the central collector certificate, and likewise the central collector has a copy of the sensor certificate, providing some assurance each end is known to the other.

I prefer using syslog-ng as the syslog daemon at both the collector and sensor even though most modern Linux systems have migrated to rsyslog by default. The open source version of syslog-ng is reliably robust and includes some features I've grown accustomed to.

The central collector logs everything from each sensor system to a daily log file based on the unique IP address of the sensor system. The DataPlane.org project receives anywhere from a few KB to a few MB per day per sensor depending on how many public IPv4 addresses are active on the sensor.

I leverage two syslog-ng features to interpret received syslog messages and extract desired insight from them for insertion into a database. First, I make use of the pattern database. This is essentially an elaborate regular expression capability applied to syslog messages. Generally, syslog messages of interest have some structure or pattern to them, even if they are essentially text. When you know this structure, you can use the pattern database feature of syslog-ng to capture fields in a log message and then refer to them later in the processing chain as you might with back references in many scripting languages. Working with the pattern database feature requires close attention to detail and will take some getting used to, but once mastered it can prove quite powerful. The following is a very simple example to match on an sshd log message capturing the incoming source IP address:

```
<pattern>Connection closed by @IPvANY:SSH.SADDR@</pattern>
```

This pattern will match not only the connection formatting shown, but will capture the IP address (IPv4 or IPv6) of the host

hitting the sensor. syslog-ng will store the IP address value in a variable named SSH.ADDR, which can be referenced later in the syslog-ng configuration. I make extensive use of the pattern database feature to capture various attributes of log messages, including source IP addresses, source ports, and application-specific detail. As log messages arrive and matches are made, the second syslog-ng feature I leverage is the ability to insert a processed version of a pattern-matched message into a database table. Once the pattern matches are defined, it is simply a matter of associating a matching pattern with a syslog-ng database destination. The following code block is an abbreviated syslog-ng configuration to demonstrate this concept with a PostgreSQL database:

```
parser p_patterndb {
  db_parser(file("/etc/syslog-ng/patterndb.d/example.xml") );
};
destination db_ssh{
  sql(type (pgsql) host("127.0.0.1") port("5432")
   database("example") table("ssh") columns ("logaddr",
         "stamp", "saddr",) values( "${SOURCEIP}", "${ISODATE}",
         "${SSH.SADDR}")
        );
};
filter f_ssh{
        match(
   "0123456789abcdef" value(".classifier. rule id")
         type("string")
        );
};
log{
  parser(p_patterndb); filter(f_ssh);
          destination(db_ssh);
};
```

Publication

The final core component of the security feeds system is to publish the final output to the community. This is a two-step process. The first step is to compile a feed from a data set in the database. The second is to push the feed to the DataPlane.org website for public dissemination. I've found an hourly update of the data feeds is generally sufficient for most users. I extract the most recent week's worth of events per feed category and generate a simple pipe-delimited text file that contains one event entry per line as defined in the commented section of the feed file. Intelligence threat providers or other interested parties can periodically pull these text-based security feeds from my website and process them further. I am currently in the process of making the security feed data available in real-time to users of the Security Information Exchange (SIE) platform run by Farsight Security (https://www.farsightsecurity.com/solutions/security-information-exchange/).

Conclusion

A number of open source projects, commercial providers, and incident response organizations make use of the security feeds DataPlane.org produces. I've been told that these security feeds are among the best and most reliably robust public set of feeds available. This seems somewhat surprising, because today I'm only producing feeds for a handful of basic network services. There are plenty more I could and want to do. The bad news is that I have not spent much time producing more varied security feeds for the past year since I started my PhD work. The good news is that I haven't had to actually do much to keep this security feeds system running as it largely runs itself. Additional detail about the implementation, including some source code for how many parts of the system are set up, can be found at the DataPlane.org GitHub project page. I invite you to take a look, contribute, or adapt what I have done to your own projects.

Perhaps one day, decades from now, the early 21st century may become known as the Internet's gangster era, a heyday where botnets, phishing emails, and DDoS attacks were commonplace. Awaiting that day implies an optimism that suggests we are now living in what will eventually be judged to be "simpler times." Whether or not this comes to bear, it seems plausible that, unlike 1920s America, the Internet do-gooders may be better remembered in the coming story than those G-men of yesteryear. Thanks to the proliferation of excellent, freely available software, sharing of insight between people and organizations, and the motivation to prevent the spread of malicious activity, few misdeeds or criminals run rampant for long.

The story, our story, is currently in progress. This article describes one modest approach to support a cast of characters helping to limit the spread of abuse on the Internet through the distillation and dissemination of security feeds. One day, we may all consider it "backward" and not worth the effort. Until that day comes, we hack.

USENIX Security and AI Networking Conference

ScAINet 2018

ALEATHA PARKER-WOOD

Aleatha Parker-Wood is a Researcher/Manager in the Center for Advanced Machine Learning at Symantec and leads a research team focused on protecting users and their data through advanced machine learning. She received a PhD in computer science from University of California, Santa Cruz, for her work on scientific data management. Her work currently focuses on differential privacy for ML and using deep learning for code analysis, with previous work in file system search, forensics, and AI for Go. She has authored several books and articles as well as numerous patent filings, and most recently served as research co-chair of MSST 2017 and on the PC of ScAINet18. Aleatha_ParkerWood@symantec.com

The USENIX Security and AI Networking conference is a one-day invited talk symposium new in 2018, with Symantec as founding sponsor. It aims to bridge the academic and industry communities in the nascent area of security machine learning and artificial intelligence (AI) and provides a complementary venue to peer-reviewed research conferences and workshops such as AISec and the IEEE S&P Deep Learning Workshop. In the spirit of bridging the two worlds, it was co-chaired by an academic, Polo Chau of the Georgia Institute of Technology, and an industry research leader, Andrew B. Gardner, Head of AI/ML and the Center for Advanced Machine Learning (CAML) at Symantec. It was held in Atlanta, GA, on May 11th, with 122 attendees from many major security companies, as well as students and faculty from Georgia Tech, Emory, UC Berkeley, and more. Audience participation was lively, and there was a parallel discussion track on Twitter at the #ScAINet18 hashtag.

In his opening remarks, Andrew Gardner said that it's an exciting time to work at the intersection of Security and AI/ML but that the challenges faced are significant. Security is characterized by adversarial rare events. The data sets are complex, noisy, heavily imbalanced, and, for the most part, private. Unlike colleagues working on computer vision and other computational perception tasks, this discipline still struggles with the basic representations required for learning on programs, graph dynamics, and the unique event streams of security. He went on to note that "as communities, we have tended to work apart. It's my hope that with greater open and collaborative interaction we can define and frame the next generation of grand problems to focus on, in the same way that self-driving cars have led to huge leaps forward in vision."

The first talk of the day was given by Elie Bursztein of Google, who spoke on abuse detection at scale, and talked about the unique challenges faced by security AI. For example, training data for security becomes obsolete quickly. A cat today is much like a cat from a hundred years ago, but a phishing email is constantly evolving. He also noted that context is critical. Two best friends might say, "I'm going to kill you!" while playing a video game, and it will no doubt be benign, whereas the same phrase in a public argument between strangers at a bar might be a huge problem. The model must account for culture, context, and setting to be accurate. Security ML must balance error costs thoughtfully. An account take-over is very dangerous, for instance, so you might choose to err on the side of false positives, locking people out and offering an extensive manual review process to restore access. He suggested relying on humans to adjudicate the long tail of hard cases wherever possible. Finally, security AI has live adversaries. He suggested limiting the amount of feedback you give to attackers in order to make the attack harder to improve, a theme that would later be reprised by David Freeman of Facebook. Last but not least, he noted that if you have a user feedback mechanism, it can and will be weaponized against you. He advised against blindly trusting feedback and emphasized putting feedback into context, filtering, and rate limiting it. Elie's

talk was delivered as a video recording, so unfortunately there was no audience discussion, but he encouraged watchers to tweet any questions at him.

Next, Jason Polakis of the University of Illinois at Chicago discussed fighting CAPTCHA bots. The evolution of AI has made distinguishing bots from real people increasingly difficult, and impersonation is both easy and cost effective. Most of the tasks that we rely on for CAPTCHAs, such as reading distorted text or recognizing named objects in pictures, are tasks that can now be done with human-level accuracy, using free or inexpensive cloud APIs. He demonstrated how an attacker can use word2vec in combination with Google's image recognition APIs to break image recognition CAPTCHAs at 66.6% success per attempt. Adversarial techniques are not yet defeating off-the-shelf image recognition, so those will not prevent bots. The net result is that CAPTCHAs, in order to defeat bots, are increasingly difficult for human users and pose a huge tax on productivity. He suggests that these techniques will need to be replaced in the near future.

David Freeman from Facebook gave a talk on practical techniques for fighting abuse at scale. In particular, he focused on how to bootstrap labeling from a small data set of ground-truth labels. He pointed out that users are both unreliable and too busy to do all your labeling for you, and that a spam label may just mean "I don't want to see this." But if you use those two sets of labels together, create new features independent of them, and avoid feedback loops, you can get much more reliable predictions. To avoid feedback loops, he reminded the audience that you can't just A/B test new security models, because independence assumptions are violated. If you test on a small set and then deploy to everyone, you cannot be sure whether the adversary gave up or iterated to avoid your classifier in the meantime. Instead, he suggested running in shadow mode to not help the spammers evolve, focusing on the spammer's motives instead of the content, as well as using data they don't control, like the social graph.

Sudhamsh Reddy from Kayak gave a talk on the various types of e-commerce bots, both benign (search engines) and malicious (DDoS, content scrapers, click bots, inventory lock-up bots, etc.). He described how simple volume-based metrics, for example, were effective at detecting the majority of bots seen by Kayak, and how using cascading classifiers, from least to most expensive, allowed them to constrain their computation costs. They save costly techniques such as activity-based analysis for low confidence samples and filter the majority into good or bad using lightweight classifiers.

Alejandro Borgia from Symantec discussed the lifecycle of an advanced persistent threat and how to automate the process of doing attack forensics and attribution. Symantec has gone from a highly manual process to a process that still uses analysts but augments them to give them superpowers. Part of that starts with the attack graph, a giant pile of hay to let them find the needle they are looking for. The attack graph contains information about files, machines, locations, and more. They sift the data to learn generalities about attacks, and then look for clusters of similar events. Rather than looking at one enterprise or event, they look across a wide variety of enterprises and events to learn these attack patterns. He mentioned that Symantec had used this framework to discover Dragonfly 2.0, an advanced threat targeting the energy sector, much faster than they would previously have been able to uncover it.

Yogesh Roy of Microsoft offered a talk on finding suspicious user logins in Azure Cloud. They pool users using similarity and use random walks on user locations. Similar users log in from similar locations, and speed of travel can be used to give a reachability score. The analytics aren't that complex in theory, but in practice, it's hard to do at scale in real time. They use Redis as a cache to partition and store model parameters and behavioral data. They have built a graph of activities across many services—with 22.5M nodes, 46M edges, and 245M security attributes—and use that to model probabilities of attack chains ("kill chain connectivity"). They make an inventory of known attack patterns, match their occurrence in the graph, and then use the rest of sub-graph for context, using the kill chain as a basic probability model to constrain the edges and build out connections using stochastic processes. A compute connectivity score is arrived at using the random walk graph. Finally, they use random forests to classify sub-graphs into scenarios. As a final interesting note, he pointed out that anomalous behavior without attack indicators seems to correlate with insider attackers. An audience member asked how similarity was computed, and Roy said it was entirely based on access patterns and their metadata. Additionally, people had several concerns around geolocation in IPv6, which Roy confessed was an open problem for them.

Le Song from Georgia Tech gave a talk on embedding spaces for graphs. Structure2vec addresses a fundamental problem in graphs—designing features for graphs based directly on data. It leverages strengths of graphical models and deep learning together, using an iterative update algorithm parameterized similarly to a neural network to create an embedding space. First it does an unsupervised pass using the features of each neighbor, pooling, and non-linear updates. Then stronger parameters can be learned downstream using supervision. He gave some examples of how to use structure2vec, including comparing code through control flow graphs and using temporal graph features to find fraudulent accounts. The audience had questions about how the update worked and whether it was unsupervised or supervised. Le explained that the training had both a supervised and unsupervised phase, where the unsupervised phase used a naïve binary label as a placeholder.

Brendan Saltaformaggio, also of Georgia Tech, gave a talk on Retroscope, a system for extracting forensic data from RAM for spatiotemporal data. They interleave execution between a live Android environment with code and data from a memory image to recreate the application's behavior in the past. By reusing the app's own drawing and other internal routines, in conjunction with in-memory data structures that have not been garbage collected, they can re-render screens from the past, even if the application has been closed and logged out of. Because the memory image code knows how to handle the app's data, it can handle all the logistics of rendering the data, and so this method doesn't require deep custom code per application. Brendan demonstrated recovering a deleted draft of a chat from Telegraph after logging out of and closing the app. He's looking at applying this technique to forensics in cases of vehicle or drone-hacking attacks. The talk sparked a lively discussion in the room and on Twitter, as people debated the right way to solve this and the performance implications, such as clearing memory completely on application switch or shut down.

