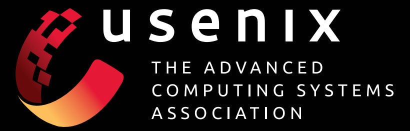
;login:

SUMMER 2016 VOL. 41, NO. 2



Causal Profiling
Charlie Curtsinger and Emery D. Berger

Fuzzing with AFL and ASAN
Peter Gutmann

HPC and Storage
John Bent, Brad Settlemyer, and Gary Grider

Adding Privacy Back to Android Apps
Michael Backes, Sven Bugiel, Oliver Schranz, and Philipp von Styp-Rekowsky

Policy-based Routing with iproute2
Jonathon Anderson

Linux FAST '16 Summary
Rik Farrow

Columns

Cutting Memory Usage in Python
David Beazley

Using the Spotify API
David N. Blank-Edelman

Vendoring in Go
Kelsey Hightower

Intrumenting a Go Service
Dave Josephsen

Security Trends
Dan Geer

Fuzzy Thinking
Robert G. Ferrell

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

USENIX ATC '16: 2016 USENIX Annual Technical Conference

June 22–24, 2016, Denver, CO, USA
www.usenix.org/atc16

Co-located with USENIX ATC '16:

SOUPS 2016: Twelfth Symposium on Usable Privacy and Security
June 22–24, 2016
www.usenix.org/soups2016

HotCloud '16: 8th USENIX Workshop on Hot Topics in Cloud Computing
June 20–21, 2016
www.usenix.org/hotcloud16

HotStorage '16: 8th USENIX Workshop on Hot Topics in Storage and File Systems
June 20–21, 2016
www.usenix.org/hotstorage16

SREcon16 Europe

July 11–13, 2016, Dublin, Ireland
www.usenix.org/srecon16europe

USENIX Security '16: 25th USENIX Security Symposium

August 10–12, 2016, Austin, TX, USA
www.usenix.org/sec16

Co-located with USENIX Security '16

WOOT '16: 10th USENIX Workshop on Offensive Technologies
August 8–9, 2016
www.usenix.org/woot16

CSET '16: 9th Workshop on Cyber Security Experimentation and Test
August 8, 2016
www.usenix.org/cset16

FOCI '16: 6th USENIX Workshop on Free and Open Communications on the Internet
August 8, 2016
www.usenix.org/foci16

ASE '16: 2016 USENIX Workshop on Advances in Security Education
August 9, 2016
www.usenix.org/ase16

HotSec '16: 2016 USENIX Summit on Hot Topics in Security
August 9, 2016
www.usenix.org/hotsec16

OSDI '16: 12th USENIX Symposium on Operating Systems Design and Implementation

November 2–4, 2016, Savannah, GA, USA
www.usenix.org/osdi16

Co-located with OSDI '16

INFLOW '16: 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads
November 1, 2016
Submissions due August 5, 2016
www.usenix.org/inflow16

Diversity '16: 2016 Workshop on Supporting Diversity in Systems Research
November 1, 2016

LISA16

December 4–9, 2016, Boston, MA, USA
www.usenix.org/lisa16

Co-located with LISA16

SESA '16: 2016 USENIX Summit for Educators in System Administration
December 6, 2016
Submissions due September 19, 2016
www.usenix.org/sesa16

USENIX Journal of Education in System Administration (JESA)
Published in conjunction with SESA
Submissions due August 26, 2016
www.usenix.org/jesa

Enigma 2017

January 30–February 1, 2017, Oakland, CA, USA
enigma.usenix.org

FAST '17

February 27–March 2, 2017, Santa Clara, CA, USA
Submissions due September 27, 2016
www.usenix.org/fast17

SREcon17

March 13–14, 2017, San Francisco, CA, USA

NSDI '17

March 27–29, 2017, Boston, MA, USA
Paper titles and abstracts due September 14, 2016
www.usenix.org/nsdi17

SREcon17 Asia

May 22–24, 2017, Singapore

;login:

SUMMER 2016 VOL. 41, NO. 2

EDITORIAL

2 Musings *Rik Farrow*

PROGRAMMING

6 coz: This *Is* the Profiler You're Looking For
Charlie Curtsinger and Emery D. Berger

11 Fuzzing Code with AFL *Peter Gutmann*

SECURITY

16 Boxify: Bringing Full-Fledged App Sandboxing to Stock Android
Michael Backes, Sven Bugiel, Oliver Schranz, and Philipp von Styp-Rekowsky

22 Using OpenSCAP *Martin Preisler*

27 Interview with Nick Weaver *Rik Farrow*

29 Interview with Peter Gutmann *Rik Farrow*

STORAGE

34 Serving Data to the Lunatic Fringe: The Evolution of HPC Storage
John Bent, Brad Settlemyer, and Gary Grider

40 Linux FAST Summit '16 Summary *Rik Farrow*

SYSADMIN

44 Improve Your Multi-Homed Servers with Policy Routing
Jonathon Anderson

48 MongoDB Database Administration *Mihalis Tsoukalos*

HISTORY

56 Linux at 25 *Peter H. Salus*

COLUMNS

60 Precious Memory *David Beazley*

67 iVoyeur: Go Instrument Some Stuff *Dave Josephsen*

71 Practical Perl Tools: Perl to the Music *David N. Blank-Edelman*

76 What's New in Go 1.6—Vendoring *Kelsey Hightower*

80 For Good Measure: Five Years of Listening to Security Operations
Dan Geer

87 /dev/random *Robert G. Ferrell*

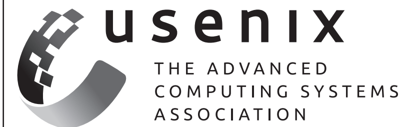
BOOKS

89 Book Reviews *Mark Lamourine, Peter Gutmann, and Rik Farrow*

USENIX NOTES

92 Results of the Election for the USENIX Board of Directors, 2016–2018

93 What USENIX Means to Me *John Yani Arrasjid*



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 bi-monthly 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 non-members are $90 per year. Periodicals postage paid at Berkeley, CA, and additional offices.

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

©2016 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

EDITORIAL

Musings

RIK FARROW

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

After having worked in one area for so many years, I decided that it is time to choose that area as a topic for musing. That area is writing and writing's relationship to programming.

Stephen King wrote that writing is telepathy [1]. Although King often writes about the metaphysical, in this case he is writing metaphorically. By telepathy, King means that the successful writer takes the ideas in his head and transfers them into someone else's head. There is no magic involved. What is involved is a lot of time, work, and careful consideration.

Technical Writing

I imagine that anyone reading this essay routinely does technical writing of one sort or another: emails, bug reports, proposals, articles, and especially papers. And this is where the telepathy had better happen. If you fail to pierce the fog between you and your readers, your thoughts will not be transmitted or will, at best, be misunderstood.

