;login:

SPRING 2017 VOL. 42, NO. 1

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

The Linux Kernel Self-Protection Project
Kees Cook

Interviews with Jeff Mogul and Amit Levy
Rik Farrow

How to Write Secure Parsers
Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon D. Momot, Meredith L. Patterson, and Anna Shubina

Warm Up Your JVMs
David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan

Columns

New Python 3.6 Features
David Beazley

Creating, then Graphing, a DB of Modify Times
David N. Blank-Edelman

Cybersecurity Workload Trends
Dan Geer and Eric Jardine

When Not to Be Truthful
Robert G. Ferrell

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

SREcon17 Americas

March 13–14, 2017, San Francisco, CA, USA
www.usenix.org/srecon17americas

NSDI '17: 14th USENIX Symposium on Networked Systems Design and Implementation

Sponsored by USENIX in cooperation with ACM SIGCOMM and ACM SIGOPS

March 27–29, 2017, Boston, MA, USA
www.usenix.org/nsdi17

SREcon17 Asia/Australia

May 22–24, 2017, Singapore
Submissions due March 6, 2017
www.usenix.org/srecon17asia

USENIX ATC '17: 2017 USENIX Annual Technical Conference

July 12–14, 2017, Santa Clara, CA, USA
www.usenix.org/atc17

Co-located with USENIX ATC '17

SOUPS 2017: Thirteenth Symposium on Usable Privacy and Security
July 12–14, 2017
www.usenix.org/soups2017

HotCloud '17: 9th USENIX Workshop on Hot Topics in Cloud Computing
July 10–11, 2017
Submissions due March 14, 2017
www.usenix.org/hotcloud17

HotStorage '17: 9th USENIX Workshop on Hot Topics in Storage and File Systems
July 10–11, 2017
Submissions due March 16, 2017
www.usenix.org/hotstorage17

Do you know about the USENIX open access policy?

USENIX is the first computing association to offer free and open access to all of our conferences 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

USENIX Security '17: 26th USENIX Security Symposium

August 16–18, 2017, Vancouver, BC, Canada
www.usenix.org/sec17

Co-located with USENIX Security '17

WOOT '17: 11th USENIX Workshop on Offensive Technologies
August 14–15, 2017
Submissions due May 31, 2017
www.usenix.org/woot17

CSET '17: 10th USENIX Workshop on Cyber Security Experimentation and Test
August 14, 2017
Submissions due May 2, 2017
www.usenix.org/cset17

FOCI '17: 7th USENIX Workshop on Free and Open Communications on the Internet
August 14, 2017
www.usenix.org/foci17

ASE '17: 2017 USENIX Workshop on Advances in Security Education
August 16, 2017
Submissions due May 9, 2017
www.usenix.org/ase17

HotSec '17: 2017 USENIX Summit on Hot Topics in Security
August 15, 2017
www.usenix.org/hotsec17

SREcon17 Europe/Middle East/Africa

August 30–September 1, 2017, Dublin, Ireland
Submissions due April 12, 2017
www.usenix.org/srecon17europe

LISA17

October 29–November 3, 2017, San Francisco, CA
Submissions due April 24, 2017
www.usenix.org/lisa17

FAST '18: 16th USENIX Conference on File and Storage Technologies

February 12–15, 2018, Oakland, CA

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

;login:

SPRING 2017 VOL. 42, NO. 1

EDITORIAL

2 **Musings** *Rik Farrow*

OPERATING SYSTEMS

4 **Operating Systems: Three Easy Pieces** *Remzi H. Arpaci-Dusseau*

8 **Teaching Operating Systems with FreeBSD through Tracing, Analysis, and Experimentation**
George V. Neville-Neil and Robert N. M. Watson

14 **Linux Kernel Self-Protection** *Kees Cook*

18 **Interview with Jeff Mogul** *Rik Farrow*

23 **Interview with Amit Levy** *Rik Farrow*

FILE SYSTEMS

26 **MarFS, a Near-POSIX Interface to Cloud Objects**
Jeff Inman, Will Vining, Garrett Ransom, and Gary Grider

PROGRAMMING

32 **Curing the Vulnerable Parser: Design Patterns for Secure Input Handling**
Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon D. Momot, Meredith L. Patterson, and Anna Shubina

SYSADMIN

40 **Postmortem Action Items: Plan the Work and Work the Plan**
John Lunney, Sue Lueder, and Betsy Beyer

46 **Don't Get Caught in the Cold, Warm Up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems**
David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan

COLUMNS

52 **Gleeful Incompatibility** *David Beazley*

57 **Practical Perl Tools: Off the Charts** *David N. Blank-Edelman*

63 **Cybersecurity Workload Trends** *Dan Geer and Eric Jardine*

68 **/dev/random** *Robert G. Ferrell*

BOOKS

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

USENIX NOTES

73 **Notice of Annual Meeting**

73 **USENIX Awards**

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.

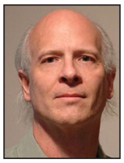
POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

COVER ILLUSTRATION
Abstract Green Sparkles Background was created by freevector.com and is distributed under the Creative Commons Attribution 4.0 license.

EDITORIAL

Musings

RIK FARROW



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

I was deeply disappointed by the operating systems class I took in 1978. An advanced CS class, the focus was on IBM mainframe architecture, and that appeared grossly inappropriate to me. By that time, the Apple II and the Altair 8080 had been out for a year, and it was obvious to me that the computers most people used would be changing.

The lab for the class used two Digital Equipment Corporation PDP 11/45s, and students were supposed to build an operating system, starting with the keyboard driver, proceeding to a file system, then the ability to load and execute code, all on a mini-computer with a very different architecture than the mainframe. Oh, and the mainframe didn't have a file system, and used Job Control Language for running programs.

In despair, I asked the teaching assistant if there wasn't something more appropriate to use as a way of understanding operating systems, and he said there wasn't. Keep in mind that AT&T had been licensing UNIX to universities for several years by then, a textbook had been written about 6th Edition UNIX, and that UNIX ran on DEC mini-computers.

I never finished that class. Competing for time on the lab systems with 200 other students, when all you could get was one-hour time slots, was too frustrating. I aced the other CS course I took that year and got a job working for a small embedded systems company, where I began learning about operating systems.

Computers have gotten a lot more complicated than they were in the seventies. I built my own computer, from a kit, in 1979. A couple of years later, I wrote my first C program, one that provided all of the file system features of CP/M [1], and the device driver, in two pages of code. With only 56 Kb of memory, having an intelligent floppy disk controller, one quite similar to the one in the DEC mini-computer, made the task simpler.

Today, Linux has more than 120 file systems, designed for different use cases. Device drivers have gotten more complex, and programmers now have gigabytes of memory to work with. Those choices are there because certain file systems perform much better for particular workloads.

Even if you are not a programmer, you still need to understand some operating systems basics. Caskey Dickson, co-chair of LISA17, has been teaching such a class at LISA, and there are several good books out about operating systems.

In this issue, we are featuring two freely available sources for learning about operating systems. The first is a three-section online book, with exercises and material for helping instructors, by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Their material most closely follows what I've seen in OS textbooks, but their tone is conversational and much less daunting.

The second example comes from classes taught using FreeBSD. George V. Neville-Neil and Robert N. M. Watson developed this material for three styles of classes: those taught as college courses, one as a tutorial for BSD conference attendees, and a third as online videos. Their classes have more pragmatic focuses, with two versions actually using BeagleBone Black computers running FreeBSD and used for probing the internals of running systems. Their materials are open source under a BSD-style license.

Either of these would have been much better than using a textbook and having class lectures about a mainframe system that was over 10 years old in 1978 and soon to be usurped.

While there were certainly good jobs to be had as systems programmers, as almost anything you did on these computers required writing assembly language patches, there really weren't that many mainframes. Especially when you compare that to the revolution that was on the horizon.

The Lineup

I've already introduced the first two articles, resources for learning about operating systems, so I'll move on to the rest.

Kees Cook has written about security improvements to the Linux kernel. Kees works on the Kernel Self-Protection Project [2], and he describes a lot of the work that has already been done to make Linux kernels more difficult to exploit.

I have two interviews for this issue, both systems-related. Jeff Mogul has done many things in his career, and in this longer than usual interview, I begin by focusing on what Jeff worked on when the Internet, and later the Web, was young. We also discuss research labs and why we have CS proceedings instead of journals.

I talked with Amit Levy about TockOS. TockOS will replace TinyOS, both operating systems for very resource-limited embedded devices. Amit's interest in TockOS includes building a secure system, something the world of IoT desperately needs.

I discovered MarFS from a talk given at SNIA's SDC conference. Jeff Inman et al. explain how they built a nearly POSIX front end for a massively parallel, object file system back end. While their focus is on HPC, MarFS and the ideas illustrated by their system can certainly be applied to other large-scale and high performance storage systems. And the software they developed is available online.

Sergey Bratus and crew have written about how to parse input securely. If you've ever written any code, including shell scripts, you likely have noticed how much time you spend on parsing input. Yet mistakes in parsing input, that often mean accepting invalid input, lead to the majority of exploits we see in both programs and operating systems. The authors use their published work, where they replace the buggy code for an industrial control system, as examples as they explain how to do this correctly, as well as how coders usually do this poorly.

In the area of system administration and SRE, Lunney et al. cover the proper handling of postmortem action items. While postmortems are now recognized as important methods for improving the quality and stability of systems, Lunney et al. explain how they take advantage of the output of postmortems to drive corrective work.

The final article comes from research published at OSDI. Lion et al. were examining the performance of popular distributed systems, like Hadoop and Spark, looking at overhead. They discovered that a large proportion of the time spent running these applications was wasted on loading and interpreting Java classes every time another request was made. They produced HotTub, a version of the JVM, that caches warmed-up JVMs for reuse, improving the performance of HDFS by 21% and Spark by 33%.

David Beazley has written about what's new in Python 3.6. Hint: it's cool and not at all backwards-compatible with Python 2.

David N. Blank-Edelman pulls off a tour de force by creating a database from the output of `ls -lR`, then creates a Web page and Perl scripts that work with Google Charts, finally creating a spiffy chart showing the number of files created each month over many years.

Kelsey Hightower and Dave Josephsen decided not to write for this issue.

Dan Geer and Eric Jardine examine cybersecurity workload trends, using the NIST vulnerability workload and data providing estimates of the number of people working on remediating vulnerabilities to produce some trend lines.

Robert G. Ferrell has written about being totally truthful, and creates an example where withholding information has both good and bad effects.

Mark Lamourine has written three book reviews. The first two cover books about Angular 2, a framework for writing Web client applications. The third review is on the third edition of *The Practice of System and Network Administration*. I've reviewed bunnie Huang's book called *Hacking Hardware*.

During LISA16, someone asked me how I'd become so interested in security. I replied that security required that I understand programming, networking, system administration, file systems, and operating systems, and that I loved having one field cut across so many other areas of interest. Exploiting computers is definitely a form of hacking because the successful exploit requires seeing the system in a manner that the designers of that system didn't anticipate. Defending systems also means going "outside the box," although I have to concede that the attackers had, and still have, the upper hand.

While your work focus may not be security, understanding as much as you can about the operating systems that provide resources, and hopefully, security to your applications should only make your work easier.

References

[1] CP/M: https://en.wikipedia.org/wiki/CP/M.

[2] Kernel Self-Protection Project: https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project.

OPERATING

Operating Systems
Three Easy Pieces

REMZI H. ARPACI-DUSSEAU

Remzi H. 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 19 PhD students in their time at Wisconsin, won nine 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

Back in 2007, a student (call him Student #1) approached me and asked a simple question: "Do I have to buy the book for this course, or can I get by without it?" The course was undergraduate operating systems (called "CS 537" at the University of Wisconsin-Madison). In teaching the course, I mostly relied upon notes developed by myself and my colleague Andrea Arpaci-Dusseau, and I still thought it was useful for students to have something to read outside of class. So I pushed back a little. "Yes, you should. It's good for you to have another source for the material. Why don't you want to buy it?" The student looked at me sheepishly and said, "Well, I, um, can't really afford it." The book we were using cost over $100, as do many textbooks today.

It was just a small moment, but it led to a big change in how we teach the OS course here at Wisconsin. Although I didn't know it at the time, that simple, honest, and slightly heartbreaking comment led to the creation of a free online operating systems textbook called *Operating Systems: Three Easy Pieces* (sometimes called *OSTEP* and available at http://ostep.org). Chapters of the book have been downloaded millions of times over the past few years, and hundreds of teachers at various colleges and universities have told me they are using the book in their classes.

In the rest of this article, I'll first provide a little more history on how the book developed, discuss how the book is organized, and make the more general case for free online textbooks; indeed, I have a strong belief that all textbooks should be made freely available online. I'll also discuss some keys to success with such an endeavor and present my thoughts on how publishers might evolve in such a free-textbook world.

History

After the encounter with Student #1, I made a snap decision for the course. "No one is required to buy a book for this course. Just come to class, take notes, and that will be enough. Everything you need to know we will cover in class," I declared. The students smiled. For at least one day, I was their $100 hero!

But then along came Student #2 and another encounter I will never forget. After a few classes in this first "no book" semester, this student said, "Professor, I sometimes miss class. For example, tomorrow, I have an interview that I couldn't move. And the other day, I overslept—the class is a bit early in the day for me. So I don't know what to do for those days that I miss and have no notes. And I don't really know anyone else to borrow them from." Now I was a little surprised at one of these comments—the class started at 1:00 p.m. that semester. Undergrads! But the general point hit home: I needed to go beyond the "take notes" approach and provide more material for them.

And thus I hit upon a simple idea. I usually leave the hour or so after class open to wind down. Why not put this hour to good use in service of the class? So after each class, if I had the energy, I would close my door and just write down, in simple text form, what I had just lectured upon in class. Just after class is a great time to do this work: the ideas are fresh in your head and it is relatively easy to write them down.

I then posted these crude "text-based lecture captures" to the class Web site. If you're interested, you can look at them here: http://pages.cs.wisc.edu/~remzi/Classes/537/Fall2008/notes.html.

Honestly, if we're going to remain friends, you probably shouldn't look these over—they're a little embarrassing. Just plain text, no real figures (just some ASCII art), and really very primitive writings.

Then a funny third thing happened: students started to give me (unsolicited) feedback on the writing. And, perhaps a little surprisingly, they were quite positive! In the world of academics, you get a lot of feedback on the work you do, and much of it is negative—those of you who have ever submitted a paper to a conference understand what I am talking about. This positive feedback was a bit like a drug; I wanted more! And so I started to plot how to take these rough notes and make them into something better. And that's what I have tried to do each semester I taught the class since that time.

Interestingly enough, many upgrades to the book were driven by student feedback. One student wished there were some better diagrams and included detailed notes to me on where to place them on each page, so I spent some time converting ASCII figures into actual EPS graphics. Another said that the raw text was a little hard on the eyes, so I started to typeset each chapter in LaTeX. Some students asked how they could obtain a print copy, which led me to self-publish the book on Lulu.com; we have sold thousands of print copies of a book that is available entirely for free online.

Finally, one student said he would buy a copy if I made a decent cover (the cover at that time: pure black, just text). I am a sucker for a sale, so I asked him, "What would you put on it?" He suggested something cool, like a dinosaur. I had to tell him that the prehistoric beast idea was already taken, but it gave me an idea, and soon enough I had a comet flying across the cover. We know what comets can do to dinosaurs, right?

As a result of all of this effort, we are nearing the completion of what we call a "version 1.0." The results can be seen at our Web site, http://ostep.org.

Organizing a Book

One major question we had in putting a book together was how to organize the material. Of course, you could just have 10–12 chapters and follow the organization of most other OS books, but that seemed less than interesting. So we started to think about different ways of organizing the material into a few major conceptual themes, and then divide these into short chapters that roughly matched a lecture or half-lecture on a particular topic.

While teaching from different textbooks, we noticed that most books introduced threads and processes early on, and thus soon had to present all thread-related topics, including locks, condition variables, race conditions, and so forth—all very detailed and hard material, and all very early in the semester. However, when we taught the material in this manner, it didn't quite seem to work; students didn't even yet understand what an address space was, and we were telling them about the differences between processes (each of which has its own address space) and threads (which share one address space). So we decided to try something different.

At this point, the idea arose to organize the course into three major conceptual pieces: **virtualization** (which covers CPU and memory virtualization), **concurrency** (which introduces threads, locks, condition variables, and related topics), and finally **persistence** (which covers storage devices and file systems). Within each section, we have a lot of short chapters, each on one subtopic (e.g., introduction to CPU scheduling, TLBs, or crash consistency in file systems). While this is a little different from other books, we've found that students make more sense of the material in this order. And, in this manner, the title of the book became obvious.

The other advantage of this organization is that it places storage systems (our research specialty) on equal footing with the other parts of the material. Some other books relegate file systems and storage to the very last chapters and thus (in our opinion) spend too much time on virtualization and concurrency at the cost of understanding this important subsystem. After all, what is more important than remembering information for the long term?

One other difference within our approach is that we tend to emphasize mechanism (and the nuts and bolts of how things work) more than policy. This decision stems from a personal belief that learning new policies is relatively easy, but understanding the machinery of systems is hard; class should thus emphasize the hard stuff and leave the easier things for students to learn later.

Why Textbooks Should Be Freely Available Online

There are many reasons textbooks should be made freely available online. Here is a list of some of the big ones:

- **It's the best way to share information with the most people.** Authors spend so much time creating these books; why trap the information inside the standard publishing wall? A casual reader is not going to drop $150 for a book with a few things inside it they are interested in. Making chapters freely available for download allows for casual usage among a much broader group of people.
- **It enables new usage models.** No professor would (likely) dare make a student buy four (expensive) books in order to use a few chapters from each. When book chapters are available online for free, this type of new model is readily available. A more competitive market for specialized sub-books could also arise.
- **It avoids needless revisions.** Authors are currently forced to do a number of silly things because of the way textbook sales work. If the author does not upgrade the book, students happily purchase used copies for very little cost; the publishers, unsurprisingly, are not happy with this, and thus essentially force authors to keep making revision after revision. With no such business model in place, material will get upgraded as needed.
- **It enables chance discovery.** Students find resources today by using search engine tools to browse the Internet or by poking around Wikipedia pages. Having book chapters available for free on the Internet makes chance discovery more likely and possible.
- **It's free.** Making a book free makes it accessible to anyone, regardless of their financial circumstances (assuming they have access to the Internet). If we wish to teach the world, we should make as much information available as inexpensively as possible to as many people as possible.

Keys to Success

In doing this work, I've tried to think about what was essential to realizing some level of success with writing one's own book. Here I list some of these tips for aspiring authors:

- **Develop a class first.** A class (for me) is just 30 lectures, telling one big story (e.g., what is an operating system?) and a number of smaller stories (e.g., what are virtualization, concurrency, persistence?). After being here for some time, Andrea and I had taught the course repeatedly and refined the message each time we rotated through. By the end of this development, we had a pretty good idea of what we wanted to say and how we wanted to say it. Once you have gone through a class a few times, writing it all down is much easier.
- **Improve something each time you teach.** I found the task of writing a book daunting—it's a lot of work! But writing a little now and then didn't sound too bad, and I enjoyed it. They say that the perfect is the enemy of the good, so I just embrace the fact that although the book will never be perfect, I can make it a little better each time. This also gives me a new focus each time I teach the class, which actually makes teaching the same class more interesting than usual.
- **Make each chapter a separate downloadable unit.** There are many reasons to do so and three particularly important ones. First, students won't get overwhelmed by a massive 800-page beast; each chapter, in contrast, is usually short (say 10–20 pages) and thus much less daunting and easier to digest. Second, short chapters enable better discovery via search engine and other related means. A person might search for "semaphores," and it is much easier to then find the exact chapter instead of searching through a book on operating systems; similarly, a Wikipedia page on multi-level page tables can point directly to the right chapter instead of vaguely to an entire book. Third, parts of the book can be used instead of the whole; a professor at another institution can pick and choose chapters from different sources, which would be much harder to do if the entire book is the only unit of usage.
- **Create homework assignments that are reusable.** Book chapters need homework questions to enable students to test their own knowledge. The thought of writing some fixed questions, and then having to update questions regularly, was a non-starter. As a result, we started using an idea we saw in Hennessy and Patterson's *Computer Architecture: A Quantitative Approach*, which was to create computer programs that can generate an infinite number of variants to a certain class of question. In our case, these programs are essentially little simulators that mimic some aspect of an OS. For example, a virtual memory simulator might generate a particular configuration (physical memory of size X, a Y-bit virtual address space) and then ask you to translate certain addresses from virtual to physical. By adding more simulators over time, you give students a richer, more interactive way to quiz themselves about the material.
- **Be responsive to feedback.** We actively encourage feedback from anyone who reads the book and credit them for any fix or update that arises from their suggestions. Many students have thus found typos for us, which we have fixed; many professors and instructors have suggested more substantial changes, which we have implemented as well. While we can't accommodate every request, we read each one carefully and then decide what to do. In all cases, we get back to the suggester as quickly as we can.
- **Realize you don't have to cover everything.** One last point about making a book: it doesn't have to be a bible. Use it to spark a student's interest and cover most important topics, especially topics you care about. It's OK if not everything is covered in a textbook; rather, what you are giving students is a way to under-

stand the major pieces and the ability (hopefully) to be able to fill details in themselves at a later time.

Of course, none of these suggestions are useful without a fair amount of hard work, for which there is little substitute. But, if done right, the work is rewarding and spread out, and you get lots of thanks from people around the world.

Aside: Why Free Doesn't Mean Open Source

People often say to me, "That's great you're doing a free book. Why isn't it on GitHub so I can hack on it, too?" My reaction to that is usually, "Uh, no thanks." Why so unfriendly, you ask?

The answer: we strongly believe that a book should have a single voice. This voice communicates one coherent body of knowledge to the reader. If each chapter were written by different people, this voice would likely be lost and the experience lessened. Whatever the model of collaboration, we believe that the important thing is that the author or group of authors work hard to maintain that single voice.

What This Means for Publishers

Probably the biggest change that will occur, should all textbooks become free, is to the world of publishers, who will find that their services (in the current form) are not much needed. However, they could save themselves by doing a number of things.

First, publishers should split out their services and offer parts of said services to authors. For example, publishers could help with marketing and advertising of free textbooks. In addition, publishers could offer editorial services as a separate service. Even printing could be split off and offered (although they are behind here, thanks to Lulu.com and other similar services). Instead of going with one publisher for all of these things, an author could pick and choose what he or she needs.

Second, publishers should figure out more ways to publish print copies at low cost. I've spoken with publishers who said they want to do low-cost books, and then turn around and say they can't do a book for less than $50 or $60. In contrast, at Lulu.com, you can print single copies of a book on demand for $20 to $30. Publishers need to get their costs down and become competitive in offering low-cost print books. Students still like print, and by selling both digital and print at low cost and high volume, publishers could still make money. There seems to be some recalcitrance in the industry that prevents this.

Conclusion

I strongly believe that textbooks should be free. *OSTEP* is just one such book, and is and will always be freely available online and at a low cost in print forms. But *OSTEP* is just one book. There needs to be more! If you are a teacher of a class, think about what it would take to convert your own personal lecture notes into something more widely shared. Soon, you might have a textbook on your hands, and the free textbook revolution can truly begin!

Teaching Operating Systems with FreeBSD through Tracing, Analysis, and Experimentation

GEORGE V. NEVILLE-NEIL AND ROBERT N. M. WATSON

George V. Neville-Neil works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, networking, and time protocols. He is the coauthor with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System*. For over 10 years he has been the columnist better known as Kode Vicious. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the USENIX Association, and IEEE. He is an avid bicyclist and traveler and currently lives in New York City. gnn@neville-neil.com

Dr. Robert N. M. Watson is a University Senior Lecturer (Associate Professor) in systems, security, and architecture at the University of Cambridge Computer Laboratory; FreeBSD developer and past core team member; and member of the FreeBSD Foundation Board of Directors. He leads a number of cross-layer research projects spanning computer architecture, compilers, program analysis, program transformation, operating systems, networking, and security. Recent work includes the Capsicum security model, MAC Framework used for sandboxing in systems such as Junos and Apple iOS, CHERI (CPU with protected memory segments), and multithreading in the FreeBSD network stack. He is a coauthor of *The Design and Implementation of the FreeBSD Operating System* (2nd edition). watson@freebsd.org

A version of this article originally appeared in the *FreeBSD Journal*, April/May 2016 issue.

Many people who study computer science at universities encounter their first truly large system when studying operating systems. Until their first OS course, their projects are small, self-contained, and often written by only one person or a team of three or four. In this article, we suggest an approach to studying operating systems we have been using with graduate students and practitioners that involves using a small ARMv7 board and tracing. All of our materials are available online, with a BSD-like license.

Since the first courses on operating systems were begun back in the 1970s, there have been three ways in which such classes have been taught. At the undergraduate level, there is the "trial by fire," in which students extend or recreate classical elements and forms of OS design, including kernels, processes, and file systems. In trial-by-fire courses the students are given a very large system to work with, and they are expected to make small, but measurable, changes to it. Handing someone a couple million lines of C and expecting them to get something out of changing a hundred lines of it seems counterintuitive at the least.

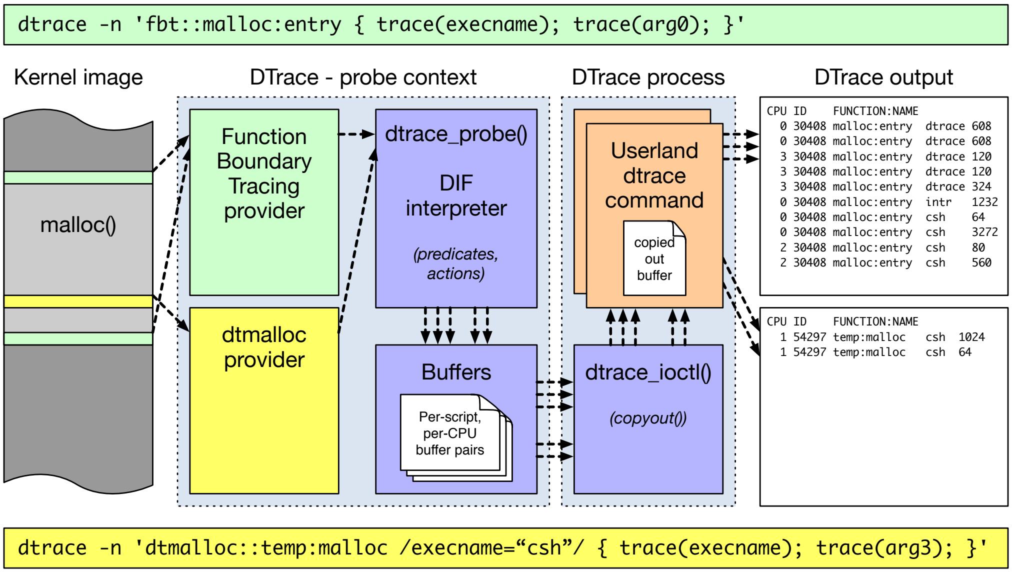
The second undergraduate style is the "toy system." With a toy system the millions of lines are reduced to some tens of thousands, which makes understanding the system as a whole easier but severely constrains the types of problems that can be presented, and the lack of fidelity, as compared to a real, fielded operating system, often means that students do not learn a great deal about operating systems, or large systems in general. For graduate students, studying operating systems is done through a research readings course, where students read, present, discuss, and write about classic research where they are evaluated on a term project and one or more exams.

For practitioners, those who have already left the university, or those who entered computer science from other fields, there have been even fewer options. One of the few examples of a course aimed at practicing software engineers is the series "FreeBSD Kernel Internals" by Marshall Kirk McKusick, with whom both authors of this article worked on the most recent edition of *The Design and Implementation of the FreeBSD Operating System*. In the "FreeBSD Kernel Internals" courses, students are walked through the internals of the FreeBSD operating system with a generous amount of code reading and review, but without modifying the system as part of the course.

For university courses at both the undergraduate and graduate level, we felt there had to be a middle way where we could use a real-world artifact such as FreeBSD, which is deployed in products around the world, while making sure the students didn't get lost in the millions of lines of code at their disposal.

Deep-Dive Experimentation

Starting in 2014, the authors undertook to build a pair of tightly coupled courses sharing pedagogy and teaching material. One version is designed for graduate students and taught by Robert N. M. Watson at the University of Cambridge. The other version is a practitioner course taught at conferences in industrial settings by George Neville-Neil.



```
dtrace -n 'fbt::malloc:entry { trace(execname); trace(arg0); }'
```

```
CPU ID     FUNCTION:NAME
  0 30408 malloc:entry  dtrace 608
  0 30408 malloc:entry  dtrace 608
  3 30408 malloc:entry  dtrace 120
  3 30408 malloc:entry  dtrace 120
  3 30408 malloc:entry  dtrace 324
  0 30408 malloc:entry  intr   1232
  0 30408 malloc:entry  csh    64
  0 30408 malloc:entry  csh    3272
  2 30408 malloc:entry  csh    80
  2 30408 malloc:entry  csh    560
```

```
CPU ID     FUNCTION:NAME
  1 54297 temp:malloc   csh  1024
  1 54297 temp:malloc   csh  64
```

```
dtrace -n 'dtmalloc::temp:malloc /execname="csh"/ { trace(execname); trace(arg3); }'
```

Figure 1: DTrace is a critical part of the course's teaching approach—students trace kernels and applications to understand their performance behavior. They also need to understand—at a high level—how DTrace works in order to reason about the "probe effect" on their measurements.

In the deep-dive course, students learn about and analyze specific CPU/OS/protocol behaviors using tracing via DTrace (Figure 1) and the CPU performance counters. Using tracing to teach mitigates the risk of OS kernel hacking in a short course, while allowing the students to work on real-world systems rather than toys. For graduate students, we target research skills and not just OS design. The deep-dive course is only possible due to development of integrated tracing and profiling tools, including DTrace and Hardware Performance Monitoring Counter (hwpmc) support present in FreeBSD.

The aims of the graduate course include teaching the methodology, skills, and knowledge required to understand and perform research on contemporary operating systems by teaching systems-analysis methodology and practice, exploring real-world systems artifacts, developing scientific writing skills, and reading selected original systems research papers.

The course is structured into a series of modules. Cambridge teaches using eight-week academic terms, providing limited teaching time compared to US-style 12-to-14-week semesters. However, students are expected to do substantial work outside of the classroom, whether in the form of reading, writing, or lab work. For the Cambridge course, we had six one-hour lectures in which we covered theory, methodology, architecture, and practice, as well as five two-hour labs. The labs included 30 minutes of extra teaching time in the form of short lectures on artifacts, tools, and practical skills. The rest of the students' time was spent doing hands-on measurement and experimentation.

Readings were also assigned, as is common in graduate level courses, and these included both selected portions of module texts and historic and contemporary research papers. Students produced a series of lab reports based on experiments done in (and out) of labs. The lab reports are meant to refine scientific writing style to make it suitable for systems research. One practice run was marked, with detailed feedback given, but not assessed, while the following two reports were assessed and made up 50% of the final mark.

Three textbooks were used in the course: *The Design and Implementation of the FreeBSD Operating System* (2nd edition) as the core operating systems textbook; *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, which shows the students how to measure and evaluate their lab work; and *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, covering the use of the DTrace system.

Although many courses are now taught on virtual-machine technology, we felt it was important to give the students experience with performance measurement. Instead of equipping a large room of servers, we decided, instead, to teach with one of the new and inexpensive embedded boards based around the ARM series of processors. Initially, we hoped to use the Raspberry Pi as it is popular, cheap, and designed at the same university at which the course would first be taught. Unfortunately, the RPi available at the time did not have proper performance counter support in hardware due to a feature being left off the system-on-chip design when it was originally produced.

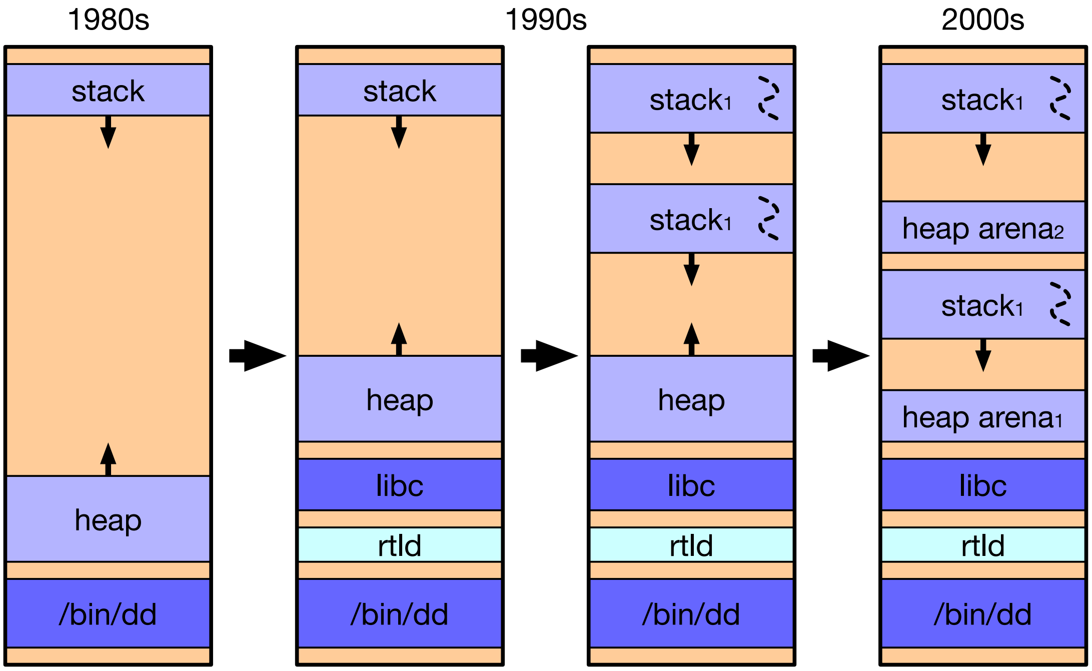


Figure 2: Students learn not just about the abstract notion of a UNIX "process," but also the evolution of the approach over the decades: dynamic linking, multithreading, and contemporary memory allocators such as FreeBSD's `jemalloc`.

With the RPi out of the running, we chose the BeagleBone Black (BBB), which is built around a 1 GHz, 32-bit ARM Cortex A-8, a superscalar processor with MMU and L1/L2 caches. Each student had one of these boards on which to do lab work. The BBB has power, serial console, and network via USB. We provided the software images on SD cards that formed the base of the students' lab work. The software images contain the FreeBSD operating system, with DTrace and support for the on-board CPU performance counters, and a set of custom microbenchmarks. The benchmarks are used in the labs and cover areas such as POSIX I/O, POSIX IPC, and networking over TCP.

Eight Weeks, Three Sections

The eight weeks of the course are broken up into three major sections. In weeks one and two, there is a broad introduction to OS kernels and tracing. We want to give the students a feel for the system they are working on and the tools they'll be working with. During these first two weeks, students are assigned their first lab, in which they are expected to look at POSIX I/O performance. I/O performance is measured using a synthetic benchmark we provide in which students look at file block I/O using a constant total size with a variable buffer size. The conventional view is that increasing the buffer size will result in fewer system calls and improved overall performance, but that is not what the students will find. As buffer sizes grow, the working set first overflows the last-level cache, preventing further performance growth, and later exceeds the superpage size, measurably decreasing performance as page faults require additional memory zeroing.

The second section, covering weeks three through five, is dedicated to the process model (Figure 2). Because the process model forms the basis of almost all modern programming systems, it is a core component of what we want the students to be able to understand and investigate during the course and afterwards in their own research. While learning about the process model, the students are also exposed to their first microarchitectural measurement lab in which they show the implications of IPC on L1 and L2 caching. The microarchitectural lab is the first one that contributes to their final grade.