Bayan Bruss from Capital One was next up, talking about financial technology phishing attacks. One out of every 4500 emails is phishing, and email is currently the number one attack vector. Capital One was interested in a solution that would accelerate their analysts and use them more efficiently. They built human-in-the-loop machine learning systems to speed up their analysis and improve defense. They still need MTA filters, which block 98% of attacks, but they couldn't afford to not catch that last 2%. Employees report emails quickly and get rapid feedback from SOC analysts to train both the users and the machine learning. By doing pre-classification, they were able to reduce their analyst workload by 70%. He regards it as empowering your tier 1 analysts by giving them better investigation tools. The goal was not to replace them but to augment them. He emphasized the importance of closing the loop with the analysts and getting the true labels for later retraining. Finally, he talked on the importance of engaging the whole enterprise more effectively. He noted that 64% of phishing drills are recognized, but only 7% of real phishing is, and suggested improving both the quality and frequency of drills. In addition, he noted that it's important to engage users by making it easy to report phishing, giving early feedback and updating the feedback after the analyst looks at it.

Flavio Villanustre from LexisNexus gave a talk on user-entity behavioral analytics (UEBA). His talk was primarily a call to action, covering open problems in UEBA, from dealing with short time series to how to realistically do continuous authentication. He noted that biometric accuracy continues to be quite low, but when used in conjunction with other independent methods, it can strengthen authentication.

Finally, the day closed with a panel session on ML in the world of startups. The panel was composed of Adam Hunt, Chief Data Scientist at RiskIQ; Sven Krasser, Chief Scientist at CrowdStrike; Sean Park, Senior Malware Scientist at Trend Micro; and Kelly Shortridge, Product Manager at SecurityScorecard. Aleatha Parker-Wood moderated and guided the discussion to cover communicating the value of machine learning in a business context, striking a balance between cutting-edge technology and tried-and-true techniques, and what emerging technologies each of them was most excited about. She then offered the closing remarks, thanking the speakers and audience for making ScAINet a success, and encouraging them to form new collaborations and connections within the community.

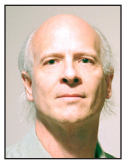
The consensus from attendees and speakers was that this was a superb lineup of speakers and open discussion, and that they looked forward to larger attendance and more speakers next year.

Special thanks to Google for sponsoring lunch and to the Program and Event committees (Polo Chau, Andrew B. Gardner, Aleatha Parker-Wood, Alina Oprea, Nikolaos Vasiloglou, and Anisha Mazumder) as well as USENIX and organizing staff, including Casey Henderson, Sarah TerHune, and Jenn Hickey.

Capacity Engineering
An Interview with Rick Boone

RIK FARROW

Rick Boone is currently a Senior Engineer on the Capacity Engineering team at Uber, where he focuses on modeling and forecasting Uber's capacity needs. He has been at Uber for 3.5 years, where he primarily worked in SRE (Site Reliability Engineering). Previously, Rick worked at Facebook for three years as a Production Engineer and, before that, at a number of tech startups in Los Angeles. In his free time, he loves traveling, swimming, and gymnastics. boone@uber.com



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

When I heard Rick Boone's talk at SREcon18 Americas, I was immediately struck by his approach. While capacity planning is really an art, relying partially on past behavior but just as much on intuition, Rick described uncovering the best metric for reliably predicting capacity as needed.

Uber's services run on their own hardware, and their goal is to always have sufficient capacity without ever having either too much or a shortage that will hurt business. Rick's approach [1] used machine learning to help pick out the appropriate metric and mathematically predict its impact on a service's capacity needs. You can watch the video of his talk to learn the approach used. In this interview, Rick discusses why Uber doesn't use capacity planning and instead relies on capacity engineering.

Rik Farrow: What's wrong with capacity planning?

Rick Boone: For those who are concerned with availability and reliability of services or platforms, service owners, Production Engineers, or SREs, capacity planning is typically one of the fuzziest and least understood parts of their job. When we speak of capacity planning, we're aiming for a "just right" amount of resources allocated for a service, which will allow that service to run both efficiently (i.e., "not using too many resources") and reliably (i.e., "not using too few resources"), even in the face of unexpected surges of traffic. This can be pretty difficult to achieve for a multitude of reasons, especially in a fast-moving, complex environment with lots of interactions between hundreds or thousands of services.

Typically, capacity planning involves a fair amount of back-of-the-napkin math and fuzzy methods that differ from service to service. Knowledge of what drives the service's needs and usage (i.e., "demand") is required, as is knowledge of how that demand will grow and change. Knowledge of the service's dependencies and operational particulars is also needed (e.g., "Does it speak to a database?"), along with an understanding of how those details affect the service's ability to serve its demand (and how it consumes resources). Once all of that is known and understood, the planner then needs to determine the best way to calculate expected demand in the future and then extrapolate expected needs from that.

All of these things tend to be very local and service-specific pieces of information, what I like to call "Jedi" knowledge—intuition which service owners tend to gain over time—which differ wildly across an engineering ecosystem. For instance, a search service will grow at a very different trajectory, and have very different resource needs, than a payment service.

You can start to see why traditional capacity planning can be so difficult across an engineering ecosystem. Its methods are typically not repeatable from one service to another, nor are they scalable beyond one or two teams. If one person or team does manage to use a method that is successful for their service, their skills and insight will not necessarily transfer. Methodology and process becomes wildly inconsistent, leading to confusion and, typically, overallocation and wastage of resources. As teams become less confident that they can capacity plan effectively, they begin to simply "throw hardware" at the problem and move on to more solvable things. Often, empirical data or mathematical reasoning is left out of the process, leading to further lack of repeatability and understanding. And even when data and analysis/mathematics are used, there is still a very worrying lack of confidence or certainty delivered

with the results. If the "plan" is to go wrong, it is unknown by how much it will go wrong. Either it will work or it won't. This leaves stakeholders and dependent services with an inability to make informed decisions or tradeoffs concerning the service.

Beyond the issues of fuzzy methodology, there are also problems that arise from the nature of software and infrastructure. At Uber, like most large-scale engineering shops, we release a lot of code and changes to a lot of services on a lot of servers throughout the day. This all adds up to an ever-changing, complex, and highly coupled environment, the entirety of which is difficult for humans to consider when predicting future usage, especially months in the future.

RF: How is capacity prediction different?

RB: With capacity *prediction*, we aim to remove all of the fuzziness and *hand-waving* from our understanding of capacity usage and needs. We do this by applying statistical and mathematical methods to large amounts of past usage data via machine learning, allowing us to create mathematically sound models of every service, which we can then use to reliably predict each service's future capacity needs.

Each model takes in, as input, a value of Uber's primary business metrics, things like *Trips Currently Online*, and returns, as output, the amount of hardware resources needed to handle that particular volume of the metric. For example, for service "FooBar," the model might indicate that to handle 100K trips online, the service will need 1000 CPU cores.

By providing a method based on a *blackbox* model of any and every service, which takes in an input that is common across all of Uber, we now have a repeatable, scalable, interpretable, and simple way of both assessing capacity usage and predicting its future values. We don't need to know about a service's dependencies, its particulars, its architecture, how its software performs, etc.—all of that is represented mathematically by the model, and we can deal solely with representative numbers, instead of human/jedi knowledge.

As is typical with statistically derived models, we are also able to construct empirically derived measures of the model's accuracy, so that we can have a very precise idea of how much confidence we can place in the model's prediction. Whereas *plans* are often and easily broken, *predictions* are made with an expectation of success (along with an empirical measure of possible failure).

RF: How did you go about creating a method for capacity prediction at Uber?

RB: The primary things needed for us to bring capacity prediction to fruition were: (1) a consideration of the fundamental thing(s) that drive resource consumption and (2) a way to represent these things via data and mathematical models.

At Uber, the demand for most services is driven by a few key high-level metrics, such as the number of drivers online or the number of trips occurring. Because of this, the levels of these metrics typically have a close correlation with resource usage. We started by building multivariate data sets comprising these metrics and the CPU usage of a single service, at a granularity of one hour. We typically use about two weeks of historical data to ensure that we're only analyzing the most recent representation of the service, including its current software releases, dependencies, clients, payloads, etc. Because we have multiple high-level metrics that can drive a service's usage, we perform correlation analysis to mathematically determine which metric has the strongest correlation with the service's resource usage. Having determined the best metric, we then use machine learning methods to build a quantile regression model, which is a variant of a linear regression model, with the high-level metric as the feature/input and the resource usage as the outcome/output. With a quantile regression, we can retrieve 99% of all possible outputs, allowing us to greatly minimize the possibility of underpredicting.

We repeat this process for every service at Uber and store the resulting model in a database, with each model relying on one of a few high-level metrics as its input. Because our high-level metrics are key performance indicators for the entire company, we have accurate forecasting for them, extending months into the future. We simply pass these forecasts into our models and are able to get a prediction for resource usage for each service for the next few months.

Any service owner, SRE engineer, etc., can now query for their service's predicted resource needs for any week within the next 2–3 months and adjust their allocations accordingly.

RF: Is capacity prediction something unique for Uber or is it easy for others to also do this?

RB: This is very doable anywhere! The toughest thing that others might run into is acquiring both historical and forecasted high-level business metric data. Once that is acquired, along with service-level resource usage data (CPU, memory, etc.), you'll need machine-learning methods to apply to the data and train a model. ML libraries are readily available in a number of libraries, primarily in Python or R. Or you could also write your own model trainer. Finally, you'll need a place to store the models (we use Cassandra) and a way to retrieve and apply them. We built a light API in front of the Cassandra model store.

Reference

[1] Rick Boone, "'Capacity Prediction' instead of 'Capacity Planning': How Uber Uses ML to Accurately Forecast Resource Utilization": https://www.usenix.org/conference/srecon18americas/presentation/boone.

Python
Shared Libraries and Python Packaging, an Experiment

PETER NORTON

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

I've been thinking about sharing some thoughts and experiments with the weird science experiment that is `memfd_create()`. It's a system call in somewhat recent versions of the Linux kernel—3.17 and later.

First let's take a trip back in time, and then we'll return to this system call with what I think is a really fun idea that could be used to explore and maybe improve an inconvenient aspect of Python packaging.

Shared Libraries

To get started, I want to talk about shared libraries.

When I was first exposed to UNIX systems in college, there was a tremendous amount of work being done to make the servers of the day more efficient. What computers of the day did was to act as time-sharing systems, allowing shell, compilation, mail, gopher, talk, netnews, and many other activities for multiple users. Like today, most users relied on software that the system administrator either compiled or installed as a package, which would benefit from the use of dynamically linked binaries.

These would help memory usage because by being dynamically linked, they were being linked at runtime to shared libraries. "Shared" in this case had more than one meaning. It meant both that they provide shared code—different programs could benefit from not having to write the same functions over and over—but by a neat trick it also meant that the read-only library codes that were used in *N* programs would all be mapped by the kernel into the same real set of bytes of memory, so each mapping of the library into a program only required a little memory overhead. This meant that even if 500 users loaded 100 KB of the same library code, via logging in and running, e.g., `pine` (which was at one point a very common mail reader), each instance of the program would see 100 KB of mappings getting linked in to its local memory, but over 500 invocations. But instead of using 50 MB of memory, an impossibly large amount at the time, all those invocations would use something more like 100 KB *total*, which is pretty cool, via some clever kernel memory mapping.

The involvement of the kernel is very important to bear in mind here. The shared mappings are done by the kernel and the dynamic linker (`ld.so` on Linux) working together to provide shared mappings of the library routines into each process's virtual memory space at an address that is only known when it's loaded. They are then "fixed-up" at runtime to point to newly assigned addresses so the executable can find them. If you've ever wondered why your Python extension modules are always compiled and linked with the `-fpic` or `-fPIC` flag, that's why. (See http://bottomupcs.sourceforge.net/csbu/c3673.htm for more about the mechanisms that are involved here.)

Even back then, you could share actual routines without bringing shared libraries into the picture by statically linking libraries. This is much simpler, but in the era where powerhouse workstations had 8 MB of memory, they didn't tell a good story about memory efficiency.

In modern systems, shared libraries aren't often first and foremost thought of as ways of saving memory by shared mappings between different processes. In fact they're often seen as

a waste of effort! With the recent abundance of memory available to systems, and the huge amount of data we're processing with that memory, the savings from the shared memory part of shared libraries that I described above has become a bit of an anachronism.

Especially when using Python, the "shared" part of shared libraries has become more about sharing C code with the Python runtime, making libraries able to be invoked from the interpreter. The benefits of this are so common they're almost a running joke—most answers to questions about making Python faster, for example, usually quickly bring up the answer, "Use `cython`" or "Write it in C and load the faster implementation from the shared library." From a more practical standpoint, a pillar of the Python community is the scientific Python stack built on top of NumPy and SciPy, and it goes one step further where FORTRAN code is built and linked so that it is compatible with C calling conventions, and then Python loads the resulting libraries for fast matrix math. Python obviously has to do more than "just load the library" for this to work, but that's where the rubber meets the road, so to speak.

Packaging

Now, the more common of these libraries are usually packaged up by the operating system maintainer—Debian, Red Hat, etc. if you're a fellow Linux user—or someone who fits into that job if your *nix is a different *nix. But once it's built, a shared lib can be dynamically linked by a Python runtime, whether it's packaged by the operating system maintainer, built yourself, or obtained from a third party like a scientific Python packager.