I ran across a wonderful example of just how technical writing can fail when done poorly and succeed when carefully crafted. I was reading a paper to be presented at a USENIX workshop and was amazed at just how clearly the authors presented their ideas. They explained each concept, which then flowed into the explanation of the following concept, until they had built both a description and an argument about the validity of the research in their paper. Their writing was crystal clear.

That paper received a Best Paper award. At this point you might be wondering why I am not citing that paper, but I have a reason. The first time these authors submitted their paper to the same conference, the paper was rejected because the program committee members didn't understand it. The authors were convinced that their work was good, but they had failed to truly communicate their ideas. The concepts were stuck in the authors' heads.

So they rewrote their paper, then edited it to within an inch of its life. I found this out after I met the authors during the conference, commended them on their wonderful writing, and asked them to reprise their paper for *;login:*. That's when I learned of their tale of failure and redemption.

I often run into very similar problems when editing articles for *;login:*. The authors clearly understand their topic, but leave out so much of *what is familiar to them* that the result is incomprehensible to everyone else. They have failed at telepathy, not because they are not psychic, but because they have not managed to communicate what other people who didn't work on their project couldn't possibly know.

When I run into this I try to coax people into explaining the missing parts. But I often fail for two reasons. There are people who just can't make the leap between ideas being in their heads, perhaps in wondrously beautiful constructs, and the very lack of the foundations necessary for these constructs to appear in the heads of their readers.

The second failing is my own. Once I have read an article many times I begin to understand the authors' points. Sometimes, instead of editing the article, the authors will attempt to explain the subtleties in email communications. So I learn what the authors have been trying

to communicate—I become contaminated with the very ideas that they failed to include in their article. At that point, I can no longer discern that the article has failed to communicate, except to those who already understand the topic and won't be reading the article anyway.

Writing and Programming

At one point I had given up on computing in general when a friend did two things that changed my life. First he asked me to store a box of materials for him while he traveled. The Zilog Z80 CPU manual was at the top of this box, and I realized that the world of computing was about to change: everyone would be able to have their own computer.

The second thing was what that friend, Madison Jones, told me. He said that if I could write, I could program.

I had programmed in college and had even written tools that my advisor used, but I had little faith in my ability to program. In retrospect, I believe that's because I spent a lot of time around people who were incredibly good programmers, and my skills had faded into inconsequence by comparison. Jones' words did serve to inspire me, and I went back to college to refresh my skills and restarted my career.

Dan Kaminsky, in a posting to the langsec-discuss mailing list [2], reminded me of the words of my friend when he wrote:

> Programming languages are about getting intent from human to machine. The human is an inherent part of this equation.

Programming is how we communicate human intent to computers. If you can write text in an organized and understandable way, you should be able to translate that text into a programming language. The translation itself is often faulty, as our mother tongue is not Go or JavaScript. But more often the problem lies in the programmer's inability to have clearly spelled out his ideas first, before attempting the difficult translation into a programming language. Having failed at description, the telepathy, the human-to-machine transmission, fails as well.

There is another way in which this failure to communicate appears in programming that is related to this very same failing. When very smart people write programs, they often begin by writing for themselves. If these people are part of a culture of very smart programmers (think Google, Facebook, LinkedIn, Apple, Microsoft, Red Hat, IBM, and many other organizations), they will also write for the people in their own cultures.

But those people are very poor at representing the rest of humanity. Most humans don't live and breathe advanced algorithms, although where I most often see these problems manifesting is in human-computer interfaces. When you are an insider, the intent of a few pixels in an interface is crystal clear, while outsiders have to guess that right-clicking or swiping over those pixels will produce a desired result and, hopefully, not be the icon for resetting the app to factory defaults, including wiping all the app's data.

Again, the program's authors have failed at their telepathy, keeping in mind that telepathy is not magic. Telepathy in practice is the art of communicating the ideas in your head to someone who doesn't share your head space.

The Lineup

We start off this issue with two articles about programming. While profiling can tell you about areas where your code is running often, causal profiling instead instruments code and tells you where improving the running time of a section of code will improve the overall performance of your code and by how much. Charlie Curtsinger and Emery D. Berger explain their technique, coz, and tell you where you can get the coz code and run your own causal profiling.

Peter Gutmann, a cyberpunk from New Zealand, is the hero of this issue. He has written an article about using AFL, an automated fuzzing toolkit, as well as a book review. Fuzzing your code has been an art form, but AFL along with a compiler like clang or a very recent version of gcc with ASAN support turns fuzzing into something you can do as part of routine testing. I also interviewed Peter because he's well established in several areas I wanted to explore: cryptography libraries, supporting open source software, and the safety of programming languages. I asked him about all of these topics and more.

I approached Michael Backes about his Boxify paper (USENIX Security '15), and he and his coauthors have taken the time to explain how they have expanded on that work to build a privacy-enhancing application for Android. Far too many Android apps want access to everything your smartphone has to offer, including sensitive information like your texts and contacts. With a newer version of Android (4.2+) and Boxify, you can limit the ability of apps to have access to anything the app developers or advertising included by the app might want to suck off your Android device.

Martin Preisler works on OSCAP, software that uses vulnerability information to audit Linux systems. OSCAP only works for some distros, but it doesn't just work on mounted file systems: OSCAP can scan VM images and containers too; OSCAP can also perform remediations, such as fixing permissions or correcting configurations.

I interviewed Nick Weaver, asking him to explain the path that led from designing FPGAs to writing about the first Warhol worm and becoming an informal expert on bulk data collection. Nick spoke on data collection at the first Enigma conference and explained how to de-anonymize Internet traffic.

EDITORIAL

Musings

John Bent, Brad Settlemyer, and Gary Grider share their perspective on how HPC (high performance computing) has changed, and will change, the way distributed storage systems work. HPC has very unusual storage requirements, but these relatively rare installations can teach us a lot about how distributed storage systems will work in the future.

I attended the Linux FAST Summit '16 and took a huge amount of notes. Instead of attempting to transcribe those notes, I have written a summary of the event this time. I also have a note about the FreeBSD Storage Summit that occurred the day before FAST '16 began.

Jonathon Anderson tells us how he is using policy-based routing at the University of Colorado Boulder Research Computing. With iproute2, you can go beyond traditional routing tables by adding policies that override default routes, techniques that really improve performance and latency for services on dual-homed servers.

Mihalis Tsoukalos provides an introduction to MongoDB administration. Like many other NoSQL databases, MongoDB has quirks that you'll want to know about before you start using it.

Peter Salus has collected many of his observations about the history of Linux into an article. The Linux system turned 25 this spring, and Peter tells us about the beginning of Linux and how various distros were born and how various distros died.

David Beazley demonstrates how to improve memory usage in Python. While creating a dict might be the easiest way to import a large amount of data, David demonstrates how using a dict is the least efficient method when it comes to minimizing the memory footprint.