The last section of the course is given over to networking, specifically the Transport Control Protocol (TCP, Figure 3). During weeks six through eight, the students are exposed to the TCP state machine and also measure the effects of latency on bandwidth in data transfers. We've moved to an explicit iPython/Junyper Notebooks framework, hosted on the BBB, to drive DTrace/PMC experimentation, and provide a consistent data analysis and presentation framework. This allows the students to be more productive in focusing on OS internals and analysis.

Challenges and Refinements

The graduate course has been taught twice at Cambridge, and we have reached out to other universities to talk with them about adopting the material we have produced. In teaching the course,

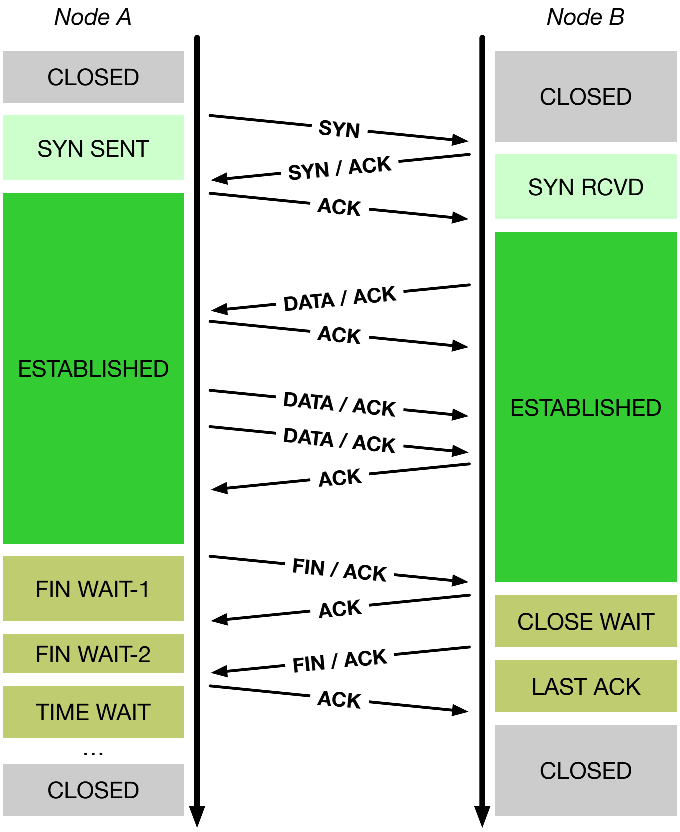


Figure 3: Labs 3 and 4 of the course require students to track the TCP state machine and congestion control using DTrace, and to simulate the effects of latency on TCP behavior using FreeBSD's DUMMYNET traffic control facility.

we discovered many things that worked, as well as a few challenges to be overcome as the material is refined. We can confirm that tracing is a great way to teach complex systems because we were able to get comprehensive and solid lab reports/analysis from the students, which was the overall goal of the course. The students were able to use cache hit vs. system-call rates to explain IPC performance. They produced TCP time-sequence plots and graphical versions of the TCP state machine all from trace output. Their lab reports had real explanations of interesting artifacts, including probe effects, superpages, DUMMYNET timer effects, and even bugs in DTrace. Our experiment with using an embedded board platform worked quite well—we could not have done most of these experiments on VMs. Overall, we found that the labs were at the right level of difficulty, but that too many experimental questions led to less focused reports— a concern addressed in the second round of teaching.

On the technical side, we should have committed to one of R, Python, or iPython Notebooks for use by the students in doing their experimental evaluations and write-ups. Having a plethora of choices meant that there were small problems in each, all of which had to be solved and which slowed down the students' progress. When teaching the course for the first time, there were several platform bumps, including USB target issues, DTrace for ARMv7 bugs, and the four-argument limitation for DTrace on ARMv7.

Teaching Practitioners

Teaching practitioners differs from teaching university students in several ways. First, we can assume more background, including some knowledge of programming and experience with UNIX. Second, practitioners often have real problems to solve, which can lead these students to be more focused and more involved in the course work. We can't assume everything, of course, since most of the students will not have been exposed to kernel internals or have a deep understanding of corner cases.

Our goals for the practitioner course are to familiarize people with the tools they will use, including DTrace, and to give them practical techniques for dealing with their problems. Along the way we'll educate them about how the OS works and dispel their fears of ever understanding it. Contrary to popular belief, education is meant to dispel the students' fear of a topic so that they can appreciate it more fully and learn it more deeply.

The practitioner's course is currently two eight-hour days. The platform is the student's laptop or a virtual machine. First taught at AsiaBSDCon 2015, the course was subsequently taught at AsiaBSDCon 2016 and BSDCan 2016.

Five-Day, 40-Hour Course Hardware or VM Platform Video Recordings

Like the graduate-level course, this course is broken down into several sections and follows roughly the same narrative arc. We start by introducing DTrace using several simple and yet powerful "one liners." A DTrace one liner is a single command that yields an interesting result. This example one-liner displays every name lookup on the system at runtime.

```
dtrace -n 'vfs:namei:lookup:entry \
      { printf("%s", stringof(arg1));}'
CPU     ID FUNCTION:NAME
  2  27847 lookup:entry /bin/ls
  2  27847 lookup:entry /libexec/ld-elf.so.1
  2  27847 lookup:entry /etc
  2  27847 lookup:entry /etc/libmap.conf
  2  27847 lookup:entry /etc/libmap.conf
```

The major modules are similar to the university course and cover locking, scheduler, files and the file system, and networking. The material is broken up so that each one-hour lecture is followed by a 30-minute lab in which students use the VMs on their laptops to modify examples given during the lectures or solve a directed problem. Unlike classes where we have access to hardware, the students do not take any performance measurements with hwpmc(4) since the results would be unreliable and uninformative.

Having taught the practitioner course several times, we have learned a few things. Perhaps the most surprising was that the class really engages the students. Walking around the class during the labs, we didn't see a single person checking email or reading social media—they were actually solving the problems.

The students often came up with novel answers to the problems presented, and this was only after being exposed to DTrace for a few hours. Their solutions were interesting enough that we integrated them back into the teaching during the next section. Finally, and obvious from the outset, handing a pre-built VM to the students significantly improves class startup time, with everyone focused on the task at hand, rather than tweaking their environment. Since the FreeBSD Project produces VM images for all the popular VM systems along with each release, it is easy to have the students pre-load the VM before class, or to hand them one on a USB stick when they arrive.

It's All Online!

With the overall success of these courses, we have decided to put all the material online using a permissive, BSD-like publishing license. The main page can be found at www.teachbsd.org, and our GitHub repo, which contains all our teaching materials for both the graduate and practitioner courses, can be found at https://github.com/teachbsd/course, where you can fork the material for your own purposes as well as send us pull requests for new features or any bugs found in the content. The third version of the Cambridge course (L41) with the Python lab environment will be online by May 2017 as the current course wraps up. We would value your feedback on the course and suggestions for improvements as well—and please let us know if you are using it to teach!

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

SAVE THE DATE!

2017 USENIX Annual Technical Conference

JULY 12–14, 2017, SANTA CLARA, CA

www.usenix.org/atc17

The 2017 USENIX Annual Technical Conference will bring together leading systems researchers for cutting-edge systems research and unlimited opportunities to gain insight into a variety of must-know topics, including virtualization, system and network management and troubleshooting, cloud computing, security, privacy, and trust, mobile and wireless, and more.

Co-Located with USENIX ATC '17

SOUPS 2017: Thirteenth Symposium on Usable Privacy and Security

JULY 12–14, 2017

www.usenix.org/soups2017

SOUPS 2017 will bring together an interdisciplinary group of researchers and practitioners in human computer interaction, security, and privacy. The program will feature technical papers, workshops and tutorials, a poster session, panels and invited talks, and lightning talks.

HotCloud '17: 9th USENIX Workshop on Hot Topics in Cloud Computing

July 10–11, 2017

www.usenix.org/hotcloud17

HotCloud brings together researchers and practitioners from academia and industry working on cloud computing technologies to share their perspectives, report on recent developments, discuss research in progress, and identify new/emerging "hot" trends in this important area. While cloud computing has gained traction over the past few years, many challenges remain in the design, implementation, and deployment of cloud computing.

HotCloud is open to examining all models of cloud computing, including the scalable management of in-house servers, remotely hosted Infrastructure-as-a-Service (IaaS), infrastructure augmented with tools and services that provide Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS).

Submissions due: March 14, 2017

Notice of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held on Thursday, July 13, in Santa Clara, CA, during the 2017 USENIX Annual Technical Conference.

HotStorage '17: 9th USENIX Workshop on Hot Topics in Storage and File Systems

July 10–11, 2017

www.usenix.org/hotstorage17

The purpose of the HotStorage workshop is to provide a forum for the cutting edge in storage research, where researchers can exchange ideas and engage in discussions with their colleagues. The workshop seeks submissions that explore longer-term challenges and opportunities for the storage research community. Submissions should propose new research directions, advocate non-traditional approaches, or report on noteworthy actual experience in an emerging area. We particularly value submissions that effectively advocate fresh, unorthodox, unexpected, controversial, or counterintuitive ideas for advancing the state of the art.

Submissions will be judged on their originality, technical merit, topical relevance, and likelihood of leading to insightful discussions that will influence future storage systems research. In keeping with the goals of the HotStorage workshop, the review process will heavily favor submissions that are forward looking and open ended, as opposed to those that summarize mature work or are intended as a stepping stone to a top-tier conference publication in the short term.

Submissions due: March 16, 2017

Linux Kernel Self-Protection

KEES COOK

Kees Cook has been working with Free Software since 1994, and has been a Debian Developer since 2007. He is currently employed as a software developer by Google, working on Android, Brillo, and Chrome OS. From 2006 through 2011 he worked for Canonical as the Ubuntu Security Team's Tech Lead and remains on the Ubuntu Technical Board. Before that, he worked as the lead sysadmin at OSDL before it became the Linux Foundation. He has written various utilities, including GOPchop and Sendpage, and contributes randomly to other projects, including fun chunks of code in OpenSSH, Inkscape, Wine, MPlayer, and Wireshark. Lately, he's been spending most of his time focused on security features in the Linux kernel. kees@outflux.net

Recent focus on defending the Linux kernel from attack has resulted in many fundamental self-protections being brought into the upstream releases across a wide spectrum of kernel internals. Getting these defenses deployed into the real world means there are fewer chances for attackers to gain a foothold on systems.

Linux systems have seen significant improvements in security over the last decade. Containers (with various combinations of namespaces) and mandatory access control policies (like SELinux) keep walls between groups of processes; privileged processes try to use only fine-grained capabilities; risky processes confine themselves with seccomp; execution chains are cryptographically integrity-checked, and the list goes on. This reduction in the attack surface of user space has resulted in more attention being given to attacks against the Linux kernel itself. Because the kernel is the mediator for all the mentioned security systems, successful exploitation of a flaw in the kernel means all these protections go out the window.

Much recent work has involved providing the Linux kernel with better self-protection. Although much of the prior security work in the kernel was designed to protect user space from user space, the Kernel Self-Protection Project [1] focuses instead on protecting the kernel from user space. Many of the ideas and technologies in this project come from the large PaX and grsecurity (https://grsecurity.net) patches, while others originate from academic research papers and similar sources. Ultimately, there are two fundamental principles: eliminate classes of bugs and remove exploitation methods.

Fixing security bugs is important, but there are always more to be found. With the average lifetime of security bugs being five years [2], kernel development needs to be aimed at eliminating entire classes of bugs instead of playing whack-a-mole. Poor design patterns that lead to bugs can be exterminated by changing APIs or data structures.

Removing exploitation methods is fundamentally about creating a hostile environment for an attack. The kernel already runs smoothly day-to-day, but when it hits unexpected situations, it needs to deal with them gracefully. These situations tend not to affect the regular operation of the kernel, but leaving them unaddressed makes exploitation easier.

Even redesigning kernel internals so that the criticality of flaws is reduced has a significant impact on security. If a bug causes a system to reboot instead of give full control to an attacker, this is an improvement. The downtime will be annoying, but it sure beats going weeks not realizing a system was backdoored and then having to perform extensive post-intrusion forensics.

There has been a steady stream of improvements making their way into the kernel, but the last three years have seen a number of significant (or at least interesting) protections added or improved. There isn't room to cover everything in this article, but what follows are highlights spanning a range of areas.

The self-protection technologies in the Linux kernel can be roughly separated into two categories: probabilistic and deterministic. Understanding the differences between these categories helps us evaluate their utility for a given system or organization's threat model. After defining what needs to be protected against, it's easier to digest what actually addresses the risks.

Probabilistic protections derive their strength from some system state being unknown to an attacker. They tend to be weaker than deterministic protections since information exposures can defeat them. However, they still have very practical real-world value. They tend to be pragmatic defenses, geared toward giving an advantage (even if small) to a defender.

Deterministic protections derive their strength from some system state that always blocks an attacker. Since these protections are generally enforced by architectural characteristics of the system, they cannot be bypassed just by knowing some secret. In order to disable the protection, an attacker would need to already have control over the system.

Probabilistic Protections

Two familiar examples of probabilistic protections, present in user space too, are the stack canary and Address Space Layout Randomization (ASLR). The stack canary is used to detect the common flaw of a stack buffer overflow in an effort to kill this entire class of bug. The protection, however, depends on the secrecy of the canary value in memory. If this is exposed, the protection can be bypassed by including the canary in the overflow. Similarly, ASLR raises the bar for attackers since they can no longer easily predict where targets are in memory. If the ASLR offset is exposed, then the memory layout becomes predictable again.

The Linux kernel has used a stack canary for a very long time. Recent improvements in the compiler (since GCC v4.9) have allowed for wider coverage of the stack canary protection, with `-fstack-protector-strong`, available in Linux since v3.14 when the kernel build configuration option `CONFIG_CC_STACKPROTECTOR_STRONG` was enabled.

ASLR in the kernel (KASLR) is a contentious issue since there have been a large number of ways to locally expose the offset. However, KASLR isn't limited to just randomizing the position of the kernel code. Improvements have been made to randomize the location of otherwise fixed data allocation positions as well.

KASLR still raises the bar for attackers, especially on systems that run without exposing user space, for example on protocol-only systems like routers, access points, or similar. An attacker facing KASLR risks crashing or rebooting their target if they make a mistake, which leads to very noticeable events from the perspective of the defender.

KASLR of the kernel code itself is controlled by `CONFIG_RANDOMIZE_BASE` and was introduced on x86 in Linux v3.14, arm64 in v4.6, and MIPS in v4.7. Other architectures are expected to gain the feature soon. In the further future, in an effort to address the weakness to exposures, the hope is to reorganize the kernel code at boot instead of just shifting it in memory by a single offset. KASLR of kernel memory is still being worked on, and is similarly architecture-specific. `CONFIG_RANDOMIZE_MEMORY` exists for x86_64 since Linux v4.8, and much of the same effect is already present on arm64 since v4.6.

Another place for randomization in the kernel is the order of the kernel's heap memory layout (not just its base offset). The introduction of `CONFIG_SLAB_FREELIST_RANDOM` in v4.7 (for the SLAB allocator) and v4.8 (for the SLUB allocator) makes it harder for attackers to build heap-spraying attacks. With this protection, an attacker has less control over the relationship between sequential memory allocations (they're less likely to be adjacent). If enough memory is allocated, though, the effect of this protection is diminished. Like KASLR, it raises the bar, if only a little.

Deterministic Protections

Two familiar examples of deterministic protections, present in user space too, are read-only memory and bounds-checking. The read-only memory flag, enforced by the CPU over designated segments of memory, will block any write attempts made within the marked regions. For an attacker trying to redirect execution flow, the less writable memory there is, the less opportunity they have to make changes to the kernel after they have found a stray write flaw. Bounds checking similarly restricts the cases where a stray write flaw may exist to begin with. If every index into an array is verified to be within the size of the given array, no amount of an attacker's wishing will escape the checks.

By far the most fundamental protection in the kernel is correct memory permissions. This is collected under the poorly named `CONFIG_DEBUG_RODATA` [3]. While it was at one time used for debugging, kernel memory permissions are used to enforce memory integrity. And while it once only controlled making read-only data actually read-only, it also now makes sure that the various safe combinations of memory permissions are in place: kernel code is executable and read-only, unchanging data is read-only and not executable, and writable data is (obviously) writable but additionally not executable. Fundamentally, nothing should ever be both executable and writable: such memory areas are trivial places attackers could use to gain control.

In the face of proper kernel memory protection, attackers tend to use user space memory for constructing portions of their attacks. As a result, the next most fundamental protection is making sure the kernel doesn't execute or (unexpectedly) read/write user space memory. The idea isn't new that kernel memory isn't available to user space (this is the whole point of system calls), but this protection is the inverse: user space memory isn't available to the kernel. If an attack confuses the kernel into trying to read or execute memory that lives in user space, it gets rejected. For example, without this protection it's trivial for an attacker to just write the executable portion of their attack in user space memory, entirely bypassing the permissions that make sure nothing is writable and executable in kernel memory.

Some models of CPUs have started providing this protection in hardware (e.g., SMEP and SMAP on x86 since Skylake, and PXN and PAN on ARM since ARMv8.1), but they are still rare, especially on server-class systems. Emulating these protections in software is the next best thing. 32-bit ARM systems can do this with `CONFIG_CPU_SW_DOMAIN_PAN` since Linux v4.3, and 64-bit ARM systems can do this with `CONFIG_ARM64_SW_TTBRO_PAN` since Linux v4.10. Unfortunately, as of v4.10, emulation for SMEP and SMAP was still not available for x86 in the upstream kernel [4].

The places where the kernel explicitly reads and writes user-space memory is through its internal calls to, respectively, `copy_from_user()` and `copy_to_user()`. Since these calls temporarily disable the restriction on the kernel's access of user-space memory, they need to be especially well bounds checked. Bugs here lead to writing past the end of kernel memory buffers, or exposing kernel memory contents to user space. While some of the bounds checking already happens at kernel compile time (especially since v4.8), many checks need to happen at runtime. The addition of `CONFIG_HARDENED_USERCOPY` in v4.8 added many types of object-size bounds checking. For example, copies performed against kernel heap memory are checked against the actual size of the object that was allocated, and objects on the stack are checked that they haven't spanned stack frames.

The kernel stack itself gained protections on x86 in v4.9 and arm64 in v4.10. Prior to `CONFIG_VMAP_STACK`, the kernel stack was allocated without any guard pages. This meant that when an attacker was able to write beyond the end of the current kernel stack, the write would continue on to the next kernel stack, allowing for the (likely malicious) manipulation of another process's stack. With guard pages, these large writes will fail as soon as they run off the end of the current stack. Introduced at the same time, the addition of `CONFIG_THREAD_INFO_IN_TASK` moves the especially sensitive `thread_info` structure off the kernel stack, making an entire class of stack-based attacks impossible.

Future Work

While not yet in the kernel as of v4.10, another interesting probabilistic protection that will hopefully arrive soon is struct randomization [5]. This will randomly reorganize the layout of commonly attacked memory structures in the kernel. This protection is less useful on distribution kernels (since the resulting layout is public), but still makes exploitation more challenging since an attacker now has to track this layout on a per-distribution and per-kernel-build basis. For organizations that build their own kernels, this makes attacks much more difficult to mount because an attacker doesn't know the layout of the more sensitive areas of the kernel without also being able to first gather very specific details through information exposures.

Building on the deterministic memory protection provided by `CONFIG_DEBUG_RODATA`, there has been some upstream work to further reduce the attack surface of the kernel by making more sensitive data structures read-only [6]. While many structures can already be easily marked read-only, others need to be written either once at initialization time or at various rare moments later on. By providing a way to make these structures read-only during the rest of their lifetime, their exposure to an attacker will be vastly reduced.

Another area under current development, as of v4.10, is protecting the kernel from reference-counting bugs. When there is a flaw in reference counting, the kernel may free memory that is still in use, allowing it to get reallocated and overwritten leading to use-after-free exploits. By detecting that a reference count is about to overflow [7], an entire class of use-after-free bugs can be eliminated. The work underway is to create a specific data type that is protected and only used for reference counting, and then replace all the existing unsafe instances.

Staying Updated

By far the best way to protect Linux systems (or any systems) is to keep them up-to-date. This isn't new advice, but it usually only takes the form of recommending that all security updates be installed. While that is absolutely a best practice to adhere to, it only addresses known flaws. The idea must be taken a step further: to get the latest kernel self-protection technologies, systems need to be running the latest Linux kernel.

If products are built using the Linux kernel, they need to be able to receive the latest kernels as part of their regular update cycle. This can end up being a fair amount of up-front cost, since drivers need to be upstreamed and proper automated testing procedures need to be implemented. The long-term results will quickly pay dividends since the burden of code maintenance is shared with upstream and the test environment will catch bugs as soon as they are introduced instead of months or years later.

If systems are built around a Linux distribution, they need to be kept upgraded to the latest distribution release. Many distributions have a "long term support" release that requires waiting a couple of years or more between upgrades. If, instead, a system is upgraded to the regular releases that usually come out on a six-month cycle, they will be much closer to the latest kernel. While distribution kernels will still lag slightly behind the latest kernel release, it's a reasonable tradeoff to make: the system has a more current kernel, but it is still supported by the distribution (unlike rolling your own kernel on top of a distribution).

The work to stay updated tends to be spread thinly across a longer time frame, rather than stacking up only to be addressed in bulk every few years. This generally means fewer emergencies and a smoother planning cycle. Beyond the other benefits of having more modern software, it'll also come with an ever increasing series of defenses designed to stop attacks before they begin.

References

[1] http://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project.

[2] https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/.

[3] Along with CONFIG_DEBUG_SET_MODULE_RONX.

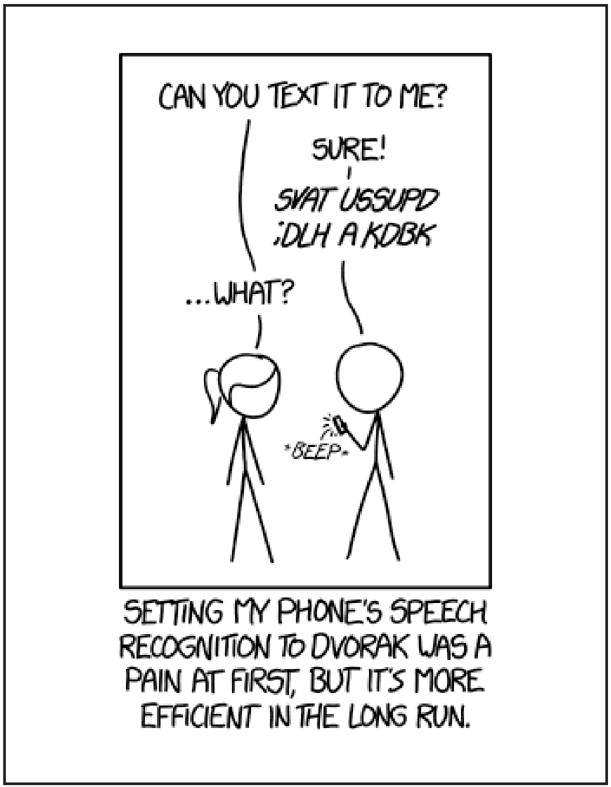
[4] Available in grsecurity via CONFIG_PAX_MEMORY_UDEREF.

[5] Available in grsecurity via the RANDSTRUCT GCC plugin.

[6] Available in grsecurity via CONFIG_PAX_KERNEXEC and the CONSTIFY GCC plugin.

[7] Available in grsecurity via CONFIG_PAX_REFCOUNT.

XKCD

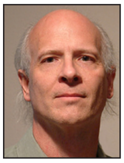

CAN YOU TEXT IT TO ME?
SURE!
SVAT USSUPD ;DLH A KDBK
...WHAT?
BEEP
SETTING MY PHONE'S SPEECH RECOGNITION TO DVORAK WAS A PAIN AT FIRST, BUT IT'S MORE EFFICIENT IN THE LONG RUN.

xkcd.com

Interview with Jeff Mogul

RIK FARROW

Jeff Mogul works on fast, cheap, reliable, and flexible networking infrastructure for Google. Until 2013, he was a Fellow at HP Labs, doing research primarily on computer networks and operating systems issues for enterprise and cloud computer systems; previously, he worked at the DEC/Compaq Western Research Lab. He received his PhD from Stanford in 1986, an MS from Stanford in 1980, and an SB from MIT in 1979. He is an ACM Fellow. Jeff is the author or co-author of several Internet Standards; he contributed extensively to the HTTP/1.1 specification. He was an Associate Editor of *Internetworking: Research and Experience*, and has been the chair or co-chair of a variety of conferences and workshops, including SIGCOMM, OSDI, NSDI, HotOS, and ANCS. jeffmogul@acm.org



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

I'm sure I met Jeff Mogul at a USENIX systems conference, but I can't remember which one. I had heard that Jeff was involved with the early Internet, but later than the groundbreaking work of Internet founders like Vint Cerf and Bob Kahn. And although I occasionally talked with Jeff, I knew little about him.

I did suspect he could shed some light on what it was like to manage an Internet connection in the mid-'80s and to help shape parts of TCP/IP. Jeff had also worked for Digital Equipment Corporation's (DEC) Western Research Lab. WRL was a small research lab in Palo Alto that produced a lot of pragmatic work and many papers too.

Rik Farrow: While at Stanford, you wrote several RFCs, including one about Reverse ARP, that allowed diskless workstations to learn their IP addresses, but also some early work on subnets. Can you tell us a little about how the Internet, and Stanford's Internets, appeared in 1984? I think that there are few people who know about early Ethernet and its limitations, as well as just how small (comparatively) the Internet was in those days.

Jeff Mogul: Actually, I think I had only a minor role in the RARP RFC. The subnet RFCs (RFCs 917, 919, 922, culminating in RFC 950) were more directly my work; I'm proud that Jon Postel co-authored that last one with me.

The Internet in 1984 was probably a lot like it was in 1983, at the time of the "TCP Transition"—I'm sure it had changed, but I don't remember what changed between 1983 and 1984. However, the TCP Transition was one of those events one remembers, because January 1, 1983 was the day that the predecessor to IP/TCP, called NCP, was disabled on the ARPANET, and so anyone who hadn't gotten TCP working by then would have been unable to send traffic [1].

At any rate, Stanford was connected to the ARPANET via Stanford's IMP; I think our IMP was number 11. IMPs had several ports, and so a few large computers could be connected to each IMP. I vaguely recall some kludges that were used to attach others. We also had an early "Experimental Ethernet" donated by Xerox PARC, along with a number of Xerox Alto computers. This Ethernet ran at 3 Mbps, and had 8-bit host addresses. Xerox had also developed a simple internetworking protocol, called PUP (PARC Universal Packet), which added an 8-bit network number, and I believe one could use Altos as routers between PUP networks. Bill Nowicki and I realized we could use some of the Stanford University Network, or "SUN," hardware (this was before Sun Microsystems was started) to build a really simple PUP router so that we didn't need to use precious Altos for that.

Once we realized that IP (and TCP) was coming, we needed a way to route IP packets from the ARPANET (effectively, the backbone of the future Internet) and the Stanford Ethernets. This meant installing an IP router at one of the IMP ports. I can't quite remember the chronology, but I do remember doing a lot of the work of installing and trying to set up this router. We used a PDP-11 for hardware, and I am pretty sure that we used J. Noel Chiappa's "C gateway" software; people then often used the term "gateway" instead of "router." I remember standing in our noisy machine room on lengthy long-distance phone calls to Noel (who was many time zones away) trying to debug his code in our router. That system was named GOLDEN-GATEWAY.STANFORD.EDU and had the address 10.1.0.11—Net 10 was the ARPANET; Stanford was IMP 11; the router was on port 1 of the IMP.

While fact-checking this, I found an old hosts.txt file [2] that included this line:

```
GATEWAY : 10.1.0.11, 36.40.0.62 : STANFORD-GATEWAY : LSI-11/23 :
MOS : IP/GW,GW/DUMB :
```

The MOS suggests that we were indeed using Noel's MIT router software.

At any rate, we also got something working by the TCP Transition date. I still have the button that Dan Lynch gave out, "I survived the TCP Transition." We also connected some of our BSD-based VAXes to the Ethernet via a card we got from Xerox, a driver we got from CMU, and some early IP/TCP software we got from BBN, the builders of the IMPs. I later took the CMU driver and generalized it in several ways. CMU had included a rudimentary packet filter in their driver, inspired by some Xerox Alto code, and I improved it enough to get an SOSP paper out of the deal [3]. Actually, I think we used the packet filter to implement PUP on the VAXes, so that might have happened before the TCP transition.

In those days, "RFC" really did stand for "Request for Comments"; pretty much anyone could write one and get a number assigned, without any actual review. The reason I wrote the original subnetting RFC was because the original "classful" IP addressing system allocated a single Class A network number to Stanford (36, or what we would call 36.0.0.0/8 once CIDR was invented). But we already had a bunch of Ethernets (18 according to RFC 917), so under this scheme we would have needed a lot more network numbers (one for each Ethernet), and we expected the number of Ethernets to grow. That would have bloated the Internet routing tables, still a problem today, even with CIDR. In those days, router memories were small—PDP11s had a 16-bit address space—and there wasn't a lot of spare bandwidth for exchanging routing updates, especially on the 56 Kbps ARPANET. Stanford was one of only a few Internet sites that actually had to worry about multiple subnets, which is why we had to invent the subnetting concept; I also wrote prototype code for BSD UNIX to implement this.

You asked about how small the Internet was in those days. It was definitely small in terms of backbone bandwidth (56 Kbps), the number of hosts (before DNS was invented, there was one Internet-wide "host table" file that we used to map names to addresses—I think SRI maintained and distributed it via FTP), and the number of people. There was a printed book that listed the name, address, phone, and email address of all known ARPANET users. And even in 1986 or 1987, people at academic networking conferences were still trying to figure out whether the Internet would ever be good for much of anything beyond email and FTP.

RF: Around 1987, you also wrote a technical report, and gave talks, about the harmfulness of fragmentation. Why had that become a problem?

JM: Internets can include different kinds of network technology, with different maximum packet sizes (so-called MTUs). Things are more homogeneous now than they were in the 1980s, when Ethernet hadn't quite taken over. At any rate, if you send a packet that fits within the MTU of the first-hop link, but some other link on the path has a smaller MTU, the router forwarding the packet at that point has to "fragment" the packet—divide it into smaller pieces that can be reassembled later. Several of us, including myself and also Chris Kent at Purdue (now Chris Kantarjiev) discovered a problem with fragmentation: sometimes it made TCP almost unusable. Why? Because our primitive Ethernet interfaces (NICs) could only buffer one or two received packets, so if packets arrived faster than the kernel could pull them out of the NIC buffer, some would get lost. This wasn't a huge problem for unfragmented packets, since the TCP receiver would get the first few packets and ACK them, and after a timeout, the sender would retransmit the rest: not ideal, but there was always forward progress.

However, when even one fragment of a fragmented packet is dropped, the receiver cannot reassemble the packet at all, so it is as if the whole packet were lost. To make matters worse, when the TCP sender eventually timed out and re-sent the packet, it would be fragmented again, and lost again with high probability, because these fragments generally arrived in bursts. So: no progress, and TCP users were sad.

This inspired Chris and me to publish a paper at SIGCOMM about the problem, and I led an IETF working group that (after a lot of debate) arrived at RFC 1191, defining "Path MTU Discovery"—which worked unless it didn't, and that's another long story that I mostly left for other people to solve.

RF: You worked on TCP, contributing the first open source firewall software, screend, to BSD UNIX. You later worked on the evolution of packet filtering in BSD, that lead to BPF. If I recall correctly, ULTRIX (DEC's UNIX) was based on BSD. Did DEC use screend as well?

JM: Screend and the packet filter were two mostly separate things. As I mentioned earlier, I think the original idea for packet filtering came from Xerox, but I think they used native code. Rich Rashid and Mike Accetta at CMU were inspired by that to add an interpreted packet filter to their Ethernet driver; interpretation (of a really simple instruction set) made it possible for user-mode programs to provide packet filters that could be safely interpreted within the kernel. I found it helpful to extend their filtering language in a variety of ways, and wrote the 1987 SOSP paper [3] describing this. But the so-called CMU-Stanford Packet Filter language was a rather inefficient stack-based execution model, and mostly one had to hand-code the filters. The Berkeley Packet Filter [4] replaced this with a register-style execution model, and they wrote a compiler for it, so overall it was much nicer, although I still think I had a cleaner solution for enabling programs such as tcpdump to put the Ethernet driver in "promiscuous mode" without having to make these programs setuid-root…but that's orthogonal to the interpreter design.

Screend came a few years later. Most of the BSD community gathered once or twice a year for a BSD summit meeting, and I believe we were at Berkeley for one of those the day that the Morris worm was unleashed. Bad timing! While that allowed a lot of people to focus on stopping the worm, they weren't able to install the patches needed.

Suddenly everyone realized that the original vision of the Internet as a place where any host could send any packet to any other host was actually not such a good one. The military had already realized this, and I think they installed "mail gateways" between the ARPANET and MILNET so that only email could get through; the rest of us thought that was rather typical of the military mind. So people started writing what we now call "firewalls."

I had already worked with Deborah Estrin (then of USC) and some of her grad students on a cryptographic approach of hers called "Visa protocols." With several decades of hindsight, you could call these "stateless SDN firewalls," since the Visa mechanisms used policy controllers separated from the routers. I believe our paper on this work was published after the Morris worm, but it was started earlier.

Anyway, at DEC in Palo Alto, Richard Johnsson (and perhaps others) needed to protect their computers against the Morris worm (and any copycats) *right now*, so he hacked a simple firewall into the BSD kernel. I think it either had a hard-coded ACL table, or perhaps there was a way to update it, but it wasn't very flexible or scalable. So I sat down and wrote screend, which did all of the fancy processing in user-mode code (in that respect, kind of like the packet-filter idea) and then kept a small cache of recent decisions in the kernel. It worked pretty well, I got a USENIX paper [5] out of the idea and helped DEC put it into the ULTRIX product, from which some colleagues ultimately built a (small) firewall business around it. I think my code even made it into the first setup for whitehouse.gov [6].

Yes, DEC's ULTRIX was very closely based on BSD, but of course with some DEC-specific additions, testing, documentation, etc.

RF: Right at the point where the Internet was growing exponentially, you worked on HTTP 1.1. What changes were you suggesting to improve the performance of HTTP around the mid-'90s?

JM: The original HTTP protocol would open a new TCP connection for each request, and then close it once the response was read. This turns out to make things really slow, because each request had to wait for the TCP handshake, which adds a network round trip. Network round-trip times (RTTs) are often tens or even hundreds of milliseconds and are the bane of good performance. Actually, it often added a lot more delay, because networks used to lose a lot more packets, and if your SYN was lost, your TCP had to time out and try again. Timeouts are usually much longer than RTTs. The other problem with the request-per-connection model was that each request-response transaction was serialized behind the previous one.

By making the TCP connections persistent [7], we avoided the setup costs. But we also enabled the use of "pipelining," a concept from computer architecture in which you can have several operations in flight at once. Since a typical Web page involves lots of HTTP requests (for images, CSS, etc.), once your browser downloads a page's HTML, it typically makes a large number of subsequent requests from the same server. With pipelining, the browser can launch a lot of those requests before any of the responses get back; this effectively allows us to hide all but one RTT.

Various things make persistent connections and pipelining harder to exploit in practice than we first realized; there are too many HTTP/1.1 servers that misbehave when asked to pipeline, so we had to wait for HTTP/2 before it became consistently safe to use. It took too long, but I think it proved to be a good idea.