There was a time when GNU `autoconf` was pretty cutting-edge. It is now considered quite unwieldy. Its heyday was in a world with literally dozens of operating systems that were sort-of-but-not-quite like a POSIX or BSD UNIX, and nothing built for one would compile on any others without inhuman knowledge of different CPU architectures, C compilers, and luck.

That was then, and the world is much simpler now (for UNIXes at least), and that's led to the current generation of popular languages being able to do better than `./configure`. Now instead of just producing a runnable program and maybe making it easy to copy the results to your local file system, modern build toolchains will also package up your work, and often turn them into a tidy single-file image that can just be executed. Golang is arguably the king of this category, where one of its main selling points is that when building your program, you will create a static binary—that's it!

Since the modern lifecycle for programs involves multiple deployments per day, there is a lot of appeal to the idea of being able to bundle up a single artifact containing everything a program needs. The prospect of having no external libraries to depend on and no OS packages to install prior—just being able to copy a file and being able to just run the program has become the gold standard of new compiled languages, and once you've done this, it's pretty nice. Golang, Java, and Rust do a great job with having their tooling provide this experience, and they set a standard for other languages to shoot for.

Python has an interesting story in this respect. Python will open a zip file that contains Python code, if the appropriate structure is in place. This is described in PEP 273, and there is some more info in PEP 441. This is the core of some cool stuff that you can get from PyPI, including pre-packaged wheels, eggs, and, outside of PyPI, other less geometrically named things like pexes and pars.

Having all of your dependencies in one place is pretty nice. You don't have to install anything special, you can just point the appropriate Python interpreter at a built zip file and get a really nice experience—both as a developer since the build process is not complicated and as a sysadmin; as long as the version of the Python interpreter is a good match for the application in the archive, you have a pretty good chance at deploying and getting a good night's sleep, too. On the face of it, something as convenient as, perhaps, Java jars. And just about the nicest thing about it is that you simply don't have to worry about installing OS packages or other dependencies.

However, there is one major weak point in using zip archives with Python. Specifically it has to do with shared libraries. If you want to have a shared library in your package, the dynamic linker on your platform, together with the kernel can't map that bit of the archive file into the running program!

That's not the end of the story, though. There is a simple hack that makes these zip archives work: the packaging tool that works with zip files will unzip a shared object from the zip file, write it out to disk, and then use the dynamic loader to make it available to your programs.

Get that? It extracts the library from the zip archive to plant it onto disk. The reason is that neither the Linux kernel nor other UNIX kernels that I'm aware of have special magic to allow portions of a zipfile to be used as a shared memory mapping for a shared library. This means that when you have a zipped-up Python archive for a project that uses common facilities that are best used via shared libraries—like MySQL, gRPC, XML, or what have you—and you want to include them in your bundled artifact in order to guarantee that there aren't dangling, unresolved dependencies, this zip-file format will need to do a few things that you'd prefer not to do:

1. Use space on disk—at least temporarily
2. Use additional disk reads+writes
3. Use additional CPU time at startup and shutdown of the program

None of that seems prohibitive, but in my experience, it can be really demoralizing when you find out that `/tmp` has filled up with detritus from your project, or when you learn that the zip file will get extracted into the running user's home directory, and that user isn't supposed to write there. Or whatever other difficulties your site may discover down in the weedy details of the specific process.

Now, returning to that cool thing I mentioned at the beginning. I heard about a pretty neat new feature in Linux a few months back. It's a system call, `memfd_create()`, that allows us to turn a region in memory into a file in `/proc/<pid>/fd/<the fd number>`. What's really interesting is that it acts like a normal file, which includes being able to be symlinked from other parts of the file system.

So, wanting to reproduce an idea that I'd heard about from some of the super tech companies, I thought it would be fascinating to have the kernel be able to map sections of the zip files—the shared libraries in particular—so that it could be used as a shared library.

This system call doesn't do exactly that, but it seems like it could get us closer to the goal of all-in-one packaging without having to extract to the file system. This works by consuming memory instead of file system space and disk I/O. The question is whether the presence of an appropriate mapping would prevent the dynamic linker from trying to load a library from the system (it should as long as the dependencies are resolved appropriately). If this worked, it would allow `libmysql.so`, for example, to be packaged up and shipped.

And it turns out that as a toy, this seems to work! The core of this is some interesting syscall work that Python lets you do via the ctypes library using the CDDL call, which maps in the library via `dlopen()`. It's pretty nifty—we can load up `libc` in order to get a hold of the syscall we want, then map in the file we have in the archive as bytes, creating the library in memory, and then use CDDL again to load it up.

An outtake of the code, which you can find at https://github.com/pcn/pymyxec, looks like this:

```
# Need to get memfd_create(), which is now in the
# syscall table at 319
# Returns the FD number
def build_a_lib(lib_name, source_bytes):
    memfd_create = 319
    libc = CDLL("libc.so.6")
    print("Lib name is {}".format(lib_name))
    so_file_name = "{}.so".format(lib_name)
    fd = libc.syscall(memfd_create, so_file_name, 0)
    for data in source_bytes:
        os.write(fd, data)
    CDLL("/proc/self/fd/{}".format(fd))
    return fd
```

I'm still smiling and happy at how this has worked. Running this as documented in the README.adoc in the repo shows how:

```
spacey@masonjar:~/dvcs/pcn/pymyxec$ bazel build
        mysql_repl.par; bazel-bin/mysql_repl.par
INFO: Analysed target //:mysql_repl.par (1 packages loaded).
INFO: Found 1 target...
Target //:mysql_repl.par up-to-date:
  bazel-bin/mysql_repl.par
INFO: Elapsed time: 1.389s, Critical Path: 0.84s
INFO: 1 process, linux-sandbox.
INFO: Build completed successfully, 2 total actions
Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license"
        for more information.
(InteractiveConsole)
>>> clientinfo = entry("libmysqlclient.so", "__main__
/libmysqlclient.so")
You got it
[u'bazel-bin/mysql_repl.par/pypi__certifi_2018_4_16', u'bazel
-bin/mysql_repl.par/pypi__chardet_3_0_4', u'bazel-bin/mysql
_repl.par/pypi__idna_2_7', u'bazel-bin/mysql_repl.par/pypi
__urllib3_1_23', u'bazel-bin/mysql_repl.par/pypi
__requests_2_19_1', u'bazel-bin/mysql_repl.par/pypi
__docopt_0_6_2', u'bazel-bin/mysql_repl.par/pypi__MySQL
_python_1_2_5', 'bazel-bin/mysql_repl.par', u'bazel-bin/mysql
_repl.par/__main__', '/usr/lib/python2.7', '/usr/lib/python2.7
/plat-x86_64-linux-gnu', '/usr/lib/python2.7/lib-tk', '/usr/lib
/python2.7/lib-old', '/usr/lib/python2.7/lib-dynload', '/home
/spacey/.local/lib/python2.7/site-packages', '/usr/local/lib
/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/gtk-2.0']
<zipfile.ZipExtFile object at 0x7ff59c934190>
Lib name is libmysqlclient.so
this pid is 14200, lib_fd is 14200
>>> modinfo = entry("_mysql",
"pypi__MySQL_python_1_2_5/_mysql.so")
You got it
[u'bazel-bin/mysql_repl.par/pypi__certifi_2018_4_16',
u'bazel-bin/mysql_repl.par/pypi__chardet_3_0_4',
  ...
<zipfile.ZipExtFile object at 0x7ff59c9341d0>
Lib name is _mysql
this pid is 14200, lib_fd is 14200
>>> link_a_lib("_mysql.so", modinfo[0], modinfo[1])
>>> import MySQLdb
```

At the end of that, we can validate that the process is using the library that we've loaded, the shared library in the par file, and not the shared library installed on the file system.

```
(aws) spacey@masonjar:~/dvcs/pcn/pymyxec$ pmap 14200 |
      grep -i mysql
14200:   python bazel-bin/mysql_repl.par
00007ff59b4d0000     40K r-x-- memfd:_mysql.so (deleted)
00007ff59b4da000   2044K ----- memfd:_mysql.so (deleted)
00007ff59b6d9000      4K r---- memfd:_mysql.so (deleted)
00007ff59b6da000     16K rw--- memfd:_mysql.so (deleted)
00007ff59bc84000   3656K r-x-- memfd:libmysqlclient.so.so
(deleted)
00007ff59c016000   2048K ----- memfd:libmysqlclient.so.so
(deleted)
00007ff59c216000     24K r---- memfd:libmysqlclient.so.so
(deleted)
00007ff59c21c000    456K rw--- memfd:libmysqlclient.so.so
(deleted)
```

This shows that the MySQL libraries that are mapped in are only those that were mapped in via ctypes.CDLL, which is doing the equivalent of a `dlopen()` call and mapping in the library. It also shows that I should update the README on GitHub with one less `.so`. The `(deleted)` is just pmap showing that it thinks the underlying file used to create the mapping was deleted.

It would be nice if `libmysql.so` could be read without having to symlink it into `/tmp` as in the previous example, but using an existing module like this, with a compiled shim library, doesn't give me that flexibility—though someone smarter than I may have an idea about how to do that. Pull requests are welcome!

As a closing thought, one remote possibility would be to see how far we could go with this. For example, could it be possible to store a subset of the required Python support files? Enough of what an interpreter needs from `lib/python<version>/` could be included into the archive, ideally in a way that `memfd_create` could be used to populate, say, a virtualenv with a bunch of symlinks into `/proc/self/fd/<various pids>`, and that virtualenv and the Python interpreter would be entirely spun from the zip file. That way the appropriate Python binary, built and tested as part of the package, would be bootstrapped by the system Python.

I don't know if anyone is interested in that, but if so maybe it'd be a good incentive for me to try to do something with Python 3.

Cheers, and have a great day. I hope this helps you smile a bit.

Practical Perl Tools

GraphQL Is Pretty Good Anyway

DAVID N. BLANK-EDELMAN

David has over 30 years of experience in the systems administration/DevOps/SRE field in large multiplatform environments and is the author of the O'Reilly Otter book (new book on SRE forthcoming!). He is one of the co-founders of the now global set of SREcon conferences. David is honored to serve on the USENIX Board of Directors where he helps to organize and engineer conferences like LISA and SREcon. dnb@usenix.org

In a past column we had the pleasure of learning about graph databases together. That particular column was a blast to write because it gave me the opportunity to dig into graphs, something I've always found interesting. In the process of researching that article, I ran into GraphQL. "Oh, goody, more graphs!" I thought. Perhaps an SQL-esque language for graphs? The bad news is GraphQL is nothing like these things or the graph databases we talked about. Even though they both have "graph" in their name, I would be hard-pressed to describe how they connect (truth be told, it isn't immediately apparent why GraphQL has "graph" in the name). The good news is GraphQL is interesting in its own right, so today we are going to give it its own column. And in keeping with my need for radical honesty, I just want to point out up front that the majority of this column will be focused on GraphQL with the Perl bits largely showing up at the end (and being straightforward-ish).

GraphQL Basics

GraphQL describes itself as "a query language for your API," which is both true and perhaps not as helpful as it could be. The official website continues with:

> GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

But I'm still not sure that helps enough. There are a few parts necessary to understanding what's behind GraphQL. To start, I think of it as being one door down from REST on the client-server interaction hallway. To see what I mean, let's use REST as the exemplar since it has been mentioned countless times in this column.

With REST, the dance goes something like this:

- `GET …/items/shoes` the shoes we have
- `GET …/items/shoe/id` the details for a particular shoe
- `GET …/items/shoe/id/laces` the color laces it can come with
- `GET …/stock/id?laces=brown` the number of those shoes with the brown laces in stock
- `GET …/stock/id?laces=black` the number of the black-laced kind in stock

I'm exaggerating a little bit, but with REST the idea is you make a request, then you follow up that request with additional requests for more specific information. Sometimes you do this a bunch of times. This is great from a data architecture perspective (especially if the URLs are legible). This is less great from a "network is slow and perhaps expensive" perspective: for example, if the client was a mobile phone. That was exactly the use case Facebook had in

mind when it created GraphQL. GraphQL attempts to provide a mechanism for saying, "Here's the data I want" and getting it back in a single interaction.

The second thing GraphQL attempts to do is to allow the client to have a simple, clear understanding of just what data the server holds and what the client can ask for. With REST, there's nothing about the interaction model that prevents the client from asking for a piece of fruit from the shoe store or querying `/those-brown-things-that-go-on-your-feet/` instead of the `/shoes/` endpoint. In that example, the server would likely tell the client to take a leap, but wouldn't it be better if the client already had an understanding of what it and the server could correctly chat about? With GraphQL, there is a schema (kinda like database schemas) that is crystal clear about what data is in play, what form it takes, and how it can be queried.

The GraphQL spec says:

> GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.

That's probably the easiest way to think about it.

Let's Play

To get a handle on how this all works in practice (at least at a very surface level), let's look at some sample GraphQL. To give you examples that will be easy for you to explore in greater depth later, I'm going to use ones that resemble those in the official doc on https://graphql.org.

Here's one of the first pieces of GraphQL in the intro tutorial:

```
{
  hero {
    name
  }
}
```

This says to query the field "name" from the hero object. The reply looks (intentionally) like the query:

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

Note, there's something funky in the docs around this example; more info on that in a moment.