David Blank-Edelman investigates the Spotify API while using Perl. David shows us techniques for exploring the RESTful interface used by Spotify and how to delve into the results returned for a richer understanding of the interface.

Kelsey Hightower explains the `vendor` directories that are now part of Go. Previously, dealing with external libraries required using an additional tool. The `vendor` directory allows you to include and manage external libraries within your Go project.

Dave Josephsen covers Go in his column too. Dave treats us to some neat tricks for instrumenting a service with a view to expanding that monitoring to cover new services with the addition of just a few lines of code.

Dan Geer explains how sometimes your metrics work better when you examine the trajectory of your data rather than their absolute values. Dan focuses on the data he has been collecting monthly about security trends for the past five years.

Robert G. Ferrell takes a deep look into Fuzzy Thinking as applied to cats, politicians, consumers, and fuzzing programs.

We have five book reviews this time, with three by Mark Lamourine, one by Peter Gutmann, and one by me.

Finally, John Yani Arrasjid writes about why he has donated so much of his time and energy to USENIX over the past 30 years.

Coding and Algebra

Like many of my contemporaries, I learned a lot of math before I ever had a chance to program a computer. My high school algebra prepared me for inorganic chemistry and classical physics, classes which I aced because they so nicely correlated to simple algebraic equations. And that appeared to lead me right into college programming classes.

I also tutored fellow classmates in chemistry during high school. While I found the relationship between algebra and chemistry crystal clear, the people I was tutoring did not. These folks were in the same college prep school I was in, equally smart, but my brain and their brains were not wired the same way. What I found simple, algebra, they had great difficulty comprehending.

When I heard of people suggesting that high school students not study algebra, I wondered if you could teach coding without algebra. After a few minutes searching, I ran across Andy Skelton's essay about coding and algebra [3]. Getting to the gist of Skelton's point, first put yourself into the mindset of algebra instead of programming. What do these three statements mean in algebra?

```
a = 2
b = 3
a = b
```

As a programmer, you might optimize this without even thinking about it to:

```
b = a = 3
```

But in algebra, the third statement, `a = b`, is nonsense because `a` does not equal `b`. What the heck! So algebra, even though it is where students learn about abstraction and how to solve word problems, doesn't map very neatly into coding. Not at all.

I also ran across a completely different approach, teaching algebra *through* coding. Bootstrap [4] provides class materials for teaching students about coding through programming a game. Instead of struggling to learn the abstractions of algebra, Bootstrap uses the design of a simple game to teach both programming and math through concrete examples.

I want to end this little essay by encouraging all of you to take the time not just to write, but to polish what you write. Stephen King is famous for his writing, but do you think that he just sits

down and knocks out best-selling novels? No way! King writes a first draft, perhaps 1000 words at a time, then puts that draft away for a month without showing it to anyone. Then he pulls out his draft and begins reworking it until he gets it into a shape where he is willing to show the early stages of a novel to some close friends.

Writing is not something that springs into existence as if a god had struck a rock and, miraculously, clear water pours forth. Just like programming, good writing is something that results from planning, careful work, debugging/editing, and, finally, testing. If you have to explain what you have written to your test audience, you have failed at telepathy. Go back and rework what you have written until a fresh test audience can understand it.

Remember the paper authors? One year their paper gets rejected in the first round, and the next year that same paper, carefully rewritten, wins a Best Paper award. The research was the same, but the exposition of that research wasn't.

Resources

[1] S. King, *On Writing: A Memoir of the Craft*, Scribner, 2000.

[2] Lang-sec discuss: https://mail.langsec.org/cgi-bin/mailman/listinfo/langsec-discuss.

[3] A. Skelton, "Programming Is not Algebra": https://andy.wordpress.com/2012/05/30/programming-is-not-algebra/.

[4] Bootstrap: http://www.bootstrapworld.org.

Do you have a USENIX Representative on your university or college campus? If not, USENIX is interested in having one!

The USENIX Campus Rep Program is a network of representatives at campuses around the world who provide Association information to students, and encourage student involvement in USENIX. This is a volunteer program, for which USENIX is always looking for academics to participate. The program is designed for faculty or staff who directly interact with students. We fund one representative from a campus at a time.

A campus rep's responsibilities include:

- Maintaining a library (online and in print) of USENIX publications at your university for student use
- Distributing calls for papers and upcoming event brochures, and re-distributing informational emails from USENIX
- Encouraging students to apply for travel grants to conferences
- Providing students who wish to join USENIX with information and applications
- Helping students to submit research papers to relevant USENIX conferences
- Providing USENIX with feedback and suggestions on how the organization can better serve students

In return for being our "eyes and ears" on campus, the Campus Representative receives access to the members-only areas of the USENIX Web site, free conference registration once a year (after one full year of service as a Campus Representative), and electronic conference proceedings for downloading onto your campus server so that all students, staff, and faculty have access.

To qualify as a campus representative, you must:

- Be full-time faculty or staff at a four-year accredited university
- Have been a dues-paying member of USENIX for at least one full year in the past

For more information about our Student Programs, please contact office@usenix.org

COZ
This *Is* the Profiler You're Looking For

CHARLIE CURTSINGER AND EMERY D. BERGER

Charlie Curtsinger is a new faculty member in the Computer Science Department at Grinnell College. His research interests include software performance, security, and reliability with an emphasis on probabilistic and statistical techniques. curtsinger@grinnell.edu

Emery Berger is a Professor in the College of Information and Computer Sciences at the University of Massachusetts Amherst, where he co-directs the PLASMA lab (Programming Languages and Systems at Massachusetts) and is a regular visiting researcher at Microsoft Research. He is the creator of a number of influential software systems including Hoard, DieHard, and DieHarder, a secure memory manager that was an inspiration for hardening changes made to the Windows 8 heap. He is currently serving as Program Chair for PLDI 2016. emery@cs.umass.edu

Causal profiling is a new approach to software profiling that tells developers which code is important for performance. Conventional profilers report where programs spend their time running, but optimizing long-running code may not improve program performance. Instead of simply observing a program, a causal profiler conducts *performance experiments* to predict the effect of speeding up many different parts of a program. During each experiment, a causal profiler uses *virtual speedup* to create the effect of optimizing part of the program, and *progress points* to measure any change in program performance as a result of the virtual speedup. A causal profile summarizes the results of many performance experiments, telling developers exactly where performance tuning would be worthwhile. Using COZ, a prototype causal profiler for Linux, we improve the performance of Memcached by 9%, SQLite by 25%, and several PARSEC applications by as much as 68%.

"Try running it with a profiler." This suggestion inevitably comes up once all reasonable ideas for improving a program's performance are exhausted. The authors of thousands of lines of code have been unable to come up with any explanation for the system's poor performance, so why do we expect a tool from 1982 to fare better [3]? Deep down, we've always known this was true. Take the historically accurate space adventure game you were playing too late last Tuesday as an example; how would a profiler know that the thread that plays lightsaber crackling sounds was less important than the thread that controls the stormtrooper you were battling? *It wouldn't.* When we look at a software profile we aren't looking for guidance, we're looking for surprises. And with parallel programs, *practically everything* is surprising. That's not to say that profilers aren't informative. Any good software profiler can tell you, with great accuracy, where a program spends its execution time. Unfortunately, this isn't the information we're looking for.