RF: After you got your PhD from Stanford, you went to work for DEC's Western Research Lab in Palo Alto, California. What was it like to work in a research lab? Did you have total freedom to pick what you wanted to work on?

JM: WRL was an unusually wonderful environment. I don't think we ever had more than 25–30 researchers, and small number of other staff, but WRL people not only invented a lot of cool things at DEC, but many of them have become stars at other companies. I now work at Google, where many of our technical leaders started at WRL. Also, WRL hired people who were both talented and genuinely fun to work with—I have more friends from WRL than from any other era of my career.

WRL was even smaller when I joined, and it was just getting out of a narrow focus on building the first practical RISC computers, called Titans. In many ways, it was an academic environment—we hired people the same way that universities hire professors, we published papers, and we solved hard problems. However, we had more ability than universities to have a large group of people work on a single system, and we had the resources to build real hardware.

While many of us tended to look for our own problems to solve, within the context of the lab's mission (and we occasionally agonized over defining a mission statement), one would have to have been a fool not to remember that our nice salaries and offices were paid for by a profit-oriented business. WRL people wanted to change DEC (initially, by trying to convince DEC that RISC machines would be half as expensive as CISC machines), and so we tended to focus on solving problems that we thought the company needed to have solved. Sometimes we were willing to get ahead of DEC (as with RISC, and much later with AltaVista), but we realized that we needed to do things in a way that DEC could adopt without having to change lots of things at once. So, for example, we usually focused on C-based software, while our sister lab in Palo Alto (SRC, the Systems Research Center) focused on building clean-slate, top-to-bottom re-designs that promised much more wonderful results—but were really hard for DEC to absorb.

As a junior member of the lab, I was encouraged to spend some time following my own interests, but it was also made clear to me that I needed to commit substantial time to a project that contributed to the overall goals of the group. So, for example, my first major effort was to port the BSD networking stack into the Titan operating system, Tunix, a rather bizarre combination of some older BSD UNIX plus a lot of code written in Modula 2. Anita Borg, who joined WRL at about the same time, did her first major work on adding demand paging to Tunix.

RF: Any thoughts on the apparent decline of research labs, like WRL and Bell Labs?

JM: WRL declined rather suddenly. Compaq bought DEC in 1998 and absorbed the three existing research labs (WRL, SRC, and the Cambridge Research Lab) more or less intact, since Compaq had never had its own research organization. The Compaq experience had its good years, but by the end there just wasn't enough money to make things work, plus we were under some VPs who were not ideally suited to running a research organization. HP bought Compaq in 2002 and incorporated WRL, SRC, and CRL into HP Labs. Originally the idea was to keep our groups as separate parts of HP Labs, but that was unsustainable: while the DEC labs were fairly generalist, the other HP labs were very topic-focused, and the other lab directors apparently didn't like the idea of keeping our labs around. Shortly after that merger, WRL's director left to become an early Google employee, and after a few months of a rather uninspiring search for another director, HP dissolved WRL and moved us into the rest of the organization. To HP's credit, any WRL person who decided to leave at that time was compensated as if they had been laid off, and HP was still generous with layoff packages in 2002. SRC and CRL lasted somewhat longer.

I stayed at HP Labs for a decade, and for a while it was still a good place to do corporate research, but there were few upticks in a general decline. One person in particular did a lot of damage to the long-term prospects of HP Labs; that's a complex story, but I think the bottom line is that it is at best extremely hard to get value out of a corporate research lab these days, compared to simply waiting for a startup to invent what you need. The problem is that the typical reaction is to manage the research organization more intensely. ("You will innovate or else! And by the way, here are some stricter rules for how you will be creative.") I believe that's exactly backwards; I think Rick Rashid had it right when he said that as leader of Microsoft Research, he tried to ensure that they hired extremely carefully, and then he got out of the way. My view is that if you hire only researchers who are smart, who understand what the company needs, and who are internally motivated to make the company succeed, then a research lab has some chance of delivering value to the company, without micro-management from above.

But today, even that might not be enough for a corporate research lab to compete either with the massive number of startups or with companies like Google that integrate researchy people into product groups. And, in any case, once you've hired badly, you end up with an organization full of people who do not self-motivate in the right direction, and then you have to manage them aggressively, and from that you can never work your way back to a team of self-motivated, creative people.

The other big problem with corporate research labs is when the company's product groups aren't allowed to reserve some spare resources, for working collaboratively on tech transfer with researchers while the technology is still a bit risky. Tech transfer does still happen to those product groups, but typically it gets delayed until the group realizes it has to catch up with competitors. So the research result doesn't have an effect until it's too late to gain a real advantage from it. Researchers can still have a big impact on products by providing guidance and design reviews, but when the VP of Research only knows how to claim success for big-splash inventions, mere expertise-transfer isn't visible enough to get support or credit.

HP Labs is still hanging on (now in two separate companies, after HP split up), and there are still some smart people there, but with top people quitting every month, I don't think it will be interesting for much longer.

I have no direct experience with Bell Labs (or with AT&T Labs Research), so you should probably ask other people for those stories.

RF: In 2008, you were involved with a group talking about the future of system conferences. Did anything actionable come out of those discussions?

JM: Some discussions never end. I recently joined the NSDI Steering Committee, and we're currently in the middle of two different email threads about how to make systems conferences work better.

I suspect the discussions about the future of systems conferences started around five minutes into the first SOSP. That is, over 50 years ago. If two or more systems researchers are sitting in a bar, or going on a hike, or waiting for a bus, they will probably start discussing what is wrong with system conferences and how to fix them. For all I know, snake researchers also sit around moaning about the sorry state of herpetology conferences…but systems researchers are a bit weird in that, unlike almost all other scientific and engineering fields, we often put more emphasis on conference papers than journal papers, so we might be unusually interested in how conferences should be organized.

After joining more than my fair share of such BS sessions, and chairing a few conferences, I thought, "What better way to solve the problems of computer systems conferences and workshops than to have a workshop on that?" So I talked USENIX into letting me a run a workshop, WOWCS (Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems), co-located with NSDI '08, and we got a pretty nice selection of papers, plus a rousing discussion (which we wrote up as a *;login:* article in August 2008 [8].

People made some interesting proposals, but I haven't gone back over the material to see whether any of them bore fruit. There were a few papers on tools that have become indispensable (HotCRP and banal). Tom Anderson wrote a follow-up paper ("Conference Reviewing Considered Harmful" [9]) that presented some great data showing that PCs should not make their decisions based on reviewer scores; after that, when I've chaired PCs, I've warned people not to argue "we should take paper X over paper Y because it had a higher average score"—that's just amplifying some noise.

Since then, I participated in another workshop debating "publication culture in computing research" that wasted considerably more CO2, but I don't think it led to much change, either.

I think our emphasis in computer systems on conferences is the worst possible system…except for all of the other ones. In particular, I think when PC chairs pick well-intentioned PC members and run a face-to-face PC meeting carefully, the social structure of the meeting encourages reviewers to discuss papers with great passion and great integrity, because it's hard to hide bad or lazy behavior. I'm not sure how else to get that kind of combination.

References

[1] Flag day: https://www.internetsociety.org/blog/2013/01/30-years-tcp-and-ip-everything.

[2] Example of hosts.txt: https://emaillab.jp/pub/hosts/19840113/HOSTS.TXT.

[3] J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The Packet Filter: An Efficient Mechanism for User-Level Network Code," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP)*, 1987, pp. 39–51.

[4] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture,"in *Proceedings of the Winter 1993 USENIX Annual Technical Conference*, pp. 259–269.

[5] J. C. Mogul, "Simple and Flexible Datagram Access Controls for Unix-Based Gateways," *in Proceedings of the Summer 1989 USENIX Technical Conference,* pp. 203–221.

[6] Section 1.2 mentions using screend as part of the firewall for whitehouse.gov: http://www.fwtk.org/fwtk/docs/documentation.html.

[7] J.C. Mogul. 1995. "The Case for Persistent-Connection HTTP," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '95)*, pp. 299–313. See also http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-4.pdf.

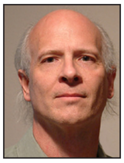
[8] J. C. Mogul (summarizer), "WOWCS '08: Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems," *;login:*, vol. 33, no. 4 (August 2008; online only): https://goo.gl/Gv5BPI.

[9] T. Anderson, "Conference Reviewing Considered Harmful," *SIGOPS Oper. Syst. Rev.,* vol. 43, no. 2 (April 2009), pp. 108–116. See also https://homes.cs.washington.edu/~tom/support/confreview.pdf.

Interview with Amit Levy

RIK FARROW

Amit Levy is a PhD student in the Department of Computer Science at Stanford University. His work focuses on building pragmatic, secure systems that increase flexibility for application developers while preserving end-user control of private data. amit@amitlevy.com



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

I'd met Amit Levy a couple of times during luncheons at system conferences. Amit is not shy about talking about his projects. I liked hearing about them, as Amit would clearly tell me about the motivations behind his projects and answer any questions I had.

So this time after we talked at OSDI '16, I asked him if I could create a more formal version of our post-luncheon conversations, and he agreed. In particular, we talked about his work on Tock using Rust and leveraging type safety.

Rik Farrow: You've done a lot of things, including your side-project MemCachier [1], but you've published more about security-related topics. What got you interested in building a replacement for TinyOS [2]?

Amit Levy: Almost all of my work has had something to do with using type safety as a means of building secure systems. Even MemCachier really started as a an exercise to learn Go and with the idea that building a memcached clone in a type-safe language would make it relatively easy to also build a safe, multi-tenant cache service. So, in that sense, rethinking the embedded operating system in the context of IoT security was a pretty natural extension of much of what I'd been working on, just a different application space. For *me* the exciting thing about Tock [3] is really figuring out how to provide safety and isolation properties to a system with extremely limited resources. And the context is allowing IoT platforms to run untrusted programs.

The actual story is just more coincidental. My roommates and I wanted to build an automatic lock for our front door after we forgot to lock it a couple times and two of our bikes were stolen. So I started looking into IoT and, particularly, low-power computers and Bluetooth low energy. Phil Levis was also interested in Bluetooth (for much less frivolous reasons), so we started reading the spec together and talking about ideas. Eventually, Phil, Prabal Dutta, and David Culler decided their students should start having weekly phone calls about software/hardware co-design, and the need for a replacement for TinyOS just came out of those weekly phone calls.

RF: You've mentioned that Tock will run on a SAM4L processor, which certainly does appear to be low power, as well as much simpler and much slower (under 100 MHz clock) than what most systems use. Do platforms like this have any hardware features that support security, things like memory management or the system call interface?

AL: Yes. Most of the new ARM Cortex-M series microcontrollers (including the SAM4L) have a feature called a memory protection unit (MPU). The MPU does not provide memory virtualization (so there is only a single address space) but does enable setting read/write /execute permission bits on ranges of memory as granular as 16 bytes. In fact, Tock uses the MPU to enable a limited number of traditional OS processes. ARM also recently released a specification for TrustZone-M, which has similarities to TrustZone on "application"-grade ARM processors like the ones in our cell-phones. TrustZone-M has some additional interesting features (like allowing interrupts to trap to untrusted code directly), which could help increase performance of embedded systems that rely on hardware protection. I think we're expecting to see some SoCs (system-on-chip) with TrustZone-M available in the next couple of years.

However, there just isn't enough memory on these microcontrollers to use a protection model based on memory isolation (e.g., processes) as a ubiquitous means of protection in the system.

In general, though, I think the simplicity of microcontrollers can be viewed as a hardware security feature. What I mean is that in many use cases, we also care about hardening embedded systems against hardware-based side-channel attacks—like timing and power analysis. TPMs (trusted platform modules), two-factor authentication devices, and HSMs (hardware security modules) are a few examples of systems where it's really important to mitigate side-channel attacks. To thwart these attacks, it's important for the hardware to be simple. Caches, like the TLB on higher-grade processors, are notoriously leaky.

RF: How does type safety improve security?

AL: Type safety serves two primary roles. It helps programmers avoid many common errors like buffer-overflows. When hardware protection is available, it's possible to catch some of these kinds of bugs at runtime. Type safety lets us catch them at compile time, before we run our program, and saves us from them when hardware protection isn't an option.

The second role is that we can leverage type safety to express really fine-grained security policies. For example, hardware protection lets me expose only certain regions of memory to untrusted code—say a memory-mapped I/O register. However, I have no control over what values are written to that memory. Type safety lets me restrict the manner in which the untrusted code uses a region of memory. For example, I can ensure that only a certain range of values is ever written to a particular register or that the value was created by a trusted module. Importantly, the compiled binary looks nearly identical to one compiled from source code in C that doesn't have these protections. There's nothing particularly magical going on. The type system just lets the compiler reject code that violates certain rules, and, in most cases when we're writing C, we don't really want to violate those rules anyway.

RF: So you have some untrusted code, and you can't distinguish it from code written in C once it's compiled. That implies to me that you can't rely on type safety here, because the untrusted code could have been compiled from C, and thus you don't know what types it can write to your target memory. I am likely just missing something here, so could you clear this up?

AL: You're right, if all you have is a pre-compiled binary, the type system doesn't help. You have to be able to compile the code yourself. In Tock, this is part of what motivates which systems components go where. Applications, which may even be loaded by an end user in some cases, typically live in a process. The process is isolated by hardware protection, so it doesn't rely on the type system and a binary is fine. Conversely, components like peripheral drivers are specific to a hardware platform—my particular embedded product has a different set of sensors, actuators, radios, etc. from other embedded products—but don't change when I change applications. The system integrator wants to make sure that if they use a driver for a particular temperature sensor they found on the Web that it's not able to leak secret encryption keys or access other peripherals on the same bus, but if they can verify safety when they compile the kernel that's fine.

RF: In some of your work [4], you talk about problems you have when using Rust. Can you explain?

AL: Rust kind of provides the lowest-level of abstraction you need to guarantee type safety. This ends up surfacing some fundamental safety tradeoffs into the language. One of the simpler examples is that if you want to use closures-based callbacks (e.g., as is common in Node.js), you need to dynamically allocate those closures—they can't be on the stack or statically allocated. Most type-safe languages assume that more or less everything is dynamically allocated, so this is implicit, while in Rust it's explicit.

In Tock, we disallow dynamic allocation in the kernel (that's a common practice for reliable systems), so this is good for us because it means we can use closures as long as we can prove to the compiler that they don't need to be dynamically allocated. However, it also means that when we try to adopt common coding styles from other frameworks that don't actually work with our system constraints, we get a compiler error. I think it's tempting as a systems builder to look at type-safe languages and think that they are magic, and so you get to stop thinking about system constraints. That's not true. There's nothing magic about type safety. It just lets you guarantee things you already knew how to do.

Unfortunately, I think it's easy to draw the wrong conclusion from that paper—that there are drawbacks with Rust that are artifactual rather than fundamental. There were three issues that we ran into building Tock in Rust, and all three of them turned out to be fundamental (or at least nearly fundamental) and, on balance, were the right design decisions for the language. There is a great paper by Dan Grossman from 2002 called "Existential Types for Imperative Languages" [5] that explains this really well. If you're going to read our paper, it's worth reading that one as well.

References

[1] MemCachier: https://www.memcachier.com/.

[2] TinyOS: http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Documentation_Wiki.

[3] TockOS: http://www.tockos.org/.

[4] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto, "Ownership Is Theft: Experiences Building an Embedded OS in Rust," in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*, October 2015: https://sing.stanford.edu/site/publications/59.

[5]: Dan Grossman, "Existential Types for Imperative Languages," in *Proceedings of the 11th European Symposium on Programming Languages and Systems (ESOP '02)*, pp. 21–35: https://homes.cs.washington.edu/~djg/papers/exists_imp.pdf.

MarFS, a Near-POSIX Interface to Cloud Objects

JEFF INMAN, WILL VINING, GARRETT RANSOM, AND GARY GRIDER

Jeff Inman is a Software Developer in LANL's High-Performance Computing Division, with surprisingly many decades of research experience in areas including parallelism, bioinformatics, GPUs, compilers, embedded computing, and scalable storage. jti@lanl.gov

Will Vining graduated from the University of New Mexico with a bachelor's degree in computer science in 2016. He is currently a graduate student at LANL and is one of the primary developers for MarFS. wfvining@lanl.gov

Garrett Ransom is a recent employee of LANL's High Performance Computing (HPC) Division. As part of the Infrastructure Team, Garrett performs system administration and assists with the development of storage technologies. gransom@lanl.gov

Gary Grider currently is the Division Leader of the High Performance Computing (HPC) Division at Los Alamos National Laboratory. Gary is responsible for all aspects of High Performance Computing technologies at Los Alamos. ggrider@lanl.gov

The engineering forces driving development of "cloud" storage have produced resilient, cost-effective storage systems that can scale to 100s of petabytes, with good parallel access and bandwidth. These features would make a good match for the vast storage needs of High-Performance Computing datacenters, but cloud storage gains some of its capability from its use of HTTP-style Representational State Transfer (REST) semantics, whereas most large datacenters have legacy applications that rely on POSIX file-system semantics. MarFS is an open-source project at Los Alamos National Laboratory that allows us to present cloud-style object-storage as a scalable near-POSIX file system. We have also developed a new storage architecture to improve bandwidth and scalability beyond what's available in commodity object stores, while retaining their resilience and economy. In addition, we present a scheme for scaling the POSIX interface to allow billions of files in a single directory and trillions of files in total.

HPC Storage Challenges

The issues faced by extreme-scale HPC sites are daunting. We use Parallel File Systems to store data sets for weeks to months, with sizes in the 100s of terabytes, and bandwidth on the order of 1 TB/sec. On the other hand, our parallel archives are used to store data forever, but can only support speeds of 10s of GB/sec. MarFS was designed to provide an economical middle-ground between the expensive capacity of PFS and the expensive bandwidth of tape, storing data sets for years, with speeds of 100s of GB/sec.

The supercomputers generating the data that is ultimately stored in MarFS are currently in the millions of cores, and multiple PBs of memory, and are expected to grow to a billion cores and 10s of PBs of memory beyond 2020. Applications that produce one file per process on such machines could produce billions of files, which a user may want to keep in a single directory. Furthermore, as we push to add value to the data we store, we expect file-oriented metadata to grow by perhaps orders of magnitude. The goal is for MarFS to easily handle up to multi-PB-sized data sets, as well as metadata for billions of files in a single directory, and 10s of trillions of files in aggregate.

Modern "cloud" storage systems provide a way to scale data storage well beyond previous approaches, using sophisticated, highly scalable erasure-coded protection schemes. These systems would allow us to build very reliable storage systems out of very unreliable (and therefore inexpensive) disk technologies. The metadata underlying cloud storage is basically a flat metadata space, which also scales very well. Reliability, economics, and scalability combine to make this technology appealing to many large-data sites. For HPC, the problem with these storage systems is that they only provide simple get/put/delete interfaces using object-names, rather than POSIX file-and-directory semantics (files, directories, ownership, open/read/write/close, etc.), and most HPC datacenters need to support legacy applications that rely on POSIX semantics. It became clear from a market survey that other products that provide POSIX-like access to scalable cloud objects were not designed to handle PB-sized files, or billions to trillions of files.

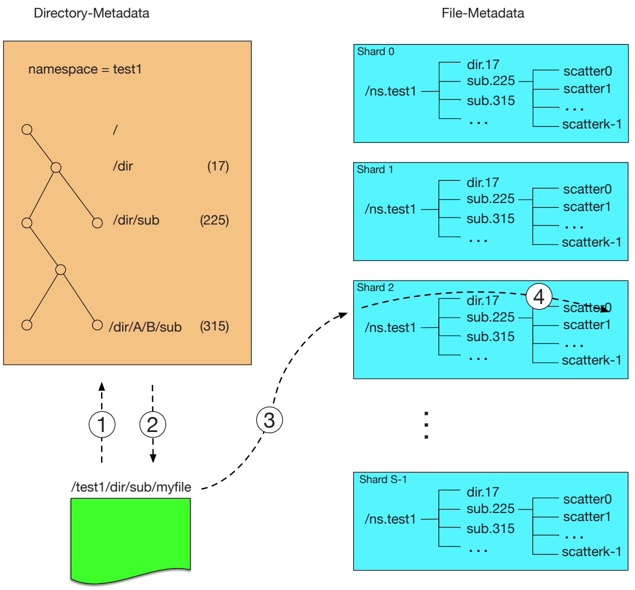


Figure 1: Storing metadata (MD) for a new file having path `/test1/dir/sub/myfile`. A directory-MD Server (dMDS) holds directory MD, and a set of file-MD Servers (fMDS) hold parts of the file MD. (1) The dMDS is consulted for access-permissions (if not in cache). (2) The dMDS also returns the inode of the leaf directory (e.g., 225). (3) A hash of the file-path, modulo the number of fMDS shards, selects the shard to hold this file MD. (4) The file-path hash, modulo number of internal "scatter" directories, identifies the internal subdirectory for the MD.

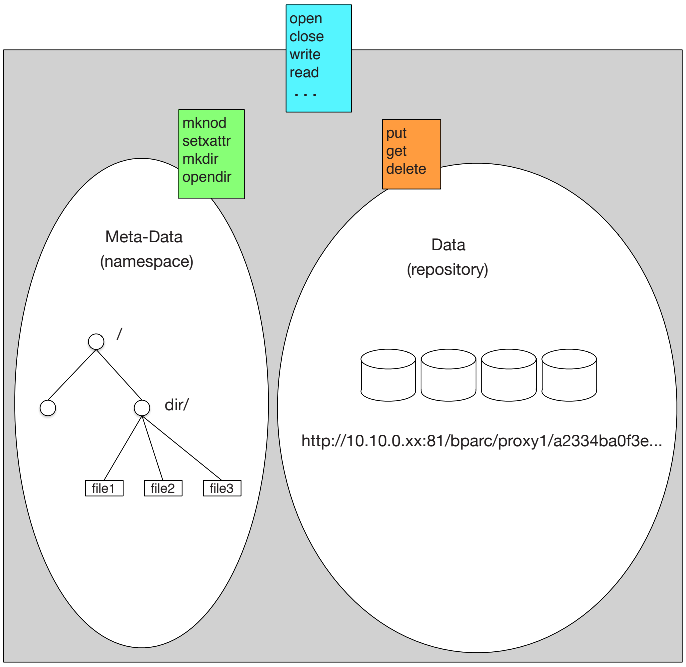


Figure 2: The default metadata scheme uses a regular POSIX file system to represent files, with object-storage holding file contents. The file system must support sparse files and extended attributes. Data and metadata schemes are installed as modular DAL and MDAL implementations, respectively.

MarFS is an open-source software technology developed at LANL to bridge this gap, putting a highly scalable POSIX metadata interface on top of highly scalable cloud object systems, making object storage systems usable by legacy applications. MarFS scales data capacity and bandwidth by splitting data across many objects, or even many object systems. For metadata, MarFS is designed to scale capacity and bandwidth in two dimensions. Currently, directory-metadata is scaled by simple directory decomposition high in the tree. We've developed a prototype file-metadata service, where we've demonstrated scaling metadata by sharding it across many file systems, as illustrated in Figure 1. This metadata sharding is not yet in use in the production version of MarFS.

MarFS Implementation Overview

Figure 2 shows the basic components of MarFS. There is a metadata implementation that handles file and directory structure, and a data implementation that stores file contents. In the default metadata implementation, user directories are implemented as regular directories, and user files are represented as sparse files truncated to the size of the corresponding data, with hidden extended attributes that hold system metadata (e.g., object-ID). This gives us basic POSIX access-control "for free."

Object-storage systems typically have a range of object-sizes for which internal storage and/or bandwidth is optimal. When storing data for files larger than this, we break the data up into distinct objects ("chunks"), transparent to the user. We refer to such multi-object files as "multi-files." Allowing data to be inserted or deleted in the middle of a multi-file (or to create sparse files) would require metadata machinery that would compromise the performance and scalability of parallel accesses. Therefore, we don't allow it. This makes us "not quite POSIX," but we gain trivial stateless computation of the object-ID and offset corresponding to any logical offset in a file, maintaining efficiency for parallel reads and writes.

Millions of small files pose another kind of metadata hazard in that they may invisibly consume significant resources from the object-store. We work around this by transparently packing many small files together into a single object, although they appear to users as distinct files. The packing is done dynamically, during data-ingest, by `pftool` (discussed below), so the packed files will typically be found together in a directory traversal, and are likely to be deleted together, avoiding packed files with many "holes." Nevertheless, we are also developing a repacker, so that multiple "Swiss cheese" packed files can be repackaged into fewer objects.

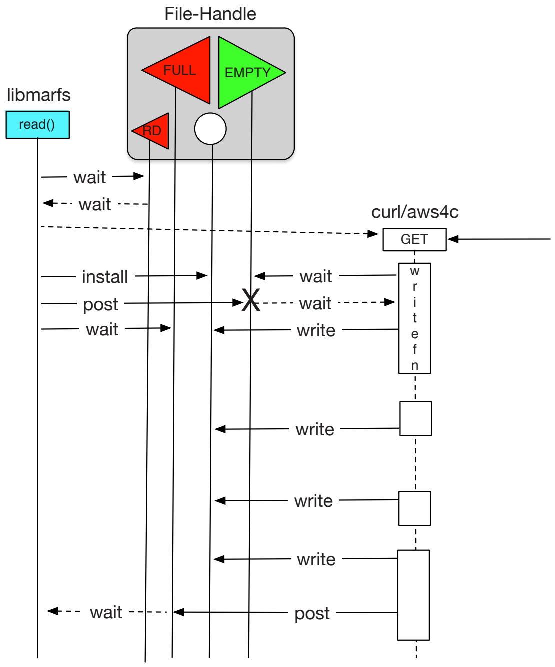


Figure 3: Sequence diagram showing interactions between a user performing a read, a file-handle containing locks, and a thread performing a GET operation on an object. The GET thread receives callbacks from `libcurl` and uses locking to coordinate across multiple `read()` calls. The colors of the locks (red/dark gray = FULL and RD, green/light gray = EMPTY) in the file-handle are represented at the moment marked "X". The circle in the file-handle is a pointer to the caller's read-buffer.

The internal MarFS data-interface must translate between the POSIX file-system semantics seen by applications (open/read /write/close) and the RESTful semantics of an object-store (get /put/delete). We do this by assigning an ongoing GET or PUT transaction to a thread at "open"-time (or at the time when data is first read-from/written-to an object). This thread can block in the `libcurl` callbacks that move data on behalf of the transaction. MarFS read or write requests then provide buffers that allow the callbacks to unblock for long enough to write data from a caller's write-buffer to a PUT, or receive GET data into a caller's read-buffer, before blocking again. When object-boundaries are crossed in a multi-file, MarFS transparently ends one transaction to the old object and begins a new transaction to the corresponding second object. This is depicted in Figure 3.

MarFS is driven by a configuration-file, allowing specification of details like the layout of namespaces and repositories, object chunk-sizes, resource quotas, types of access that are enabled, file systems used for metadata, etc.

Flexibility

Our initial development utilized an object store supporting the S3 protocol, but we are now in production with a Scality RING, using Scality's `sproxyd`. This protocol eliminates the need for maintenance of some internal S3 metadata, improving bandwidth. However, in our relentless quest for economical capacity and bandwidth, we have developed an alternative to cloud-style object-storage, doing our own erasure coding and storing the coded parts in distinct ZFS pools, which themselves are also erasure protected, forming a two-tier erasure arrangement.

Intel's Intelligent Storage Acceleration Library (ISA-L) provides an efficient implementation of Galois Field erasure code generation, allowing an arbitrary number of erasure blocks to be generated for a set of data blocks. Up to that number of corrupted blocks can then be regenerated from the surviving data and erasure blocks. We wrapped ISA-L functionality within a utility library (`libne`) to provide POSIX-like manipulation of sets of data and erasure blocks through higher-level `open`, `close`, `read`, and `write` functions. For example, data provided to the high-level `write` function is subdivided into N blocks. The functions of ISA-L are applied across the N data blocks to produce E additional erasure-code blocks, making a "stripe" of N+E blocks. The stripe is then written across N+E internal files, with one block per file.

The failure tolerance of the system depends on the number of erasure blocks produced. Given (N+E) blocks written with `libne`, we can survive the complete loss of up to E blocks of any stripe. If desired, checksums are also calculated across each block, providing a means of identifying corrupted blocks while reading, and are stored within either the parts themselves or in their extended attributes. Both N and E are configurable, allowing for a customized balancing of the tradeoffs between computation overhead and reliability.

Should a problem be detected, whether that be in the form of a corrupted block, offline server, failed disk, or a failed checksum verification, the erasure utilities will continue to service read requests by automatically performing regeneration on the fly. Such reads will also return an error code, indicating the blocks that are corrupt or missing, but will not attempt to repair the stored data itself. This approach preserves information about failures while avoiding interference with other ongoing accesses.

The Data and Metadata Abstraction Layers (DAL/MDAL)

The desire to experiment with swapping out storage-protocols leads us to the idea of a Data Abstraction Layer (DAL). This is an abstract interface to internal RESTful storage functions (e.g., GET, PUT, and DELETE), which can be implemented and installed in a modular way, swapping out the storage component of Figure 2. We have used this approach to provide a new kind of scalable data-store based on libne, where erasure-coded blocks are written across a set of independent file systems. This should allow us to overcome the overhead of the internal communication and metadata management required of an object-store, improving our overall storage throughput without compromising reliability. We refer to this architecture as multi-component storage.

We refer to a storage-server and its associated JBODs as a Disk Scalable Unit (DSU). A DSU holds one or more *capacity units*, and each capacity unit hosts an independent ZFS pool. All DSUs have an identical configuration of capacity units. So, to expand capacity, one would add an identical new capacity unit to every DSU. ZFS provides its own erasure encoding and checksum protection for each data and erasure block, but it remains vulnerable to large-scale failures. To maximize resilience and bandwidth, each of the N+E files of a stripe is written to a different DSU, all on the same-numbered capacity unit. Thus, we can survive the complete loss of any E DSUs in the set of N+E that hold an object.

The parallel nature of this design allows for independent read /write operations across each of the ZFS systems, without the opacity and overhead of an object store. Our expectation is that this architecture will provide improved bandwidth, with more than sufficient reliability.

Multi-component (MC) storage is realized as an implementation of the Data Abstraction Layer, utilizing libne to perform low-level accesses. The MC DAL depends on a directory tree of NFS mounts, which groups capacity units (hosting ZFS pools) into DSUs, and DSUs into *pods*, as shown in Figure 4. A pod is just a set of N+E DSUs, where N and E are the parameters of the erasure coding used in the repository. The blocks of a stripe are written across a pod, starting at some DSU and wrapping within the pod.

To reduce the number of files in any one of the internal directories of the individual storage systems, we add another layer of k sub-directories (scatter0, scatter1, etc.) inside each ZFS pool. For a repository that has 3+1 erasure coding, two pods of four DSUs, and two capacity units per DSU, the directory *scaffolding* might look like this:

```
/repo3+1/pod[0..1]/block[0..3]/cap[0..1]/scatter[0..k-1]/
```

To determine the location of the blocks for an object, we compute a hash of the object-ID and use that to fill in the pod and scatter-directory, in a path-template provided by the MarFS configuration. For new data, computation of the capacity-unit may follow from policy guidance (e.g., favor newly added capacity, or spread load in a given ratio) rather than a simple hash. Filling-in these fields of the scaffolding template produces a new template (shown below), which is used by libne, along with a starting block (also computed from the hash), to write the object across the N+E independent storage systems in the selected pod:

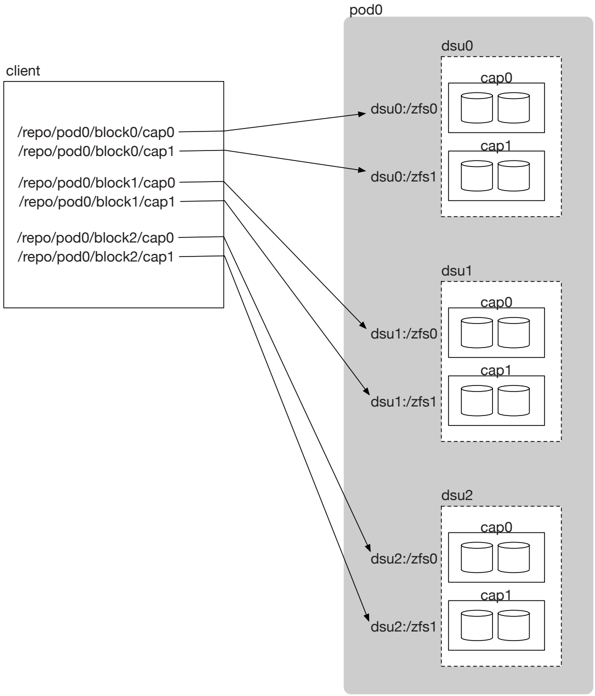


Figure 4: NFS mounts and exports supporting the multi-component DAL. This example shows a single "pod" of 3 DSUs (e.g., N=2, E=1), each having two capacity units. The capacity units each host a single ZFS file system which is exported via NFS. On the client, NFS mounts are made to each of the exports. A stripe of three blocks would be written across the DSUs. The *scatter* directories are internal to the ZFS file systems and are not shown here.

```
/repo3+1/pod1/block%d/cap1/scatter7/object-id
```

In stripes where some blocks are all-zero, ZFS can store the zero blocks much more compactly. By computing the starting block from the hash, we can ensure that capacity is utilized at roughly the same rate in each ZFS pool; otherwise, the capacity in block0 might be used up more quickly if a large number of small objects are created. For access to existing data for which the capacity unit can't be predicted from metadata (e.g., from the creation-date), we will generate a set of paths covering the available capacity units and issue *stat* requests to all of them in parallel.

The MC DAL is configurable on per-MarFS-repository basis, allowing for different storage configurations to be used simultaneously. The configurable parameters are the path template, the number of pods, the erasure parameters (N and E), the number of capacity units per DSU, and the number of scatter directories in each capacity unit.

Multi-component storage provides a high level of data integrity through two layers of erasure coding; data on any individual disk is recoverable in two decoupled erasure regimes. ZFS allows recovery of individual blocks, and data-blocks are stored along with erasure-blocks across ZFS pools. Even moderately sized multi-object files will tend to have objects in all pods. Because the pods are independent, we could lose E pools from each of the pods without data loss.

In conjunction with libne, the MC DAL can read through missing blocks or corrupted data. Errors are detected when an object is read. When that happens, the object-ID is flagged as degraded and logged to a file so the object can be rebuilt, either by an offline program run by an administrator, or by a daemon that is notified when there is rebuild work to be done.

We also support a Metadata Abstraction Layer (MDAL), allowing modular replacement of the metadata system. This is how we would swap-in something like the scalable MD system of Figure 1, replacing the metadata implementation in Figure 2.

Metadata Performance

MarFS teammates wrote an MPI application to measure pure metadata (MD) performance and scalability in the forward-looking scheme of Figure 1. The goal was to benchmark only internal MD activity, ignoring any overhead associated with the persisting of data or metadata. Thus, we installed a "no-op" DAL that does nothing for data-write operations, and an MDAL that integrates with the application. Specific MPI ranks acted as clients, file-MD shards, a directory-MD shard (one instance only), or as the master. File and directory MD were stored in tmpfs. Clients performed scripted MD operations, organized by the master rank.

Using 8800 * 16 cores, and one MPI rank/core, we were able to create approximately 820M files/sec, and we stored 915 billion files in a single directory. Because the MD is distributed, and resides in a broad directory-tree per shard, a *stat* of any one of these files can return quickly. We are exploring semantics for parallel *readdir* and *stat* in this model.