We can add more fields and more objects as desired:

```
{
  hero {
    name
        appearsIn
        friends {
          name
        }
  }
}
```

Did you catch the interesting part? Objects can have both fields and sub-objects (that can have fields). In this case, in addition to asking for a new field, I've also asked for both the name fields in the hero object and the name fields in the friends object in that hero object. That would yield something like:

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
            "appearsIn": [
                  "NEWHOPE",
                  "EMPIRE",
                  "JEDI"
            ],
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

This example also shows that, if desired, objects can hold lists of values fields, not just single strings.

If we want to query for a specific object, we can pass in arguments:

```
{
  hero(episode:EMPIRE) {
    name
  }
}
```

and get just the results we need:

```
{
  "data": {
    "hero": {
      "name": "Luke Skywalker"
    }
  }
}
```

Wait, what? If you are puzzled at this response given the material we've seen before, don't sweat it. I was, too. I could not figure out why the initial "{hero {name} }" didn't yield all of the possible heroes. It took me a bunch of spelunking around in the source for the documentation to find the reason, but when I found it, it yielded an important truth. Let me explain.

The reason why we only saw R2-D2 when there wasn't an "episode" argument was this little piece of code called from the source of the page:

```
/* Allows us to fetch the undisputed hero of
   the Star Wars trilogy, R2-D2.
 */
function getHero(episode) {
  if (episode === 'EMPIRE') {
    // Luke is the hero of Episode V.
    return humanData['1000'];
  }
  // Artoo is the hero otherwise.
  return droidData['2001'];
}
```

GraphQL isn't a database. Remember, "GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions." How those interactions take place are (1) language agnostic and (2) defined by the code you do wire up to it. The code assigned for returning heroes (the GraphQL "resolver" for hero) had its own opinion as to what it should return. This particular lesson took me longer to grok than I would have liked; hopefully, I've saved you a little time.

Want to see both heroes? For that, we would use a syntax (aliases) that allows us to ask for two objects that share the same field name, but with different arguments:

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

The result makes a bit more sense now:

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

There are a number of syntactical sugar extensions to the language including those that make it easier to repeat parts of a query without writing it out repeatedly, ways to pass variables into the language, and ways to change the data (mutate it) instead of just querying. There are also some spiffy introspection capabilities that allow a client to ask the server questions about the schema.

In the interest of brevity, rather than diving into these things (or schema construction itself), I recommend you take a look at the tutorial at https://graphql.github.io/learn/. Instead, let's actually see how we can use GraphQL with Perl.

GraphQL and Perl

The heart of all (present day) support of GraphQL in Perl comes from a port of the reference JavaScript implementation. Quick warning: when you install the GraphQL Perl module, it has a number of dependencies. Make that a large number of dependencies (because the dependencies have dependencies). When I installed it on a fresh Perl distribution, the count was 80. I used "cpanm" (which we've talked about in a past column), so it was only a matter of waiting, but I thought I'd give you fair warning.

For the client-server interaction aspect of GraphQL, the client support is pretty trivial. Your client just needs to be able to spit some GraphQL at the server. It could in theory do some more interesting things like schema validation, but let's leave that for a moment. That is probably just by constructing and sending an HTTP request with the right payload in it like we've done a ton of times before in this column. The harder part is the server-side support. That's where the Perl module mostly comes into play.

In the past we've looked at a few Perl web frameworks with the most emphasis on Mojolicious. We'll use Mojolicious::Lite to handle the server duties for this super quick example as well. The key to using Mojolicious is the plugin module called Mojolicious::Plugin::GraphQL, which is a separate dependency you will need to install. Let's take a look at a piece of sample code from a Rosetta Stone-esque blog post here:

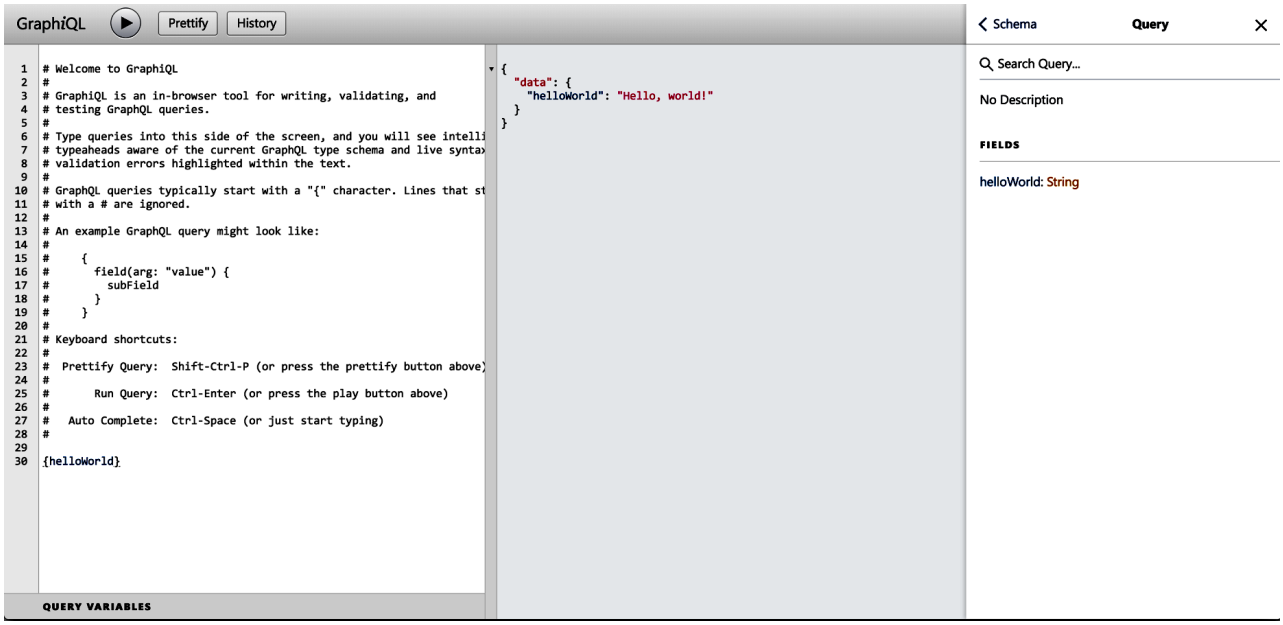


Figure 1: The GraphiQL interface

http://blogs.perl.org/users/ed_j/2017/10/graphql-perl---graphql-js-tutorial-translation-to-graphql-perl-and-mojoliciousplugin-graphql.html.

I call this a Rosetta Stone because this blog post shows the Perl equivalent code for one of the more well-known tutorials whose examples are in JavaScript (https://graphql.org/graphql-js/). Here's one of the code samples from that blog post:

```
use Mojolicious::Lite;
use GraphQL::Schema;
my $schema = GraphQL::Schema->from_doc(<<'EOF');
type Query {
        helloWorld: String
}
EOF
plugin GraphQL => {
        schema => $schema,
        root_value => { helloWorld =>
         'Hello, world!' },
        graphiql => 1,
};
app->start;
```

The first part of the sample includes a definition of a GraphQL schema (a very simple one). The second part loads the GraphQL plugin and sets up the value that will be returned when `{helloWorld}` gets queried. Then we start the Mojolicious event loop and are off to the races.

The one fun part of this plugin shown in the code that I want to highlight is this line:

```
graphiql => 1,
```

GraphiQL is an in-browser IDE that is super spiffy. It allows you to interactively play with GraphQL queries, find errors, see all of the possible objects/fields from a schema, auto-complete them when typing, and so on. When you include this in the plugin configuration as above, it will automatically load GraphiQL for you. So if we start up this code snippet with:

```
$ perl ./test2.pl daemon -l http://*:5000/graphql
[Mon Jun 25 10:43:11 2018] [info] Listening at
"http://*:5000/graphql"
Server available at http://127.0.0.1:5000/graphql
```

and browse to that URL, we see something like Figure 1.

I have opened up the Docs section and clicked through a bit, so you can see that it stands at the ready to show you what's available in the schema. I have also typed something into the left window pane and executed the query, so you can get the full idea from the screen shot.

With this little tip on how to play with GraphQL, I'm going to wind the column down. GraphQL has a bit of a learning curve, but it is some great stuff and there is strong support for it in the community. I hope you'll take a moment to play with it a bit. Take care, and I'll see you next time.

Yes, Virginia, There Is Still LDAP

CHRIS "MAC" MCENIRY

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

With the current trend of adapting web-based single-sign-on solutions, it is easy to forget about one of the most prominent authentication and user information systems still in use: LDAP. At its base, LDAP is a collection of objects that:

1. have a Distinguished Name to identify them,
2. have attributes that follow predetermined schema, and
3. are structured and related to each other as nodes on a branching tree.

The above properties affect how you identify and manipulate them.

Two of the most common implementations of LDAP are Microsoft's Active Directory and OpenLDAP. Microsoft's Active Directory (AD) underpins many corporate infrastructures. While you may not want to use it for all AD operations, AD provides LDAP as a first-class way of searching and modifying objects inside of it. OpenLDAP is commonly found in many open source shops and large cluster installations.

In this article, we will look at two common interactions with LDAP:

- How do you find a user in LDAP?
- How do you add a user to a group in LDAP?

Along with properly assigned group ownership, these two can be used to help users manage their own groups.

To help us out, we're going to focus on the go-ldap library (https://github.com/go-ldap/ldap). In addition to that, we will use the Go Subrepository library for password handling (https://golang.org/x/crypto/ssh/terminal).

Setup

The code for this is found in the `useldap` directory of the GitHub repository (https://github.com/cmceniry/login). It includes Gopkg configurations to pull in dependencies. Both the `search` and the `group` commands are expected to be run with a simple `go run ...` command.

In addition to the code, you will need access to an LDAP server. If you are familiar with LDAP, you can probably modify the examples as necessary for your situations.

If you are new to LDAP, one of the fastest ways to get up and running is to run OpenLDAP as a docker container:

```
docker run --hostname ldap.example.com \
   --name ldap -d -p 389:389 -p 636:636 \
   osixia/openldap
```

Once up and running, you will want to load the included `data.ldif` file:

```
ldapadd -H ldap://localhost \
   -D "cn=admin,dc=example,dc=org" -w admin \
   -f ./data.ldif
```

While there are common conventions that appear between LDAP installs, the specific locations and paths used for objects can vary. In the examples here, we limit our users to the `ou=people,dc=example,dc=org` subtree, and our groups to the `ou=groups,dc=example,dc=org` subtree. If you are attempting the same thing against Active Directory, its structure will depend entirely on your Forest, Domains, and Organizational Unit structures. You may have to change the search filters widely to find the appropriate objects there.

For the sake of brevity, we will ignore TLS in this example. However, if you are using LDAP, you should be using it securely. Luckily, the LDAP Go library referenced here has simple support for TLS. Add the TLS configuration after the `ldap.Dial` calls:

```
err = l.StartTLS(&tls.Config{
   …
})
```

Safely Reading Passwords

LDAP does not maintain a constant session across multiple connections, but does require authentication, known as "binding" inside of LDAP. Our examples are simple command line tools which will create new connections every time that they are invoked. This means that we're going to have to authenticate every time as well. To do that, we'll want a safe and cross-platform way to obtain the user's password. In this case, it is the simple "admin" password, but we should still handle it safely.

passwd.go: GetPassword.

```
func GetPassword() (string, error) {
   fmt.Printf("Password: ")
   pw, err := terminal.ReadPassword(int(os.Stdin.Fd()))
   fmt.Println()
   if err != nil {
      return "", err
   }
   return string(pw), nil
}
```

We begin the above function by asking for a password via our "Password:" prompt. We don't end this `Printf` with a new line in order to preserve it as a prompt. This doesn't change the behavior of it, but it is the common convention for the user interface. The magic comes in the form of `terminal.ReadPassword`, which is the cross-platform method of obtaining input without echoing it back to the screen.

We finish the main prompting with the `Println` for two reasons. First, since `terminal.ReadPassword` disables echo, any new line entered by the user will not be echoed and so the next printed characters will end up on this line. In addition, the `Println` statement resets the echo state of the terminal. Any `Print*` would do, but we are taking out two birds with one stone.

Finding a User

When doing group changes, the first step is to identify the users to be added or removed from the group. Our first utility will help us identify users. In the simple case, we're going to accept a command line option, the name of a user to find, and return the distinguished name (DN) for that user.