Code that runs for a long time is not necessarily a good choice for performance tuning. Developers need to know which code is important—where successful performance tuning would improve the program's end-to-end performance. Consider a function that draws a "loading …" animation while you wait for the next level of your game to load. The animation runs just as long as the loading code, but we would never expect to speed up the program by making the animation faster.

This problem is not limited to programs that perform I/O. Figure 1 shows a simple parallel program with a similar issue. This program creates two threads, one to run the function `a()` and another to run the function `b()`. The program exits once both threads have finished. A conventional profiler like gprof, whose output for this program is shown in Figure 2, reports that the program spends roughly equal time running `a()` and `b()`. While accurate, this information is misleading; optimizing `a()` alone will speed the program up by just 4.5%, and optimizing `b()` will have no effect on performance.

example.cpp

```
void a() { // ~6.7 seconds
  for(volatile size_t x=0; x<2000000000; x++) {}
}
void b() { // ~6.4 seconds
  for(volatile size_t y=0; y<1900000000; y++) {}
}
int main() {
  // Spawn both threads and wait for them.
  thread a_thread(a), b_thread(b);
  a_thread.join(); b_thread.join();
}
```

Figure 1: A simple multithreaded program that illustrates the shortcomings of existing profilers. Optimizing `a()` will improve performance by no more than 4.5%, while optimizing `b()` would have no effect on performance.

The key issue with conventional profilers is that they only *observe* a program's execution. Through observation alone, they cannot tell you which code to optimize, because long-running code is not necessarily important to program performance. Speeding up a line of code might shorten an important path through the program, or it may speed up a thread that does background work, increasing contention on a critical data structure, which in turn hurts overall program performance.

```
% time  cumulative     self              self    total
           seconds  seconds   calls   Ts/call  Ts/call   name
 55.20        7.20     7.20       1                       a()
 45.19       13.09     5.89       1                       b()

% time      self    children      called        name
                                                <spontaneous>
  55.0      7.20        0.00                    a()
-----------------------------------------------------------------------
                                                <spontaneous>
  45.0      5.89        0.00                    b()
```

Figure 2: A conventional profile for `example.cpp` collected with gprof

Causal Profiling

Causal profiling is a novel approach to profiling that identifies code where optimizations will have the largest impact [2]. A causal profiler is fundamentally different from a conventional profiler; rather than simply observing program execution, a causal profiler intentionally perturbs program performance to conduct *performance experiments.* During a performance experiment, a causal profiler creates the effect of speeding up some piece of a program using *virtual speedup* (more on this later). While virtually speeding up one piece of a program, a causal profiler then measures program performance to determine the effect of this speedup. Given enough performance experiments with varying locations and sizes of virtual speedup, we can construct a causal profile, which tells you both where optimizations would have an effect and how large that effect would be.

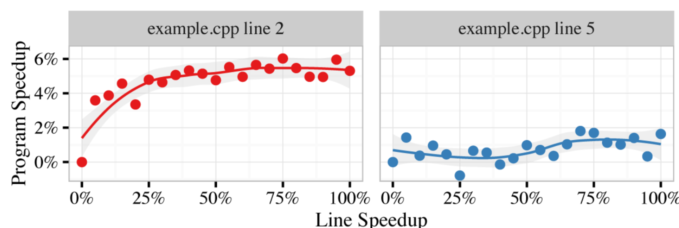


Figure 3: Causal profile for `example.cpp`

Figure 3 shows a real causal profile for the program in Figure 1 collected with COZ, a prototype causal profiler for Linux. This causal profile suggests that speeding up `a()` alone could improve program performance by up to 5.0%, very close to the actual 4.5%. Beyond this point, the thread running `b()` becomes the program's critical path. The causal profile correctly indicates that speeding up `b()` alone would have a negligible effect on performance.

Producing a causal profile requires three key pieces: we need a way to create the effect of an optimization, the profiler must apply a virtual speedup for the duration of a performance experiment, and we need a way to measure a program's performance at the end of each experiment.

Virtual Speedup

A causal profiler cannot magically speed up a part of a program and measure the effect of that speedup; if this were possible, we would just magically speed up the entire program. Instead, a causal profiler creates the effect of speeding up one part of a program by slowing everything else down. The amount that other threads are slowed down determines the size of the virtual speedup. The size of the virtual speedup can range from 0% (the code's runtime is unchanged) to 100% (the code's runtime is reduced to zero).

Figure 4 illustrates a virtual speedup in a simple parallel program. Part (a) shows the original execution of two threads running functions `f()` and `g()`, and part (b) shows the effect of *actually* speeding up `f()` by 40%; the size of this speedup was chosen arbitrarily and could be any value from 0% to 100%. Finally, part (c) shows the effect of *virtually* speeding up `f()` by 40%.

Each time `f()` runs in one thread, all other threads pause for 40% of `f`'s original execution time. While virtual speedup does not actually shorten the program's runtime, the difference between the program's original runtime and its runtime with a virtual speedup is known: it is just the number of times `f()` ran multiplied by the delay size. Given that we know both quantities, we

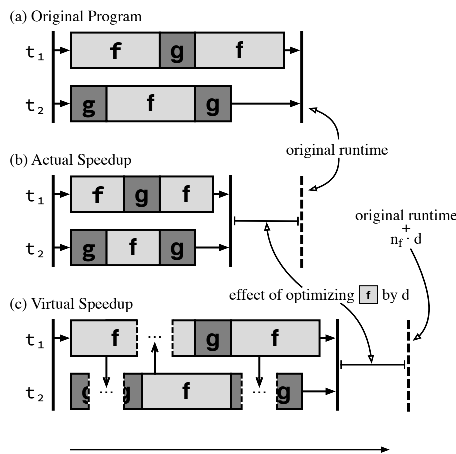


Figure 4: An illustration of a program's (a) original execution, (b) a real speedup of function `f()` by 40%, and (c) a virtual speedup of `f()` by 40%

can adjust the baseline runtime by this extra time to predict the effect of an actual speedup.

Instrumenting a program to track visits to `f()` and signaling other threads to pause them every time it runs would be prohibitively expensive. Instead, COZ uses sampling to approximate this approach. Instead of pausing other threads each time `f()` runs, COZ would delay other threads every time it sees a sample in `f()`. The size of the delay is proportional to the sampling interval rather than the execution time of a single call to `f()`.