Data Performance

Our production hardware uses SMR drives everywhere, and there has been concern about sustained throughput in this technology. On an object-storage testbed with 48 DSUs, we were able to achieve 28.5 GB/sec, for sustained low-level writes. With production workloads on similar hardware (but with incomplete JBODs), we are typically seeing less than 15 GB/sec. To support the pre-tape tier of the storage hierarchy for the new Trinity supercomputer, this is less-than-hoped-for performance. The multi-component architecture was developed to boost bandwidth, while also increasing reliability.

We are building a new testbed with 12 DSUs. There, we will debug and benchmark the MC DAL back end in a 10+2 configuration to prepare for a transition to production, where the 48 DSUs will be treated as four pods of 10+2.

Parallel Data-Movement with **pftool**

pftool is an open-source tool for moving data in parallel from one mounted file system to another and is the de facto production workhorse for performing data-movement at scale between storage systems at LANL. Moving data is coordinated by a scheduler which distributes subtasks to worker processes scattered across a cluster. As workers become idle they are given new subtasks, including performing one portion of the parallel traversal of the source-directory tree (returning sets of source-files for copy/compare as new subtasks) or executing one such copy/compare subtask. For large files, a copy/compare subtask can refer to a set of offset+size "chunks" of the large file to be copied, allowing large individual files to be copied in parallel, as well. pftool coordinates with file systems to choose this chunk-size. For MarFS, this means large files are broken into chunks that match up with back-end objects in a multi-file, and a special exemption from our sequential-writes-only rule is granted.

The subtasks are executed independently of each other and are asynchronous with respect to the scheduler. If the overall operation fails or is cancelled, it can be restarted and will efficiently resume with any portions of the work that were not previously performed. The duties of the scheduler are light (dispatching subtasks from a work-queue), so the scheduler doesn't become a bottleneck even at very large scales. The result is a self-balancing parallel data-movement application.

Future Work

We are exploring several new development paths, including the MD scalability of Figure 1, pftool extensions to allow cross-site transport, custom-RDMA protocols to improve storage bandwidth, and power management schemes for cold storage.

Acknowledgments

We thank our hard-working teammates. Thanks to Rik Farrow for helpful comments. This work was done at the Los Alamos National Laboratory, supported by the US Department of Energy contract DE-FC02-06ER25750. This publication has been assigned the LANL identifier LA-UR-16-28952. MarFS can be downloaded at https://github.com/mar-file-system/marfs. A commercial product using MarFS has been announced, with Lustre for metadata and SpectraLogic Black Pearl for data storage.

Curing the Vulnerable Parser

Design Patterns for Secure Input Handling

SERGEY BRATUS, LARS HERMERSCHMIDT, SVEN M. HALLBERG, MICHAEL E. LOCASTO, FALCON D. MOMOT, MEREDITH L. PATTERSON, AND ANNA SHUBINA

Sergey Bratus is a Research Associate Professor of Computer Science at Dartmouth College. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. He has a PhD in mathematics from Northeastern University and worked at BBN Technologies on natural-language processing research before coming to Dartmouth. sergey@cs.dartmouth.edu

Lars Hermerschmidt is currently working as Information Security Officer at AXA Konzern AG, where he is leading software security activities. He is a PhD candidate in software engineering at RWTH Aachen University, where he started to work on correct unparsers to prevent injections and on automated security architecture analysis. hermerschmidt@se-rwth.de

Sven M. Hallberg is a programmer by passion, a mathematician by training, and calls himself an applied scientist of insecurity by profession. He contributed large parts to the Hammer parser library and wrote the DNP3 parser based on it. He is currently pursuing a doctoral degree at Hamburg University of Technology, Germany, where he tries to further apply LangSec principles to cybernetic systems. pesco@khjk.org

Programs are full of parsers. Any program statement that touches input may, in fact, do parsing. When inputs are hostile, ad hoc input handling code is notoriously vulnerable. This article is about why this is the case, how to make it less so, and how to make the hardened parser protect the rest of the program.

We set out to make a hardened parser for an industrial control protocol known for its complexity and vulnerability of previous implementations: DNP3 [1]. We started with identifying known design weaknesses and protocol gotchas that resulted in famous parser bugs; we soon saw common anti-patterns behind them. The lesson from our implementation was twofold: first, we had to nail down the protocol syntax with precision beyond that of the standard, and, second, we formulated and followed a design pattern to avoid the gotchas.

We've used this approach with other protocols. Our parser construction kit *Hammer* (https://github.com/UpstandingHackers/hammer) allows a programmer to express the input's syntactic specification natively in the same programming language as the rest of the application. Hammer offers bindings for C, C++, Python, Ruby, Java, .NET, and others, and is suitable for a wide variety of binary protocols.

Sadly, there is no silver bullet one could implement in a library and simply reuse in every program to fix unsafe input-handling once and for all. However, we found several design patterns for handling input correctly, and thus making programs resilient against input-based attacks. In the following sections we describe three of them: the *Recognizer*, the *Most Restrictive Input Definition,* and the *Unparser.*

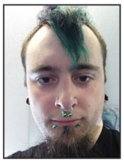
These patterns came from studying famous input-handling code flaws and what made them that way. Importantly, we found that the problems started with the choice of the input syntax and format that forced additional complexity on the code. The code flaws were made more likely by the choices of input structure; in a word, data format doomed the code.

"Don't trust your input" doesn't help to write good parsers. First, we need to deal with the standing advice of "Don't trust your input." This advice doesn't give the programmers any actionable solution: what to trust, and how to build trust? Without giving developers a recipe for establishing whether the input is trustworthy, we cannot expect correct software. This is a design issue, which no amount of penetration testing and patching can fix.

The problem of trust in the data is old. This is what types in programming languages arose to mitigate: the problem of *authenticating* the data, as James H. Morris Jr. called it in 1973 [2], before operating on it. We now call it *validating* the data, although our opponent is not Murphy—randomly corrupted data that leads to crashes—but Machiavelli: purposefully crafted data that leads to state corruption and compromise, aka unexpected computation.

Trustworthy input is input with predictable effects. The goal of input-checking is *being able to predict the input's effects on the rest of your program*. Already, as we speak of checking the input, we assume that there is a checker separate and distinct from the rest of the code; we will make this distinction precise in the design patterns discussed below. The standing advice to validate input implicitly assumes that, if the input is valid, then its effects are *predictable* and do not include unexpected computation; it is safe to pass on to the rest of the program.

Michael E. Locasto is a Senior Computer Scientist at SRI International, where he works in the Infrastructure Security Group and leads several projects dealing with IoT security and secure energy systems research. He was previously an Associate Professor at the University of Calgary and an I3P Fellow at George Mason University. He is interested in why computer programs break and how we can get better at fixing them. michael.locasto@sri.com



Falcon Darkstar Momot is a Senior Security Consultant with Leviathan Security Group. He leads security reviews and penetration tests of software and networks at various large software companies, with an eye to process improvements. He received a BSc in computer science from the University of Lethbridge and is an MS student at Athabasca University. In his spare time he teaches people how to use amateur radios and works on a team to maintain an operational Bell System No. 1 Crossbar. falcon@iridiumlinux.org

Meredith L. Patterson is the founder of Upstanding Hackers. She developed the first language-theoretic defense against SQL injection in 2005 as a PhD student at the University of Iowa and has continued expanding the technique ever since. She lives in Brussels, Belgium. mlp@upstandinghackers.com

Anna Shubina is a Research Associate at the Dartmouth Institute for Security, Technology, and Society. She was the operator of Dartmouth's Tor node when the Tor network had about 30 nodes total. ashubina@cs.dartmouth.edu

Safety—that is, predictability of execution—comes from the combination of both the input format and the code checking it being simple and well-structured.

How do we know that reading a file that contains a hundred records is safe? How can we be sure that the execution is predictable? We'll have to start with the idea that a single record can be predictably read, and that the actions required for the reading are repeatable. One way to do so is to make sure the validity of each record can be judged apart from the contents of others, and that any objects constructed from it depend only on that record. This means that the records encode independent objects that follow each other (rather than nesting in each other). A pattern is allowed to repeat without limit, or up to a certain number of times, but its structure must be rigid; a pattern cannot contain itself recursively, directly or indirectly. Then, if it's safe to call the code that parses a record once, it's safe to call it repeatedly.

Some nesting of objects in a record is allowed but only in a pre-defined pattern: if we draw the objects containing each other as a tree, the shape of that tree is rigid except for possible repetition of a node where that kind of node is allowed. Supposing that each object is parsed by a separate function, the shape of the call graph is similar to the shape of the tree. This roughly corresponds to so-called regular syntax (as in regular expressions).

In short, when parsing such regular formats, the answer to "What should I do next?" or "Is the next part of the input valid?" doesn't depend on reexamining any previous parts. Thus the code that works predictably once is sure to work again.

However, not all formats can be so restricted. In HTML or XML, for example, elements can be nested in elements like themselves to an arbitrary depth. The same is true for file systems that have directories and for formats that emulate such file systems such as Microsoft's OLE2. Other formats, like PDF, have other kinds of container objects that can nest to any depth.

For such formats, whether it is safe to invoke the code that parses an object again and again may not be predictable, because it could be called under a potentially infinite set of circumstances. Should the result depend on the path to the top of the tree of objects or, worse, on the sibling nodes in that tree, such dependencies may now pile up infinitely. Unlike the regular case above, the shape of the tree is no longer rigid; much variation in its form can occur. Now the code needs to foresee a potentially unlimited number of possible paths and histories after which it gets called; the more its behavior is supposed to depend on reexamining other objects, the harder it is to get it right (and the harder it is for a programmer to have a succinct mental model of its behavior that has any predictive power whatsoever).

Thus the simpler the better; and only with the simplest formats can some assurance be obtained. The simplest syntax patterns are *regular* and *context-free*. Context-sensitive patterns are much harder to parse, and the code is much harder to reason about. In fact, such reasoning poses undecidable or intractable problems for formats that seem fairly intuitive and straightforward. We refer the reader to [3] and http://langsec.org/ for the theory; here, we'll look at the common scenarios of how things go wrong instead.

How Input Handling Goes Wrong

From a certain perspective, input data is "just" a sequence of symbols or bytes. But this sequence drives the program logic involved in construction and manipulation of some objects. These objects drive the rest of the program and must do so predictably.

The program should make no assumptions about these objects beyond those that the parser constructing them validates. If it does, the likely effect of its code working on data it does not expect will be exploitation. This relationship between the parser's results and assumptions made by the rest of the program is crucial, but the absolute majority of programming languages do not provide any means of expressing it. Yet it has multiple ways of going wrong. Either the data's design is so complex that it invites bugs, or the programmer misunderstands the kind of validation that the data needs. Let us look at some of these examples.

Input too complex for its effects to be predictable. Safety is predictability. When it's impossible to predict what the effects of the input will be (however valid), there is no safety.

Consider the case of Ethereum, a smart contract-based system that sought to improve on Bitcoin. Ethereum operators like the decentralized autonomous organization (DAO) accepted contracts—that is, programs—to run in a virtual environment on their system; the code was the contract. The program that emptied the DAO's bank was a valid Ethereum program; it passed input validation. Yet it clearly performed unintended computation (creative theft of funds) and should not have been allowed to run.

Could the DAO have made this determination beforehand, algorithmically? Certainly not; Rice's theorem says that no general algorithm for deciding non-trivial properties of general-purpose programs may exist, and predicting the effects of a program on a bank such as DAO's is beyond even "non-trivial"—even the definition of malice in this context may not be amenable to complete computational expression. We will not dig into this theory here but will instead appeal to intuition: how easy would it be to automatically judge what obfuscated program code does before executing it? A Faustian Ethereum smart contract is hardly any easier. From the viewpoint of language-theoretic security, a catastrophic exploit in Ethereum was only a matter of time: one can only find out what such programs do by running them. By then it is too late.

Arbitrary depth of nesting vs. regexp-based checking. The arrangement (ordering and relative location) of objects in input requires a matching code structure to validate. Famously, regular expressions do not work for syntactic constructs that allow arbitrary nesting, such as elements of an HTML or XML documents or JSON dictionaries. These constructs may contain each other in any order and to any depth; their basic well-formedness and conformance to additional format constraints must be validated at any depth.

Regular expressions (regexps), which many Web applications erroneously use to check such structures, cannot do it. Regexps were originally invented to represent finite state machines, and those are incompatible with arbitrary-depth nesting. Thus regexps are best suited to checking sequences of objects that contain and follow each other in a particular order, repeat one or more times (or zero or more times), but do not infinitely nest; in other words, a finite state machine has no way of representing trees that can go arbitrarily deep. One can write a pattern that nests to some given depth *N*, but what about an input byte sequence where objects nest to depth $N + 1$? The attacker can craft just such an input and bypass the check.

Although regexp extensions found in modern scripting languages such as Perl, Python, and Ruby extend the power of their regexps beyond finite state machines, it is quite hard to write such patterns and get them right. Put differently, finite state machines cannot handle recursion well; a stack is needed there, and stack machines make a different, more powerful class of automata. Try writing a regexp without back references to match a string where several kinds of parentheses must nest in a balanced way. It cannot be done; the same problem arises with matching nesting XML elements of several kinds.

Perhaps the best known example of this mistake was the buggy anti-XSS system of Internet Explorer 8. Using regexps to "fix" supposed XSS led to non-vulnerable HTML pages being rewritten into vulnerable ones, the fix adding the actual vulnerabilities [5]. Web app examples of vulnerable checks of (X)HTML snippets are many and varied.

Context sensitivity. The lesson of the previous pitfall—still not learned by many Web apps—is that judging input must be done with appropriate algorithmic means, or else the program won't be able to tell if the data is even well-formed. But this is not the only trouble there can be.

Besides being well-formed, objects should only appear where it is legal for them to appear in the message. Judging this legality can be troublesome when the rules that determine validity depend not just on the containing object or message (i.e., the "parent" of the object we are judging), but on other objects as well, such as "sibling" objects in that parent or even some others across protocol layers.

For example, imagine that an object contains a relative time offset, in a shorter integer field, which is relative to another object that has the longer absolute value. For the relative value to appear legally, there has to be an absolute value somewhere preceding it, and the checker must keep track of this. This situation actually occurs in DNP3.

A closer-to-home example is nested objects that each include a length field. Since these lengths specify where each (sub)object ends (and another begins), all these lengths must agree with each other, and with the overall length of the message; that may be a *quadratic* number of checks on these fields alone!

The infamous Heartbleed bug arose from just such a construct: the agreement between the length fields of the containing `SSL3_RECORD` and the `HeartbeatMessage` contained in it was not checked, and the inner length was used to grab the bytes to echo back. Set that inner length to 65535, and that's how many bytes

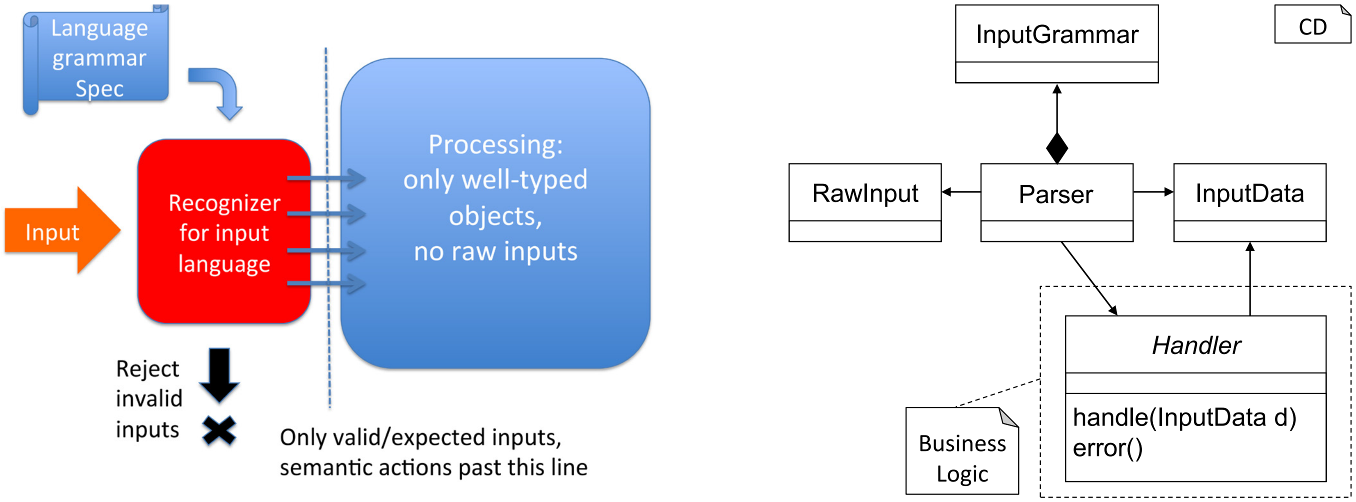

Language grammar Spec
Input
Recognizer for input language
Processing: only well-typed objects, no raw inputs
Reject invalid inputs
Only valid/expected inputs, semantic actions past this line
InputGrammar
CD
RawInput
Parser
InputData
Handler
handle(InputData d)
error()
Business Logic

Figure 1: The Recognizer Pattern for validating raw input and providing it to the business logic: (a) input data flow through the Recognizer Pattern; (b) the Recognizer Pattern as a UML class diagram

OpenSSL included, even though the overall message length was set at a modest four bytes. The GNU TLS Hello message CVE-2014-3466 similarly took advantage of three nested lengths that were expected to agree but didn't get checked.

An older but equally famous example was the 2002 pre-authentication bug in OpenSSH 3.3 exploited by GOBBLES; there, the lengths of all SSH options would need to sum up to the length of the packet, and instead overflowed an integer allocation size variable before the crafted packet could be discarded.

Such formats where validity rules require checking properties across object boundaries are called *context-sensitive*. They require more complex checkers—and are more error-prone. Indeed, it is violating these relationships that's first tried by exploiters: a forgotten check is more likely there and thus an action on unchecked data that's not what the code expects.

It's best to be able to judge an object's legality based either just on its own content or on what type its parent object is; that is, *context-free* syntax is preferable to the more complex *context-sensitive*.

Transformation before validation. Another lesson from vulnerable ways of handling such a seemingly straightforward format as XML is that input messages should be checked as they arrive, *without additional transformations*, least of all those driven by the elements of these messages themselves. This has been a source of famous vulnerabilities with XML entities.

An XML document may include *entities*, syntactic elements that will be resolved and replaced, by string substitution, throughout the body of the document. Substitutions may occur in many rounds if entities include other entities, which, in turn, will be parsed and substituted, all before the document can be finally validated. The simplest consequence of this is that a short document can expand to gigabytes in size by using several levels of entities and repetition, the so-called "billion laughs" attack.

XML entities may also include references to external documents that need to be fetched and inserted before the input data object can be constructed and validated. Fetching an XML external entity (XXE) may already be an undesirable action in and of itself, and can trigger execution of other code; at the very least it creates network connections and can exfiltrate files, or even lead to remote code execution.

It becomes instantly clear that XXEs are trouble when you consider that an *action is taken based on input before that input has been fully validated*. XXEs bring actions into the recognition process, thus breaking the separation between recognition and processing. By comparison, JSON has no such feature, and JSON objects are judged as they are received. This may account for an order of magnitude difference in CVEs related to XML (850 at the time of this writing, of which 216 are XXE-related) vs. JSON (96).

The shotgun parser. We say we have a *shotgun parser* where validation is spread across an implementation, and program logic grabs data from the input-handling code before the full data's correctness is assured. This makes it very hard to follow the dependencies and assumptions made by the code, which, in turn, leads to vulnerabilities and unexpected behavior. The antidote for this is separation of concerns: validation first, then a clear boundary at which the data has been validated to a clear specification—and not used before.

But what comes out at that boundary? It is data as objects: constructed and fully conforming to the definitions of the data structures to be extracted from input. Reaching in to use them before they are ready is an anti-pattern that resulted in Heartbleed (a remote memory leak) and many remote code executions like the 2002 OpenSSH bug or the GNU TLS Hello bug.

A million-dollar misnomer. Another key misconception about input data is that it is generally benign but can contain unsafe elements that should (and can) be "sanitized" or "neutralized." The choice of words suggested that having these elements removed or altered makes the data safe overall.

As a typical result of this (mis)understanding, the input is transformed by filtering it through regexp-based substitutions, where the regexps match the "bad" syntactic elements and replace them with some "safe" ones or suppress them.

The problem with this intuition is immediately clear: validity as predictability of execution is the property of the entire input, not of a few characters!

Deserialization is parsing, too! It should be clear by now that deserialization is not a trivial concern to be handled by some auxiliary code; it is a security boundary. This boundary exists between every pair of components that communicate outside a strong typing system or that use different structures to represent data.

It is the deserialization code's responsibility to create the conditions that the rest of the program can trust; otherwise any assurance of good program behavior is lost. That's why the properties of the serialized payload should be as simple as possible to check and, once checked, reliable enough to ensure predictable behavior.

Simply put, what a deserializer cannot check, the rest of the code should not assume. If serialized objects aren't self-contained and validatable on their own, the game is already lost; so many Java deserialization bugs, Python unpickling bugs, Remote Procedure Call bugs, and so on have turned into exploits.

The Recognizer Design Pattern for Input Validation

Input validation needs design patterns. Ensuring that input data is safe to process is a distinct, specialized role for code. As a matter of program architecture, any specialized code should be isolated in a dedicated component. Design patterns are a natural way to express the relationships of this component with others.

The main input-handling pattern we discuss is the Recognizer Pattern. As a whole, a recognizer has the sole task of accepting or rejecting input: it enforces the rule of full recognition before processing. This pattern concentrates the logic responsible for strictly matching the input's syntax with the specification and discarding any inputs that don't match.

The Recognizer Pattern in Figure 1 describes the relationships between five main elements: the InputGrammar, the Parser, the RawInput, the Handler, and the data type representing the input data within the program (called InputData in Figure 1 (b)). The locus of the Recognizer Pattern is the Parser. The Parser uses the InputGrammar as a definition of the valid input syntax. For input sequences read from the RawInput that comply with that syntax, the Parser produces a correctly instantiated InputData object representing the input in the programming language's type system. Importantly, the Parser only invokes the `handle()` method of the Handler interface after creating InputData objects. The Handler interface must be implemented by the "business logic" of the application. This arrangement cleanly separates the parsing logic from subsequent processing within the business logic, as the Handler can only access InputData validated by the Parser. This provides a crucial guarantee to the remainder of the business logic that the data has been validated and that such validation is structurally sound (i.e., it cleanly handles InputData objects nested within each other).

Most Restrictive Input Definition

In order to fully take advantage of this pattern, the input syntax specification expressed as the Grammar component should have a minimum of complexity needed to represent input objects. This point is very important because it openly acknowledges the price of adopting the Recognizer Pattern. Part of the value of adopting this approach is that you have a clear idea of what data you accept, but you give up attempting to accept arbitrarily complex data. Practically speaking, this means purposeful, thoughtful subsetting of many protocols, formats, encodings, and command languages, including eliminating unneeded variability and introducing determinism and static values. The design principle for creating predictable programs is to *choose the most restrictive input definition for the purpose of the program;* we acknowledge that it may be challenging to completely articulate the purpose of the program well enough, and that errors may still exist deeper in the program logic.

Parser Combinators: Don't Fear the Grammar!

At the heart of the Recognizer Pattern is keeping the admitted inputs to a strict definition of valid syntax. Being definite about the input gives the pattern its power; but how to do so without undue burden?

Historically, computer scientists wrote such definitions in special languages such as Augmented Backus-Naur Form (ABNF). Unfortunately, that's one more language—and another set of tools—for developers to learn; too much investment for handling what might seem a simple binary format! Moreover, after having written the input data definitions as a grammar (say, for yacc or Bison), one would need to write them *again*, in code, to construct the actual objects.

To add to developer confusion, yacc and Bison focus primarily on compiler construction, not binary parsing. The code they generate is quite unreadable: it's a large state machine with none of its internals named in a way to make sense to humans. Interfacing processing code with it is hard and has led to many mistakes.

Finally, another concern about grammars is that they have subtle gotchas to confuse their developers, such as left recursion's incompatibility with classic *LL(k)*-parsing algorithms.

Fortunately, the *parser combinator* style of writing input handling code provides a graceful way around these obstacles. The parser combinator style of programming defines the grammar of the input language and implements the recognizer for it at the same time. Thus it repackages strict grammar constraints on input in a form much more accessible to developers than do bare grammars, while retaining all of the rigor and power.

We took the parser combinator approach, and implemented the Hammer parser construction kit to specifically target parsing of binary payloads (e.g., describing bit flags and fields that cross byte boundaries is simple in Hammer, unlike in character-oriented parsing tools). Hammer targets C/C++, where the need for secure parsing is the strongest, yet modern tools for it (such as ANTLR) are not available.

Hammer supports hand-writing code that looks like the grammar and captures the definition of the recognized language in an eminently readable form. However, it does not preclude code generation. For example, Nail [4], a direct offshoot of Hammer, comes with a code-generation step.

But didn't ASN.1 solve this problem? The formidable ASN.1 standard was expected to solve the problem of unambiguously representing protocol syntax. Separating the syntax from encoding and specifying the encoding rules separately was supposed to open the way for automatically validating data against specification. The security gain from this would be obvious.

In reality, ASN.1 encoding rules and code generation tools created enough complexity and confusion to result in a series of high-profile bugs. The more permissive BER seems to be doing worse than DER: 45 vs. 26 related entries out of a total 95 ASN.1-related CVEs (based on a simple keyword search). Overall, the security record of ASN.1 does not suggest an equivalent security win for code generation.

Specifying a format with combinators. Here is an excerpt showing what our parser combinator code looks like. Remember, under this style everything gets its own parser, even a bit flag. This may seem excessive, but it truly defines the format from the ground up, and makes it clear, at every point, what structure is expected from inputs, and which properties have been checked and are being checked. Since Hammer targets binary protocols, it provides primitives for a field containing a given number of bits, h_bits, and a way to limit such a bit field to a range of possible integer values, h_int_range.

These individual parsers are connected up to parsers for each sub-unit of the message with *combinators*, such as sequencing (h_sequence, arguments are a NULL-terminated sequence of constructs that must follow each other), repetition (h_many, h_many1, h_repeat_n for the same respective meanings as *, + and {n} in regexps), or alternatives (h_choice).

Let's build up the parser for a DNP3 application header, which starts with a four-bit sequence number followed by four single-bit flags, then a one-byte function code (FC), and is optionally followed by a 16-bit field called "internal indications" (IIN), of which two bits are reserved. Whether a payload is a response or a request is determined by the flag combination. Not all combinations of flags are legal, and IIN is only legal in payloads that represent protocol responses, not requests. All these dependencies must be checked before the payload can be acted upon—or else memory corruption awaits.

We start with building up the bits for flags and their allowed combinations:

```
bit =  h_bits (1, false );
one = h_int_range(bit, 1, 1); // bit constant 1
zro = h_int_range(bit, 0, 0); // bit constant 0

conflags = h_sequence(bit, zro, one, one, NULL); // confirm
reqflags = h_sequence(zro, zro, one, one, NULL); // fin, fir
unsflags = h_sequence(one, one, ign, ign, NULL); // unsolicited
rspflags = h_sequence(zro, bit, bit, bit, NULL); // response
```

Then comes the start of the header, with its several valid alternatives. The rest are illegal and will be discarded.

```
seqno = h_bits(4, false /* unsigned */ );
conac = h_sequence(seqno, conflags, NULL );
reqac = h_sequence(seqno, reqflags, NULL );
unsac = h_sequence(seqno, unsflags, NULL );
rspac = h_sequence(seqno, rspflags, NULL );
iin  =  h_sequence(h_repeat_n(bit, 14), reserved (2) , NULL );
...

req_header =
  h_choice(h_sequence(conac, confc, NULL),
      h_sequence(reqac, reqfc, NULL), NULL);

rsp_header =
  h_choice(h_sequence(unsac, unsfc, iin, NULL) ,
      h_sequence(rspac, rspfc, iin, NULL), NULL);
```

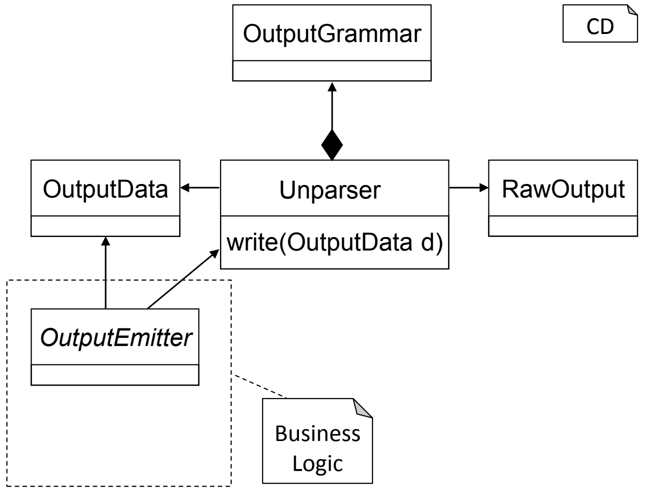


Figure 2: The Unparser Pattern for creating valid output illustrated as a UML class diagram.

Not shown here are the parsers for the one-byte function code field (`confc`, `reqfc`, `unsfc`, and `rspfc`), which enforce the appropriate value ranges. For example,

```
fc = h_uint8();
reqfc = h_int_range(fc, 0x01, 0x21);
```

and so on.

This example shows how the parser combinator-style code defines the expectations regarding the input precisely and implements a recognizer for them at the same time. But there's more—this recognizer doubles as the constructor of the parsed objects! For more detail, see [1].

Handling Output: The Unparser Pattern

So far we've only considered the case of an adversary that can directly provide input to a program. However, in interconnected systems, e.g., a Web server and a database, there are back-end systems like the database that only process input provided by the front-end Web server. Nevertheless, unexpected input to the front end may manipulate its output so that the back end interprets this in a way not intended by the developer. Therefore we need to discuss how to make back-end systems safe from indirect input attacks, where hostile inputs are passed by another program. Examples include SQL injection (SQLi) and cross-site scripting (XSS) and are most common in, but not limited to, text-based languages like SQL and HTML.

This injection into the output of the front end cannot, generally speaking, be prevented by the Recognizer at the front end. The reason is simple: the Recognizer enforced the specification of the *input* language; the language expected to be *output* by a program is different, and the Recognizer has no information about it. Hence it cannot reject those inputs that cause problems in output.

Commonly, textual output is created by concatenating fixed strings like SQL query parts with program input. Since textual languages like SQL use special tokens such as quotation marks to separate data from code, those tokens must be encoded when used within the program's output. Otherwise, input might change the meaning of the created output by using these tokens. Using templates where variables are replaced by input data, e.g., to create HTML, suffers from the same core problem: naïve creation of output with string concatenation that is not aware of the string being a language parsed by another program.

A defensive design pattern must encapsulate this awareness. For creating output and ensuring it is well formed, we developed the Unparser Pattern shown in Figure 2. Its operation is essentially reverse to that of the Recognizer: it uses a language specification (an OutputGrammar) to serialize existing valid objects to that specification.

Just as the Parser is the only class meant to read from RawInput, only the Unparser writes output to the RawOutput. Therefore, creating output from the perspective of the business logic works by instantiating OutputData objects and filling them with data without caring whether this data might contain special tokens of the output language. The Unparser takes these objects and creates a serialized output. It uses the definition of the OutputGrammar to ensure tokens possibly contained in the OutputData are encoded properly.

SQL's prepared statements interface is a special case of this pattern that had not been generalized to other output languages; we correct that. Our OutputData class provides an interface similar in function but more general and strongly typed. More about unparsers can be found in [6]; *McHammerCoder* (https://github.com/McHammerCoder) is our binary unparser kit for Java.

Finally, connecting the Recognizer, Most Restrictive Input Definition, and Unparser patterns using a business logic that translates InputData to OutputData results in a *Transducer*. The special case when InputGrammar and OutputGrammar are the same can be employed as a transparent filter at the trust boundary of a system. It acts like a syntactic firewall, improving the system's predictability by enforcing a strict input specification. We implemented this approach in our DNP3 exhaustive syntactic validation proxy and recommend it for other protocols.

Conclusion

After decades of repeated embarrassing failure, the larger programmer community accepted that "rolling your own crypto" was simply the wrong approach; effective cryptography required using professional tools.

This realization came none too soon, but a bigger realization awaits: *rolling your own parser is just as bad or worse*. Faulty input-handling is a bigger threat to security than faulty crypto, simply because, as a target, it comes *before* crypto and leads to full compromise. Solid design and professional tools are needed, just as with crypto; otherwise, the insecurity epidemic will continue.

References

[1] S. Bratus, A. J. Crain, S. M. Hallberg, D. P. Hirsch, M. L. Patterson, M. Koo, and S. W. Smith, "Implementing a Vertically Hardened DNP3 Control Stack for Power Applications," Annual Computer Security Applications Conference (ACSAC), Industrial Control System Security Workshop (ICSS), December 2016, Los Angeles, CA.

[2] J. H. Morris, Jr., "Types Are Not Sets," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*, October 1973, pp. 120–124.

[3] L. Sassaman, M. L. Patterson, S. Bratus, M. E. Locasto, and A. Shubina, "Security Applications of Formal Language Theory," *IEEE Systems Journal*, vol. 7, no. 3, September 2013; Dartmouth Computer Science Technical Report TR2011-709.

[4] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-bangert.pdf.

[5] E. V. Nava and D. Lindsay, "Abusing IE8's XSS Filters," 2010: http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf.

[6] L. Hermerschmidt, S. Kugelmann, and B. Rumpe, "Towards More Security in Data Exchange: Defining Unparsers with Context-Aware Encoders for Context-Free Grammars," in *Proceedings of 2015 IEEE Security and Privacy Workshop*, pp. 134–141: http://spw15.langsec.org/.

Postmortem Action Items

Plan the Work and Work the Plan

JOHN LUNNEY, SUE LUEDER, AND BETSY BEYER

John Lunney is a Senior Site Reliability Engineer at Google Zürich. His team manages G Suite, productivity apps for Enterprise customers. He holds a degree in computational linguistics from Trinity College in Dublin, Ireland. Before Google, he worked on several lexicography projects for the Irish language. lunney@google.com

Sue Lueder is a Site Reliability Program Manager in Google's Mountain View office. She's part of the team responsible for disaster testing and readiness, incident management processes and tools, and incident analysis. Before Google, Sue worked as a Systems Engineer in the wireless and smart energy industries. She has an MS in organization development from Pepperdine University and a BS in physics from UCSD. slueder@google.com

Betsy Beyer is a Technical Writer for Google Site Reliability Engineering in NYC. She has previously provided documentation for Google Data Center and Hardware Operations teams. Before moving to New York, Betsy was a lecturer in technical writing at Stanford University. She holds degrees from Stanford and Tulane. bbeyer@google.com

In the 2016 O'Reilly book *Site Reliability Engineering*, Google described our culture of blameless postmortems and recommended that operationally focused teams and organizations institute a similar culture of postmortems in their approach to production incidents. A postmortem is a written record of an incident that details its impact, the actions taken to mitigate or resolve it, the root cause(s), and the follow-up actions taken to prevent the incident from recurring. The chapter "Postmortem Culture: Learning from Failure" describes criteria for deciding when to conduct postmortems, some best practices around postmortems, and advice on how to cultivate a postmortem culture based upon the experience we've gained over the years.

We write postmortems to ensure we achieve a few primary goals:

- We understand all contributing root causes.
- The incident is documented for future reference and pattern discovery.
- We enact effective preventive actions to reduce the likelihood and/or impact (i.e., duration and/or scope) of recurrence.

We refer to the preventive actions identified during root cause analysis as *postmortem action items*, which in aggregate form the *postmortem action item plan*.

This article addresses the challenges in designing an appropriate action item plan and then executing that plan. We discuss best practices for developing high-quality action items (AIs) for a postmortem, plus methods of ensuring these AIs actually get implemented. If the AIs are not closed out, you are implicitly agreeing that it is acceptable to suffer the exact same outage again. Furthermore, if you are successful as a service, the outage will be larger the next time around.