We start by getting the admin password using our `terminal.ReadPassword` wrapper. In this example, we're going to `panic` if anything goes wrong.

search/main.go: getpw.

```
pw, err := useldap.GetPassword()
if err != nil {
   panic(err)
}
```

With password in hand, we open our connection to the LDAP server. `ldap.Dial` follows the same form that any of the `Dial` functions do: protocol and hostname:port. After checking for error, we defer closing the connection so that it will properly shut that down when we are finished (probably not needed in this case, but good practice nonetheless).

search/main.go: connect.

```
l, err := ldap.Dial("tcp", "localhost:389")
if err != nil {
   panic(err)
}
defer l.Close()
```

After connecting, we need to identify ourselves. In LDAP terms, this is called binding. Binding takes a distinguished name and a password. Our DN is the LDAP admin account.

search/main.go: bind.

```
err = l.Bind("cn=admin,dc=example,dc=org", pw)
if err != nil {
   panic(err)
}
```

Once fully into the server, we can perform our search with the `Search` method of our LDAP connection. `Search` takes one argument, `*SearchRequest` which is constructed with the general `NewSearchRequest` func. `NewSearchRequest` takes nine arguments:

1. The base DN or section of the tree to search under
2. The scope or how deeply into the tree to search
3. The "Deref" flag to show if there are any objects pointed to
4. The limit on the number of resulting entries to get (this can be further restricted by the server, so the response may not always be the same)
5. The time limit to wait for a response
6. The "TypesOnly" flag to indicate whether to show attributes' names only or names and values

7. The filter to use to search what matches which attributes to select one
8. The limit of the attributes to return
9. The controls that affect how a search is processed (e.g., to support paging of results)

Of these, the most common one to change is the search filter, or what to search for (no. 7), and the second most commonly changed is the base DN, or where to search for it (no. 1). For our example, we want to look only under the `ou=people,dc=example,dc=org` part of the tree and only for those entries where the common name, or `cn`, attribute matches our command line options.

search/main.go: search.

```
results, err := l.Search(ldap.NewSearchRequest(
   "ou=people,dc=example,dc=org",
   ldap.ScopeWholeSubtree, ldap.NeverDerefAliases,
   0, 0, false,
   fmt.Sprintf("(cn=%s)", os.Args[1]),
         nil, nil,
))
```

Now we show the output with three loops. The `results` struct has a primary field, `Entries`, which is an array of all of the returned LDAP objects. We can iterate over the array of objects. Each object has a DN and an array of attributes. By iterating over this array, we can see that each attribute can have multiple values (e.g., multiple `member` attributes for group membership), so we finally iterate over those and display them.

search/main.go: show.

```
for _, r := range results.Entries {
   fmt.Printf("------- %s -------\n", r.DN)
   for _, attr := range r.Attributes {
      for _, v := range attr.Values {
         fmt.Printf("%s: %s\n", attr.Name, v)
      }
   }
}
```

Updating a Group

Once we have the reference to the user object, we can make sure that that is a member of the group. In our second tool, `group`, we're going to accept a DN (**note: not user cn or name**) and ensure that that exists on our mygroup group (i.e., add it if it doesn't exist, or just leave it there if it does).

We start by getting the password, connecting, and binding as we did before:

group/main.go: getpw,connect,bind.

```
pw, err := useldap.GetPassword()
...
l, err := ldap.Dial("tcp", "localhost:389")
...
err = l.Bind("cn=admin,dc=example,dc=org", pw)
```

Modifying an LDAP object with something it already has results in an error. So we first want to check that our user addition doesn't already exist on the group. We perform an LDAP search, but this time on the group.

group/main.go: search.

```
results, err := l.Search(ldap.NewSearchRequest(
   "ou=groups,dc=example,dc=org",
   ldap.ScopeWholeSubtree, ldap.NeverDerefAliases,
   0, 0, false,
   "(cn=mygroup)",
   nil, nil,
))
```

With this result, we iterate through the `member` values and exit out successfully if the DN is already there.

group/main.go: exist.

```
members := results.Entries[0].GetAttributeValues("member")
for _, v := range members {
   if v == os.Args[1] {
      os.Exit(0)
   }
}
```

Once we confirm the addition isn't already there, we proceed to update the group object. Similar to the `NewSearchRequest`, we construct a `NewModifyRequest` that we can feed to `Modify`. The main difference between using the two is that we create the request struct and then add our modifications to it. In this case, we `Add` our DN as a value for the `member` attribute. Again, attributes can have multiple values, so we add the array (even if it's only one value).

group/main.go: modify.

```
m := ldap.NewModifyRequest(
   "cn=mygroup,ou=groups,dc=example,dc=org",
)
m.Add("member", []string{os.Args[1]})
err = l.Modify(m)
if err != nil {
   panic(err)
}
```

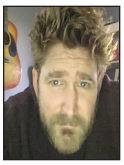
And with that, we've ensured that our user is on the group.

Conclusion

This example shows that Go has the chops to exercise even what many forgot is a common protocol underlying a lot of infrastructures. The above could be done with the appropriate invocations of `ldapsearch` and `ldapmodify`, but we can encode some of our conventions (tree structure, attribute names) and simplify what we must know to achieve our goals. Add to that that we can distribute these tool binaries as single files, and we can provide simple interfaces for our users and ourselves to manage our resources. This is a very useful method to keep operations running smoothly in any organization.

iVoyeur
Flow

DAVE JOSEPHSEN



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

Little Mission Creek roars and tumbles and thrashes against its banks, along with, it seems, every watershed in all of western Montana. The Clark Fork in Missoula, the Gallatin in Bozeman, the Great Missouri River in Helena, and the Yellowstone River a dozen miles south from where I sit—they've all crested their banks and tested their spillways in the last several weeks.

But Little Mission Creek is my home, though I barely recognize the violent torrent it has become this spring. Watching it churn impatiently about my legs, it's easy to forget what an arid place this is. The notion of water-rights and violent disputes over creeks like this one have shaped the landscape here every bit as much as the flowing water itself.

I like to sit here at the bank and attempt to imagine how the water flowing past me now will, in roughly 11 minutes, make its way to the head of our valley and join forces with Mission Creek proper—itself busily running north out of the foothills. How in another 20 minutes the water below me will spill crashing into the Yellowstone River and turn east, running for 150 miles into Billings before turning back north, and joining the Missouri just past the North Dakota border.

That's about the extent of my imagination. I can't really wrap my head around the scope of the journey these H_2O molecules are about to make, but that doesn't stop the water in Little Mission Creek. On it flows, heedless of my cognition and indifferent to my doubts, winding halfway across North Dakota before veering back down south to Kansas City, where it turns east again to join the Mississippi in St. Louis before finally making a 700-mile beeline for the Gulf of Mexico at New Orleans.

That's just inconceivable to me. It seems mythical, otherworldly. Someday I'm going to drive it. I'll plan it carefully, taking small roads as necessary to remain as close to the water as possible. Hopefully, I'll get a tangible sense of it—a concrete understanding of what it means to flow like the water in my creek. When I go, I will take some of my creek water with me in a bottle. I'll carry it to the Gulf like a riverbed on wheels and, like an orphan reunited, return it to the Atlantic at the end of my own journey. I wonder if I'll be giving it a head start or delaying its arrival; or maybe it's the journey that matters, not the destination. Maybe when I get there I'll understand.

Data Lake

At work I'm helping out on a project called "The Data Lake." We're all very excited about it. For example, the other day I got a meeting invite whose description read (I promise I'm not making this up): "Break out your data-paddles, because it's time to go data-canoeing in the data-lake." That's how excited we are. Just absolutely dancing-away-with-the-metaphor-in-public excited.

What on earth is a data-lake?

Great question, I'm glad you asked. The data-lake is just a colorful name for a series of S3 buckets. S3 buckets?! What's so great about a bunch of S3 buckets, you ask? Well, it's not so much the storage medium that's cool as what's stored there, how it's stored, and what we can do with it later by way of a few Python scripts and AWS's Athena service. Have you read about schema-on-read, columnar data storage formats, and the rise of the SQL query engines? If not, prepare yourself, because these are the substrate into which the data-lake is carved.

For the entire length of the history of people interacting with databases so far, we have been mapping our data to a schema at write time. Like anal-retentive scriveners whose very nature prevents us from just writing anything down all willy-nilly, we take the raw data in one hand and a description of what the data should look like in the other, and we combine the two, writing the result to disk in a binary, pre-formatted way. Users can subsequently make queries against the data because we have it stored in a schemafied, normalized, queryable format.

Schema-on-read systems, by comparison, map the schema to raw data at query time. That is, the data is not preformatted—it is not "queryable" in the database-sense. It's just bytes sitting somewhere on disk in its native format (JSON, newline-separated lines of text, whatever...). The schema itself is stored as a set of ETL-like (extract, transform, load) instructions (or even a regular expression), which the query engine can use to map the at-rest data into named fields on-demand. So really, there is no "database" in a schema-on-read system. There is just some metadata linking the location of some at-rest data to a schema we can use to parse it when we want to.

Bereft of a proper database to pamper and worship, users instead interact with a *query engine*. When a user makes a query, the query engine finds the data, maps it to the schema in memory, executes the query on the resultant in-memory data blob, and returns the result. When the query engine speaks SQL (most of them speak a dialect of SQL like Presto (https://prestodb.io/)), we call it, unimaginatively, an "SQL query engine."

Why would you *ever* do that?

I know, if you want a database, use a database, right? Well, databases have their own suite of problems, related mostly to getting data *into them*. Engineers often turn to schema-on-read systems to provide an SQL interface to some vast quantity of already at-rest data that would otherwise be too onerous for a traditional database to ingest.

Say, for example, that there's an S3 bucket containing a yottabyte of raw (un-summarized) monitoring check output for every computer ever owned by some corporation since the beginning of time, and you need to query it. You could spend the better part of a month writing custom ETL and using it to get all that data into MySQL, all just to run a couple of queries and then throw it all away, or you could just point your Apache Drill (https://drill.apache.org/) SQL query engine directly at the data.

This sort of ad hoc access to analyze ponderously huge data sets stored across a widely distributed medium is the bread-and-butter use-case for schema-on-read, but those of us who struggle with a preponderance of monitoring data might also find a compelling story herein.

Imagine that you could just flip a switch and enable SQL querying of all of your organizational Nginx logs. What a treasure trove it would suddenly become for managers, engineers, account managers, support personnel...anyone able to formulate an SQL query. A common, self-service interface for anyone with questions like, "What was the 99th percentile response time on the /accounts endpoint this morning?" or "How many people signed up last month?" And you can make it happen without any of the headache of ETL, provisioning, scaling, and managing a proper database or rolling an ELK-style log analysis system.

That's pretty much the data-lake concept in a nutshell: a low-maintenance, self-service SQL interface into timely operational data that you just happen to have lying around anyway.

Keeping Things Low Cost

For the data-lake, our chosen SQL query engine is an AWS-hosted service called Athena (https://aws.amazon.com/athena/). It's easy to use, wholly hosted, and it obviously works flawlessly with data stored in S3. You only pay for the queries you make, but here's the rub: it costs $5 per terabyte of data scanned by each query.

How is THAT going to work?!

I know. You have a LOT of data (so do I). So the game becomes a process of getting the answers you need from the data set with the minimum amount of actual reading data. There are two hacks that make our data-lake cheap enough that so far, we aren't worried about restricting access to it.

The first is data partitioning. It's possible to write the data to S3 in chunks, labeling these in such a way that Athena can intuit the chunk names. A very common partitioning scheme, which many log-writers support without even knowing it, is partitioning by year/month/day. Simply write the data to S3 using year/month/day "directories" (there aren't really any directories in S3), and identify these to Athena as partitions. Then make queries like this one, which I just used to count the number of API calls made by a customer in the first 10 days of April:

```
SELECT COUNT(*) FROM "data-lake.nginx-json" where
"customer_id"='1234' AND "partition_0"='2018' AND
"partition_1"='04' AND "partition_2"
in('01','02','03','04','05','06','07','08','09','10');
```

It seems stupid-obvious, but without partitioning you'll too often find yourself reading more data than you want. You can read more about data-partitioning for Athena at the AWS support site (https://docs.aws.amazon.com/athena/latest/ug/partitions.html).

The second hack to minimize the amount of data you scan is to use a columnar data storage format. I know I said you didn't need to pre-format your data, and you don't. But if you're building a semi-permanent log-query solution on Athena, like we are, and want to save a considerable amount of money, I'd highly recommend running your queries against a columnar-transformed copy of your logs.

So what's a columnar data store? Well, start by imagining a typical database as a spreadsheet, where you have a row of headers followed by rows of data records and where each column represents a schema entry in that data record. You know what it looks like:

```
first, last, middle, num, street, state, pet
dave, josephsen, j, 11, may street, MT, cat
jill, gomez, f, 114, epic road, CA, goldfish
jose, cardona, r, 210, turbine ct, TX, hedgehog
```

A columnar data store is pretty much a broken spreadsheet. We take all the column entries and store them on top of each other, along with a small header which maps the line numbers of each column. I'm simplifying things for instructional purposes but it basically looks like this:

```
first 1, last 4, middle 7, num 10, street 13, state 16, pet 19
dave
jill
jose
josephsen
gomez
cardona,
j
f
r
11
114
210
may street
epic road
turbine ct
MT
CA
TX
cat
goldfish
hedgehog
```

Now imagine what happens when I make a query like

```
select * where middle="j"
```

In a traditionally laid out record-per-line text file, the query engine needs to traverse and parse essentially the entire file, ingesting each record to search for the `middle` field, string compare it against j, and then return the whole line if it matches.

With a columnar format, we can use index numbers to look around rather than scanning the data itself. First, we parse the header for the line-number of the `middle` field, and then we simply seek directly down to line 7, comparing just the individual bytes from each record's middle column, and return the matching records. The records we can also reconstruct from line-number offsets without having to actually scan the data (e.g., the offset between line 7 and the matching record (line 7) is 0. So we reconstruct the entire record by walking the header and forming the union of lines 1+0, 4+0, 7+0, and so on...).

This is a *way* more efficient means of querying data, which translates to both faster responses and smaller Amazon bills.

Flows

Okay, so what have we learned?