Performance Experiments

A causal profiler can use virtual speedup to test the effect of a potential optimization, but how does it decide which code to virtually speed up, and by how much? Coverage is particularly important: given a large code base, we would like to find the one line of code with the largest possible payoff from optimizations. COZ applies virtual speedup at the granularity of source lines, which means there are potentially tens of thousands of program fragments that could be virtually sped up. Rather than choosing uniformly from all source lines, COZ selects from the distribution of where a program spends its time running. While long-running code may not be the best place for performance tuning, code that never runs is certainly a bad place to focus our limited energy. Once a COZ selects a line to virtually speed up, it selects a speedup size between 0% and 100% in increments of 5%.

During a performance experiment, COZ applies the same fixed virtual speedup to the selected line. All that is required to speed up a specific line is to map program execution samples, which are memory addresses of code, to source information. This is a relatively straightforward process using DWARF debugging information. While COZ currently uses source lines as the unit of virtual speedup, any fragment of code that can be mapped to addresses could be virtually sped up.

At the end of a performance experiment, COZ measures the program's performance with the virtual speedup in place. But how can we measure performance in the middle of an execution or for programs that run indefinitely? For the simple example in Figure 3, COZ runs a single performance experiment for the entire execution of the program. While this approach works for small programs, it does not scale well; large programs would require, at minimum, thousands of runs to get reasonable coverage with performance experiments. COZ solves this problem by allowing developers to specify progress points.

Progress Points

A progress point is some point in a program that should happen as frequently as possible, completing a user request, for example, or processing a block of data. Developers mark one or more progress points in their application by adding the `COZ_PROGRESS` macro, which keeps a count of the visits to this point in the code. COZ measures the rate of visits to a progress point as a proxy for performance. This allows COZ to conduct many performance experiments in a single run of a program or in programs where end-to-end runtime is not meaningful such as servers and interactive applications.

This basic notion of progress points allows us to measure throughput at some point in the code, but COZ can also use progress points to measure the latency between two points. Instead of specifying a single point, developers mark the beginning and end of a transaction using two macros, `COZ_BEGIN` and `COZ_END`. COZ does not track individual transactions as they flow through the system, but by measuring the rate of arrivals at the beginning point and the number of outstanding requests, COZ can use Little's Law to compute the average latency between the two points [4].

Using COZ

Running a program with COZ requires just three steps: (1) find one or more places to add progress points that allow COZ to measure the program's performance; (2) run the program with the command-line driver: `coz run --- <program> <args>`; and (3) use COZ's Web-based profile interface to plot the results and rank lines by potential impact. Our SOSP 2015 paper on causal profiling includes case studies where we use COZ to optimize eight different applications, three of which are included below [2]. These case studies include the compression program dedup, where COZ led us to a degenerate hash function; the embedded database SQLite, where COZ guided us to an inefficient coding practice that prevented function inlining; and the in-memory key-value store Memcached, where COZ identified unnecessary contention on a shared lock. Fixing these issues led to whole-program performance improvements of 9% for dedup, 25% for SQLite, and 9% for Memcached.

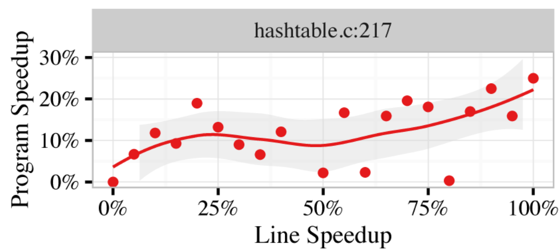


Figure 5: The causal profile for the source line `hashtable.c:217` in dedup shows the potential performance improvement of fixing a hash bucket traversal bottleneck.

Case Study: dedup

The dedup application, part of the PARSEC suite, performs parallel file compression via deduplication [1]. We added a progress point to dedup's code just after a single block of data is compressed (`encoder.c:189`).

COZ identifies the source line `hashtable.c:217` as an opportunity for optimization; Figure 5 shows the causal profile results for this line. This plot shows that improving the performance of the code that runs this line will result in a nearly one-to-one performance improvement in program performance up to 20%, with modest additional gains for performance improvements over 20%. This code is the top of the `while` loop in `hashtable search` that traverses the linked list of entries that have been assigned to the same hash bucket. This suggests that dedup's shared hash table has a significant number of collisions. Hash collisions could be caused by two things: a hash table that is too small or a hash function that does not evenly distribute elements throughout the hash table. Increasing the hash table size had no effect on performance, so the only remaining culprit is the hash function. It turns out dedup's hash function was mapping keys to just 2.3% of the available hash table buckets; over 97% of hash buckets were never used during the entire execution, and the 2.3% of buckets that were used at all contained an average of 76.7 entries.

The original hash function adds characters of the hash table key, which leads to virtually no high-order bits being set. The resulting hash output is then passed to a bit-shifting procedure intended to compensate for poor hash functions. Removing the bit-shifting step increased hash table utilization to 54.4%, and changing the hash function to use bitwise XOR on 32-bit chunks of the key increased hash bucket utilization to 82.0%. This three-line change resulted in an 8.95% ± 0.27% performance improvement. Figure 6 shows the rate of bucket collisions of the original hash function, the same hash function without the bit shifting "improvement," and our final hash function. The entire optimization required changing just three lines of code. This entire process, from profiling to a completed patch, took just two hours.

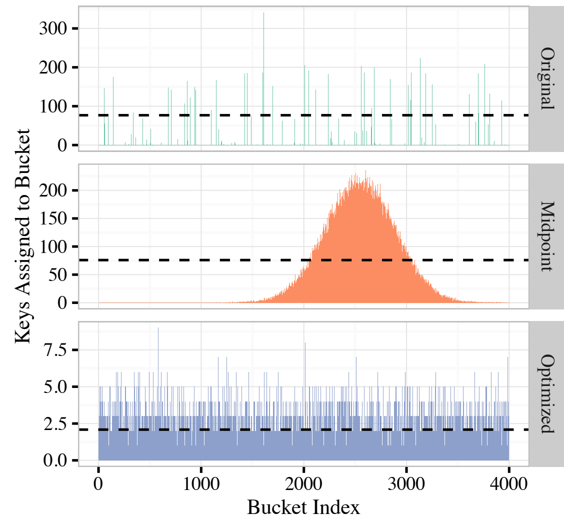


Figure 6: Hash collision rate before, during, and after performance tuning for a subset of dedup's hash buckets. Dashed black lines show the average number of items per utilized bucket. Note the different y-axes. Fixing dedup's hash function improved performance by 9%.

Case Study: SQLite

The SQLite database, which can be included as a single large C file, is used for many applications—including Firefox, Chrome, Safari, Opera, Skype, iTunes—and is a standard component of Android, iOS, Blackberry 10 OS, and Windows Phone 8. We evaluated SQLite's performance using a simple write-intensive parallel workload, where each thread rapidly inserts rows to its own private table. While this benchmark is synthetic, it exposes any scalability bottlenecks in the database engine itself because all threads should theoretically operate independently. This benchmark executes a progress point each time an insert to the database is completed.