It's worth noting that Google teams are by no means perfect at formulating and executing postmortem action items. We still have a lot to learn in this challenging area and are sharing our approach to give a starting point for discussion throughout the industry.

Action Item Best Practices

Successful AIs require careful thought at both ends of their life cycle: formulation and follow-through. The following sections detail best practices we've cultivated as we continually refine our methods.

Enacting AIs

Classifying Action Items for Full Coverage

We classify action items by category (Investigate, Mitigate, Repair, Detect, Prevent) to make sure that the action item plan covers both very short-term and longer-term fixes. Making sure to consider AIs for each category can inspire simple but effective changes, particularly around detection (as early detection is often the best way to reduce time to resolution).

When an outage has multiple contributing causes, you need a multi-dimensional action item plan that will address each root cause and all systems that contributed to the outage.

At a minimum, your postmortem must include AIs to **Mitigate** and **Prevent** future incidents, but it should also include all other relevant categories listed below. Note that many teams initiate incident investigation and mitigation (bullets one and two) before conducting the postmortem.

- **Investigate** this incident: what happened to cause this incident and why? Determining the root causes is your ultimate goal.
  Examples: logs analysis, diagramming the request path, reviewing heapdumps
- **Mitigate** this incident: what immediate actions can we take to resolve and manage this specific event?
  Examples: rolling back, cherry-picking, pushing configs, communicating with affected users
- **Repair** damage from this incident: how can we resolve immediate or collateral damage from this incident?
  Examples: restoring data, fixing machines, removing traffic re-routes
- **Detect** future incidents: how can we decrease the time to accurately detect a similar failure?
  Examples: monitoring, alerting, plausibility checks on input/output
- **Mitigate** future incidents: how can we decrease the severity and/or duration of future incidents like this? how can we reduce the percent of users affected by this class of failure the next time it happens?
  Examples: graceful degradation; dropping non-critical results; failing open; augmenting current practices with dashboards, playbooks, incident management protocols, and/or war rooms
- **Prevent** future incidents: how can we prevent a recurrence of this sort of failure?
  Examples: stability improvements in the code base, more thorough unit tests, input validation and robustness to error conditions, provisioning changes-

When filing issues or bugs for these action items, make sure to use the appropriate classification (bug vs. feature request). Although this differentiation may seem subjective, in our view, a **bug** is a deviation from required behavior, while a **feature request** is new required behavior. Typically, you should use the type your team tracks most strictly (see the later section "Prioritizing Action Items " for more details).

Wording Action Items

The right wording for an AI can make the difference between easy completion and indefinite delay due to infeasibility and/or procrastination. A well-crafted AI should manifest the following properties:

- **Actionable**: Phrase each AI as a sentence starting with a verb. The action should result in a useful outcome, not a process. For example, "Enumerate the list of critical dependencies" is a good AI, while "Investigate dependencies" is not.
- **Specific**: Define each AI's scope as narrowly as possible, making clear what is and what is not included in the work.
- **Bounded**: Word each AI to indicate how to tell when it is finished, as opposed to leaving the AI open-ended or ongoing.

Table 1 provides examples of poorly worded vs. well-crafted AIs.

Poorly Worded	Better
Investigate monitoring for this scenario.	(Actionable) Add alerting for all cases where this service returns >1% errors.
Fix the issue that caused the outage.	(Specific) Handle invalid postal code in user address form input safely.
Make sure engineer checks that database schema can be parsed before updating.	(Bounded) Add automated presubmit check for schema changes.

Table 1: Examples of action items

We recommend implementing automated fixes when possible, as opposed to prevention/mitigation that requires ongoing manual intervention.

Consider grouping AIs either by theme or by team. In addition to providing a clear organizational or responsibility-focused structure, this categorization may also help you spot an unbalanced AI plan (see "Unbalanced Action Item Plans").

After the post-incident dust settles, don't be afraid to update a poorly worded AI to make it more tractable.

Prioritizing Action Items

It's crucial to properly prioritize action items because the priority guides future attention each AI will receive. At Google, we use the following priority levels, based on estimated risk:

- **P0**: *High risk of unmitigated recurrence of the incident if this AI is not resolved.* Resolving this AI will directly address a root cause. Resolution will either completely prevent such incidents from recurring or greatly reduce their impact to a negligible level.
- **P1**: *Medium risk of unmitigated recurrence of the incident if this AI is not resolved.* Resolving this AI will directly address the root cause. Resolution will either significantly mitigate the impact of a recurrence or have a high chance of preventing a recurrence.

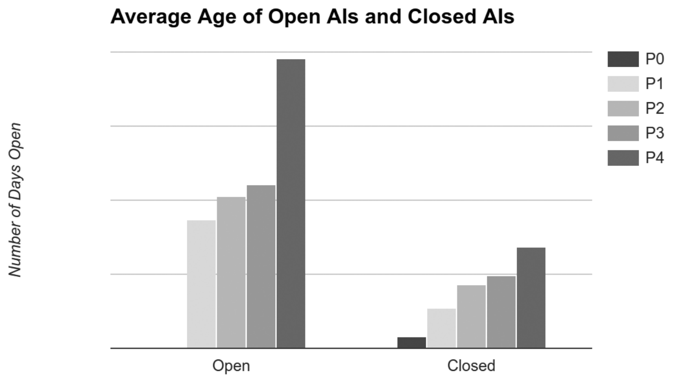


Figure 1: Time to close out action items

- **P2**: *Low risk of unmitigated recurrence of the incident if this AI is not resolved.* Resolving this action item will only superficially mitigate a recurrence of this issue or will address only peripheral contributing conditions.
- **P3**: *Trivial risk of unmitigated recurrence of a similar incident if this AI is not resolved.*

We require every postmortem stemming from a user-visible event to have at least one P0 or P1 action item. If the outage was bad enough to disrupt users, it's important enough to require high priority follow-up work to avoid or mitigate recurrence.

Figure 1 shows that on average, high priority actions are closed more quickly than low priority AIs. However, when it comes to AIs that are still open, priority doesn't significantly influence their age—on average, outstanding P1 AIs have been open almost as long as outstanding P3 AIs. We use this data to implement initiatives to bring more attention to open actions from postmortems.

Following Up on AIs

Postmortem Reviews

Many teams at Google that participate in incident response conduct postmortem review sessions. These reviews are helpful in bringing key parties together to ensure that the postmortem is complete and that the action item plan covers required categories and avoids anti-patterns. Most postmortem reviews have the following general format:

- **Walkthrough of incident timeline, impact, and root cause**: Include clarifications and address open discussion threads.
- **Review of lessons learned**: Discuss updates, additions, and mappings to action items.
- **Review of action items**: Review the checklist (see the Appendix) to make sure AIs have owners, wordings are clear, priorities make sense, and that no category (Investigate, Mitigate, Repair, Detect, Prevent) is missing.

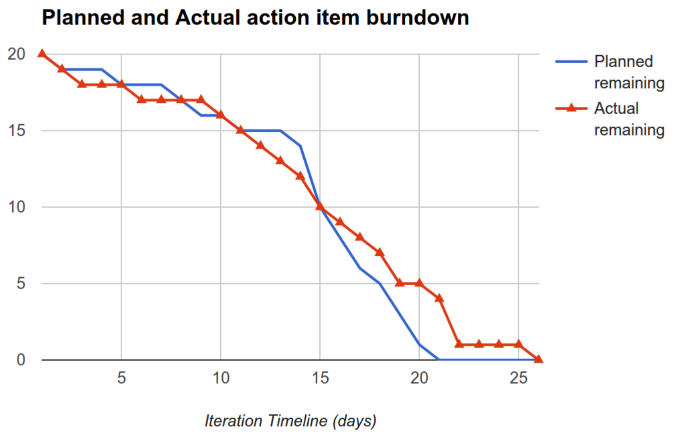


Figure 2: AI burndown for a single postmortem

These reviews should happen soon after an incident so that the parties involved remember what happened. You can hold reviews on a small scale with just the impacted team(s), or on a large scale with many parties and observers.

Action Item Closure Tracking

Encourage action item owners to close AIs that you'll never have time to address—don't keep them around forever. If an AI is obsolete or infeasible, it just distracts you from the AIs that still need work.

It's a good idea to provide periodic visibility into team progress towards reducing the technical debt identified in postmortems. Consider adding postmortem action item burndown progress (that is, AI follow-through) to your regular service or team reporting. For example, you might build postmortem AI reports in your bug/issue-tracking system and track these issues against a closure-time objective, following up with outliers.

In many cases, an action item requires considerable effort and must fit in with work that's already scheduled. Keeping an eye on how long it takes to close out action items on average helps us identify where slow action item closure leads to additional risk to reliability.

We actively monitor bug burndown over time. There are multiple ways to visualize this data. Figure 2 shows how we might track burndown for a single postmortem. In this example, the team planned out an action item completion schedule for all 20 actions to be completed over 21 days. They monitored progress until the final action item was complete on the 25th day.

Figure 3 shows how we might track AIs across any part of the organization by measuring the number of postmortem AIs created vs. closed by day. The widening distance between the two lines indicates accumulating technical debt over time, a pattern that you should seek to avoid.

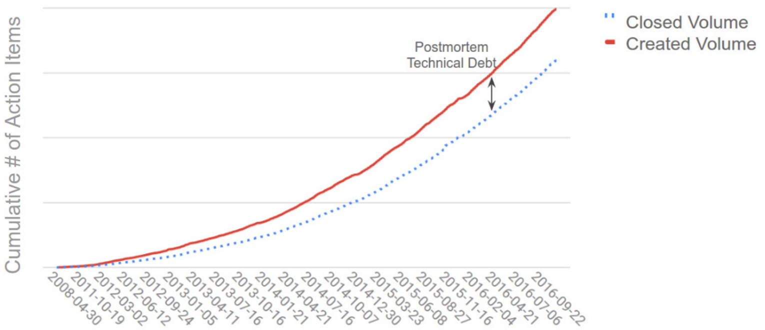


Figure 3: Postmortem AIs created vs. closed, by day

Executive Focus

You can further shine light on postmortem AI follow-up through close attention from senior leaders in your organization. We regularly review postmortems with VPs and Directors to ensure that high priority postmortems and action items receive the attention they deserve.

Action Item Anti-Patterns

In reviewing the thousands of postmortems we've conducted at Google over the years, we've identified a number of common deficiencies when it comes to both constructing and handling action items. The most common shortcoming is lack of follow-up (and many of our best practices aim to mitigate this problem).

The following section presents several other anti-patterns, which relate to how we structure or enact postmortem action items. Our experience shows that if either of these steps goes wrong, no amount of follow-up will help because vague or misleading AIs can't be completed.

Structuring AIs

Unbalanced Action Item Plan

If your postmortem action item plan contains only long-term, unrealistic, or infeasible actions, it's likely that you won't resolve any AIs before the next outage hits. On the other hand, a plan that only includes tactical items and never explores better ways to architect a more robust system is a missed opportunity to increase reliability.

Mitigation: Create a balanced action item plan that includes both:

- Near-term fixes to prevent/mitigate a similar outage
- Strategic improvements to the design of implicated systems to increase reliability

Strike a healthy balance between local/incremental/Band-Aid solutions and loftier long-term improvements. Covering all the categories in "Classifying action items" helps in this effort.

Tossing Work "Over the Wall"

An action item plan often requires work from partner teams in other parts of the organization. Don't draft and file action items against other teams without some discussion with the team that owns the component. Without this discussion, you're essentially throwing work over the wall and hoping that it gets done.

Mitigation: Include partner teams in the postmortem drafting process. Make sure each owner or team is satisfied with their assigned action items before publishing the postmortem. You might want to discuss the action item plan in person or via videoconference and ask for commitment to a resolution time frame. If conflict arises, "Executive focus" (see the best practice) may help with escalation.

Focusing on Elimination (at the Cost of Mitigation)

It can be tempting to design an action item plan that will eliminate the chance of the incident from ever happening again. Of course, you should take those actions when appropriate, but you should also spend time evaluating how to reduce the duration and impact of the incident—especially if such a fix will take effect sooner than a potential "elimination fix."

Mitigation: Take a look at how an incident unfolds and consider writing detection and mitigation action items that address the following:

- Could we have detected the incident sooner?
- Could we have triaged the impact sooner, leading to a more appropriate incident response?
- Could we have understood the root cause sooner, leading to faster rollback?
- Could the rollback have proceeded faster or more smoothly?
- Could we have scaled back the initial faulty rollout, thereby impacting a smaller percentage of users?

Thinking Only of the Current Incident (Missing Patterns)

One of our colleagues appropriated Mark Twain to observe, "We rarely repeat incidents, but we sometimes have incidents that rhyme." If we only consider a given incident in isolation, we may overfit a specific solution to the incident at hand. We also might create duplicate actions by missing information about improvements that are underway as part of another postmortem action item plan. Even worse, we might miss an opportunity to kill two risks with one stone.

Mitigation: Review postmortems for similar incidents and their accompanying action items. You might identify an opportunity to add resources to an action item that's not getting the attention it deserves, or an opportunity to collaborate on a new action item that would help in both types of incidents.

Enacting AIs

Lack of Ownership

The surest way for a postmortem author to ensure that an action item never gets completed is to leave it without an owner.

Mitigation: Always assign an owner for every action item as it is enacted, even if that owner's primary task is to find the best person for the job (e.g., the Tech Lead for the team responsible for that product or software). Your issue tracker is an appropriate place to assign ownership.

Overly Specific Monitoring Changes

With the benefit of hindsight, it's easy to say we should have monitored an XYZ-specific signal, which would have alerted us to the problem before it became a huge incident. However, this strategy only helps if that very specific failure mode recurs in the exact same way (which is frequently not the case).

Mitigation: Make the effort count: look for ways to improve monitoring for a whole class of issues. Preferably, these improvements should target user-focused symptoms rather than internal metrics. Otherwise, you risk tying alerts to implementation details.

Fixing Symptoms (Not Root Causes)

Root cause analysis (RCA) is one of the most crucial parts of a postmortem. The outcome of this analysis should drive the construction of the action item plan. Shallow RCA limits action items to impermanent fixes or surface patches to problems.

Mitigation: A thorough RCA is key to defining action items that will prevent or mitigate future incidents of this nature. Use the five-whys RCA (or another methodology [2]) to help determine which contributing causes in the chain your AIs should target.

Blaming Humans (Missing System Fixes)

It's very rarely productive to think of humans as the ending "why" in a root cause chain. During an emergency, people are typically doing the best they can under intense pressure and when faced with ambiguous data. As a result, what looks like an obvious point in the cold light of day can be quite non-obvious in the heat of the moment.

Attributing blame to a specific person or group doesn't improve your system or spur development of systematic defenses. The next time there is an emergency, the hapless on-duty person will be faced with a similarly difficult problem to solve in real time. If you trust your engineers to make the best decision given available information, it's more helpful to consider an error to be a failure of the entire system, as opposed to the fault of one or more humans.

Mitigation: Rather than finger-pointing, it's much more helpful to think about:

- How we can give people better information to make decisions?
- How we can make our environment, systems, tools, and processes more immune to human fallibility?

When you feel tempted to use human error as a root cause, use a critical eye to avoid one-off fixes. A useful stance is to believe, "The system should not have been able to fail this way." Ask yourself the following questions:

- How likely is the next person to cause the same problem? Could a new hire or sleepy SRE at 4 a.m. have made this mistake? Why did the system let them?
- Was information flawed, misleading, or poorly presented? Can we fix that misinformation (preferably through the use of automation)?
- Could software have prevented/mitigated this error? Can we automate this activity so it doesn't require human intervention?

Fixes Late in the Software Life Cycle (Missing Earlier Chances)

It can be tempting to stop a badly behaving system from impacting users by implementing a check or safeguard at the last step before changes enter production. For example, you might implement additional checks right before a config file is pushed to production but fail to consider adding configuration file coding standards, automated testing, improved training, or making sure there are fewer ways to break configuration files in the first place. The fact that bugs are much more expensive to fix late in the software life cycle is well understood in the industry [3].

Mitigation: When reviewing the postmortem timeline and lessons learned, look for ways to address the root cause (and possibly, the event trigger) as early as possible. The fix might be the same (e.g., input validation) but applied to the first system as opposed to the last one.

Conclusion

Years of conducting postmortems at Google have taught us that there's no one-size-fits-all approach to conducting this exercise successfully. However, this accumulation of experience—what we've done right, what we've done wrong, and how we've iterated to improve—has led to a certain amount of insight, which we hope can benefit other companies and organizations. We believe that it's very important to both construct high quality postmortem action items and follow up on them in a timely and comprehensive manner. Only by completing these AIs can we hope to avoid recurrence of costly and time-consuming production incidents.

The checklist appended to this article is a good starting point if you're new to conducting postmortems, or perhaps a useful honing tool for veterans of this process. As we continue to refine our approach to this imperfect science, we hope to learn equally valuable lessons from others in the field.

References

[1] B. Beyer, C. Jones, J. Petoff, and N. Murphy, eds., *Site Reliability Engineering* (O'Reilly Media, 2016).

[2] More formal methodologies exist for those looking for more rigor. For example, Ishikawa fishbone diagrams, 8Ds, fault tree analysis, and failure mode and effects analysis (FMEA). See https://en.wikipedia.org/wiki/Root_cause_analysis for more ideas.

[3] J. Leon, "The True Cost of a Software Bug: Part One," Celerity blog, Feb 28, 2015: http://blog.celerity.com/the-true-cost-of-a-software-bug.

Checklist

Structuring

□ Each lesson learned is addressed with at least one AI.

□ AI plan is balanced between near-term fixes and strategic design improvements.

□ AIs address both prevention and decreasing resolution time.

□ "Rhyming" incidents and their action plans have been reviewed.

□ No work is tossed "over the wall": all involved teams are committed to relevant AIs.

Enacting

□ AIs cover the two most critical categories (Mitigate, Prevent) + all other relevant categories (Investigate, Repair, Detect).

□ AIs are worded to be actionable, specific, and bounded.

□ AIs are prioritized, with at least one P0 or P1 to avoid or mitigate recurrence.

□ All AIs have an owner.

□ AIs aren't overly specific (for example, could you monitor something more general?).

□ Problem is caught as early as possible in the software life cycle.

□ AI plan addresses a root problem (as opposed to just patching symptoms).

□ AIs don't blame humans (focus instead on automatic system detection).

Follow-up

□ AI plan is shared with your team, stakeholders, and those involved in the incident.

□ AIs are appropriately filed and tagged/tracked to appear in your reporting system.

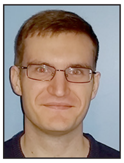
□ AI plan was reviewed with an executive or group of leads for visibility.

□ Postmortem and AI plan were reviewed/approved per team policy.

Don't Get Caught in the Cold, Warm Up Your JVM

Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems

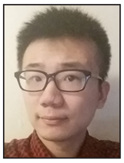
DAVID LION, ADRIAN CHIU, HAILONG SUN, XIN ZHUANG, NIKOLA GRCEVSKI, AND DING YUAN



David Lion is a graduate student in the Electrical and Computer Engineering Department of the University of Toronto. His research interest is in software systems and their performance. david.lion@mail.utoronto.ca

Adrian Chiu is an undergraduate student in Electrical Engineering at the University of Toronto. His research interests are in operating systems, distributed systems, and compilers. adrian.chiu@mail.utoronto.ca

Hailong Sun is an Associate Professor in the School of Computer Science and Engineering at Beihang University. His research interests include distributed systems, software engineering, and crowdsourcing. sunhl@ece.utoronto.ca



Xin Zhuang is a graduate student at the University of Toronto, studying computer engineering. His research interest is in software systems. xin.zhuang@mail.utoronto.ca

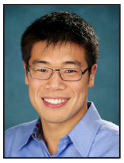
Many widely used, latency sensitive, data-parallel distributed systems, such as HDFS, Hive, and Spark choose to use the Java Virtual Machine (JVM) despite debate on the overhead of doing so. By thoroughly studying the JVM performance overhead in the above-mentioned systems, we found that the warm-up overhead, i.e., class loading and interpretation of bytecode, is frequently the bottleneck. For example, even an I/O intensive, 1 GB read on HDFS spends 33% of its execution time in JVM warm-up, and Spark queries spend an average of 21 seconds in warm-up. The findings on JVM warm-up overhead reveal a contradiction between the principle of parallelization, i.e., speeding up long-running jobs by parallelizing them into short tasks, and amortizing JVM warm-up overhead through long tasks. We therefore developed HotTub, a new JVM that reuses a pool of already warm JVMs across multiple applications. The speed-up is significant: for example, using HotTub results in up to 1.8x speed-ups for Spark queries, despite not adhering to the JVM specification in edge cases.

The performance of data-parallel distributed systems has been heavily studied in the past decade, and numerous improvements have been made to the performance of these systems. A recent trend is to further process latency sensitive, interactive queries with these systems. However, there is a lack of understanding of the JVM's performance implications in these workloads. Consequently, almost every discussion on the implications of the JVM's performance results in heated debate. For example, the developers of Hypertable, an in-memory key-value store, use C++ because they believe that the JVM is inherently slow. They also think that Java is acceptable for Hadoop because "the bulk of the work performed is I/O" [4]. In addition, many believe that as long as the system "scales," i.e., parallelizes long jobs into short ones, the overhead of the JVM is not concerning [7].

Our research asks a simple question: what is the performance overhead introduced by the JVM in latency sensitive data-parallel systems? We answer this by presenting a thorough analysis of the JVM's performance behavior when running systems including HDFS, Hive on Tez, and Spark. We had to carefully instrument the JVM and these applications to understand their performance.

Surprisingly, after multiple iterations of instrumentation, we found that JVM warm-up time, i.e., time spent in class loading and interpreting bytecode, is a recurring overhead. Specifically, we made the following three major findings. First, JVM warm-up overhead is significant even in I/O intensive workloads. For example, reading a 1 GB file on HDFS from a hard drive requires JVM to spend 33% of its time in warm-up. In addition, the warm-up time does not scale but, instead, remains nearly constant. For example, the warm-up time in Spark queries remains at 21 seconds regardless of the workload scale factor, thus affecting short-running jobs more. The broader implication is the following:

Nikola is the VP of Engineering at Vena Solutions Inc. Prior to this he worked at the IBM Compiler Group for 12 years, most notably as a Technical Lead for the x86 JIT Optimizer and Code Generator. He holds a master's degree in computer engineering from the University of St. Cyril and Methodius in Skopje, Macedonia. grcevski@gmail.com



Ding Yuan is an Assistant Professor in the Electrical and Computer Engineering Department of the University of Toronto. He works in computer systems, with a focus on their reliability and performance. yuan@ece.toronto.edu

There is a contradiction between the principle of parallelization, i.e., speeding up long-running jobs by parallelizing them into short tasks, and amortizing JVM warm-up overhead through long tasks.

Finally, the use of complex software stacks aggravates warm-up overhead. A Spark client loads 19,066 classes executing a query, which is three times more than Hive despite Spark's overall latency being shorter. These classes come from a variety of software components needed by Spark. In practice, applications using more classes also use more unique methods, which are initially interpreted. This results in increased interpretation time.

To solve the problem, our key observation is that the homogeneity of parallel data-processing jobs enables a significant reuse rate of warm data, i.e., loaded classes and compiled code, when shared across different jobs. Accordingly, we designed HotTub, a new drop-in replacement JVM that transparently eliminates warm-up overhead by reusing JVMs from prior runs. The source code of HotTub and our JVM instrumentations are available at https://github.com/dsrg-uoft/hottub.

Analysis of JVM Warm-up Overhead

What follows is an in-depth analysis of the JVM warm-up overhead in three data-parallel systems, namely HDFS, Hive running on Tez and YARN, and Spark SQL running with Spark. We will show that on each system the JVM warm-up time stays relatively constant. The HDFS experiment further shows how warm-up can dwarf I/O, while the Spark and Hive experiments explain the implications of warm-up overhead for parallel computing. All experiments are performed on an in-house cluster with 10 servers connected via 10 Gbps interconnect. Each of them has at least 128 GB DDR4 RAM and two 7,200 RPM hard drives. The server components are long running and fully warmed-up for weeks and have serviced thousands of trial runs before measurement runs. Details on our study methodology and the JVM instrumentation can be found in our OSDI paper [5].

HDFS

We implement three different HDFS clients: sequential read; parallel read, with 16 threads, that runs on a server with 16 cores; and sequential write. We flush the OS buffer cache on all nodes before each measurement to ensure the workload is I/O bound. Note that interpreter time does not include I/O time, because I/O is always performed by native libraries.

Figure 1 shows the class loading and interpreter time under different workloads. The average class loading times are 1.05, 1.55, and 2.21 seconds for sequential read, parallel read, and sequential write, respectively, while their average interpreter times are 0.74, 0.71, and 0.92 seconds. The warm-up time does not change significantly with different data sizes. The reason that HDFS write takes the JVM longer to warm up is that it exercises a more complicated control path and requires more classes. Parallel read spends less time in the interpreter than sequential read because its parallelism allows the JVM to identify the "hot spot" faster.

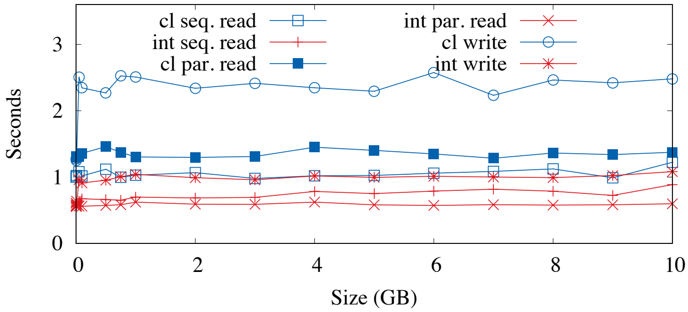


Figure 1: JVM warm-up time in various HDFS workloads. "cl" and "int" represent class loading and interpretation time, respectively. The x-axis shows the input file size.

Figure 2 further shows the significance of warm-up overhead within the entire job. Short-running jobs are affected the most. When the data size is under 1 GB, warm-up overhead accounts for more than 33%, 48%, and 30%, respectively, of the client's total execution time in sequential read, parallel read, and sequential write. According to a study [8] published by Cloudera, a vast majority of the real-world Hadoop workloads read and write less than 1 GB per-job as they parallelize a big job into

Don't Get Caught in the Cold, Warm Up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems

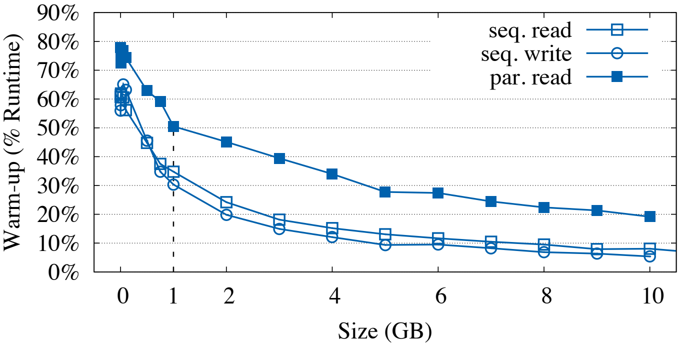


Figure 2: The JVM warm-up overhead in HDFS workloads measured as the percentage of overall job completion time

smaller ones. The study further shows that for some customers, over 60% of their jobs read less than 1 MB from HDFS, whereas a 1 MB HDFS sequential read spends over 60% of its time in warm-up.

Next we break down class loading and interpreter time using the 1 GB sequential read as an example. Figure 3 shows the warm-up time in the entire client read. A majority of the class loading and interpreter execution occurs before a client contacts a datanode to start reading.

Further drilling down, Figure 4 shows how warm-up time dwarfs the datanode's file I/O time. When the datanode first receives the read request, it sends a 13-byte ACK to the client, and immediately proceeds to send data packets of 64 KB using the `sendfile` system call. The first `sendfile` takes noticeably longer than subsequent ones since the data is read from the hard drive. However, the client takes even longer (15 ms) to process the ACK because it is bottlenecked by warm-up time. By the time the client finishes parsing the ACK, the datanode has already sent 11 data packets, and thus the I/O time is not even on the critical path. The client takes another 26 ms to read the first packet, where it again spends a majority of the time loading classes and interpreting the computation of the CRC checksum. By the time the client finishes processing the first three packets, the datanode has already sent 109 packets. In fact, the datanode is so fast that the Linux kernel buffer becomes full after the 38th packet and has to block for 14 ms so that the kernel can adaptively increase its buffer size. The client, on the other hand, is trying to catch up the entire time.

Figure 4 also shows the performance discrepancy between interpreter and compiled code. Interpreter takes 15 ms to compute the CRC checksum of the first packet, whereas compiled code only takes 65 µs per-packet.

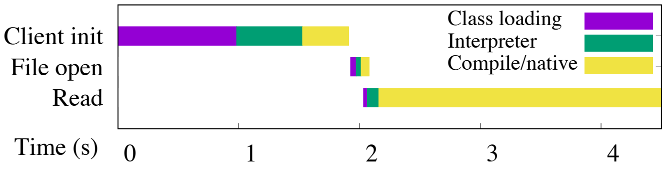


Figure 3: Breakdown of sequential HDFS read of 1 GB file

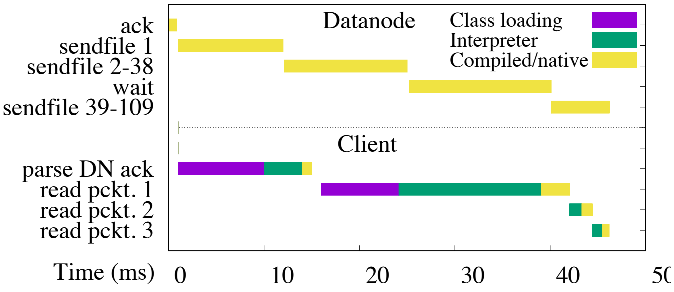


Figure 4: Breakdown of the processing of data packets by client and datanode

Break Down Class Loading

The HDFS sequential read takes a total of 1,028 ms to load 2,001 classes. Table 1 shows the breakdown of class loading time. Reading the class files from the hard drive only takes 170 ms. Because Java loads classes on demand, loading 2,001 classes is broken into many small reads: e.g., 276 ms are spent searching for classes on the classpath, which is a list of file-system locations. The JVM specification requires the JVM to load the first class that appears in the classpath in the case of multiple classes with identical names. Therefore it has to search the classpath linearly when loading a class. Another 411 ms are spent in define class, where the JVM parses a class from file into an in-memory data structure.

	Read	Search	Define	Other	Total
Time (ms)	170	276	411	171	1,028

Table 1: Breakdown of class loading time

Spark versus Hive

Figure 5 shows the JVM overhead on Spark and Hive. Surprisingly, *each query spends an average of 21.0 and 12.6 seconds in warm-up time on Spark and Hive, respectively.* Similar to HDFS, the warm-up time in both systems does not vary significantly when data size changes, indicating that its overhead becomes more significant in well parallelized short-running jobs. For example, 32% of the Spark query time on 100 GB data size is on warm-up. In practice, many analytics workloads are short running. For example, 90% of Facebook's analytics jobs have under 100 GB input size [1, 2], and a majority of the real-world Hadoop workloads read and write less than 1 GB per-task [8].

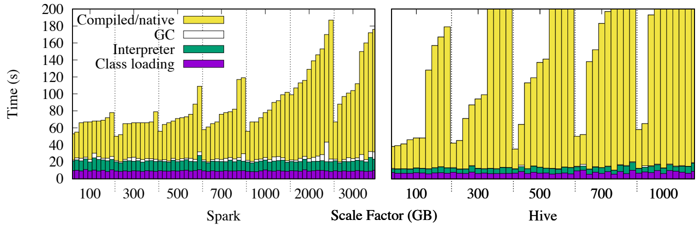


Figure 5: JVM overhead on BigBench. Overhead breakdown of queries from BigBench [3] across different scale factors. Only the 10 shortest queries from BigBench are analyzed because of our focus on latency-sensitive queries. The scale factor corresponds to the size of input data in GB. The queries are first grouped by scale factor and then ordered by runtime. Note that Hive has a larger query time compared to Spark.

Software Layers Aggravate Warm-up Overhead

The difference in the warm-up times between Spark and Hive is explained by the difference in number of loaded classes. The Spark client loads an average of 19,066 classes, compared with Hive client’s 5,855. Consequently, the Spark client takes 6.3 seconds in class loading whereas the Hive client spends 3.7 seconds. A majority of the classes loaded by Spark client come from 10 third-party libraries, including Hadoop (3,088 classes), Scala (2,328 classes), and Derby (1,110 classes). Only 3,329 of the loaded classes are from Spark packaged classes.

A large number of loaded classes also results in a large interpreter time. The more classes being loaded, the greater the number of different methods that are invoked, where each method has to be interpreted at the beginning. On average, a Spark client invokes 242,291 unique methods, where 91% of them were never compiled by JIT-compiler. In comparison, a Hive client only invokes 113,944 unique methods, while 96% of them were never JIT-compiled.

Breaking Down Spark’s Warm-up Time

We further drill down into one query (query 13 of BigBench with scale factor 100) to understand the long warm-up time of Spark. While different queries exhibit different overall behaviors and different runtimes, the pattern of JVM warm-up overhead is similar, as evidenced by the stable warm-up time. Figure 6 shows the breakdown of this query. The query completion time is 68 seconds: 24.6 seconds are spent on warm-up overhead of which 12.4 seconds are spent on the client while the other 12.2 seconds come from the executors. Note that a majority of executors’ class-loading time is not on the critical path: executors are started immediately after the query is submitted, which allows executors’ class loading time to be overlapped with the client’s warm-up time. However, at the beginning of each stage the executor still suffers from significant warm-up overhead that comes primarily from interpreter time.

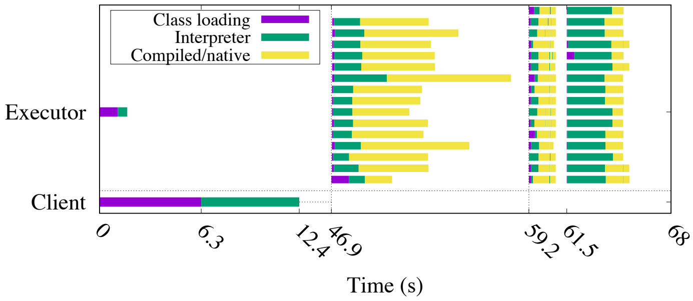


Figure 6: Breakdown of Spark’s execution of query 13. It only shows one executor (there are a total of 10 executors, one per host). Each horizontal row represents a thread. The executor uses multiple threads to process this query. Each thread is used to process three tasks from three different stages.

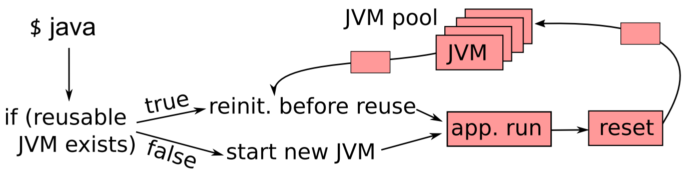


Figure 7: Architecture of HotTub

Hive

Hive parallelizes a query using different JVM processes, known as containers, whereas each container uses only one computation thread. Therefore within each container the warm-up overhead has a similar pattern to the HDFS client shown earlier. Hive and Tez also reuse containers to process tasks of the same query, and therefore the JVM warm-up overhead can be amortized across the lifetime of a query.

HotTub

The design goal for HotTub is to allow applications to share the “warm” data, i.e., loaded classes and compiled code, thus eliminating the warm-up overhead from their executions. HotTub is implemented by modifying OpenJDK’s HotSpot JVM and is made to be a drop-in replacement. Users simply replace `java` with HotTub and run their Java application with normal commands.