- Schema-at-read query engines can effectively query at-rest data using SQL-like syntax.
- Most watershed from western Montana winds up in the Gulf of Mexico.
- You can roll your own query-engine with something like Apache Drill or use a hosted one like Amazon Athena.
- You can query raw data but it's expensive and slow.
- If you transform it into structured data and store it in a columnar format like Parquet (https://parquet.apache.org/), things get orders-of-magnitude faster and cheaper.

But how do we get our log data from text files on individual server instances into Parquet-formatted data in the data-lake? Well, a detailed description of our ingestion pipeline will have to wait until next time, but the short answer is—rather obviously—it *flows* there. Nginx to Rsyslogd to Fluentd to Kinesis to EMR, like rivers winding, maybe our data-lake metaphor isn't really as absurd as it sounds at first. Deeper than a pond and yet perhaps not so final a destination as an ocean, our humble Data Lake is already solving pretty big observability conundrums for us internally, so maybe our excited overuse of metaphor is similarly justifiable. Anyway, grab your hip-waders, because next time we'll wade into the stream and measure the flow.

Take it easy.

For Good Measure

Numbers Are Where You Find Them

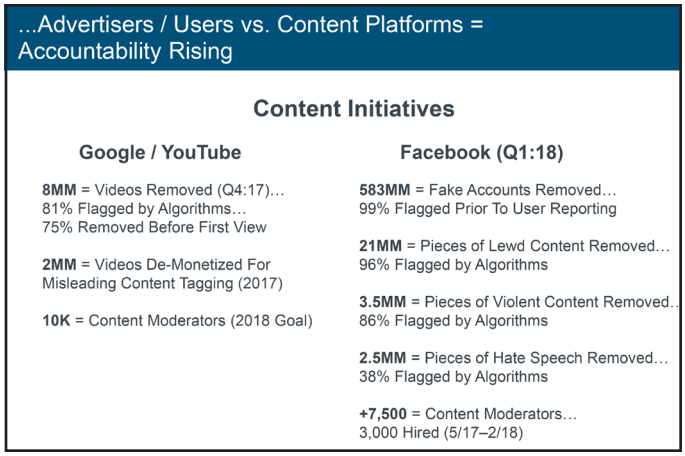
DAN GEER

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

It will come as no surprise that 50, or even 20, years ago inquisitive minds were often at a loss for bodies of numbers upon which they could rely. Putting aside the precise meaning of "rely" for the moment, a shortage of numbers is less and less a reason for inaction in any domain. Just as obvious as the sunrise, soon enough the issue will be too many numbers. Sensors, radios, and AI, oh my.

Security metrics study is possibly out in front of some other fields, but only so much and likely not even that for much longer. The idea that managing a risk requires measuring that risk or its precursors has long since become standard operational thinking in the security game, yet we are living proof that while collecting numbers is necessary, it is not sufficient to deliver security.

Some would argue that it isn't our tools and our scorekeeping (with numbers) that is the "thing" that is not sufficient—rather, it is incentives that are wrong. Whole conferences are on this topic, and there is no way to summarize them in the context of this column, but study of, and suggestions for, incentive structures, be they rewarding or punishing, are surely needful. As an example, the organizers of the Code Conference (CodeCon) said, "[For] 2018, we invited the people in charge of enforcing regulations, and those creating new ones." That's a stab at incentive structures to be sure, but let's specifically look at some numbers from Mary Meeker's "Internet Trends 2018" slide deck at CodeCon, beginning with Slide 99 [1].

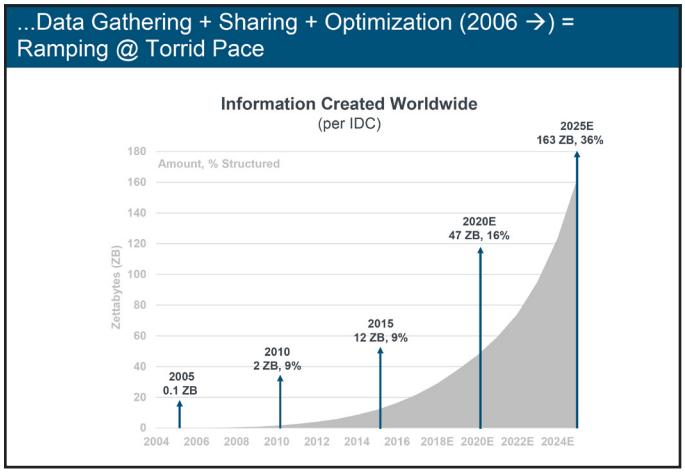


Content initiatives, Slide 99

Note the role of algorithms in the above, which, for the purpose of this column, we will take as a form of security metrics even if the algorithms in question are not open for inspection. Algorithms as censors is a worthy topic in its own right.

With Google/YouTube, that 81% were flagged by algorithms presumably means that the average Content Moderator sees those algorithms as automated assistance to making faster

Online prices are falling, Slide 111



Projection of global information creation, Slide 189

decisions. A month before Meeker's speech, *The Guardian* [2] said this about the algorithms, per se:

> Those systems broadly work in one of three ways: some use an algorithm to fingerprint inappropriate footage and then match it to future uploads; others track suspicious patterns of uploads, which is particularly useful for spam detection. A third set of systems use the company's machine learning technology to identify videos that breach guidelines based on their similarity to previous videos. The machine learning system used to identify violent extremist content, for instance, was trained on 2 million hand-reviewed videos.
>
> While machine learning catches many videos, YouTube still lets individuals flag videos. Members of the public can mark any video as breaching community guidelines. There is also a group of individuals and 150 organisations who are "trusted flaggers"—experts in various areas of contested content who are given special tools to highlight problematic videos. Regular users flag 95% of the videos that aren't caught by the automatic detection, while trusted flaggers provide the other 5%. But the success rates are reversed, with reports from trusted flaggers leading to 14% of the removals on the site, and regular users just 5%.

Facebook's use of algorithms is undoubtedly similar to that of Google/YouTube.

But measurement is not a problem just for us here in security metrics land; take something as important as economics. Everyone knows something about the Consumer Price Index (CPI). As Wikipedia puts it, "In most countries, the CPI, along with the population census, is one of the most closely watched national economic statistics." Yet even the calculation of the CPI is having trouble these days, as Slide 111 shows.

Think of the spread of things that the CPI is baked into, from labor contracts to entitlements to financial instruments to you-name-it. Surely the CPI is easier to measure than security, but here we are.

Of course, everyone knows that the world is creating lots of data. Defining "structured" data as "data that has been organized so that it is easily searchable and includes metadata and machine-to-machine (M2M) data," we have (by way of the market intelligence firm IDC) the curve you see in Slide 189.

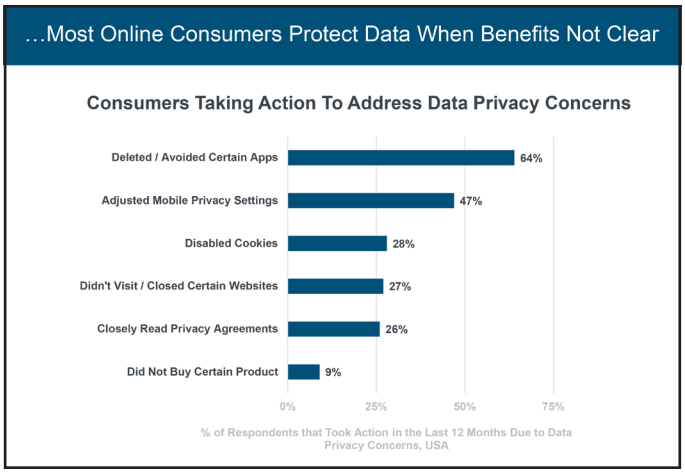
For those of us working in data protection, the message is obvious—data protection must be automated; the algorithms have to make the "kill decisions." And other algorithms will have to summarize things for us, summaries that will be ever more distant from the raw numbers.

Putting aside the argument over whether security and privacy are mutually supportive or fundamentally at odds, Slide 206 has a few somewhat encouraging numbers that consumers are at least thinking about it:

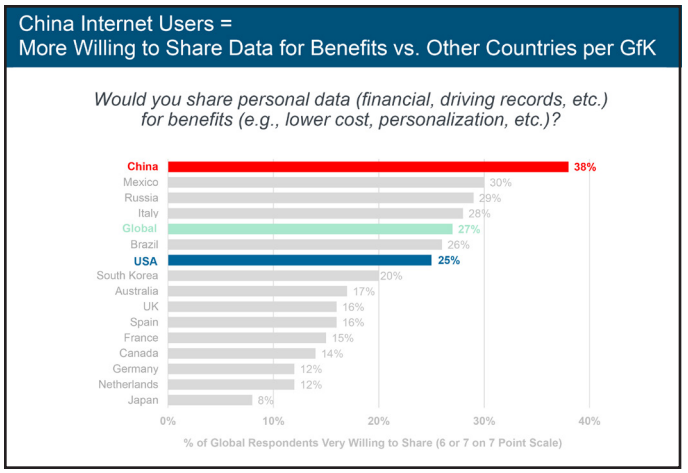
On the other hand, trading short-term gain for long-term risk is still blithely popular, but, as measured by the German marketing firm GfK, blitheness is culturally diverse—see Slide 223.

That one slide, Slide 223, probably says more than we know how to evaluate both as to privacy (the question GfK actually investigated) and to security (as in risk/benefit tradeoffs generally). Later on (Slide 266), the founder of Slack hits the nail on the head: "When you want something really bad, you will put up with a lot of flaws." We, the global "we," want our toys ever harder, ever faster. There've been a lot of demonstrations of that phenomenon, but let's use *The Economist*'s numbers in Table 1 [3].

To get adoption rates accelerating like that a lot of flaws must be put up with, security flaws in particular, one might presume,



How consumers address data privacy concerns, Slide 206



Willingness to share data by country, Slide 223

since security is so generally a feature worth adding only after there is good consumer uptake.

So where does this get us? It is not as if anyone needs to be told that things are changing faster than we understand. It is not as if the present author's selected numbers from the "Internet Trends Report" are unbelievable individually, but collectively they predict a world where prediction (as we humans understand the term) is less and less possible because of the number of moving parts, their opacity, their interdependence, their cardinality, their specificity of purpose, their autonomy, their speed. Clausewitz would no doubt call this a deepening fog of war. Modern military doctrine trades off precision and certainty for speed and agility, or, as Army Chief of Staff Gen. Mark Milley says [4], "On the future battlefield, if you stay in one place longer than two or three hours, you will be dead." Is that not the future of cybersecurity in a nutshell?

Years until used by one-quarter of American population	
46	Electricity
35	Telephone
31	Radio
26	Television
16	Personal Computer
13	Mobile Phone
7	The Web

Table 1: Technology adoption

References

[1] M. Meeker, "Internet Trends 2018": http://www.kpcb.com/file/2018-internet-trends-report.

[2] A. Hern, "YouTube Reveals It Removed 8.3m Videos from Site in Three Months," *The Guardian*, April 23, 2018: https://www.theguardian.com/technology/2018/apr/24/youtube-reveals-it-removed-83m-videos-from-site-in-three-months.

[3] "Happy Birthday World Wide Web," *The Economist*, March 12, 2014: https://www.economist.com/graphic-detail/2014/03/12/happy-birthday-world-wide-web.

[4] S. Freedberg, Jr., "Miserable, Disobedient and Victorious: Gen. Milley's Future US Soldier," *Breaking Defense*, October 5, 2016: https://breakingdefense.com/2016/10/miserable-disobedient-victorious-gen-milleys-future-us-soldier/.

/dev/random
No Bots About It

ROBERT G. FERRELL

Robert G. Ferrell is a fourth-generation Texan, literary techno-geek, and finalist for the 2011 Robert Benchley Society Humor Writing Award. rgferrell@gmail.com

The term "artificial intelligence" has been lobbed about in the semantic tennis match we call the Internet so often over the past decade that I don't think it retains any real meaning for most of us. Our TVs, watches, doorbells, thermostats, toasters, and cars are called "smart" now, and "smart" is another word for "intelligent," so "artificial intelligence" means we can control the crispness of heated bread from the shower. Let us not forget that one of the definitions of "artificial" is: pretended; assumed; insincere. From that perspective, I would argue that much of the recent egregious behavior of our government officials could be termed "artificial intelligence," although whether the "intelligence" part applies at all is debatable. Maybe "artificial leadership" is more apropos.

I've written before (ad nauseum) on the somewhat irrational fear of technologists that the machine singularity will automatically lead to the inevitable extinction of the human race at the appendages of our cold, unfeeling robot overlords. No, if the machines do in fact take over, it won't be mechanoids or automatons or network-controlled front-end loaders with unconstrained bloodlust that carry out the executions, it will be us. Humanity. We will off ourselves, and we'll do it because bots drove us to do it.

"Bots!" I hear you sneer, rolling your eyes. What kind of threat are bots? What are they going to do, index your website without permission? Steal your CPU cycles to mine digital currencies that may or may not have any actual value at any given moment? Inflate your popularity on Instagram? I shake my head sadly at your naively myopic techno-worldview. There is so much more bot-related ruckus to be raised, my friends.