COZ identified three important optimization opportunities, shown in Figure 7. Interestingly, the profile suggests that a small improvement to these lines' performance would speed up the program, but large performance improvements could actually be detrimental; this is evidence of contention elsewhere in the program. While resolving contention did not play a role in optimizing SQLite, contention is a factor in the next case study, which examines Memcached.

At startup, SQLite populates a large number of structs with function pointers to implementation-specific functions, but most of these functions are only ever given a default value determined by compile-time options. The three functions COZ identified unlock a standard pthread mutex, retrieve the next item from a shared page cache, and get the size of an allocated object. These simple functions do very little work, so the overhead of the indirect function call is relatively high, particularly because

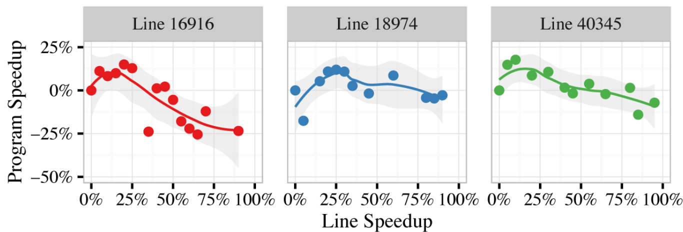


Figure 7: COZ's output for SQLite before optimizations

these functions are all likely candidates for inlining. Replacing these indirect calls with direct calls—which only required changes to seven lines of SQLite code—resulted in a 25.60% ± 1.00% speedup.

Case Study: Memcached

Memcached is a widely used in-memory key-value store, typically used as a cache in front of a database server. To evaluate Memcached's performance, we ran a version of the Redis performance benchmark ported to Memcached (available at https://github.com/antirez/mc-benchmark). This program spawns 50 parallel clients that collectively issue 100,000 `SET` and `GET` requests for a variety of keys. We added a progress point at the end of the `process_command` function in Memcached, which will execute after each client request is completed.

The vast majority of the source lines COZ profiles have virtually no potential for performance impact; this is hardly surprising given the level of performance tuning attention Memcached has received [5]. Excluding lines with little or no potential performance impact—which have a flat causal profile—most of the lines COZ identifies are cases of contention with a characteristic downward-sloping causal profile plot. This downward slope shows that optimizing this particular line of code would *hurt* rather than help program performance. If speeding up a line of code would hurt program performance, then some action that follows this line must contend with the program's critical path. One such line is at the start of the `item_remove` function, which locks an item in the cache, decrements its reference count, and frees the item if its reference count is zero.

To reduce lock-initialization overhead, Memcached uses a static array of locks to protect items, where each item selects a lock using a hash of its key. Consequently, locking any one item can potentially contend with independent accesses to other items whose keys happen to hash to the same lock index. However, Memcached uses atomic increment and decrement operations for reference counts; locking at this point is completely unnecessary. Resolving this issue along with two similar fixes required changing just six lines of code and resulted in a 9.39% ± 0.95% performance improvement.

Conclusion

Causal profiling is a radical departure from previous approaches to software profiling. Conventional profilers simply observe a program's execution, leaving developers to apply some intuition or a performance model to decide which parts of the program are important for performance. With a causal profiler, the program is the performance model. Instead of simply observing a program while attempting to minimize changes to that program's performance, a causal profiler *intentionally* alters program performance to conduct performance experiments. By carefully coordinating delays across a program's execution, a causal profiler can create the effect of optimizing a specific code fragment. By directly measuring the effect of a performance change, a causal profiler can tell developers exactly where optimizations will make a difference.

COZ is available at http://coz-profiler.org.

References

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architecture and Computation Techniques (PACT 2008)*, pp. 72–81: http://parsec.cs.princeton.edu/doc/parsec-report.pdf.

[2] Charlie Curtsinger and Emery D. Berger, "COZ: Finding Code that Counts with Causal Profiling," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 2015)*, pp. 184–197: http://dx.doi.org/10.1145/2815400.2815409.

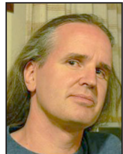
[3] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, "gprof: a Call Graph Execution Profiler," in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices,* vol. 17, no. 6, pp. 120–126: http://dx.doi.org/10.1145/989393.989401.

[4] John D. C. Little, "A Proof for the Queueing Formula: L = λW," *Operations Research,* vol. 9, no. 3 (1961), pp. 383–387.

[5] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani, "Scaling Memcache at Facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13),* pp. 385–398: https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf.

Fuzzing Code with AFL

PETER GUTMANN



Peter Gutmann is a Researcher in the Department of Computer Science at the University of Auckland working on design and analysis of cryptographic security architectures and security usability. He helped write the popular PGP encryption package, has authored a number of papers and RFCs on security and encryption, and is the author of the open source cryptlib security toolkit *Cryptographic Security Architecture: Design and Verification* (Springer, 2003) and an upcoming book on security engineering. In his spare time he pokes holes in whatever security systems and mechanisms catch his attention and grumbles about the lack of consideration of human factors in designing security systems. pgut001@cs.auckland.ac.nz

Most programs are only ever used in fairly stereotyped ways on stereotyped input and will often crash in the presence of unexpected input. Test suites designed by humans, assuming there even is a test suite, are only as good as the people creating them and often only exercise the common code paths. This problem is where fuzzing comes in, the creation of input that exercises as many different code paths as possible in order to show up problems in the code. Until recently fuzzing has been a complex and tedious process, but with the appearance of instrumentation-guided fuzzers like AFL the task has become much easier. This article looks at how you can apply AFL to your code.

Fuzzing Software with AFL

Most software is pretty buggy. The reason why it works a lot of the time is that we use it in ways that don't trigger the bugs, either because the bugs are in obscure parts of the code that never get exercised or because they're in commonly used parts of the code but we know about them and avoid triggering them. Since most programs are used in stereotyped ways that exercise only a tiny portion of the total number of code paths, removing obvious problems from these areas will be enough to keep the majority of users happy. This was shown up in one study of software faults which found that one-third of all faults resulted in a mean time to failure (MTTF) of more than 5,000 years, with somewhat less than another third having a MTTF of more than 1,500 years [1].

On the other hand, when you feed unexpected input to these programs, meaning you exercise all the code paths that are normally left alone, you reduce the MTTF to zero. A study [2] that looked at the reliability of UNIX utilities in the presence of unexpected input, and later became famous for creating the field of fuzz-testing or fuzzing, found that one-quarter to one-third of all utilities on every UNIX system that the evaluators could get their hands on would crash in the presence of random input.

Unfortunately, when the study was repeated five years later [3] the same general level of faults was still evident.