Figure 7 shows the architecture of HotTub. When `java` is first called there are no existing JVMs to reuse, so a new JVM must be created for the application to run on as it normally would. Once the application finishes, the JVM must first be reset before it can be added to a pool of JVMs for later reuse. When there are JVMs in the pool, a call to `java` will attempt to find a valid JVM for reuse. If a JVM is found it will be reinitialized, and then the application will run on the already warm JVM with nearly zero warm-up overhead.

The main challenge of this process is to ensure that the application's execution on HotTub is *consistent* with the execution on an unmodified JVM. Next we discuss some techniques HotTub uses to ensure consistency. More detailed discussions can be found in our OSDI paper [5].

Class Consistency

When choosing a JVM to reuse we must make sure any class that will be reused is the same as the class that would have been dynamically loaded in a normal execution. To do this HotTub ensures for a JVM the classpath and classes on the classpath are the same for both the new application and the previously run applications. This also ensures there is a large amount of potential overlap between the new application and the already loaded classes and compiled code. It is possible to be more strict and only reuse a JVM if the application is more similar to what was previously run, but being less strict would not be able to guarantee consistency.

Data Consistency

At the reset phase all stale data is cleaned up off of the critical path before the JVM is put back in the pool. All application threads are cleaned up, so there are no more stacks left, and all file descriptors opened by the application are closed. HotTub also zeroes out all static data from classes. HotTub now runs garbage collection to remove all the stale data, but since there are no root references from the stack at this point, and roots from static data are all zero, practically all heap data is dead and collected quickly.

Once a JVM has been chosen to be reused it will perform the reinitialization phase, which sets the new file descriptors and runs the class initialization code of all loaded classes to correctly initialize the static data since it had been previously set to zero. The order this is done in is important because dependencies between classes can exist. HotTub maintains the correct order by recording the order of class initializations when they are first initialized and replaying the initializations in the same order before each reuse. There are some limitations to reinitializing static data, since known bad practices such as class dependence cycles and real time static initialization dependencies will cause HotTub to be inconsistent. However, these cases are extremely uncommon in practice.

Handling Signals and Explicit Exit

HotTub has to handle signals such as SIGTERM and SIGINT and explicit exit by the application, otherwise it will lose the target server process from our pool. If the application registers its own signal handler, HotTub forwards the signal. If SIGKILL is used or the application exists through a native library, the JVM will die and cannot be reused.

Privacy Limitation

The use of HotTub raises privacy concerns. HotTub limits reuse to the same Linux user, as cross-user reuse allows a different user to execute code with the privileges of the first user. However, our design still violates the principle "base the protection mechanisms on permission rather than exclusion" [6]. Although we carefully clear and reset data from the prior run, an attacker could still reconstruct the partial execution path of the prior run via timing channel since previously loaded classes and JIT-compiled methods can be seen.

Workload	Completion Time (s) Unmod.	Completion Time (s) HotTub	Speed-up
HDFS read 1 MB	2.29	0.08	30.08x
HDFS read 10 MB	2.65	0.14	18.04x
HDFS read 100 MB	2.33	0.41	5.71x
HDFS read 1 GB	7.08	4.26	1.66x
Spark 100 GB best	65.2	36.2	1.80x
Spark 100 GB median	57.8	35.2	1.64x
Spark 100 GB worst	74.8	54.4	1.36x
Spark 3 TB best	66.4	41.4	1.60x
Spark 3 TB median	98.4	73.6	1.34x
Spark 3 TB worst	381.2	330.0	1.16x
Hive 100 GB best	29.0	16.2	1.79x
Hive 100 GB median	38.4	25.0	1.54x
Hive 100 GB worst	206.6	188.4	1.10x

Table 2: Performance improvements by comparing the job completion time of an unmodified JVM and HotTub. For Spark and Hive we report the average times of the queries with the, best, median, and worst speed-up for each data size. Speed-up values were calculated using full-precision values, not the rounded values shown as completion times in this table.

Performance of HotTub

We conduct a variety of experiments on HotTub in the same manner as our JVM warm-up performance analysis to evaluate its performance. Table 2 shows HotTub's speed-up compared with an unmodified HotSpot JVM. We ran the same workload five times on an unmodified JVM and six times on HotTub. We compared the average runtime of the five unmodified runs with the average runtime of the five reuse HotTub runs, excluding the initial warm-up run. For Spark and Hive, we ran the same 10 queries that we used in our study.

The results show that HotTub significantly speeds up the total execution time. For example, HotTub reduces the average job completion time of the Spark query with the highest speed-up by 29 seconds on 100 GB data, and can speed up HDFS 1 MB read by a factor of 30.08. Among nearly 200 pairs of trials, a job running in a reused HotTub JVM always completed faster than an unmodified JVM. Enabling our performance counters, we observe that indeed HotTub eliminates the warm-up overhead. In all the experiments, the server JVM spends less than 1% of the execution time in class loading and interpreter.

In addition to evaluating the speed-up of HotTub in our paper, we evaluated many other aspects. We also found that the majority of speed-up comes in the first reuse run. When inspecting hardware performance counters we saw a large reduction in memory accesses due to avoidance of class loading and interpretation. We found that when reusing JVMs that were warmed up with a different query than the one being run, HotTub still achieved similar speed-ups since different jobs still tend to use similar framework code in these systems. Also, the management overhead of HotTub turned out to be low, only adding a few hundred milliseconds to the critical path.

Conclusion

We started this project curious to understand the JVM’s overhead on data-parallel systems, driven by the observation that systems software is increasingly built on top of it. Enabled by non-trivial JVM instrumentations, we observed the warm-up overhead and were surprised by the extent of the problem. We then pivoted our focus on to the warm-up overhead by first presenting an in-depth analysis on three real-world systems. Our results show the warm-up overhead is significant, bottlenecks even I/O intensive jobs, increases as jobs become more parallelized and short running, and is aggravated by multi-layered systems. We further designed HotTub, a drop-in replacement of the JVM that can eliminate warm-up overhead by amortizing it over the lifetime of a host. Evaluation shows it can speed up systems like HDFS, Hive, and Spark, with a best case speed-up of 30.08x.

Acknowledgments

We greatly appreciate the insightful feedback from our anonymous OSDI reviewers, our OSDI shepherd Andrea Arpaci-Dusseau, and Rik Farrow. We thank Yu Luo, Serhei Makarov, Michael Stumm, Jenny Ren, Kirk Rodrigues, Guangji Xu, Yongle Zhang, and Xu Zhao for the useful and thought-stimulating discussions. We thank Yu Luo for setting up and maintaining the server cluster environment used in our experiments. His help was invaluable. This research is supported by NSERC Discovery grant, NetApp Faculty Fellowship, and an NSERC USRA award. Hailong Sun is supported by National Key Research and Development Program of China (2016YFB1000804) and National Natural Science Foundation of China (61370057).

References

[1] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, “PACMan: Coordinated Memory Caching for Parallel Jobs,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI ’12)*, 2012: https://www.usenix.org/system/files/conference/nsdi12/pacman.pdf.

[2] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, “Scale-up vs. Scale-out for Hadoop: Time to Rethink?” in *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC ’13)*, 2013.

[3] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, “Bigbench: Towards an Industry Standard Benchmark for Big Data Analytics,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD ’13)*, 2013.

[4] Hypertable: “Why We Chose CPP over Java”: https://code.google.com/p/hypertable/wiki/WhyWeChoseCppOverJava.

[5] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan, “Don’t Get Caught in the Cold, Warm Up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI ’16)*, 2016: https://www.usenix.org/system/files/conference/osdi16/osdi16-lion.pdf.

[6] J. H. Saltzer, “Protection and the Control of Information Sharing in Multics,” *Communications of the ACM*, vol. 17, no. 7 (1974), pp. 388–402.

[7] “StackOverflow: Is Java Really Slow?”: http://stackoverflow.com/questions/2163411/is-java-really-slow.

[8] Yanpei Chen, Cloudera, “What Do Real-Life Apache Hadoop Workloads Look Like?”: http://blog.cloudera.com/blog/2012/09/what-do-real-life-hadoop-workloads-look-like/.

Gleeful Incompatibility

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Recently someone asked me when I thought that Python 2 and Python 3 might converge. They were a bit dismayed when I replied "never." If anything, Python 3 is moving farther and farther away from Python 2 at an accelerating pace. As I write this, Python 3.6 is just days from being released. It is filled with all sorts of interesting new features that you might want to use if you dare. Of course, you'll have to give up compatibility with all prior versions if you do. That said, maybe an upgrade is still worth it for your personal projects. In this article, I look at a few of the more interesting new additions. A full list of changes can be found in the "What's New in Python 3.6" document [1].

But First, Some Reflection

Since my earliest usage of Python, I've mostly viewed it as a personal productivity tool. I write a lot of custom scripts and use it for all sorts of tasks ranging from system administration to data processing. When I see new features, I think about how I might use them to make my life easier and more interesting. To be sure, this is a different view than that of a typical library writer who wants to maintain backwards compatibility with prior versions of Python. If you're mainly writing scripts for yourself, it is liberating to free yourself from the constraints of backwards compatibility. In this regard, Python 3.6 does not disappoint. However, if you're maintaining code for others, everything you're about to read should be taken with a grain of caution. So, with that said, let's begin!

String Formatting

Suppose you had a list of tuples like this

```
portfolio = [
  ('IBM', 50, 91.1),
  ('MSFT', 100, 63.45),
  ('HPE', 35, 42.75)
]
```

and you wanted to produce a nicely formatted table. There are many approaches to string formatting you might take. For example, you could use the classic string formatting operator (%):

```
>>> for name, shares, price in portfolio:
...     print('%10s %10d %10.2f' % (name, shares, price))
...
       IBM         50      91.10
      MSFT        100      63.45
       HPE         35      42.75
>>>
```

Or you could use the more verbose .format() method of strings:

```
>>> for name, shares, price in portfolio:
...     print('{:>10s} {:10d} {:10.2f}'.format(name, shares,
price))
...
       IBM         50      91.10
      MSFT        100      63.45
       HPE         35      42.75
>>>
```

Starting in Python 3.6, you can now use so-called "f-strings" to accomplish the same thing using far less code:

```
>>> for name, shares, price in portfolio:
...     print(f'{name:>10s} {shares:10d} {price:10.2f}')
...
       IBM         50      91.10
      MSFT        100      63.45
       HPE         35      42.75
>>>
```

f-strings are a special declaration of a string literal where expressions enclosed in braces are evaluated, converted to strings, and inserted into the resulting string [2]. In the above example, the name, shares, and price variables are picked up from the enclosing loop and inserted into the string. There's no need to use a special operator or method such as % or .format().

At first glance, it might appear that f-strings are a minor enhancement of what is already possible with the normal format() method. For example, format() already allows similar name substitutions:

```
>>> '{name:>10s} {shares:10d} {price:10.2f}'.
format(name=name, shares=shares, price=price)
'       HPE         35      42.75'
>>>
```

However, f-strings allow so much more. The greater power comes from the fact that nearly arbitrary expressions can be evaluated in the curly braces. For example, you can invoke methods and perform math calculations like this:

```
>>> f'{name.lower():>10s} {shares:10d} {price:10.2f}
{shares*price:10.2f}'
'       hpe         35      42.75    1496.25'
>>>
```

That's pretty neat and possibly rather surprising. For the most part, any expression can be placed inside the braces. The only restriction is that it cannot involve the backslash character (\). So attempts to mix f-strings and regular expressions might be thwarted. Of course, that's probably a good thing. Maybe.

Supervising Subclasses

Another interesting feature of Python 3.6 is the ability of a parent class to supervise the creation of child subclasses [3]. This can be done by providing a new special class method __init_subclass__(). For example, suppose you have this class:

```
class Base(object):
    @classmethod
    def __init_subclass__(cls):
        print('Base Child', cls)
        super().__init_subclass__()
```

Now, if you inherit from the class, you'll see the method fire:

```
>>> class A(Base):
...     pass
...
Base Child <class '__main__.A'>
>>> class B(A):
...     pass
...
Base Child <class '__main__.B'>
>>>
```

The use of super() in this example is to account for multiple inheritance. It allows for all of the parents to participate in the supervision if they want. For example, if you also had this class:

```
class Parent(object):
    @classmethod
    def __init_subclass__(cls):
        print('Parent Child', cls)
        super().__init_subclass__()
```

Now watch what happens with multiple inheritance:

```
>>> class C(Base, Parent):
...     pass
...
Base Child <class '__main__.C'>
Parent Child <class '__main__.C'>
>>>
```

Supervising subclasses might seem like a fairly esoteric feature, but it turns out to be rather useful in a lot of library and framework code because it can eliminate the need to use more advanced techniques such as class decorators or metaclasses. Here's an example that uses the __init_subclass__() method to register classes with a dictionary that's used in a convenience function.

```
class TableFormatter(object):
    _formats = {}
    @classmethod
    def __init_subclass__(cls):
        cls._formats[cls.name] = cls

def create_formatter(name):
    formatter_cls = TableFormatter._formats.get(name)
    if formatter_cls:
      return formatter_cls()
    else:
      raise RuntimeError('Bad format: %s' % name)

class TextTableFormatter(object):
    name = 'text'

class CSVTableFormatter(object):
    name = 'csv'

class HTMLTableFormatter(object):
    name = 'html'
```

In this code, the TableFormatter class maintains a registry of child classes. The create_formatter() function consults the registry and makes an instance using a short name. For example:

```
>>> create_formatter('csv')
<__main__.CSVTableFormatter object at 0x10ae9f748>
>>>
```

There are many other situations where a base class might want to supervise child classes. We'll see another example shortly.

Ordering Some (All?) of the Dicts

One of the more dangerously interesting features of Python 3.6 is that there are many situations where dictionaries are now ordered—preserving the order in which items were inserted. A dictionary like this

```
>>> s = { 'name': 'ACME', 'shares': 100, 'price': 385.23 }
>>>
```

now preserves the exact insertion order. This makes it much easier to turn a dictionary into a list or tuple in a way that respects the original structure of data. For example:

```
>>> keys = list(s)
>>> keys
['name', 'shares', 'price']
>>> row = tuple(s.values())
('ACME', 100, 385.23)
>>> dict(zip(keys, row))
{ 'name': 'ACME', 'shares': 100, 'price': 385.23 }
>>>
```

The fact that order is preserved may simplify a lot of data-handling problems: e.g., preserving the order of data found in files, JSON objects, and more. So, on the whole, it seems like a nice feature.

This ordering applies to other dictionary-related functionality. For example, if you write a function involving **kwargs, the order of the keyword arguments is preserved [4]:

```
>>> def func(**kwargs):
...     print(kwargs)
...
>>> func(spam=1, bar=2, grok=3)
{ 'spam': 1, 'bar': 2, 'grok': 3 }
>>>
```

Since the order is preserved, it seems to open up more possibilities for interesting functions involving **kwargs. For example, maybe you want to convert a sequence of lists to dictionaries:

```
rows = [
  ['IBM', '50', '91.1'],
  ['MSFT', '100', '63.45'],
  ['HPE', '35', '42.75']
]

def parse_rows(_rows, **columns):
  types = columns.values()
  names = columns.keys()
  for row in _rows:
     yield { name: func(val)
           for name, func, val in zip(names, types, row) }

for r in parse_rows(rows, name=str, shares=int, price=float):
   print(r)
```

Similarly, modules and classes now capture the definition order of their contents [5]. This is potentially useful for code that performs various forms of code introspection. For example, you can iterate over the contents of a class or module in definition order using a loop like this:

```
>>> import module
>>> for key, val in vars(module).items():
...     print(key, val)
...
>>>
```

As noted, this is one of the more dangerous features of Python 3.6. Past versions of Python do not guarantee dictionary ordering. So, if you rely upon this, know that your code will not work on any prior version. Also, the ordering seems to be provisional—meaning that it could be removed or refined in future Python versions.

Annotating All the Things

Since the earliest release of Python 3, it was possible for functions to have annotated arguments. For example:

```
def add(x:int, y:int) -> int:
    return x + y
```

The annotations didn't actually do anything, but served more as a kind of documentation. Tools could obtain the annotations by looking at the function's __annotations__ attribute like this:

```
>>> add.__annotations__
{'x': <class 'int'>, 'y': <class 'int'>, 'return': <class 'int'>}
>>>
```

The annotation idea is now extended to class attributes and variables [6]. For example, you can write a class like this:

```
class Point:
    x:int
    y:int
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Like their function counterparts, the annotations do nothing. They are merely collected in a class __annotations__ attribute.

```
>>> Point.__annotations__
{'x': <class 'int'>, 'y': <class 'int'>}
>>>
```

You can also annotate free-floating variables in a module. For example:

```
# spam.py

x:int = 0
y:int = 1
```

In this case, they become part of a module level __annotations__ dictionary.

```
>>> import spam
>>> spam.__annotations__
{'x': <class 'int'>, 'y': <class 'int'>}
>>>
```

It's important to note that the annotations don't change any aspect of Python's behavior. They are extra metadata that can be used by other tools such as frameworks, IDEs, or program checkers.

Summoning the Genie

Now that we've seen a few new features, it's time to gleefully put them into practice with something more interesting. How about a typed tuple object with a silly name?

```
import operator

class Toople(tuple):
    @classmethod
    def __init_subclass__(subcls):
        types = list(subcls.__annotations__.items())

        @staticmethod
        def __new__(cls, *args):
            if len(args) != len(types):
                raise TypeError(f'Expected {len(types)} args')
            for val, (name, ty) in zip(args, types):
                if not isinstance(val, ty):
                    raise TypeError(f'{name} must be an {ty.
__name__}')
            return super().__new__(cls, args)
        subcls.__new__ = __new__

        def __repr__(self):
            return f'{subcls.__name__}{super().__repr__()}'
        subcls.__repr__ = __repr__


        # Make properties for the attributes
        for n, name in enumerate(subcls.__annotations__):
            setattr(subcls, name, property(operator.itemgetter(n)))
```

Good god—f-strings, annotations, subclassing of the tuple built-in, and an __init_subclass__ method that's patching child classes. What is going on here? Obviously, it's a small bit of Python 3.6 code that lets you write typed-tuple classes like this:

```
class Point(Toople):
    x:int
    y:int

class Stock(Toople):
    name:str
    shares:int
    price:float
```

Check it out:

```
>>> p = Point(2, 3)
>>> p
Point(2, 3)
>>> p.x
2
>>> p.y
3
```

```
>>> s = Stock('ACME', 50, 98.23)
>>> s
Stock('ACME', 50, 98.23)
>>> s.name
'ACME'
>>> s.shares
50
>>>

>>> Stock('ACME', '50', '98.23')
Traceback (most recent call last):
  ...
TypeError: shares must be an int
>>>
```

Okay, that's kind of awesome and insane. Don't try it on anything earlier than Python 3.6 though. It requires all of the features discussed including the reliance on newfound dictionary ordering. In fact, your coworkers might chase you out of the office while waving flaming staplers and hurling single-serve coffee packets at you if you put code like that in your current application. Nevertheless, it's a taste of what might be possible in the Python of the distant future.

Final Words

Over the last few years, a lot has been said about the Python 2 vs. Python 3 split. There are those who claim that Python 3 doesn't offer much that's new. Although that might have been true five years ago, it's becoming much less so now. In fact, Python 3 has all sorts of interesting new language features that you might want to take advantage of (e.g., I haven't even talked about the expanded features of async functions that were introduced in Python 3.5). Python 3.6 pushes all of this to a whole new level. Frankly, Python 3 has become a lot of fun that rewards curiosity and an adventurous spirit. If you're starting a new project, it's definitely worth a look.

References

[1] What's New in Python 3.6: https://docs.python.org/3.6/whatsnew/3.6.html.

[2] PEP 498—String literal interpolation: https://www.python.org/dev/peps/pep-0498/.

[3] PEP 487—Simpler customization of class creation: https://www.python.org/dev/peps/pep-0487/.

[4] PEP 468—Preserving the order of **kwargs in a function: https://www.python.org/dev/peps/pep-0468/.

[5] PEP 520—Preserving class attribute definition order: https://www.python.org/dev/peps/pep-0520/.

[6] PEP 526—Syntax for variable annotations: https://www.python.org/dev/peps/pep-0526/.

Practical Perl Tools

Off the Charts

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the Technical Evangelist at Apcera (the comments/views here are David's alone and do not represent Apcera/Ericsson) . He has spent close to 30 years in the systems administration/DevOps/SRE field in large multiplatform environments including Brandeis University, Cambridge Technology Group, MIT Media Laboratory, and Northeastern University. He is the author of the O'Reilly Otter book *Automating System Administration with Perl* and is a frequent invited speaker/organizer for conferences in the field. David is honored to serve on the USENIX Board of Directors. He prefers to pronounce Evangelist with a hard 'g'. dnb@usenix.org

I recently had the pleasure of giving another invited talk to the LISA 2016 audience. Part of preparing that talk involved performing some basic forensics on a machine that I could no longer directly access. I wanted to explore how its file systems had changed over time. I like pretty pictures, so my first goal was to attempt to draw a diagram that represented this evolution. In this column, we'll take a look at the code I wrote to achieve this. Just a warning up front: there are a number of moving parts for the approach I took (not all of which are Perl), but I'll do my best to explain all of the plates that are being spun.

The Best Tool for Storing Data Is a…

When I started out, I wasn't exactly clear what sort of representation of the data I needed or even what was the best way to wade through the information I had on hand. I was given access to a set of directory listings (basically the output of recursive `ls -lR {something} > output` commands for the file system with different flags used). The beginning of one of these files looked like this:

```
/etc:
total 851
drwxr-xr-x   67 root    sys             5120     Sep 19 12:19 .
drwxr-xr-x   39 root    root            2048     Jan  8  2016 ..
drwxr-xr-x    2 adm     adm              512     May 15  2006 acct
lrwxrwxrwx    1 root    root         14 May 15   2006 aliases -> ./mail/aliases
drwxr-xr-x    2 root    bin         512 May  5   2009 apache
drwxr-xr-x    2 root    bin         512 May 15   2006 appserver
-rw-r--r--    1 root    bin          50 May 15   2006 auto_home
-rw-r--r--    1 root    bin         113 Mar 20   2008 auto_master
-rw-r--r--    1 root    other     47389 Mar 31   2009 bootparams
-rw-r--r--    1 root    other     47389 Mar 31   2009 bootparams.old
-rw-r--r--    1 root    other     47397 Mar 27   2009 bootparams.orig
lrwxrwxrwx    1 root    root         18 May 15   2006 chroot -> ../usr/sbin/chroot
-rw-r--r--    1 root    other            314     Jun 15 15:04 coreadm.conf
lrwxrwxrwx    1 root    root         16 May 15   006 cron -> ../usr/sbin/cron
drwxr-xr-x    2 root    sys              512     Jun 15 15:05 cron.d
```

My thinking was that if I could get all of this information into a database, it would allow me to play around with the data through ad hoc queries. The tricky part was parsing the files because, as you can see above, it is basically a hot mess. The actual directory name itself appears in a different format before the entries. Some files have explicit years in their dates, some do not. Some files aren't even files (in the classic sense), they are links to files or directories. Whee!

Output like this is notoriously hard to parse as a Web search on the question will quickly reveal. Much to my delight, it turns out that a module that attempts to handle this unpleasantness actually exists called File::Listing. Here's the code I wrote to use that module to slurp the contents of a directory listing into a SQLite database. SQLite was used because of its lightweight nature and ability to install as part of a single Perl module (DBI::SQLite) install. This was easier than installing/configuring a database, its libs, and a separate Perl module before I could make progress.

```
use strict;
use DBI qw(:sql_types);

my $file = shift;

open my $L, '<', $file or die "Can't open $file:$!\n";
my $dir = File::Listing::parse_dir( $L, undef, 'unix', 'warn' );
close $L;

my $dbh = DBI->connect( "dbi:SQLite:dbname=$file.db", "", "" );

# throw an error if something fails so we don't have to check
# the results of every statement and turn off committing the
# data to the file on every insert
$dbh->{RaiseError} = 1;
$dbh->{AutoCommit} = 0;

$dbh->do(
  "CREATE TABLE dir (filename text, filetype text, filesize
integer, filetime integer, filemode text)"
);

my $sth = $dbh->prepare(
  "INSERT INTO dir (filename, filetype, filesize, filetime,
filemode) VALUES (?,?,?,?,?)"
);

my $rowcount = 0;
foreach my $listing (@$dir) {
    $sth->execute(@$listing);
    if ( $rowcount++ % 1000 == 0 ) {
        $dbh->commit;
        print STDERR ".";
    }
}

$dbh->commit;
$dbh->disconnect;
```

We've talked about using DBI (the Perl DataBase Independent) framework before, so we won't go into depth about how that part of this code works. Instead, let me just give a brief summary of what is going on and mention a few salient points. The first thing this code does is read in the listing file as specified on the command line and parse it. The file gets parsed and stored in memory (the listings I had were only about 30 MB so I could get away with it).

We then do the DBI magic necessary for "connecting" to a SQLite database file (creating it if it does not exist—in this case we use the name of the listing file as the start of that database file name), create a table called "dir" into which we'll store the info, and then start to populate it. We iterate through the parsed file info we have in memory, inserting the info into the database. After every 1000 records, we actually commit those inserts to the file and print a dot to let us know the process is working. We didn't have to turn off autocommit and commit explicitly like this, but we get a wee bit of a performance boost if we do so. After the script runs we are left with a nice `filename.db` SQLite file we can query to our heart's content.

After much playing around with SQL queries and Web searches about SQLite SQL queries, I finally hit upon this SQL statement to do what I needed:

```
SELECT strftime('%Y-%m', filetime, 'unixepoch') yr_mon,count(*)
num_dates FROM dir GROUP BY yr_mon;
```

It produces results that looks like this:

```
yr_mon:num_dates
1973-05:1
1992-09:1
1993-04:2
1993-06:1
1993-07:4
1993-08:12
1993-09:4
1993-10:6
1993-11:1
1993-12:1
...
```

Here we have the number of files created in each of the listed months (e.g., in August of 1993, 12 files were created). Now all we have to do is represent this information in a chart.

Google Charts Ho!

I could have fed these results into any number of applications or services that draw graphs, but I thought it might be fun to learn how to use Google Charts from Perl. Plus, it had a wide variety of charts available, so I thought it would be good to hedge my bets.

The tricky thing with Google Charts is it is not meant to run client-side. We won't be running a program on local data and have it spit out a chart. Instead, the process is roughly: you load a Web page, that Web page loads some Google Chart libraries, creates the necessary JavaScript objects, makes a call to get the data for the chart (if it isn't embedded in the page), and then asks Google to return the desired chart, which is shown by your browser as embedded in the Web page. If that sounds like a little bit of work, it definitely is (at least the first time you are trying it). We'll go slow.

One thing to note here is that in order for this to work, you will need to load this Web page (and its surprise guest that we'll get to in a moment) from a Web server. You can't just load it from the File->Open menu items in your browser. Your server doesn't have to be anything high-powered (I used Apache via MAMP PRO running on my laptop to serve the files, but that was just because it was already handy), but you do need one for Google Charts to function properly.

The first thing to do is to create the Web page mentioned above. It is going to have a small amount of HTML and a bunch of JavaScript. The docs at https://developers.google.com/chart are really quite good, so you can get very far via simple cut-and-pasting even if your JavaScript isn't so hot (phew). Here's the .html page I used (I'll break it down in a sec):

```
<head>
 <script type="text/javascript" src="https://www.gstatic.com/
charts/loader.js"></script>
 <script type="text/javascript" src="//ajax.googleapis.com/
ajax/libs/jquery/1.10.2/jquery.min.js"></script>
 <script type="text/javascript">
   google.charts.load('current', {'packages':['scatter']});
   google.charts.setOnLoadCallback(drawChart);

   function drawChart () {

       var jsonData = $.ajax({
           url: "get_data.pl?filename=listing.db",
           dataType: "json",
           async: false
           }).responseText;


      var data = new google.visualization.
DataTable(jsonData);

      var options = {
        width: 2000,
        height: 700,
        chart: {
         title: 'File Creation Dates',
                  subtitle: 'listing',
        },
        hAxis: {title: 'Date'},
        vAxis: {title: 'Number'}
       };

      var chart = new google.charts.Scatter(document.
getElementById('scatterchart_material'));

      chart.draw(data,   google.charts.Scatter.
convertOptions(options));
   }
  </script>
 </head>
 <body>


    <div id="scatterchart_material"></div>
 </body>
</html>
```

Much of the above is straight from the docs, so I'll just briefly mention what is going on. It can be a bit of a challenge to read because most of the listing consists of definitions that get triggered at the right moment. After we load the right libraries and set up something that will cue the function that does all of the work after everything is loaded, we define that function `drawChart()`. In `drawChart()` we specify how we are going to pull the data (more on that in a moment), various options on how the chart should look, what HTML element in the document will "hold" the resulting chart, followed by a call to actually kick off the drawing. When the page loads, it will call `drawChart()` and we are off to the races.

How Does the Data Get into the Chart?

Yeah, that's one of the fun questions. The key part was in our description of how the data should be loaded:

```
var jsonData = $.ajax({
              url: "get_data.pl? filename=listing.db",
              dataType: "json",
              async: false
             ).responseText;
var data = new google.visualization.DataTable(jsonData);
```

Google Charts lets you specify the data for a chart inline (i.e., you can put JSON right in the .html file), but that only works for smallish data sets. The chart I was hoping to build had 278 rows of data, so I wasn't keen on embedding that all in the same doc. Instead, we're going make an AJAX call to another URL (`get_data.pl`) and ask it to send us the data set. We can then take those results and put them in the proper object for graphing.

Time for some more Perl. We'll need a CGI script that will query our database using the SELECT statement we previously saw and format the results into the proper JSON output expected by the Google Charts API. When I heard the requirements "CGI" and "JSON output," a couple of the frameworks we've seen in this column before (Mojolicious and Dancer) leapt right to mind. My choice was cemented when I saw Joel Berger's excellent blog post "Some code ports to Mojolicious, just for fun" [1]. It was a post about porting another person's work on Google Charts from

Perl to Mojolicious::Lite. The code we're about to see is a direct descendant of Joel's example with a few fun twists.

Let's take this task piece by piece. The CGI portion of the script is only a few lines:

```
use Mojolicious::Lite;
use DBIx::Connector;

any '/' => sub {
  my $c = shift;

  my $filename = $c->param('filename');
  my $data = $c->get_data($filename);
  $c->render( json => $data );
};
```

This just says that when a request for the URL "/?filename=something" comes in we will parse out the parameter (the file name of the database we'll be using), the proper database query will be made and results returned in the right form, and this will be converted into JSON and sent to the requester.

More interesting are the two helper functions we will define. The first is responsible for getting us a safe database handle for the right SQLite database:

```
helper db => sub {
  my $filename = $_[1];
  state $db =
   DBIx::Connector->connect( "dbi:SQLite:dbname=$filename",
'', '' );
};
```

By the way, if you haven't seen DBIx::Connector before (I hadn't), it is worth looking up because it is quite spiffy.

Now for the more complex part of the script, a helper that does the actual query for data and then transforms the results so the JSON will be correct.

First step, perform the actual query:

```
helper get_data => sub {


  my $filename = $_[1];
  my $db = shift->db($filename);

  my $query =
"SELECT strftime('%Y-%m', filetime, 'unixepoch') yr_mon,count(*)
num_dates FROM dir GROUP BY yr_mon;";

  my $data = $db->selectall_arrayref($query);
```

Now for some annoying stuff. Google Charts expects to receive the data set in a very specific JSON format. This means we're going to have to transform the data coming out of the database in a very particular way such that we match the format expected when the Perl data structure to JSON conversion is made.

The JSON format Google Charts expects looks like this [2]:

```
{
 cols: [{id: 'A', label: 'NEW A', type: 'string'},
    {id: 'B', label: 'B-label', type: 'number'},
    {id: 'C', label: 'C-label', type: 'date'}
 ],
 rows: [{c:[{v: 'a'},
      {v: 1.0, f: 'One'},
      {v: new Date(2008, 1, 28, 0, 31, 26), f: '2/28/08
12:31 AM'}
    ]},
    {c:[{v: 'b'},
      {v: 2.0, f: 'Two'},
      {v: new Date(2008, 2, 30, 0, 31, 26), f: '3/30/08
12:31 AM'}
    ]},
    {c:[{v: 'c'},
      {v: 3.0, f: 'Three'},
      {v: new Date(2008, 3, 30, 0, 31, 26), f: '4/30/08
12:31 AM'}
    ]}
 ]
}
```

I'm going to describe this JSON blob in terms of Perl data structures because I think it will make it easier to understand the Perl code we are about to see. You can look at this like a hash with two keys, 'cols' and 'rows'. The cols part is basically a definition of the contents of the rows that will follow. If it helps, think of this as the column heading of a spreadsheet followed by a bunch of rows.

The cols portion is constructed from an array that holds three separate hashes, one for each column being defined. So the first column has a key of 'id' whose value is "A," a key of "label" whose value is "NEW A," and a key of "type" whose value is "string." This is how we specify the id of the first column, how it will be labeled, and what kind of values it will contain.

Here's how we build our version of that part of the data structure in Perl:

```
my $response->{'cols'} = [
  { 'id'    => 'Date',
   'label' => 'date',
   'type'  => 'string' },
  { 'id'    => 'Count',
   'label' => 'count',
   'type'  => 'number' },
];
```

This code creates a similar array with two hashes in it, one for each column. We'll have a column for the date (e.g., "1993-08") and the number of files in that time period ("12").

Now let's tear apart one of the rows. A row consists of an array of cells containing values (that makes sense, yes? if just from your use of spreadsheets).

Here's an example row (the first one):

```
rows: [{c:[{v: 'a'},
           {v: 1.0, f: 'One'},
           {v: new Date(2008, 1, 28, 0, 31, 26), f: '2/28/08 
12:31 AM'}
        ]},
```

It shows a row that consists of cells containing the values "a," "1.0," and a newly defined date. So, something like this:

```
|a|1.0|2/28/08 12:31 AM|
```

Each value is stored in a hash with the key 'v' (for value). There is also another (optional) key 'f' in the example above for "formatted value" (i.e., how the value should be displayed).

To review:

- We're going to create a key called 'rows' in the hash we created above when defining the columns.
- 'rows' will contain an array that holds the hashes defining the cells in each row.
- Each cell hash needs to have a single key of 'c' to mark it as cell data.
- 'c' will hold an array of hashes, each representing a cell value.
- Each hash holding a value will have a key called 'v' whose value is the value for that cell.

Phew! See why I call this annoying? Now let's take on creating this in Perl:

```
    foreach my $row (@$data) {
        my $c->{'c'} = [ map { { 'v' => $_ } } @$row ];
        push( @{ $response->{'rows'} }, $c );
    }
```

It may be a bit surprising to realize all of that rigmarole can be implemented in just three lines of code. It is probably not surprising that it is three lines of fairly gnarly/compact code. Let's unravel it so it all makes sense.

The DBI query we made above

```
$db->selectall_arrayref($query);'
```

returns a reference to an array containing the results of our query:

```
0  ARRAY(0x7fccb45eaa90)
   0  ARRAY(0x7fccb45ebca0)
      0  '1973-05'
      1  1
   1  ARRAY(0x7fccb45eb060)
      0  '1992-09'
      1  1
   2  ARRAY(0x7fccb4002c68)
      0  '1993-04'
      1  2
   3  ARRAY(0x7fccb45ec880)
      0  '1993-06'
      1  1
   4  ARRAY(0x7fccb45ec988)
      0  '1993-07'
      1  4
   5  ARRAY(0x7fccb45eb090)
      0  '1993-08'
      1  12
```

As you can see, each array is a row from the results of the query. Our code is going to iterate over these results, one row/array at a time:

```
    foreach my $row (@$data) {
```

For each value in the results, we're going to return an anonymous hash with a key of 'v' whose value is the value in the result. Here's the part that creates the anonymous hash for a value:

```
{ 'v' => $_ }
```

That's for a single value in the results. Here's how we iterate over all of the values in the results array, returning anonymous hashes as we go:

```
map { { 'v' => $_ } } @$row
```

We collect all of the {v}=something hashes the map{} returns into an array

```
[ map { { 'v' => $_ } } @$row ]
```

and stuff that array into a hash under the key of 'c' (representing the cells of the row):

```
my $c->{'c'} = [ map { { 'v' => $_ } } @$row ];
```

At this point, we now have a hash for the cells of that row. Go us! We need to store that hash into the array holding all of the rows of cells, so we append it to that array:

```
        push( @{ $response->{'rows'} }, $c );
```

If you find all of the punctuation in that line confusing, don't feel bad. Here's what's going on:

```
$response->{'rows'}
```

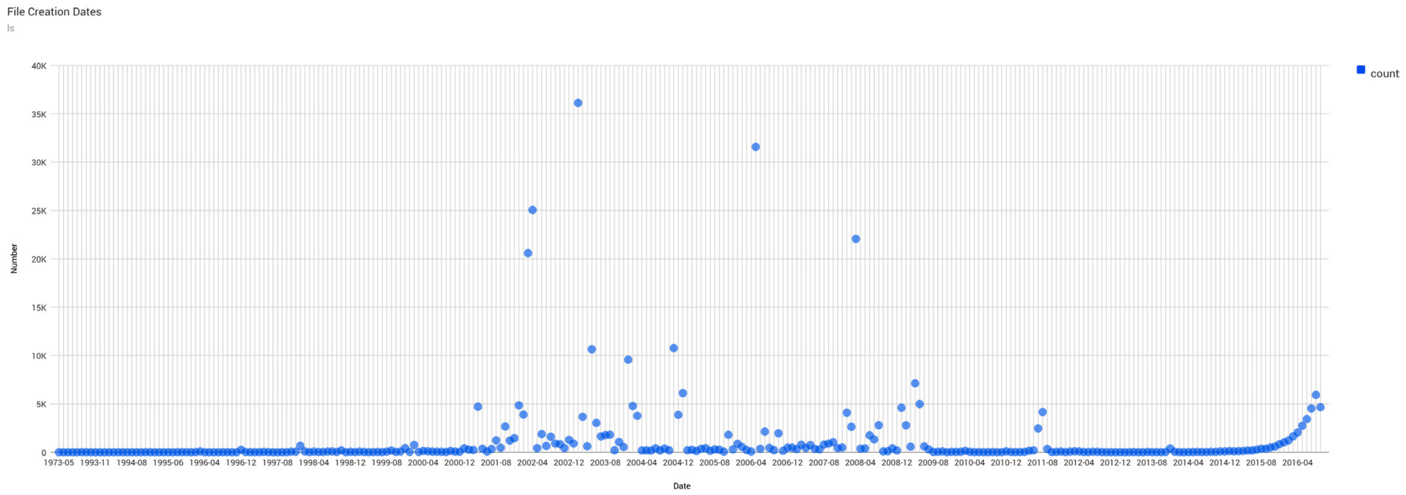


Figure 1: This chart appears in our browser when we load our Web page, and it all comes together.

$response is a reference to an anonymous hash, which has a key called 'rows'. So far so good?

The value for this key is a reference to an anonymous array (the one that is going to hold all of the row information). We need to de-reference it to get at the array itself, hence:

```
    @{ $response->{'rows'} }
```

Once we've done that, we can add this set of cells as another row in that array:

```
            push( @{ $response->{'rows'} }, $c );
```

And with that, we've done the work of retrieving the info from the database and transforming it into the right data structure.

If that felt a bit painful, I'll be the first to agree. It took me a while to build all that up piece by piece. To add insult to injury, well after I had completed the work I happened to stumble on the module Data::Google::Visualization::DataTable, which describes itself as "attempts to hide the gory details of preparing your data before sending it to a JSON serializer—more specifically, hiding some of the hoops that have to be jump[ed] through for making sure your data serializes to the right data types."

Sigh.

It hadn't come up during any of my other searches for Google Chart modules, so I (and now you) learned how to do it the hard way.

The last step for the CGI script is to translate the data structure into JSON and send it along to the requester; this happens in the last line of the script because Mojolicious::Lite makes it this simple:

```
    any '/' => sub {
        my $c = shift;
        my $filename = $c->param('filename');
        my $data = $c->get_data($filename);
        $c->render( json => $data );
    };
```

If we browse to the page we made, we get the lovely graph shown in Figure 1.

As I said, there are a number of moving parts. But once you get a sense of how they all work, you now get to bring to bear all of the power Google Charts has to offer you. Take care, and I'll see you next time.

References

[1] http://blogs.perl.org/users/joel_berger/2013/10/some-code-ports-to-mojolicious-just-for-fun.html.

[2] Google Visualization API Reference: https://developers.google.com/chart/interactive/docs/reference.

Cybersecurity Workload Trends

DAN GEER AND ERIC JARDINE

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

Eric Jardine is an Assistant Professor of Political Science at Virginia Tech and a Fellow at the Centre for International Governance Innovation. His research focuses on issues to do with measurement and cybersecurity, the uses and abuses of the Dark Web, and trust and the Internet ecosystem. ejardine@vt.edu.

> I became interested in long-term trends because an invention has to make sense in the world in which it is finished, not the world in which it is started.—*Ray Kurzweil*

A small bit of statistical wisdom: trend analysis can derive real guidance even when the measurement being examined is subject to consistent (relatively constant) error. Hold that thought…

NIST (the US National Institute of Standards & Technology) has for years collated and published vulnerability information, with the Common Vulnerability Scoring System (CVSS) being the best known of NIST's cybersecurity metrics. CVSS scores are numeric and calculated by a defined, constant formula [1]. Putting aside that calculation formula, CVSS is a stable system for which the errors are relatively constant.

From the CVSS data, NIST publishes on a daily basis what it calls a Workload Index, defined this way [2]:

> This [Workload Index] calculates the number of important vulnerabilities that information technology security operations staff are required to address each day. The higher the number, the greater the workload and the greater the general risk represented by the vulnerabilities.
>
> The NVD workload index is calculated using the following equation:
>
> (
> (number of high severity vulnerabilities published within the last 30 days) +
> (number of medium severity vulnerabilities published within the last 30 days/5) +
> (number of low severity vulnerabilities published within the last 30 days/20)
>) / 30
>
> The index equation counts five medium severity vulnerabilities as being equal in weight with 1 high severity vulnerability. It also counts 20 low severity vulnerabilities as being equal in weight with 1 high severity vulnerability.

Taking the Workload Index to be, just as it says, a composite estimate of the workload imposed on information technology security operations staff by the changing inventory of vulnerabilities in the CVSS catalog, we can begin to ask some questions.

The first and most obvious would be simply whether the workload due to known vulnerabilities is improving (going down) or worsening (going up). In finance, a typical measure of how a company is doing is "trailing twelve month" income—the income for the twelve-month period immediately prior to the date of the report. In Figure 1, we show the trailing 12-month value of the Workload Index over the past decade (overlain with a fitted order-2 polynomial, and with the X axis crossing the Y at Y=0).

Does that curve tell us anything? It certainly appears that information technology security operations staff had a few years of declining workload but may now be in a period of rising workload. One almost imagines a suite of arguments paralleling those about global warming to break out here—is workload rising or is this just natural variation?

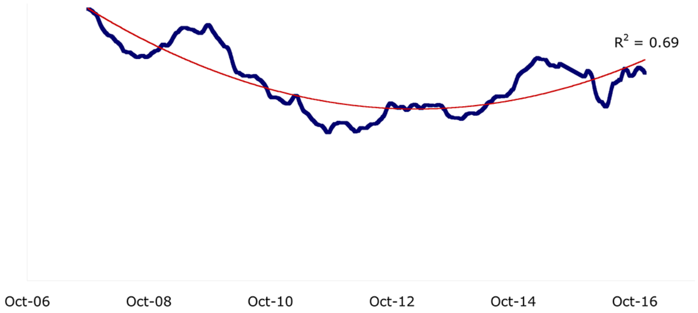


Figure 1: Trailing 12-month Workload Index

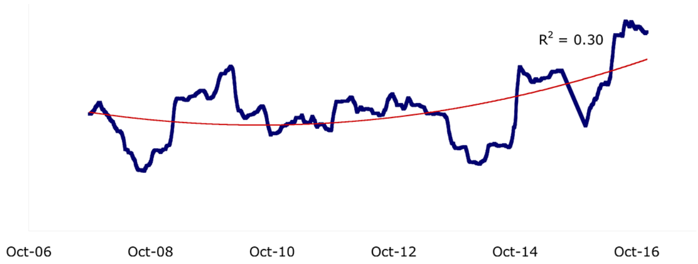


Figure 2: Trailing 12-month standard deviation (volatility)

In finance, the measure of variation is called "volatility," usually expressed as the trailing 12-month standard deviation. So, in Figure 2 we show exactly that, the trailing 12-month standard deviation of the Workload Index (again overlain with a fitted order-2 polynomial, and with the X axis crossing the Y at Y=0).

We might now ask (ourselves) how strong is the indication that volatility in the Workload Index is rising? Nassim Taleb, whom you may know from having read some of his *Incerto* tetralogy [3], has characterized a system with rising interconnectedness as one where a "black swan" event can (will) occur. In particular, he suggests that our hyper-connected society is "undergoing a switch between [continuous low grade volatility] to … the process moving by jumps, with less and less variations outside of jumps." The NVD Workload Index cannot itself answer a conjecture that serious vulnerabilities are becoming rarer except for the few that slip through and are found to be more serious than ever. What do you see in Figure 2?

So what is the meaning of "workload" anyhow? Can we think of it as interest on technical debt? Does it need some sort of normalization to be a worthy basis for decision-making? There is no doubt that the source of risk is dependence, particularly dependence on the stability of system state, so is this workload measure, along with other measures, a way to price our dependence? Or is it something else?

Let's think first about economy-wide effects. The number of schools offering instruction in cybersecurity has skyrocketed in the last decade [4]. All those people entering the field should have the effect of divvying up the workload, shouldn't they? The Index of Cyber Security [5] looked at one form of that question, asking it twice, 40 months apart: "As you look to fill vacancies in your organization, which of the following describes the status of the current job market for information security professionals?"

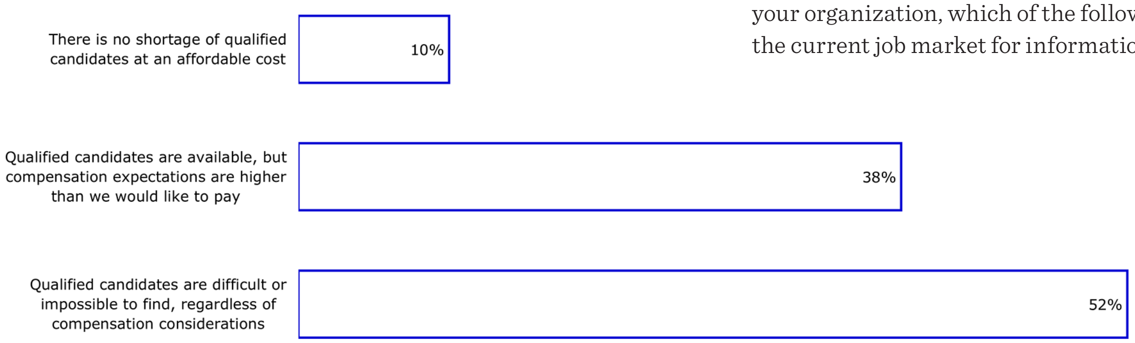


Figure 3A: November 2012

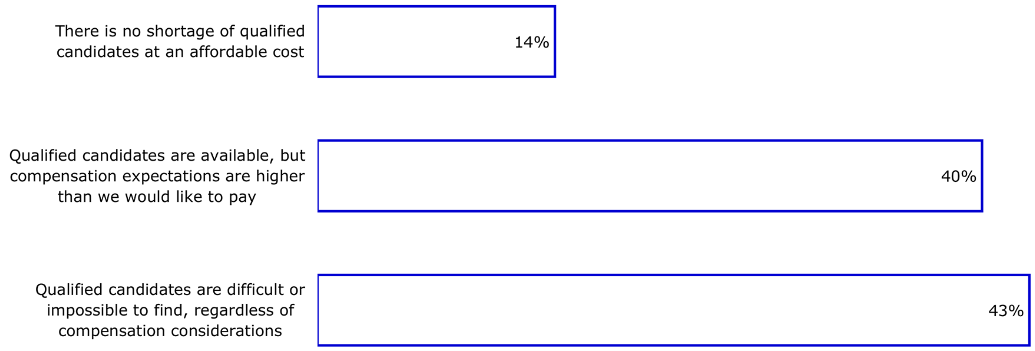


Figure 3B: April 2016

	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
mean WI	10.79	9.59	8.69	8.96	5.99	5.98	6.51	6.05	7.88	7.65
100K workers	4.01	4.67	4.75	4.71	5.37	5.53	6.05	6.02	6.29	6.52
WI/100K	2.69	2.05	1.83	1.90	1.12	1.08	1.08	1.01	1.25	1.17

Table 1: Workload Index normalized by number of workers

2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
54,111	47,480	42,170	38,476	37,992	39,593	43,066	47,406	50,962	55,367

Table 2: Bachelor's degrees in computer and information sciences

Figure 3A shows the answer in November 2012, while Figure 3B shows the answer in April of 2016.

From the first sample in 2012 to the second in 2016, the idea that "qualified candidates are difficult or impossible to find" fell by almost 10 percent. The answer that those frontline security managers gave implies an increasing supply of competent individuals with whom to share the workload. Can we normalize to that? And if we do, might that tell us more about the level of cybersecurity risk from technical vulnerabilities in the economy?

Table 1 shows the yearly average Workload Index number from 2006 to 2015, which can, in turn, be normalized by the US Bureau of Labor Statistics dataview for the number of workers in the category "Computer and information systems managers" [6]:

The data in Table 1 is redrawn as a chart in Figure 4, again overlain with a fitted order-2 polynomial. If you imagine plotting the mean Workload Index onto Figure 4 as well, you would have a line that declines into 2011, but then increases a fair amount from there on in. In this case, we see a steady decline and flattening of the curve when the index is normalized to the number of workers. Framed in this light, the "workload" posed by new vulnerabilities has gotten better since 2006 and remained relatively flat ever since. (Note that BLS data for the preferred category "Information security analysts" only began in 2011, so that category cannot yet be used for decadal views.)

Managing a variable amount of risk in a large system is only partially about the particular risks currently in that system; it is about the history (and future) of scaling factors as well. Sometimes, from 2006 to 2011, for example, when the mean score on the Workload Index was declining, one might naturally have inferred that cyberspace was becoming safer. Should we now infer that that welcome decline has stopped?

Over the last decade, the number of new graduates entering the workforce with computer science degrees fell and then rose, as seen in Table 2.

Those annual graduation numbers, as it turns out, are not correlated with the numbers of "Computer and information systems managers" in the workforce (r = .18), so either there is a lot of turnover among those jobs or the graduates are going somewhere else. So we will stick with "Computer and information systems managers" as our description of who is handling the vulnerability workload. But the Workload Index is really about how much work there is to be done. If we think of the work to be done as handing each member of the workforce a to-do list, then we would multiply the workforce count by the Workload Index and call that a measure of the work pending in the economy at large, viz., the size of the to-do list in the economy at large. That gets you Figure 5.

This mathematical manipulation generates an economy-wide to-do list, but labor markets can be sticky, as evidenced by the lack of a correlation between new computer science graduates and computer and information systems managers. This means that the "real" level of risk in the system might not have translated over into enough workers to actually handle the daily updates and patches needed to address the Workload Index. In

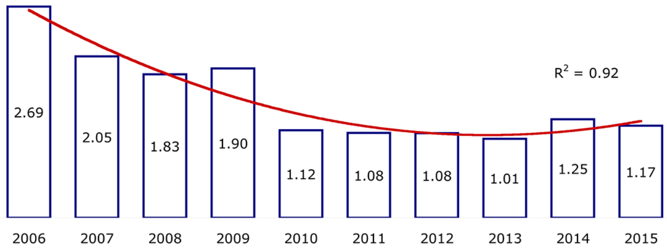


Figure 4: Workload Index normalized to number of workers

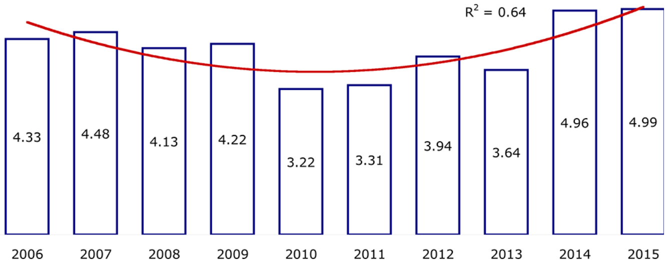


Figure 5: Millions of Items on the economy-wide to-do list

such a situation of a labor market failure, the formula for the economy-wide to-do list would be something like Y = Workload Index * (computer and information systems managers + X), where X is the number of workers who should be working in the system but are not due to market lags. X, in a world of imperfect information and with humans who need to be educated and trained, would be some positive value—at least until we swing past the labor market saturation point into a surplus labor situation.

What would this do to the numbers in Figure 5? If the component parts of the product get bigger, so will the resulting to-do list. Does that look like things are getting tougher? Probably, which certainly makes the case for automation at some level.

So we are left with the original Workload Index and two transformed measures—the normalized Workload Index per worker and an economy-wide to-do list—but which of these is "right"? Which aids and which distorts our understanding of the level of technical risk in cybersecurity? There is more to be done on questions of measurement and cybersecurity [7], but the three measures illuminate three different things and are useful for different purposes.

First, the Workload Index works. It is consistent in how it measures vulnerabilities, providing a replicable time-series measure of the technical problems that plague our systems. The almost u-shaped structure of the Workload Index between 2006 and 2016 suggests, tentatively at least, that technical vulnerabilities might be a bit cyclical. That is useful information to have; firms and the economy can adjust accordingly.

Second, the normalized measure shows that with an expanding IT workforce, the total technical work per worker in the system is not too much worse than it was before. These numbers suggest calm in the face of sensational data breaches that affect millions (or possibly billions). The average network size that people can access once they have breached a system is probably getting bigger, but this measure suggests that keeping any particular part of the system secure on a technical front is not yet a mounting task.

Third, the economy-wide to-do list shows how an increasing worker count and a relatively constant Workload Index can generate a lot of work overall. These numbers suggest that things are getting worse, because the economy is exerting so much effort to keep things afloat. A real trouble here is that more work can mean more room for error, especially if humans remain at the forefront. Additionally, opportunity costs are real. Every hour a worker spends keeping the network safe is an hour which that person could have spent doing something else, something productive rather than protective. At a certain point, the economy-wide to-do list will get too big, the wasted hours will grow too large, and we will have to move towards more automation to keep the networks working and our workers free to do other things.

References

[1] NIST Vulnerability Workload Calculator: nvd.nist.gov/CVSS/v3-calculator.

[2] NIST Vulnerability Workload Index: nvd.nist.gov/Home/Workload-Index.cfm.

[3] Nassim Taleb, *Incerto* tetralogy: *Fooled by Randomness, The Black Swan, The Bed of Procrustes, Antifragile.*

[4] Cybersecurity and higher education: digitalguardian.com/blog/cybersecurity-higher-education-top-cybersecurity-colleges-and-degrees.

[5] Index of Cyber Security: www.cybersecurityindex.org.

[6] US Bureau of Labor Statistics: www.bls.gov/cps/tables.htm#charemp.

[7] E. Jardine, "Garbage In, Garbage Out: Measuring the Effectiveness of Remedial Cybersecurity Policies," working paper.

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Writing for *;login:*

We are looking for people with personal experience and expertise who want to share their knowledge by writing. USENIX supports many conferences and workshops, and articles about topics related to any of these subject areas (system administration, programming, SRE, file systems, storage, networking, distributed systems, operating systems, and security) are welcome. We will also publish opinion articles that are relevant to the computer sciences research community, as well as the system adminstrator and SRE communities.

Writing is not easy for most of us. Having your writing rejected, for any reason, is no fun at all. The way to get your articles published in *;login:*, with the least effort on your part and on the part of the staff of *;login:*, is to submit a proposal to login@usenix.org.

PROPOSALS

In the world of publishing, writing a proposal is nothing new. If you plan on writing a book, you need to write one chapter, a proposed table of contents, and the proposal itself and send the package to a book publisher. Writing the entire book first is asking for rejection, unless you are a well-known, popular writer.

;login: proposals are not like paper submission abstracts. We are not asking you to write a draft of the article as the proposal, but instead to describe the article you wish to write. There are some elements that you will want to include in any proposal:

- What's the topic of the article?
- What type of article is it (case study, tutorial, editorial, article based on published paper, etc.)?
- Who is the intended audience (syadmins, programmers, security wonks, network admins, etc.)?
- Why does this article need to be read?
- What, if any, non-text elements (illustrations, code, diagrams, etc.) will be included?
- What is the approximate length of the article?

Start out by answering each of those six questions. In answering the question about length, the limit for articles is about 3,000 words, and we avoid publishing articles longer than six pages. We suggest that you try to keep your article between two and five pages, as this matches the attention span of many people.

The answer to the question about why the article needs to be read is the place to wax enthusiastic. We do not want marketing, but your most eloquent explanation of why this article is important to the readership of *;login:*, which is also the membership of USENIX.

UNACCEPTABLE ARTICLES

;login: will not publish certain articles. These include but are not limited to:

- Previously published articles. A piece that has appeared on your own Web server but has not been posted to USENET or slashdot is not considered to have been published.
- Marketing pieces of any type. We don't accept articles about products. "Marketing" does not include being enthusiastic about a new tool or software that you can download for free, and you are encouraged to write case studies of hardware or software that you helped install and configure, as long as you are not affiliated with or paid by the company you are writing about.
- Personal attacks

FORMAT

The initial reading of your article will be done by people using UNIX systems. Later phases involve Macs, but please send us text/plain formatted documents for the proposal. Send proposals to login@usenix.org.

The final version can be text/plain, text/html, text/markdown, LaTex, or Microsoft Word/Libre Office. Illustrations should be EPS if possible. Vector formats (TIFF, PNG, or JPG) are also acceptable, and should be a minimum of 1,200 pixels wide.

DEADLINES

For our publishing deadlines, including the time you can expect to be asked to read proofs of your article, see the online schedule at www.usenix.org/publications/login/publication_schedule.

COPYRIGHT

You own the copyright to your work and grant USENIX first publication rights. USENIX owns the copyright on the collection that is each issue of *;login:*. You have control over who may reprint your text; financial negotiations are a private matter between you and any reprinter.

/dev/random

ROBERT G. FERRELL

Robert G. Ferrell is an award winning author of humor, fantasy, and science fiction, most recently *The Tol Chronicles* (www.thetolchronicles.com).

rgferrell@gmail.com

By the time you read this the shock of Election 2016 will hopefully have worn off to some extent. I don't customarily engage in political commentary, even in my life outside *;login:*, as people plummet into ad hominem-laden irrationality at blinding speed in such "discussions." Irrational arguments make my toes itch, and nothing interferes with developing a devastatingly clever comeback like having to take your shoes off to claw at your metatarsal digits.

I only broach the subject because of a single arresting quote, "the information trumps all," made in the course of a discussion of whether or not to publicize alleged state-sponsored hacking in connection with the US election. (Yes, I noticed the apropos transitive verb. I don't know whether or not it was intentional.) This ends-justifies-the-means paradigm is, of course, hardly a new concept. WikiLeaks is founded on it. Since we are in the information technology business here, it has particular relevance to our pursuits.

At first glance revealing the bare-bones truth about everything might seem a noble undertaking. I mean, who can make sound decisions in a factual vacuum, right? But I would argue that from a social, and at times even a technical, perspective having too much truth is as damaging as having none at all. We each build up comfortable mythologies surrounding the validity of our cherished institutions and the moral underpinning of our vaunted heroes. When the blunt truth is laid before us—that our institutions have inherent flaws and our heroes are subject to human foibles—those mythological foundations crumble and we are left with nothing much to admire or trust. Is this bald veracity an improvement? Not for me.

I am getting to the "relevant" part. Trust me. I just saw *Rogue One* and my brain hasn't yet made the long journey back from a galaxy far, far away.

We, and by that I mostly mean "some of you," have spent a great deal of time, effort, and coffee creating a wide variety of software and hardware tools designed to reveal to us what's really going on in our systems and networks. We as systems managers have an insatiable desire for the real scoop; the bottom line; the raw data; the dank underbelly; the misapplied metaphor. We tell ourselves we need to know precisely how our systems are performing, and why that's the case. But is this really true? Moreover, is conveying that information intact really the best course of action?

In some cases, I suppose a brutal reckoning is necessary, but I would argue that most of the time an approximation erring on the side of optimism might be better suited to the workplace. Submitted for your consideration: you're running low on disk space. You have two utilities for analyzing this. One shows the average disk usage per node, the other a more granular absolute user-by-user value. The first tool indicates that the average storage is approaching quota across the board and that (presuming no extraneous data is being kept) it's probably time to spring for more disks, or at least up the quota and have less reserve available. The other tool demonstrates quite clearly that the only users abusing the quotas are the boss and his two top assistants. Everyone else is way below the max, but those three users are blowing out the average egregiously.

You as the sysadmin need to deal with this problem. Which tool's results are you going to present to your documentation-crazy boss in support of your solution? Too much information might lead to hard feelings at best and unemployment at worst. Sustaining your rosy outlook concerning the practices and motivations of your coworkers has clear advantages here. There are myriad other instances where this is true.

Once upon a time there was a systems manager named Joan who was well-loved by all of her users. She had been with the company for many years and knew everyone's birthdays, their children and spouses, and each of their birthdays, too. She went to all of their parties and social functions. She had them over for cake, tea, and Canasta. She almost never missed a day of work. She kept the computers running most of the time.

One day, while this beloved sysadmin was out of the office for a week attending training, the IT staff member who'd been assigned to cover her duties was running routine network monitoring operations, looking for choke points. She pulled up the system log aggregator and noticed that the status panel indicating critical patch installations was showing red. She decided to investigate further.

The patch management system log showed that all recommended operating system patches had been faithfully installed enterprise-wide until three months ago, when they'd abruptly ceased. The weekly reports to senior management, however, failed to reflect that. This particular company had zero tolerance for risk. When executive management read the report filed by the IT team member, they summarily dismissed Joan for professional negligence that seemingly placed the entire IT landscape in danger by not installing recommended patches.

Her replacement was ordered to install all the missed patches immediately. The older switches and firewalls were incompatible with those patches, as Joan had tried in vain to explain before her dismissal, and this left the network wide open to a variety of malware as a result. Inept attempts to combat a massive distributed denial-of-service attack launched by an unscrupulous business rival further eroded the once-solid information security barriers surrounding the network, and repeated ransomware demands stemming from spearphishing operations eventually bankrupted the firm entirely.

Notice the consistent negative correlation between full disclosure and longevity of employment in the preceding examples? The truth really will set you free.

Okay, do I really expect to draw a direct, meaningful comparison between journalists who sit on a scoop for fear they might unwittingly be doing some foreign potentate's bidding and a sysadmin hiding the fact that software patches haven't been installed in a timely fashion in order to protect her network from incompatibility issues? You bet I do. In this post-rational world linear arguments based on logic and deductive reasoning are, like, so passé.

I desperately need to scratch my toes now.

Book Reviews

MARK LAMOURINE AND RIK FARROW

Learning Angular 2

Pablo Deeleman
Packt Publishing 2016, 326 pages
ISBN 978-1-78588-207-4

Reviewed by Mark Lamourine

If there's one thing I've discovered from my attempt to learn client-side Web programming, and Angular 2 in particular, it's that I'm glad I'm not a Web programmer. Creating a Web app these days, even with frameworks to standardize many of the constructs and behaviors, requires the use of at least four languages (I count JavaScript, HTML, CSS, and templating as distinct languages). In many cases, one or more of these are interlaced in a single file. Web design frameworks take some of the burden by providing a well-defined set of tools for the developer. Angular is Google's attempt to create a JavaScript framework to assist in and standardize the creation of single-page client-side Web applications.

Learning Angular 2 doesn't really help my impression much. The book is based on RC1 of Angular 2 and was published in May 2016. Angular 2 went to first release (2.0) in September, after six more release candidates. While Google is promising increment-only releases (using "semantic versioning"), this doesn't give me warm fuzzies to start. I was less happy when, on downloading the sample code from GitHub and trying to follow the installation process, I found that the required libraries were already advanced and out of sync with each other. An experienced JavaScript coder and NPM user would have solved this in moments, but it took me an hour or so merely to get to where I could start the actual samples in the book.

Once I got past this, the text runs in a fairly typical way. In Chapter 2, Deeleman introduces TypeScript, a superset of ECMAScript 6. The language itself is very straightforward. It should be safe as it is managed by Microsoft and is the source language that Google selected for Angular 2 itself. But again, I'm a bit disconcerted by the explanation for the existence of TypeScript, which is essentially that none of the standards bodies could agree on what client-side scripting should look like, so Microsoft took it on themselves to decide. It may be a good thing, and it's not Deeleman's fault in any case, but it doesn't instill confidence in a new learner.

Deeleman's "hello world" example is a Pomodoro timer. He explains that this is a kind of work-tracking device to help break down tasks. He guides the reader through the creation of a simple app in Chapter 3, and the rest of the book extends the application with new features. I like how he presents his code samples, offering a complete file or feature first, then breaking down the parts and explaining how they interact or relate. I prefer this to a style that presents small, digestible but apparently unrelated fragments and then composes them at the end.

There are places where the narrative gets lost, though. Deeleman states that he expects the reader to have a comprehensive understanding of JavaScript, but it feels at times as if he's presenting incomplete or circular definitions for terms: "Angular 2 defines directives as components without views. In fact, a component is a directive with a view." But there's very little time spent on what a component is and why that is significant.

Deeleman manages to cover the major points and features of Angular 2: component design, composition, standard directives (logic for producing and laying out the custom content that components present), HTML templating language, and client-server communications. He doesn't go deep into the theory or philosophy behind how and why these elements work together the way they do. His approach is mechanical, but it is effective on that level.

Packt tends to publish early books, and it seems sometimes that they spend less effort on editorial work than some of the more prestigious imprints. In an environment where frameworks like Angular can come and go in a publishing cycle, this makes some sense. They provide for a market of readers eager to learn new things that more conservative publishers might pass over or miss completely. Having lamented the thin supply of books on Go, Docker, and Kubernetes, I appreciate what they do. Readers should be aware, though, of what they are getting.

If you're an experience client-side Web developer looking for a self-tutorial on Angular 2 to supplement the documentation already on line, *Learning Angular 2* will serve. Anyone hoping to learn to design Web apps from scratch will have to work harder to grasp the context and operations that Deeleman leaves out.

Mastering Angular 2 Components

Gion Kunz
Packt Publishing, 2016, 352 pages
ISBN 978-1-78588-464-1

Reviewed by Mark Lamourine

My first experience learning Angular 2 was a challenge at least in part because of my own inexperience with client-side Web development, but I didn't want to stop with a single try. My reading of *Mastering Angular 2 Components* gets the benefit of that experience.

This book is also based on Angular 2 RC1 and was released in June 2016, so the same risks apply regarding bit rot as applied with *Learning Angular 2*, but I didn't have any problems preparing the working environment this time.

Kunz begins by introducing terminology and tooling, and he spends significant time both defining terms and explaining why they matter and how they relate. While I understand classes and decorators from other languages, I appreciated the paragraph or two he gave to each, explaining how they are defined and used in TypeScript and how this relates to ECMAScript and JavaScript standards. I'm still not comforted much by the state of language development for Web programming, but at least I now better understand the technical aspects of the decisions.

Kunz alternates well between developer and application user realms, which clarifies the reasoning and the choices that the Angular 2 developers made when designing the framework. He has peppered the text with diagrams to help clarify the relationships between components and directives and how these are related to views and templates. Chapter 7, "Components for the User Experience," makes clear who we are actually writing our apps for. The composition of complete services, both the presentation and logic, seems natural and meshes well with Kunz's exposition of the language and framework features that Angular 2 provides to the developer.

I especially liked the section which treats CSS and how the CSS elements are bound back to the HTML to influence the visual presentation. It is easy for a coder to treat visual presentation as subsidiary to data structures and logic (I am guilty). Kunz spends time showing how the design of the templates and data bindings in Angular 2 components can be influenced by the intended presentation and why it is important to consider the presentation hooks during development of the components.

I also appreciated his clear treatment of how data, both input and output, is bound to HTML template elements. It is an aspect of Web programming that had confounded me for some time. He shows how to create structures to present data both as text and graphically, using both CSS and SVG to create dynamic visual elements: graphs, charts, sliders, and interactive controls.

The final significant topic that piqued my interest is a section on the interactions between client and server, including timing and response mechanisms. While others have described the syntax necessary to create and respond to triggers and data exchange, Kunz is the first I have seen to clearly diagram the sequence of real-time events and communications that result from these coded elements.

As an experienced developer in other realms, I was comfortable working through each step of the learning process as presented by *Mastering Angular 2 Components*. I think this is one I'm going to come back to as I work on my own first Web service.

The Practice of System and Network Administration, Volume 1, 3rd Edition

Tom Limoncelli, Christine Hogan, Strata R. Chalup
Addison-Wesley, 2016, 1168 pages
ISBN 978-0-321-91916-8

Reviewed by Mark Lamourine

It's been a decade since the release of the second edition of *The Practice of System and Network Administration*. In that time, the character of system administration has changed and expanded in ways few of us anticipated. Each edition has been a comprehensive survey of the aspects of system administration in its era. Since the release of the second edition, configuration management has become commonplace, virtualization has moved from the desktop to the datacenter and the cloud, software development has accepted the tenets of Agile processes, and software revision control has become a public service. Containers have been rediscovered, though I don't think we can see yet what the results will be. (There is a Volume 2 which deals specifically with cloud administration. This is not that book.)

From the first, *TPoSaNA* (I generally avoid acronyms and abbreviations, but I make an exception for a title this long) has been an encyclopedia of the profession. It is a welcome anomaly in the sea of technical tutorials and references. You're not going to learn to be a system administrator by reading it, but you can become a better one by scanning it and then keeping it handy for those times when you're not sure what to think or do. More than once I have pulled it out to show to a colleague or manager when I have needed an authority to back me up in some point of discussion, and it has proved very useful in educating managers in the scope of the work their people are expected to do.

The updates start with the table of contents. As an encyclopedia, it isn't surprising that this book has as many sections as most books have chapters (11). Thirty-two of the 56 chapters are new or updated (indicated by a marker on the title line). I liked the fact that I could thumb through and so easily find the places I needed to reread.

In this edition, the authors have dropped their 1st edition conceit of offering "The Basics," "The Standard," and "The Icing" levels of support for each topic. They still open with a clear discussion of the topic scope and goals, but then use a more conventional approach to the detailed discussion. Often they justify or illustrate their choices with anecdotes from their own work, showing how the problems arise in the real world and how they responded. Every chapter is peppered with references to other resources and ends with a set of exercises, which are really prompts for readers to think about what they've just read in the context of their own work environments. This works well to help readers relate the new ideas to their own work.

Most people would expect a book on system administration to include operating system and software installation and configuration management. Many would expect to see guidelines for help-desk management. I think many (non-sysadmin) people would be surprised to see power and air-conditioning management under the umbrella of system administration. I know I have welcomed the chapters on how to manage time, not just to do the job well but to remain sane and happy. I haven't needed the chapters on hiring and firing, but I know technical people who have grown to lead or manage groups of developers or admins and appreciated it. In a technical field, it's just not something you have to think about…until you do.

For people who call themselves system administrators, I can't recommend having a handy copy of this third edition of *TPoSaNA* highly enough. For managers of system administrators, I recommend it even more highly. If you've been in the profession for long, there's likely a lot here you already have heard, but this book is the perfect starting resource for those times when you or your colleagues find yourself having to extend yourselves to Get the Job Done.

The Hardware Hacker: Adventures in Making and Breaking Hardware

Andrew "bunnie" Huang
No Starch Press, 2017, 416 pages
ISBN: 978-1-59327-758-1

Reviewed by Rik Farrow

Based on the title and subtitle of this book, I thought Huang was writing about, well, hardware hacking. But a lot of the book is about manufacturing hardware in China for small production runs. It wasn't long after I realized that the title poorly describes the content that I got over my disappointment, however.

bunnie Huang is not just a brilliant hardware designer, he's a great writer, too. His style is conversational, clear, and concise, and I found myself wishing that more people could write like Huang. He explains that he started visiting factories in China to support the manufacturing of the Chumby, a dedicated MP3 player with touch screen he designed in the early noughts. Huang describes just how fascinating he found the mega-bazaars of Shenzhen, China. But his real focus is the factories, how important it is to find the right factory and to communicate clearly what you want them to do.

Huang uses a bill-of-materials for a bicycle safety light as an example. While the device just requires eight parts, getting it manufactured correctly requires an entire page full of detailed information. Almost as an aside, I learned what RoHS means (Restriction of Hazardous Substances) and how one region's standard can mean safer products for everyone.

Huang's writing does tend to wander, but always in directions that I found fascinating. For example, he gets to visit a factory that makes zippers, starting with zinc/aluminum ingots. He asks the same question I might have about why one processing line required a human to align zipper pulls while other lines did not. The answer is subtle, a tiny tab found on most zippers. Yet this diversion helps to illustrate an important point about the manufacturing process: that what might be important to a designer, lack of the tiny tabs, causes problems for manufacturing.

Huang uses several of the projects he has worked on to illustrate the problems that a hardware hacker, intent on actually going on to produce product runs in the low thousands, will encounter. One issue is fake parts, parts that appear authentic but are actually just the casing with no electronics inside. Other issues appear truly ridiculous but are no less real. When Huang and a partner required a spiral notebook where the spiral had to be non-conducting for Chibitronics, the manufacturer didn't understand what that meant. Huang bought a pair of volt-ohm meters and taught the manufacturer how to use them to test the spiral bindings for conductance.

The chapter on fake parts, something that can be a problem when manufacturing in China, is the closest Huang comes to hardware hacking in my mind. Huang provides photos of SD cards, then has them stripped down to the chips hidden inside the epoxy resin. He does talk about some of his other hardware designs: for example, the hacker's laptop he co-designed and manufactured, but manufacturing is still the largest theme.

Toward the end of the book (Chapter 10), Huang veered off into an area I felt he couldn't possible handle: biology and bioinformatics. Huang compares a metabolic diagram to an Apple II schematic, then imagines DNA and RNA as configuration bits. I was wrong about this chapter, as Huang manages his comparisons brilliantly, using hardware as a way to explain genes, proteins, and amino acids. He goes on to describe the flu virus and how it manages to continue to evade vaccine designers using just 3.2 KB of "data" in its genome.

I found Huang's book both easy and fun to read. If you are curious about the manufacturing culture in China, including its own version of "open source," I recommend reading this book.

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, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks and operating systems, and book reviews

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

Discounts on registration fees for all USENIX conferences

Special discounts on a variety of products, books, software, and periodicals: www.usenix.org/member-services/discount-instructions

The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers

For more information regarding membership or benefits, please see www.usenix.org/membership/or contact office@usenix.org. Phone: 510-528-8649.

Notice of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held on Thursday, July 13, in Santa Clara, CA, during the 2017 USENIX Annual Technical Conference.

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

David N. Blank-Edelman, *Apcera*
dnb@usenix.org

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

Daniel V. Klein, *Google*
dan.klein@usenix.org

EXECUTIVE DIRECTOR
Casey Henderson
casey@usenix.org

USENIX Awards

USENIX honors members of the community with two prestigious awards which recognize public service and technical excellence:

- **The USENIX Lifetime Achievement (Flame) Award**
- **The LISA Award for Outstanding Achievement in System Administration**

The winners of these awards are selected by the USENIX Awards Committee. The USENIX membership may submit nominations for either or both of the awards to the committee.

Call for Award Nominations

USENIX requests nominations for these two awards; they may be from any member of the community. Nominations should be sent to the Chair of the Awards Committee via awards@usenix.org at any time. A nomination should include:

- Name and contact information of the person making the nomination
- Name(s) and contact information of the nominee(s)
- A citation, approximately 100 words long
- A statement, at most one page long, on why the candidate(s) should receive the award
- Between two and four supporting letters, no longer than one page each

The USENIX Lifetime Achievement (Flame) Award

The USENIX Lifetime Achievement Award recognizes and celebrates singular contributions to the USENIX community in both intellectual achievement and service that are not recognized in any other forum. The award itself is in the form of an original glass sculpture called "The Flame," and in the case of a team based at a single place, a plaque for the team office.

Details and past recipients can be found at www.usenix.org/about/flame.

The LISA Award for Outstanding Achievement in System Administration

This award goes to someone whose professional contributions to the system administration community over a number of years merit special recognition.

Details and past recipients can be found at www.usenix.org/lisa/awards/outstanding.

WOOT '17: 11th USENIX Workshop on Offensive Technologies

August 14–15, 2017 • Vancouver, BC, Canada

Sponsored by USENIX, the Advanced Computing Systems Association

WOOT '17 will be co-located with the 26th USENIX Security Symposium (USENIX Security '17) and take place August 14–15, 2017.

Important Dates

- Paper submissions due: **Wednesday, May 31, 2017, 8:59 p.m. PDT**
- Notification to authors: **Tuesday, June 27, 2017**
- Final papers files due: **Tuesday, July 25, 2017**

Workshop Organizers

Program Co-Chairs

William Enck, *North Carolina State University*
Collin Mulliner, *Square Inc.*

Program Committee

Lorenzo Cavallaro, *Royal Holloway University of London*
Sandy Clark, *University of Pennsylvania*
Erinn Clark, *FirstLook*
Scott Coull, *FireEye*
Lucas Davi, *University of Duisburg-Essen*
Razvan Deaconescu, *University POLITEHNICA of Bucharest*
Manuel Egele, *Boston University*
Mario Heiderich, *Cure53*
Alexandros Kapravelos, *North Carolina State University*
Zach Lanier, *Cylance*
Per Larsen, *University of California, Irvine, and Immunant*
Tarjei Mandt, *Azimuth Security*
Charlie Miller, *Uber ATC*
Adwait Nadkarni, *North Carolina State University*
Ben Nell
Christin Pöpper, *New York University*
Kapil Singh, *IBM T. J. Watson Research Center*
Julien Vanegue, *Bloomberg LP and Cornell University*
Ralf-Philipp Weinmann, *Comsecuris*
Georg Wicherski, *CrowdStrike*
Glenn Wurster, *BlackBerry*
Yves Younan, *Cisco Talos*

Overview

The USENIX Workshop on Offensive Technologies (WOOT) aims to present a broad picture of offense and its contributions, bringing together researchers and practitioners in all areas of computer security. Offensive security has changed from a hobby to an industry. No longer an exercise for isolated enthusiasts, offensive security is today a large-scale operation managed by organized, capitalized actors. Meanwhile, the landscape has shifted: software used by millions is built by startups less than a year old, delivered on mobile phones and surveilled by national signals intelligence agencies.

In the field's infancy, offensive security research was conducted separately by industry, independent hackers, or in academia. Collaboration between these groups could be difficult. Since 2007, the USENIX Workshop on Offensive Technologies (WOOT) has aimed to bring those communities together.

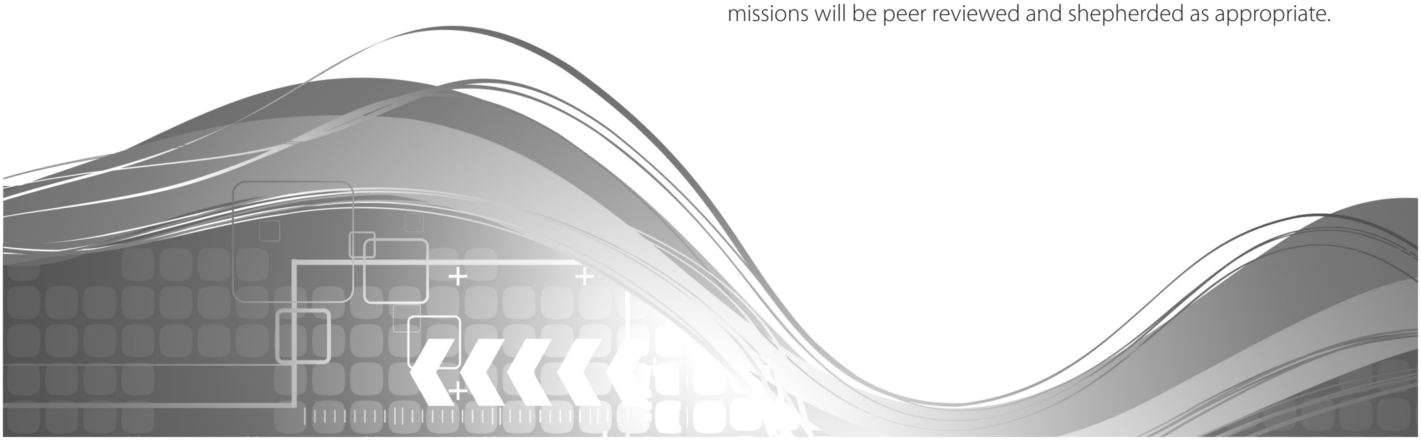
WOOT '17 will feature a Best Paper Award and a Best Student Paper Award.

Symposium Topics

Computer security exposes the differences between the actual mechanisms of everyday trusted technologies and their models used by developers, architects, academic researchers, owners, operators, and end users. While being inherently focused on practice, security also poses questions such as "what kind of computations trusted systems are and aren't capable of?," which harken back to fundamentals of computability. State-of-the-art offense explores these questions pragmatically, gathering material for generalizations that lead to better models and more trustworthy systems.

WOOT provides a forum for high-quality, peer-reviewed work discussing tools and techniques for attack. Submissions should reflect the state of the art in offensive computer security technology, exposing poorly understood mechanisms, presenting novel attacks, or surveying the state of offensive operations at scale.

WOOT '17 accepts papers in both an academic security context and more applied work that informs the field about the state of security practice in offensive techniques. The goal for these submissions is to produce published works that will guide future work in the field. Submissions will be peer reviewed and shepherded as appropriate.



Submission topics include but are not limited to:

- Vulnerability research
- Offensive applications of formal methods (solvers, symbolic execution)
- Practical attacks on deployed cryptographic systems and kleptography
- Offensive aspects of mobile security (including location, payments, and RF)
- Attacks on content protection and DRM
- Hardware attacks and attacks on the "Internet of Things"
- Internet-scale network reconnaissance
- Application security (web frameworks, distributed databases, multi-factor authentication)
- Malware design, implementation and analysis
- Vulnerabilities in browser and client-side security (runtimes, JITs, sandboxing)
- Mass surveillance and attacks against privacy

Workshop Format

The presenters will be authors of accepted papers. There will also be a keynote speaker and a selection of invited speakers.

Regular Submission

WOOT '17 welcomes submissions without restrictions of formatting (see below) or origin. Submissions from academia, independent researchers, students, hackers, and industry are welcome. Did you just give a cool talk in the hot Miami sun at Infiltrate? Got something interesting planned for Black Hat later this year? This is exactly the type of work we'd like to see at WOOT '17. Please submit—it will also give you a chance to have your work reviewed and to receive suggestions and comments from some of the best researchers in the world. More formal academic offensive security papers are also very welcome.

Systemization of Knowledge

Continuing the tradition of past years, WOOT '17 will be accepting "Systematization of Knowledge" (SoK) papers. The goal of an SoK paper is to encourage work that evaluates, systematizes, and contextualizes existing knowledge. These papers will prove highly valuable to our community but would not be accepted as refereed papers because they lack novel research contributions. Suitable papers include survey papers that provide useful perspectives on major research areas, papers that support or challenge long-held beliefs with compelling evidence, or papers that provide an extensive and realistic evaluation of competing approaches to solving specific problems. Be sure to select "Systematization of Knowledge paper" in the submissions system to distinguish it from other paper submissions.

All accepted papers will be available online to registered attendees prior to the workshop and will be available online to everyone beginning on the first day of the workshop, August 14, 2017. If your paper should not be published prior to the event, please notify production@usenix.org.

Submission

Papers must be received by 8:59 p.m. PDT on Wednesday, May 31, 2017.

What to Submit

Submissions should be in PDF format. Apart from this, there is no mandatory formatting requirement. Even though the submission format is open, the program committee will have to evaluate the submissions, and the guidelines below will help the program committee to evaluate the quality and originality of the submission.

Papers should be succinct but thorough in presenting the work. The contribution needs to be well motivated, clearly exposed, and compared to the state of the art. Typical research papers are 4–10 pages long (not counting bibliography and appendix). Shorter, more focused papers are encouraged and will be reviewed like any other paper. Papers whose lengths are incommensurate with their contributions will be rejected. The submission should be formatted in 2 columns, using 10-point Times Roman type on 12-point leading, in a text block of 6.5" by 9". Please number the pages. If possible, use the USENIX Templates for Conference Papers at https://www.usenix.org/conferences/author-resources/paper-templates when preparing your paper for submission.

Authors of accepted papers will have to provide a paper for the proceedings following the above guidelines. A shepherd may be assigned to ensure the quality of the proceedings version of the paper (but not to write the paper for the author).

All submissions will be electronic and must be in PDF. Submissions are single-blind; author names and affiliations should appear on the title page. Submit papers using the Web form on the WOOT '17 Web site, www.usenix.org/woot17/cfp.

Submissions accompanied by non-disclosure agreement forms will not be considered. Accepted submissions will be treated as confidential prior to publication on the WOOT '17 Web site; rejected submissions will be permanently treated as confidential.

Policies and Contact Information

Simultaneous submission of the same work to multiple competing venues, submission of previously published work without substantial novel contributions, or plagiarism constitutes dishonesty or fraud. USENIX, like other scientific and technical conferences and journals, prohibits these practices and may take action against authors who have committed them. See the USENIX Conference Submissions Policy at https://www.usenix.org/conferences/author-resources/submissions-policy for details.

Note: Work presented at industry conferences, such as Black Hat, is not considered to have been "previously published" for the purposes of WOOT '17. We strongly encourage the submission of such work to WOOT '17, particularly work that is well suited to a more formal and complete treatment in a published, peer-reviewed setting. In your submission, please do note any previous presentations of the work.

Authors uncertain whether their submission meets USENIX's guidelines should contact the program co-chairs, woot17chairs@usenix.org, or the USENIX office, submissionspolicy@usenix.org.

Registration for Authors

One author per paper will receive a discount on registration. If the registration fee poses a significant hardship for the presenting author, contact conference@usenix.org.

Rev. 2/7/17

Announcement and Call for Papers www.usenix.org/ase17/cfp

ASE ’17: 2017 USENIX Workshop on Advances in Security Education

August 15, 2017 • Vancouver, BC, Canada

Sponsored by USENIX, the Advanced Computing Systems Association

ASE ’17 will be co-located with the 26th USENIX Security Symposium (USENIX Security ’17) and take place August 15, 2017.

Important Dates

- Paper submissions due (full and short papers): **Tuesday, May 9, 2017, 8:59 pm PDT (no extensions)**
- Notification to paper authors: **Thursday, June 8, 2017**
- Lightning Talk abstracts due: **Wednesday, June 28, 2017**
- Notification about Lightning Talks: **Wednesday, July 5, 2017**
- Final paper files due: **Thursday, July 6, 2017**

Workshop Organizers

Program Co-Chairs

Mark Gondree, *Sonoma State University*
Ashley Podhradsky, *Dakota State University*

Program Committee

Adam Aviv, *US Naval Academy*
Rakesh Bobba, *Oregon State University*
Tom Chothia, *University of Birmingham*
Kevin Du, Syracuse University
Márk Félegyházi, *Budapest University of Technology and Economics CrySyS Lab*
Wai Yi Feng, *University of Cambridge*
Wu-Chang Feng, *Portland State University*
Nathan Fisk, *University of South Florida*
Andreas Haggman, *Royal Holloway University of London*
Michael Hicks, *University of Maryland*
Cynthia Irvine, *Naval Postgraduate School*
Colleen Lewis, *Harvey Mudd College*
Jelena Mirkovic, *University of Southern California Information Sciences Institute*
Zachary N J Peterson, *Cal Poly, San Luis Obispo*
Portia Pusey, *CyberSecurity Competition Federation*
Z. Cliffe Schreuders, *Leeds Beckett University*
Ambareen Siraj, *Tennessee Tech University*
Richard Weiss, *The Evergreen State College*

Steering Committee

Adam Aviv, *US Naval Academy*
Matt Bishop, *University of California, Davis*
Mark Gondree, *Sonoma State University*
Zachary N J Peterson, *Cal Poly, San Luis Obispo*
Giovanni Vigna, *University of California, Santa Barbara*

Overview

The 2017 USENIX Workshop on Advances in Security Education (ASE ’17) is co-located with the 26th USENIX Security Symposium and is intended to be a venue for cutting-edge research, best practices, and experimental curricula in computer security education.

The workshop welcomes a broad range of paper submissions on the subject of computer security education in any setting (K-12, undergraduate, graduate, non-traditional students, professional development, and the general public) with a diversity of goals, including developing or maturing specific knowledge, skills and abilities (KSAs), or improving awareness of issues in the cyber domain (e.g., cyber literacy, online citizenship). ASE is intended to be a venue for educators, designers, and evaluators to collaborate, share knowledge, improve existing practices, critically review state of the art, and validate or refute widely held beliefs.

ASE is the evolution of the USENIX Summit on Gaming, Games, and Gamification (3GSE), expanded to welcome a wider range of contributions to security education research. The broad workshop scope is intended to attract those already working in this space within the traditional USENIX Security community, as well as those from other communities, including education researchers, social scientists, and practitioners. The workshop attempts to represent, through invited talks, paper presentations, panels, and tutorials, a variety of approaches and issues related to security education.

Format

ASE is intended to be a venue for informal collaboration and community-building. The current program includes:

- A keynote address
- Sessions for full papers; authors accompany these with presentations at the workshop, with time for follow-up discussion
- Sessions for short papers; authors accompany these with “live lessons” at the workshop, demonstrating a successful or innovative lesson, activity, exercise, or tool
- A session for Lightning Talks and community announcements
- A panel discussion exploring popular and/or controversial issues in security education

All sessions are intended to stimulate group discussion and impact future work. We encourage attendees to participate in Lightning Talks, where they can bring attention to new results, distribute materials, or make announcements of interest to the education community (new events, projects, funding opportunities, venues, etc.).

Topics

The core mission of ASE is to disseminate cutting-edge, practitioner-oriented, computer security education research. Specific topics of interest include, but are not limited to:

- Novel pedagogical approaches and experimental curricula
- Outreach and mentorship of groups underrepresented in security
- Education technology research in a security education context
- Tools and techniques for measurement, evaluation, and assessment
- Frameworks and infrastructures supporting education
- Experiences with standards, certifications, and accreditation
- Security games and competitions
- Extramural and extracurricular education programs
- Experience with alternative teaching modalities for computer security, including MOOCs, flipped classrooms, peer-instruction and inquiry-based instruction, and distance learning
- Security education geared toward non-technical audiences

Full Papers

Full paper submissions should be no more than eight pages long (excluding references). Full papers are expected to follow style and format of a traditional academic format, featuring an abstract, introduction, related work, conclusion and references. As a workshop paper, these may highlight early work, in-progress work, lessons-learned, position papers, or program summaries; however, full papers are intended do at least one of the following: highlight some technical solution of merit to the education community, feature some analysis or survey work of value to the education community, or employ some assessment based on community-accepted practices for the scholarship of teaching and learning.

Each full paper will be accompanied by a presentation delivered at the workshop by one of the paper's authors (approximately 15–20 minutes in duration).

Short Papers

We are excited to provide educators with a venue to share an exercise, problem set, activity or tool with the workshop. Short papers supplement these presentations and may take the form of extended abstracts, stand-alone lesson plans (e.g., featuring learning objectives and related materials to help educators reproduce the lesson) or technical descriptions to accompany a demo.

Short paper submissions should be between 2–6 pages, but no more than 6 pages long (including references). At a minimum, short papers should feature an abstract, introduction, and references, and the paper's introduction should contain a summary of what the "live lesson" at the workshop will demonstrate. Beyond this, short papers should choose a form that complements their topic. For example, an in-class activity might provide a lesson plan, learning objectives, activity description, sample follow-on activities; a software demo might include a description of its capabilities and a short case study of its prior use. When appropriate, the paper is encouraged to reference external, supplemental, and/or multimedia resources. Short papers for lessons, in particular, may consider paralleling the format of SIGCSE Nifty presentations (http://nifty.stanford.edu/), i.e., letting ASE host all assignment materials and using the short paper as a brief summary/commentary on those. All supplemental materials should be submitted with the paper or otherwise be accessible to reviewers at the time of submission and throughout the review period.

Each short paper will be accompanied by a "live lesson" delivered at the workshop by one of the paper's authors (approximately 15–20 minutes in duration), but extra time may be afforded during breaks or after sessions for continued exploration. Potential "live lessons" include scaffolded exercises, abbreviated lessons, tool demonstrations, or classroom activities (engaging the workshop audience, either as students or fellow practitioners). They may include a short video of a classroom practice, a live demo of an instructional technique, an interactive exercise with the workshop attendees, a technology demonstration, etc.

Lightning Talks

Lightning Talks highlight fresh ideas, unique perspectives, valuable experiences, and emerging trends in computer security education. Short talks are five-minute presentations on work and ideas not ready or suitable for peer-reviewed publication but worth sharing to jump-start discussion among and solicit feedback from attendees.

Short talk presentations are five minutes in duration with an additional five minutes for discussion. If you would like to present a short talk at the event, please email a talk abstract to ase17talks@usenix.org. There are no length or content requirements for the short talk abstract, but a few sentences describing what you'd like to do or announce, informally, is appropriate.

Paper Submissions

Full paper submissions must be no more than eight pages long, excluding references. Short paper submissions should be no more than six pages long, including references.

For all submissions, text should be formatted in two columns on 8.5" x 11" paper using 10-point type on 12-point leading ("single-spaced"), with the text block being no more than 6.5" x 9" deep. Text outside the 6.5" x 9" block will be ignored. Submissions need not be anonymized. Submissions must be in PDF and must be submitted via the Web submission form on the ASE '17 Web site, www.usenix.org/ase17/cfp.

All accepted papers will be available online to registered attendees before the workshop. If your paper should not be published prior to the event, please notify production@usenix.org. The papers will be available online to everyone beginning on the day of the workshop. At least one author from every accepted paper must attend the workshop and present.

Simultaneous submission of the same work to multiple venues, submission of previously published work, or plagiarism constitutes dishonesty or fraud. USENIX, like other scientific and technical conferences and journals, prohibits these practices and may take action against authors who have committed them. See the USENIX Conference Submissions Policy at www.usenix.org/conferences/submissions-policy for details. Questions? Contact your program co-chairs, ase17chairs@usenix.org, or the USENIX office, submissionspolicy@usenix.org.

Papers accompanied by nondisclosure agreement forms will not be considered. Accepted submissions will be treated as confidential prior to publication on the USENIX ASE '17 Web site; rejected submissions will be permanently treated as confidential.

Rev. 2/7/17

Announcement and Call for Papers www.usenix.org/cset17/cfp

CSET '17: 10th USENIX Workshop on Cyber Security Experimentation and Test

August 14, 2016 • Vancouver, BC, Canada

Sponsored by USENIX, the Advanced Computing Systems Association

CSET '17 will be co-located with the 26th USENIX Security Symposium (USENIX Security '17) and take place August 14, 2017.

Important Dates

- Submissions due: **Tuesday, May 2, 2017, 11:59 p.m. PDT (no extensions)**
- Notification to authors: **Tuesday, June 13, 2017**
- Final papers due: **Tuesday, July 11, 2017**

Workshop Organizers

Program Co-Chairs

José M. Fernandez, *École Polytechnique de Montréal*
Mathias Payer, *Purdue University*

Program Committee

John Aycock, *University of Calgary*
Saurabh Bagchi, *Purdue University*
Kevin Borgolte, *University of California, Santa Barbara*
Sergey Bratus, *Dartmouth College*
Lucas Davi, *University of Duisburg-Essen*
Sven Dietrich, *CUNY John Jay College & The Graduate Center*
Brendan Dolan-Gavitt, *New York University*
Simon Edwards, *SE Labs*
Sonia Fahmy, *Purdue University*
Ryan Gerdes, *Virginia Tech University*
Fanny Lalonde-Lévesque, *École Polytechnique de Montréal*
Antoine Lemay, *École Polytechnique de Montréal*
Dave Levin, *University of Maryland*
Stefan Mangard, *TU Graz*
Jelena Mirkovic, *USC Information Sciences Institute (ISI)*
Cristina Nita-Rotaru, *Northeastern University*
Aravind Prakash, *Binghamton University*
Anil Somayaji, *Carleton University*
Peter Stelzhammer, *AV-Comparatives*
Gianluca Stringhini, *University College London*
Laura S. Tinnel, *SRI International*
Erik van der Kouwe, *Vrije Universiteit Amsterdam*
Chao Zhang, *Tsinghua University*

Steering Committee

Terry V. Benzel, *USC Information Sciences Institute (ISI)*
Sean Peisert, *University of California, Davis, and Lawrence Berkeley National Laboratory*
Stephen Schwab, *USC Information Sciences Institute (ISI)*

Overview

The CSET workshop invites submissions on cyber security evaluation, experimentation, measurement, metrics, data, simulations, and testbeds for software, hardware, or malware.

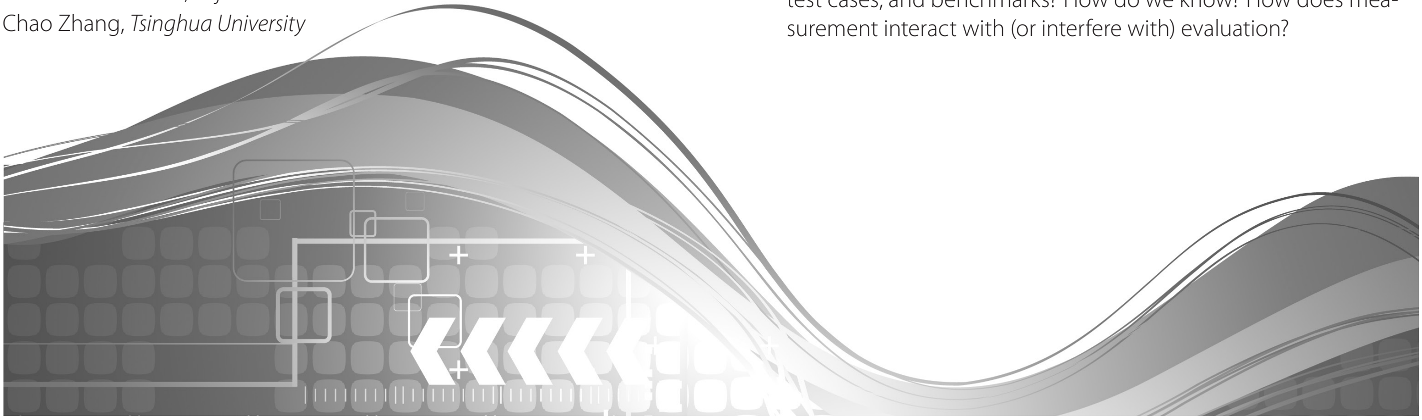
The science of cyber security poses significant challenges. For example, experiments must recreate relevant, realistic features in order to be meaningful, yet identifying those features and modeling them is very difficult. Repeatability and measurement accuracy are essential in any scientific experiment yet hard to achieve in practice. Few security-relevant datasets are publicly available for research use and little is understood about what "good datasets" look like. Finally, cyber security experiments and performance evaluations carry significant risks if not properly contained and controlled yet often require some degree of interaction with the larger world in order to be useful.

Addressing all these challenges is fundamental not only for scientific advancement in the field of Computer Security but also in order to enable evidence-based decision making on security products and policies by industry, government and individual users. Meeting these challenges requires transformational advances, including understanding the relationship between scientific method and cyber security evaluation, advancing capabilities of underlying experimental infrastructure, and improving data usability.

Topics

Topics of interest include but are not limited to:

- **Benchmarks for security:** e.g., development and evaluation of benchmark suites that evaluate certain security metrics
- **Research methods for cyber security experiments:** e.g., experiences with and discussions of experimental methodologies; experiment design and conduct addressing cybersecurity challenges for software, hardware, and malware
- **Measurement and metrics:** e.g., what are useful or valid metrics, test cases, and benchmarks? How do we know? How does measurement interact with (or interfere with) evaluation?



- **Data sets:** e.g., what makes good data sets? How do we know? How do we compare data sets? How do we collect new ones or generate derived ones? How do they hold up over time?
- **Security product evaluation methodologies:** e.g. what product evaluation methodologies provide more accurate prediction of real-world performance? How should user-related characteristics (behaviour, demographics) be modeled for in security product performance evaluation?
- **Simulations and emulations:** e.g., what makes good ones? How do they scale (up or down)?
- **Design and planning of cyber security studies:** e.g., hypothesis and research question, study design, data (collection, analysis, and interpretation), accuracy (validity, precision)
- **Ethics of cyber security research:** e.g., experiences balancing stakeholder considerations; frameworks for evaluating the ethics of cyber security experiments
- **Testbeds and experimental infrastructure:** e.g., tools for improving speed and fidelity of testbed configuration; sensors for robust data collection with minimal testbed artifacts; support for interconnected non-IT systems such as telecommunications or industrial control

Special note: Papers that primarily focus on computer security education are likely a better fit for the 2017 USENIX Workshop on Advances in Security Education (ASE '17), also co-located with the USENIX Security Symposium. Authors of education-centered papers should strongly consider submitting their work to ASE.

Workshop Format

Because of the complex and open nature of the subject matter, CSET '17 is designed to be a workshop in the traditional sense. Presentations are expected to be interactive, and presenters should ensure that sufficient time is reserved for questions and audience discussion. Audience participation is encouraged. To ensure a productive workshop environment, attendance will be limited to 80 participants.

Submission Instructions

Research papers and position papers are welcome as submissions. Research papers should have a clearly stated methodology including a hypothesis and experiments designed to prove or disprove the hypothesis. Position papers, particularly those that critique past work, should present detailed solutions, either proposed or implemented. Submissions that recount experiences (e.g., from experiments or deployments) are especially desired; these should highlight takeaways and lessons learned that might help researchers in the future. For all submissions, the program committee will give greater weight to papers that lend themselves to interactive discussion among attendees.

Submissions must be no longer than eight pages including all tables, figures, and references. Text should be formatted in two columns on 8.5"x11" paper using 10-point type on 12-point leading ("single-spaced"), with the text block being no more than 6.5"x9". Text outside the 6.5"x9" block will be ignored. Authors are encouraged to use the LaTeX and Word guides from the USENIX paper templates page at www.usenix.org/conferences/author-resources/paper-templates. The review process will be single-blind; submissions do not need to be anonymized.

All papers must be submitted in PDF format via the Web submission form on the CSET '17 Web site, www.usenix.org/cset17/cfp. Please do not email submissions.

All papers will be available online to registered attendees before the workshop. If your accepted paper should not be published prior to the event, please notify production@usenix.org. The papers will be available online to everyone beginning on the day of the workshop. At least one author from every accepted paper must attend the workshop and present the paper.

Simultaneous submission of the same work to multiple venues, submission of previously published work, or plagiarism constitutes dishonesty or fraud. USENIX, like other scientific and technical conferences and journals, prohibits these practices and may take action against authors who have committed them. See the USENIX Conference Submissions Policy at www.usenix.org/conferences/submissions-policy for details. Questions? Contact your program co-chairs, cset17chairs@usenix.org, or the USENIX office, submissions-policy@usenix.org.

Papers accompanied by nondisclosure agreement forms will not be considered. Accepted submissions will be treated as confidential prior to publication on the USENIX CSET '17 Web site; rejected submissions will be permanently treated as confidential.

Rev. 2/7/17

usenix

LISA17

Preliminary Call for Participation

Sponsored by USENIX, the Advanced Computing Systems Association

LISA17 will take place October 29–November 3, 2017, at the Hyatt Regency in San Francisco.

LISA is the premier conference for operations professionals, where systems engineers, IT operations, SRE practitioners, and academic researchers share real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

Industry Call for Participation

We invite both industry leaders and people on the front lines to propose topics that demonstrate the present and future of operations. LISA submissions should inspire and motivate attendees toward action that improves their day-to-day work as well as the tech industry as a whole.

LISA encourages submissions from people from a wide range of backgrounds. Our early proposal program allows first-time submitters and/or submitters of controversial topics to receive feedback and improve their chances for acceptance.

Important Dates

- Early proposals deadline*:
  Monday, February 27, 2017, 11:59 pm PST
- Notification to early proposal submitters:
  Monday, March 20, 2017
- Standard proposals deadline*:
  Monday, April 24, 2017, 11:59 pm PDT
- Notification to standard proposal submitters:
  Tuesday, June 13, 2017

*For early proposals, instead of being declined, feedback will be given so that it can be re-submitted for the main deadline. The standard deadline provides the opportunity to submit revised proposals, as well as providing more time for tutorial instructors to create new content prior to the conference.

Conference Organizers

Program Co-Chairs
Caskey L. Dickson, Microsoft
Connie-Lynne Villani, Grilled Cheese Invitational

Steering Committee
David Blank-Edelman, Apcera
Mark Burgess, Oslo University College
Brendan Gregg, Netflix
Casey Henderson, USENIX Association
Andrew Hume, Ericsson
Amy Rich, Mozilla
Ben Rockwood, Chef Software, Inc.
Carolyn Rowland, National Institute of Standards and Technology (NIST)

USENIX Tutorials Staff
Natalie DeJarlais, USENIX Association
Rik Farrow, USENIX Association

Topic Categories

Architecture

- Scalability and Resiliency
- Infrastructure Design
- Machine Learning
- Performance Planning
- Strategic Vision
- On the Horizon

Culture

- Building Dev/Ops Relationships
- Business Communication
- Standards and Regulatory Compliance
- On-Call Challenges
- Workplace Diversity
- Mentorship, Education, and Training

Engineering

- Dynamic Service Implementation
- Continuous Delivery
- Monitoring and Instrumentation
- Machine and Service Hardening
- Analytics of System Data
- Release Engineering

Proposals We Are Seeking

- **Talks:** 30 and 45 minute talks, with time for Q&A.
- **Mini Tutorials:** 90-minute courses teaching practical, immediately applicable skills.
- **Tutorials:** Half-day or full-day courses taught by experts in the specific topic, preferably with interactive components such as in-class exercises, breakout sessions, or use of the LISA Lab space.
- **Panels:** Moderator-led groups of 3–5 experts answering moderator and audience questions on a particular topic
- **Vendor-neutral interactive demonstrations** of hardware and software use in practical situations for operations professionals.

All proposal submissions are due by April 24, 2017, 11:59 pm PDT.
www.usenix.org/lisa17/cfp

Rev. 1/12/17

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