Bots, not you, control what you see and do on the Internet. Really. Reactive content, for example—that is, content generated based on current events and news items—is deeply dependent on bot activity. If bots generate ten million views for some useless doodad and you happen to fall in the fake demographic it was spoofing, ads for that doodad are going to get displayed prominently on your social media account, even if nothing in your actual profile suggests you'd have any interest in doodads. If you try turning them off, you'll get a stern warning that ads cannot be turned off without risking the complete collapse of all the world's economies. Do you really want that on your conscience? Just buy a stupid doodad, for Pete's sake. Then, of course, be prepared to see ads for the exact doodad you just bought for several weeks because bots hate you.

Not only do bots determine what ads you'll see or music you'll listen to or videos you'll watch, there is research to suggest they may even control your basic perception of reality. "Emotionally volatile users" (also known as "Everyone on the Internet") are particularly susceptible to manipulation by the malicious misapplication of personal data. Most people surrender a ridiculous amount of information about themselves to social media sites, and then act all surprised and betrayed when that data is used to target them. Boo hoo.

/dev/random: No Bots About It

Why did you think they kept nagging you to fill out that profile, patting your virtual head as positive reinforcement for every scrap of privacy you gave up? Did you believe Facebook just wanted to make sure to buy you the perfect birthday present? Or that maybe you were in the running for Who's Who among Gullible Computer Users? Every time you accept the invitation by some new application to make use of the "convenience" of logging into it via a social media account, you've just stripped yet another layer off the already pea-sized onion of your privacy.

"Live chat" bots are one of the more ironically named primary growth industries in the bot landscape. Most observers classify them as "benign," but benign tumors can still mess you up, believe me. These chatbots' ostensible purposes are to help you find things on a website, place orders, or engage in some other automatable customer service activity. Just remember that anytime you interface with a bot, you have no real guarantees as to what information that code might be collecting on you from places like your browser history or various caches. Oh, you told it not to look at any of those? Well, that's all right, then: no piece of software has ever been used to engage in duplicitous activity. Crisis averted.

Even without that level of intrusion, the answers you give to its questions will be used to flesh out your all-important marketing target profile. Some of them are subtler about this collection process than others. If the live chat bot you're talking to while getting tickets to the theater starts asking you what kind of socks you wear or whether you prefer stick to roll-on, you have stumbled upon one of the less-subtle varieties.

It is poetic justice to me, then, that a lot of the information supplied to potential advertisers by the various harvesting bots is downright erroneous. Some studies have shown that as much as 60% of all reported ad traffic stems from click fraud perpetrated by bots. Those 2.5 million views of your ad last month? Only six of them were by actual living human beings. Sorry. Would you like to file a complaint? We have a live chat bot for that. It's a good listener and hardly ever interrupts with profanity. And it has 1.2 million likes.

All of this is well and good, you're probably muttering to yourself, but how does any of it contribute to the thesis that bots will be responsible for our downfall as Earth's dominant land-based species? To answer this, let us turn once again to our old nemesis, social media. Is it a coincidence that the generation on whose shoulders humanity's hopes and dreams squarely rest can't bear to be parted from their social media for even one moment or they experience full-on withdrawal? I say it is not. I say the bots have positioned us, and themselves, right where they want us. There's a reason suicide rates have gone up, and it isn't fluoride in the water.

Mood swings, depression, hopelessness, frustration…what's the source of all this negative baggage? In my day it would have been a combination of bills, bad news, academic/job disappointments, errant romances, and possibly a car that doesn't run. These would have been woes of more or less discrete origin, however, deriving largely from face-to-face imbroglios. Digital technology has amalgamated the disparate elements of your misery today and served them up as a homogeneous, quivering mass of shock gelatin.

The genius inherent in this approach is that you can no longer treat one of the symptoms to improve the disease, any more than putting new tires on your car will make you more satisfied with the job to which it conveys you every morning. The bots who feed you your every emotion have seen to that. Do I sound paranoid? It's not me talking, it's the schizobot who intercepts my keystrokes.

You could, of course, avoid bots to some extent by skipping out on the Internet altogether, but if that isn't practical you can try my tactic: fibbing shamelessly. I fill out every survey, answer each and every question I am asked, with complete and utter fiction. I suspect my data is probably archived by cryptozoologists and alien hunters worldwide. After all, I'm a 262-year-old gender-fluid goblin entomological proctologist named Mortallica Lazarkolun who enjoys heavy water sports and harbors a penchant for deep-fried lymph nodes (with fat). Fantasy novelists: we have our uses.

Book Reviews

MARK LAMOURINE AND RIK FARROW

UNIX and Linux System Administration Handbook, 5th Ed.

Evi Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley, and Dan Mackin
Pearson Education, 2018, 1180 pages
ISBN 978-0134277554

Reviewed by Mark Lamourine

There are few books that I would recommend to every working sysadmin at every level of ability and at any point in their career. This is one. I'm going to refer to it merely as [Nemeth5] to avoid writing the whole title repeatedly. This is also a deliberate tribute to Evi Nemeth, one of the original authors and a pioneer of learning and teaching system administration as an art and profession.

I've owned all five editions of [Nemeth] as soon as they were released, and I've learned or re-learned something from each of them. [Nemeth4] was the last one that Evi worked on. Evi was crew on the *Niña*, a 50-foot sailing yacht that went missing in the Tasman Sea in 2013. In [Nemeth5], the remaining co-authors have maintained the range and quality of the previous editions.

From the first edition, [Nemeth] has been a *handbook*. Although it's big and has fine paper pages, it is meant to be kept close and thumbed through often. A handbook doesn't have the narrative of a tutorial or the depth of a topical reference manual.

When the first edition appeared, the World Wide Web didn't exist. Today you can find everything in [Nemeth5] through a search engine. The paper book has one often overlooked advantage: compactness. By this I mean that all of the of the searching and sorting and question refinement has been done for you, the reader. All you need to do is flip to the table of contents or the index to find what you need.

Each edition has been based on a set of four or five currently popular vendors or distributions. For the 5th edition, the authors chose FreeBSD and three flavors of Linux: Debian, Ubuntu, and CentOS. This selection is broader than it seems because these generalized distributions are commonly used as a base for more targeted flavors. Users of these derivative distributions will still find a lot of value here. The authors make note of another 10 distributions, including their strengths and their relationships to the selected core set.

Nemeth et al. have never been shy about expressing an opinion, and you'll find a lot of it here still, though usually couched in wry humor. For example, a paragraph comparing boot time init systems is entitled "inits judged and assigned their proper punishments." The authors address all of the options that a reader is likely to encounter. Their goal is to assist the reader, but they don't feel the need to appear impartial in their evaluation of the tools they're describing.

I can't possibly enumerate all of the sections and topics the authors cram into this two-inch-thick tome. Instead, I'm just going to note a few of the things that caught my attention as new or interesting as I leafed through.

The first item that I came across was a scheduling-tool alternative to cron. I've worked with systemd since it was introduced in Fedora, but I've never seen systemd timers before. It may or may not replace cron, but it certainly presents an alternative, offers much finer control, and allows explicit sequencing capabilities with other systemd controlled events.

The section on scripting I would recommend to beginning system administrators even over most books on the topic. The authors give very good advice on style and approach. They describe and provide examples for all of the most critical language features and a number that are more obscure, useful, and commonly overlooked. The chapter concludes with brief introductions to both Python and Ruby. While it may be true that one can be a good system administrator without programming, I would claim that anyone would find the job easier with some skill in scripting.

When the 4th edition was released in 2010, cloud computing was in its infancy. Modern containers were introduced with Docker in 2013. [Nemeth5] includes a complete chapter on commercial cloud service concepts, providers, and a few examples. It includes chapters on virtualization and containers. None of these are deep, but they are broad and touch on all the important ideas. The writing is clear, without any of the hyperbole or misplaced enthusiasm that is common in dedicated books. Like most chapters, these end with a list of external references and suggested reading.

I don't think I can express just how encyclopedic this book is. It has one of the best technical introductions I've seen on DNS and DNSSEC; a fairly complete examination of SMTP interactions; reasonable default configurations of Sendmail, Exim, and Postfix; and good comparisons of recent CM tools such as Ansible and Salt. Puppet and Chef get mentioned, but they're not the cool kids now, apparently. See: Opinions. The book closes with a chapter on datacenter management that includes the merits of various DC layouts, with example floor plans. I've left out nearly

half of the topics in the book and haven't touched on the obscure-but-valuable little knowledge tidbits sprinkled in every section.

Throughout, [Nemeth5] is readable and accessible. It's perfect for either thumbing through or finding just the start you need on any topic. It is an ongoing tribute to Evi and her lifetime of work.

Accelerate: Building and Scaling High Performing Technology Organizations

Nicole Forsgren, PhD, Jez Humble, and Gene Kim
IT Revolution Press, 2018, 256 pages
ISBN 978-1-942788-33-1

Reviewed by Mark Lamourine

Finally, someone applies science to the Agile/DevOps practice.

For years the Agile/DevOps movement has had only hype, surmise, and anecdote to support a counterintuitive idea: that giving individuals more agency in their work with less managerial gatekeeping (along with the tools to detect and respond rapidly to problems) leads to faster, better, more reliable software and services. That's not to say that there was no evidence. The anecdotes are many, and, when done well, numerous informal case studies have made people confident that there's something to the ideas. Agile practice is derived from the Toyota Production System (TPS), first formally described in 1988 [1]. The advantages of TPS are backed by strong commercial and academic research, but TPS is designed for manufacturing production, and it is not a given that it would translate trivially to software development. Some additional confirmation is needed.

In *Accelerate*, Forsgren et al. have applied modern sociological methods, first to define and then to measure the effectiveness of Agile practices in software development and service delivery. You won't learn how to run a scrum stand-up or use a Kanban board (unless you follow the references in the bibliography). What you will find is the first real demonstration that Agile practices, writ large, are effective, and specifically which aspects have the most demonstrable benefit. They also show proper recognition that there is more work to be done to design and implement good practices and to keep learning how to measure and evaluate them.

Forsgren et al. provide the three elements you expect in a peer reviewed paper, but in a narrative form that non-academic readers will fine comfortable. Don't let the form put you off. If you can read a good technical reference, you can follow their exposition and arguments. If you have read any RFCs, this will be a breeze.

The reason for the somewhat different presentation from most other books in this arena is that *Accelerate* is based on the same data set that the authors used for two peer reviewed papers [2, 3] in 2016, and continue to use for more recent papers. You can find references to the ongoing work on their website [4].

They begin by defining the question under study: What is meant by "Agile practice"? What is meant by "effectiveness" and how do you measure it? The first third of the book defines what her team will try to measure and what they will not.

In the middle section, they lay out their findings so far. They start by describing their data collection methods and briefly justify the use of sociological models before proceeding to explain the data and what it means.

The final section is the most technical. Here the authors explain the methodology and models that they chose to use when gathering the data. They justify the selection and design of the models, questions and data analysis in terms that will be familiar to anyone in the social sciences. They go further, to explain briefly the more technical terminology and offer references for those who want to learn more about the details.

There's nothing earth-shattering in the results. This is primarily because that's not what they were trying to do. The metrics they chose for software delivery quality are deliberately constrained: software delivery rate, delivery lead time, software failure rate, and time to fix. Then they surveyed a broad range of people and companies using varying degrees of Agile methods in their software development. They hoped to test for correlations between those who use Agile methods and those who achieve high marks in their quality metrics.

In general, they find that "higher quality" as defined by their metrics are associated with groups and companies that conform to Agile tenets. A couple of things that raised an eyebrow: the best performers commit directly to a master SCM branch and have no or very short term development branches. The change in lead times in the highest performers were measured in hours. I'm not so much skeptical of these findings as I am wondering whether the strict definitions required for a good metric are out of line with my colloquial understandings of these terms.

What most worries me is the possibility that the metric definitions are in some way forming a logical circle with the Agile methods supposedly under test. The descriptions of the metrics align fairly strongly with my understanding of the purpose and goals behind Agile philosophy and methods design. Is it possible that what is being measured is the effectiveness of the methods used to achieve the stated behavioral goals, without ever evaluating whether those behaviors actually improve the software being delivered? I'm inclined to view Agile methods as indeed effective and beneficial. I'm not sure how to show something stronger than "They do what they claim, and aim, to do."

In every case, Forsgren et al. are careful to qualify any claims. There are muddy and unanswered questions. There are anomalies in the data that need explanation and resolution. Mostly, we need more data and a longer history to work with. Over time the

metrics will, I hope, be broadened and refined. A larger longitudinal data set will make trends more evident and the conclusions more sound. This is what distinguishes research from advocacy.

This isn't a book for the beginning coder. It's not even for most people who are already faithfully attempting to use Agile methods (or not). *Accelerate* is for the doubters who need evidence to show that Agile methods aren't merely a buzzword fad, and for developers and managers who might need some talking points when trying to pitch or improve the software development practices where they work.

There's still a lot of work to be done to find the best ways to manage software development and delivery, but we have a foundation on which to build.

References

[1] T. Ohno, *Toyota Production System: Beyond Large-Scale Production,* 1st Edition (Productivity Inc., 1988).

[2] N. Forsgren, A. Durcikova, P. F. Clay, and X. Wang, "The Integrated User Satisfaction Model: Assessing Information Quality and System Quality as Second-Order Constructs in System Administration," *Communications of the Association for Information Systems* 38 (2016), pp. 803–839.