Windows was no better. A study that looked at 30 different Windows applications [4]—including Acrobat Reader, Calculator, Ghostscript, Internet Explorer, MS Office, Netscape, Notepad, Paintshop, Solitaire, Visual Studio, and Wordpad, coming from a mix of commercial and non-commercial vendors—found that 21% of programs crashed and 24% hung when sent random mouse and keyboard input, and every single application crashed or hung when sent random Windows event messages.

Before the Apple fans get too smug about these results, OS X applications, including Acrobat Reader, Apple Mail, Firefox, iChat, iTunes, MS Office, Opera, and Xcode, were even worse than the Windows ones [5].

So what can we do about this?

Fuzz Testing

The answer to the question posed in the previous section is to test your app with random input through fuzz testing before someone else, possibly with less than good intentions, does it for you. Until now this has been quite a pain to deal with since the tools were under-documented and required a large amount of manual intervention to do their job. The process would typically involve downloading a fuzzer, staring at the extensive half-page-long manual for a while, and then settling down to trying to figure out whatever arcane scripting language the fuzzer used to get it to generate input for your app.

Even if you got that far, it was often a case of trial and error with a code profiler to determine whether you were getting any useful code coverage from the fuzzing or just wasting CPU cycles.

Eventually, with a large amount of effort and more than a little luck, you could start fuzzing your code.

All of this changed a few years ago with the introduction of instrumentation-guided fuzzers. These compile the code being fuzzed with a custom build tool that instruments the code being compiled and tries to ensure that the fuzzer generates input that exercises all of the different code paths. As a result, the fuzzer doesn't spend forever randomly generating test cases that exercise the same paths over and over, but generates cases that exercise as many different paths as possible. In addition it can prune the test cases to eliminate ones that are covered by other cases, minimizing the amount of effort expended in trying to find problems.

This strategy produces some truly impressive results. The fuzzer I'll be talking about here, American Fuzzy Lop (named after a breed of rabbit), or AFL, managed to produce valid JPEG files recognized by djpeg starting from a text file containing the string "hello" [6]. The files didn't necessarily decode to produce a photo of the Eiffel Tower but did produce valid if rather abstract-looking JPEG images.

When I ran it on my code, I was somewhat surprised to find myself stepping through PGP keyring code when I'd started with a PKCS #15 key file, which has a completely different format. The input file sample for fuzzing that AFL had started with was:

```
00000000  30 82 04 BA 06 0A 2A 86 48 86 F7 0D 01 0F 03 01
00000010  A0 82 04 AA 30 82 04 A6 02 01 00 30 82 04 9F A0
00000020  82 01 96 A0 82 01 92 A0 82 01 8E 30 18 0C 16 54
00000030  65 73 74 20 45 43 44 53 41 20 73 69 67 6E 69 6E
```

What AFL mutated this into over time was:

```
00000000  99 01 A2 04 37 38 F7 27 11 04 00 97 AB 53 62 04
00000010  7F 8C BB 1A 25 0A 58 CA 63 20 9D 43 D4 8D 50 15
00000020  70 68 E3 76 3D 7B C2 76 78 28 23 B6 9A 40 BC CF
00000030  14 88 A3 80 47 3B 5F 17 5F 73 72 5A 60 1F D3 1B
```

Like the JPEGs that started as the text string "hello," it was a syntactically valid PGP keyring, although semantically meaningless.

Building AFL

Using AFL requires AFL itself [7], a compiler, and a compiler tool called Address Sanitizer [8], or ASAN, that's used to detect code excursions. ASAN requires a fairly recent compiler, quite possibly a more recent one than whatever crusty old version your OS ships with, so don't use the version you find in some repository but build it yourself so that you know it'll be done right. You can use either gcc or clang; if you value your sanity I'd recommend clang.

Start by getting the various pieces of the compiler suite that you'll need (clang is part of the LLVM toolset):

```
svn co https://llvm.org/svn/llvm-project/llvm/trunk LLVM
svn co https://llvm.org/svn/llvm-project/cfe/trunk LLVM/tools
    /clang
svn co https://llvm.org/svn/llvm-project/compiler-rt/trunk
    LLVM/projects/compiler-rt
```

Then build clang and the related tools:

```
cd LLVM
mkdir build
cd build
export MAKEFLAGS="-j`getconf _NPROCESSORS_ONLN`"
cmake -DCMAKE_BUILD_TYPE=RELEASE ~/LLVM
cmake --build .
```

If you don't have CMake installed, then either get it from your favorite repository or build it from source [9].

The clang build process will take an awfully long time even spread across multiple CPUs (which is what the `MAKEFLAGS` line does), so you can go away and find something else to do for a while. If you run out of virtual memory during the build process, decrease the -j argument, which spreads the load across fewer processors and uses less resources.

Eventually, the whole thing will be built and you'll have the necessary binaries present in the LLVM/build directory tree.

Now that you've got the tools that you need to build AFL, you can build AFL itself:

```
wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
tar xvfz afl-latest.tgz
rm afl-latest.tgz
cd `find . -maxdepth 1 -type d -print | sort -r | head -1`
export PATH=~/LLVM/build/bin:~/LLVM/tools/clang/tools
    /scan-build:$PATH
make
cd llvm_mode
make
```

After all that, you've finally got the AFL tools ready to go.

Building Your App

Now you need to build your app. First, you need to modify it to take as input the data generated by AFL. If your app is something that takes a filename on the command line, as the djpeg example mentioned earlier does, this is pretty straightforward. If the app is a bit more complex than that, for example a GUI app, then you'll need to modify your code to allow the test data to be fed in. In my code I use a custom build that no-ops out a lot of the code that isn't relevant to the fuzzing and allows the test data to be injected directly into the data-processing code.

If you're fuzzing a network app then things get a bit more complicated. There's ongoing work to add support for fuzzing programs that take input over network sockets, one example being [10], but since it's work-in-progress it could well be superseded by the time you read this. A much easier option is to modify your code to take input from a file instead of a network socket, which also avoids the overhead of dealing with a pile of networking operations just to get the test data into your app.

Finally, if your app has a relatively high startup overhead, then AFL provides additional support for dealing with this, which I'll describe later in the section on optimizing AFL use.

Once you've got your code set up to take input from AFL, you can build it as you normally would, specifying the use of the AFL tools instead of the usual ones. For example, if you build your app using a makefile, you'd use:

```
export AFL_HARDEN=1 ; export AFL_USE_ASAN=1 ;
make CC=afl-clang-fast CFLAGS=-fsanitize=address
```

which builds the code with instrumentation and ASAN support. `afl-clang-fast` is the AFL-customized version of clang that adds the necessary instrumentation needed by the fuzzing. As the code is built, you'll see status reports about the instrumentation that's being applied.

One thing that you need to make sure of is that your app actually crashes on invalid input, either explicitly by calling `abort()` (typically via an assertion) or implicitly with an out-of-bounds memory access or something similar that ASAN can detect. ASAN inserts guard areas around variables and can detect out-of-bounds and other normally undetectable errors. But if your program simply continues on its way with invalid input, then AFL can't detect a problem. The easiest way to ensure an AFL-detectable exit is to sprinkle as many sanity-check assertions as possible throughout your code, which means that if any pre- or post-condition or invariant is violated by the input that AFL generates, it can be detected.