[3] N. Forsgren and J. Humble, "DevOps: Profiles in ITSM Performance and Contributing Factors," in *Proceedings of Western Decision Sciences Institute (WSDI) 2016*.

[4] https://devops-research.com/research.html.

The Computer Book: From the Abacus to Artificial Intelligence, 250 Milestones in the History of Computer Science (Sterling Milestones)

Simson L. Garfinkel and Rachel H. Grunspan
Sterling, 2018, 528 pages
ISBN 978-1454926214

Reviewed by Rik Farrow

I got to see a proof of this book, coming out November 2018, and have to say I was pleasantly surprised. Not that I didn't expect Simson, both a friend and someone who has written many books, to succeed at this task. But because in reading the book, I kept saying to myself, "Damn, that's how these events fit together."

The authors chose 250 milestones, ranging from a clay tablet abacus to artificial general intelligence—not that we are anywhere near that. The format of the book consists of a page of text on the left with a color photo or illustration on the right. The photos always add some real context to the page of history, besides often being beautiful (the Babbage replica) or just plain interesting. The writing is concise, as it must be to fit on a single page, but always held my attention and was easy to read. Each page ends with cross-references to other pages.

One of the benefits of reading this book is that I learned about the origin of many of the technologies and the terms we use. While some were obscure, like the meaning of "RS" in RS-232, others were real eye-openers.

While this book won't help you with programming or sysadmin, it is a lot of fun to read. And I guarantee that you will find lots of surprises, whether you read it cover-to-cover or just pick it up and flip through the pages at random. And finally, for a book full of photos, the price is very reasonable.

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

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

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

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

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

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

USENIX Board of Directors

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

PRESIDENT
Carolyn Rowland, *National Institute of Standards and Technology*
carolyn@usenix.org

VICE PRESIDENT
Hakim Weatherspoon, *Cornell University*
hakim@usenix.org

SECRETARY
Michael Bailey, *University of Illinois at Urbana-Champaign*
bailey@usenix.org

TREASURER
Kurt Opsahl, *Electronic Frontier Foundation*
kurt@usenix.org

DIRECTORS
Cat Allman, *Google*
cat@usenix.org

Kurt Andersen, *LinkedIn*
kurta@usenix.org

Angela Demke Brown, *University of Toronto*
angela@usenix.org

Amy Rich, *Nuna Inc.*
arr@usenix.org

EXECUTIVE DIRECTOR
Casey Henderson
casey@usenix.org

The Big Picture

Liz Markel, Community Engagement Manager

Summer and fall are my favorite seasons, with my current preference being determined by the weather of the moment; both are equally dazzling where I grew up in New England. Consequently, I was thrilled to find myself in Boston for USENIX ATC '18.

Every visit to Boston as an adult is an opportunity to discover this city through fresh eyes. My opinion about the city evolves with the pursuit of activities I now enjoy, such as bicycle rides along the Charles River. Participating in ATC as a still-new USENIX employee whose background is not in computer science also offered an additional perspective on a previously unfamiliar part of the landscape: the advanced computing systems research space and the tech industry in and around Boston.

In addition to broadening my perspectives on the constituencies that USENIX serves, ATC was an opportunity to observe the research side of USENIX following my exposure to the practice-focused side at SREcon18 Americas. I also met more of our amazing volunteers, including the newly elected Board of Directors, the multitude of program committee members, and the LISA18 organizing committee. Interacting with these groups made me appreciate the diversity of our leadership teams at USENIX, who represent a wide variety of sectors, genders, backgrounds, and experiences. I saw this diversity reflected in our conference attendees as well. It seems that we are at the leading edge of the social and community aspects of computer systems research and engineering just as we are with the technical content that comprises our programs and talks. Furthermore, where

we are and where we are headed—driven by a thoughtful and strategic decision-making process—is consistent with our values as an organization, as well as my personal values about equity, opportunity, and the value and richness diversity brings to life.

All of this talk about the people at USENIX conferences is a perfect segue to important news about an upcoming survey that offers you an opportunity to share valuable information with us and express your opinion about:

- How effectively you feel USENIX is implementing its mission
- Your thoughts about trends in your field
- Your story: how your professional career evolved, what role professional development has played in that progression and growth, and how USENIX has been involved in both
- The ways you're interested in engaging with your peers and with USENIX
- What value USENIX membership offers you

As a nonprofit organization, USENIX exists to serve you and your colleagues across the advanced computing systems profession. The information you provide through your survey responses will help us understand the needs of both the broad computing systems community and those of different segments, such as different generations and different genders. When we have a clear picture of your needs, we can better serve you and more effectively fulfill our mission.

The questions in the survey are a blend of data that we'll gather year over year to track trends as they're in development, plus questions that are pertinent to strategic and operational decisions we'll make within the next 12 to 18 months. Many questions are quick, but others are open-ended inquiries that may require a few minutes to answer. We hope that you'll take the time to answer all of these questions thoughtfully: your investment in this survey will be met with an equal investment on the part of USENIX staff and volunteer leadership to convert this data into actionable items.

Keep an eye on your inbox for a link to the survey coming in late August. If you have feedback or ideas beyond what's asked in the survey, I would love to hear from you at liz@usenix.org. I've long believed that alone we'll go faster, but together we'll go farther. I am looking forward to traveling that road with you.

Meet the Board

Meet Amy Rich, one of the new members of the USENIX Board of Directors. Liz Markel asked Amy a few questions about her professional activities, her personal interests, and her relationship with USENIX to help you get to know her better.

Tell me about your current role and what kinds of problems you're working on solving.

I'm one of the engineering directors at Nuna, Inc., a healthcare technology company in San Francisco that builds data platforms and analytics solutions to help healthcare industry decision-makers understand cost and quality trends. With the insights we provide, they can make changes that increase access to effective, affordable care.

I lead the organization called Foundational Engineering, which includes Infrastructure, Security, and IT. Together, those three teams provide operational resilience through continuous integration and deployment, security, developer productivity, and system user support and support for the entire company.

Each of these teams has a specific focus, but the overall problem they're all trying to tackle revolves around transitioning from a startup to a young company. We walk the fine line between being agile and fast enough to find a product fit in an emerging market while still being secure and not accumulating so much technical debt that it significantly hampers our progress. Because we're a healthcare company that performs significant work for the Medicaid and Medicare arm of the Federal Government, we also have a number of regulatory and compliance constraints to add to the mix.

Are there any emerging trends you're observing in your field?

When it comes to US healthcare, one of the most prominent trends over the last decade has been the move toward payment delivery models oriented towards value—that is, rewarding healthcare providers when they deliver high-quality, affordable care, instead of paying the same for all care, regardless of whether it was cost effective or achieved the desired outcome. To do this right, the government, and companies who provide healthcare for their employees, need good data platforms and analytics with which to measure the cost, quality, and experience of care, as well as to administer these new payment models. Significant investments have also been made to modernize the Medicaid program in its structure, policies, and information systems. This year we finally celebrated the final US state's conversion to submitting digitized healthcare claim records to the Federal Government! The emerging popularity of the fields of data science and machine learning is a huge benefit to programs like these and is one of the ways we can create systems to change US healthcare for the better.

When it comes to infrastructure engineering and IT, most things are moving towards the cloud, automated continuous integration and deployment, and serverless where workloads permit. We won't escape the need to understand and run the infrastructure anytime soon, but the ways in which we do so are drastically changing. Cloud has lowered the barrier to entry for a number of small companies who don't have dedicated infrastructure/security/IT roles, but the complexity of abstraction and scale almost always results in needing those folks after an initial minimum viable product phase. DevOps and SRE have become hot topics with different meanings, depending on who you talk to, but we've come back around in the cycle of trying to more closely align the developers and operations people, if

not looking for both sets of skills in one individual.

Regarding security, with the ubiquity of the Internet and people being online on both their work and personal devices 24x7 also comes the ubiquity of personal and corporate information theft. Security is more important than ever, and yet is neglected, as businesses try to move fast and be disruptive in creating the next big thing. Even more so than DevOps and SRE, there are few qualified people in the field, making those with the skills extremely sought after.

How were you first introduced to USENIX?

As an undergraduate work-study student in college, I had far more interest in system administration than programming. I was responsible for helping run the servers in the Computer Science department and also moonlighted in various other computer labs on campus. As my senior thesis, I chose to perform a risk analysis on our CS department systems and write up the results. I argued that attending the 1994 USENIX Summer Technical Conference (conveniently held nearby) would provide significant background and training in my field, help with my degree and thesis, and also directly benefit the department since I could immediately apply the skills I learned. I presented a good enough case that they paid for a student ticket and subsidized my hotel for half the week.

What involvement have you had with USENIX?

For the most part, I've always preferred to work behind the scenes than be a presence on the stage. In the early days, I volunteered with the registration desk to help offset the cost of attendance to USENIX ATC and LISA. After several years, I was well-known and respected enough to be asked to volunteer as a paper reviewer for the LISA program committee. That eventually led to being the LISA Invited Talks Co-Chair, the LISA Program Co-Chair, and then a member of the LISA Steering Committee and conference liaison. At various points I've also acted as an unofficial volunteer to help publicize and solicit speakers/trainers (or just carry around boxes) for LISA, ATC, and SREcon. I've gotten on stage in front of the crowd as part of a WiAC panel and also led a number of BoF sessions over the years.

Why did you decide to pursue a seat on the board?

USENIX had a profound impact on my life and career. The USENIX ideals about research, education, and OSS encouraged me to focus my skills on projects that positively impacted the world and shaped my career progression from sysadmin to technology director at mission-driven organizations. At this point in my career, I hope to exert a positive influence on the future focus and direction of the organization to ensure its continued relevance and ability to provide similar exposure to technology, professional networking, and peer support.

Why should someone consider becoming involved in USENIX?

The rise of social media and availability of online instructional content was a huge leap forward for those trying to learn today's fast-moving skills that aren't taught in traditional classroom environments. USENIX is a premier provider of such content and also provides a venue for academics to showcase their latest research. Beyond that, USENIX conferences also provide ample opportunity for professional networking, something you still can't obtain by watching an online video. The people you meet at USENIX conferences may be your next coworkers, co-authors, research partners, or lifelong friends.

Aside from your work (and USENIX), tell me about your passions and how you spend your time.

Tangential to work, I'm passionate about diversity and inclusion and building strong, healthy leadership and management structures. I mentor folks from other companies and participate in various slack workspaces geared towards these topics. To unwind from all the serious stuff, I occupy my mind by reading fiction, playing games, solving puzzles, and building LEGO.

Do you have one unique fact about yourself you can share with us?

I'm an Adult Fan of LEGO (AFOL) and have accumulated an extensive collection (specializing in Space, Star Wars, and Speed Challenge sets) since becoming solvent.

In the spirit of the Board Game Night BoF, what's your favorite board game?

I'm an avid board and card gamer, so trying to pick just one favorite might be the hardest question of this whole interview. I'm a particular fan of "hidden traitor" mechanics or "one against many" deduction games, but I also enjoy a good brutal co-op game that kicks your butt. After rolling 3D6 + 1D4 damage bonus, the randomized answer on my lookup table is Dead of Winter. It's the other people, not the zombies, you really need to guard against.

Tell me a bit about the region of the country you live in: what you like about it, the tech scene, and why someone might consider visiting or relocating there.

I grew up in the country (hometown of 200 people), so I like having a perimeter of personal space, but I also like being relatively close to the convenience of stores, culture, and a major airport. This means I'm going to live in the suburbs of a decently sized, but not huge, city. I also like trees, hills, having four seasons (yes, I love my snow), and being on the ocean. All of those things together mean that the northeast, and specifically the coastal suburbs of Boston, are where I make my home.

Boston has a number of very well-respected universities with excellent STEM programs, and therefore also has a burgeoning tech scene. Visitors to the area come to satisfy a wide array of interests including, but not limited to, US history, animals and nature (no matter the season), foodie lifestyle, recreational or professional sports, liberal ideals and politics, art, theater, science, and craft beer.

Anything else you'd like to share?

I'm excited to join the other great members of the Board and ready to do some work!

Register Now!

usenix

LISA18

October 29–31, 2018
Nashville, TN, USA

LISA: Where systems engineering and operations professionals share real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

Featured Speakers


Janna Brummel, ING


Nicholas Hunt-Walker, Starbucks


Edward Hunter, Netflix


Jeri-Elayne Smith, The Citadel


Madhu Akula, Appsecco


Jon Masters, Red Hat


Jeffrey Snover, Microsoft


Thomas Limoncelli, Stack Overflow

Register by Monday, October 8 and save!

www.usenix.org/lisa18

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

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

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

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

Register Today!

OSDI 18

13th USENIX Symposium on Operating Systems Design and Implementation

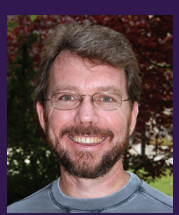
October 8–10, 2018 • Carlsbad, CA, USA

Sponsored by USENIX in cooperation with ACM SIGOPS

OSDI 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 OSDI Symposium emphasizes innovative research as well as quantified or insightful experiences in systems design and implementation.

Program Co-Chairs

Andrea Arpaci-Dusseau
University of Wisconsin—Madison



Geoff Voelker
University of California, San Diego

The full program and registration are now available.
Register by September 17 and save!

www.usenix.org/osdi18