Fuzzing Your App

Now you're finally ready to fuzz your app. To do this, run the fuzzer as `afl-fuzz`, giving it an input directory to take sample files from and an output directory telling it where to store statistics and copies of files that produce crashes. If your app was built as `a.out`, you'd use:

```
afl-fuzz -i in -o out ./a.out @@
```

The `@@` is a placeholder that `afl-fuzz` replaces with the path to the fuzzed data files that it generates.

When the fuzzer is running, the results will be displayed on an annoying screen-hogging live status page that prevents you from running more than one copy on multicore systems. To deal with this, use `nohup` to get rid of the full-screen output. The AFL tools have built-in support for running across multiple cores or servers but the details are a bit too complex to go into here and have evolved over time; see the AFL Web pages for more information.

Alongside the status screen, stats are written to a file `fuzzer_stats` in the fuzzer output directory. The important values are `execs_per_sec`, which indicate how fast you're going; `execs_done`, how far you've got; `unique_crashes` and `unique_hangs`, which are pretty self-explanatory (although the hangs aren't terribly useful unless you set the threshold fairly high; they'll be mostly false positives due to timing glitches like page faults and I/O); and finally `cycles_done`, the number of full sets of mutations exercised. Depending on the complexity of your input data, it can take days or even weeks to complete a cycle. There's a tool `afl-tmin` that tries to help you minimize the size of the test cases; again, see the AFL Web page for details.

For each crash or hang, AFL will write the input that caused it to the `crashes` or `hangs` subdirectories in the output directory. You can then take the files and feed them to your app running under your debugger of choice to see what's going on.

If you're running AFL on someone else's machine, you're going to make yourself somewhat unpopular with it. It uses 100% of the CPU per AFL task, and if you maximize the overall utilization with one task per '`getconf _NPROCESSORS_ONLN`' you're going to also have a load average of '`getconf _NPROCESSORS_ONLN`'.

In addition, ASAN uses quite a bit of virtual memory, around 20 terabytes on x64. Yes, that's 20,000,000 megabytes, which it uses as shadow memory to detect out-of-bounds accesses. While this may seem like a gratuitous stress test of your server's VM subsystem, when I ran it on someone else's Linux box it ran without any problems. Just be aware that you're going to really hammer anything that you run this on.

Optimizing the Fuzzing

Many applications have a high startup overhead. As part of its operation AFL uses a fork server in which it preloads the app once rather than reloading it on each run [11], but this still triggers the startup overhead. The way to avoid this is to defer the forking until the startup has completed and tell AFL to fork after that point. So if your app has a code flow that's a bit like:

```
init_app();
process_input();
```

then you'd insert a call to the function `__afl_manual_init()` between the two:

```
init_app();
__afl_manual_init();
process_input();
```

which defers the forking until that point. This means the startup code is run once and then the initialized in-memory image is cloned on each fuzzing run, which can greatly speed up the fuzzing process. If you use this optimization, make sure that you insert the call at the right place. If, for example, you do some of the input processing before the AFL fork-server call, then you'll be reusing a copy of the same input on each fuzzing run rather than reading new input each time.

Beyond that there are various other tweaks that you can apply, which you can also find on the AFL Web page.

So that's how you can test your code's behavior on unexpected input. Using fuzzing may seem like a lot of work to set up initially, but once it's done you can roll it into an automated test system that both identifies existing problems in your code and later checks that you haven't introduced new ones in any updates you make.

References

[1] Edward Adams, "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Development,* vol. 28, no. 1 (January 1984), pp. 2–14.

[2] Barton Miller, Lars Fredriksen, and Bryan So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM,* vol. 33, no. 12 (December 1990), pp. 32–44: http://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf.

[3] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan and Jeff Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," *University of Wisconsin—Madison Computer Sciences Technical Report,* #1268, April 1995: ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf.

[4] Justin Forrester and Barton Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," *Proceedings of the 4th USENIX Windows Systems Symposium (WinSys '00),* August 2000, p. 59: https://www.usenix.org/legacy/events/usenix-win2000/full_papers/forrester/forrester.pdf.

[5] Barton Miller, Gregory Cooksey, and Fredrick Moore, "An Empirical Study of the Robustness of MacOS Applications Using Random Testing," *SIGOPS Operating Systems Review,* vol. 41, no. 1 (January 2007), pp. 78–86: ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-nt.pdf.

[6] Michal Zalewski, "Pulling JPEGs Out of Thin Air," November 7, 2014: http://lcamtuf.blogspot.co.nz/2014/11/pulling-jpegs-out-of-thin-air.html.

[7] Michal Zalewski, "American Fuzzy Lop": http://lcamtuf.coredump.cx/afl/.

[8] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Annual Technical Conference,* June 2012, p. 309: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf.

[9] "CMake": http://www.cmake.org/.

[10] Hanno Böck, "Network Fuzzing with American Fuzzy Lop," October 27, 2015: https://blog.fuzzing-project.org/27-Network-fuzzing-with-american-fuzzy-lop.html.

[11] Michal Zalewski, "Fuzzing Random Programs without execve()," October 14, 2014: http://lcamtuf.blogspot.co.nz/2014/10/fuzzing-binaries-without-execve.html.

ENIGMA

MORE TO DECIPHER

It's time for the security community to take a step back and get a fresh perspective on threat assessment and attacks. This is why in 2016 the USENIX Association launched Enigma, a new security conference geared towards those working in both industry and research. Enigma will return in 2017 to keep pushing the community forward.

Expect three full days of high-quality speakers, content, and engagement for which USENIX events are known.

The Call for Participation will be available soon.

enigma.usenix.org

JAN 30–FEB 1 2017

OAKLAND, CALIFORNIA, USA

SECURITY

Boxify

Bringing Full-Fledged App Sandboxing to Stock Android

MICHAEL BACKES, SVEN BUGIEL, OLIVER SCHRANZ, AND PHILIPP VON STYP-REKOWSKY

Michael Backes is a full Professor at the Computer Science Department of Saarland University and has the chair for Information Security and Cryptography. He is the Director of the Center for IT-Security, Privacy, and Accountability (CISPA) and is a Max Planck Fellow of the Max Planck Institute for Software Systems. backes@mpi-sws.org

Sven Bugiel is a Postdoctoral Researcher at the Center for IT-Security, Privacy, and Accountability (CISPA)/ Saarland University. His research interests lie in (mobile) systems security and trusted computing, with a strong focus on Android. bugiel@cs.uni-saarland.de

Oliver Schranz is a first year PhD student in the Information Security and Cryptography Group at CISPA/Saarland University. His research focuses on compiler-assisted security solutions on Android that are deployable at the application layer. schranz@cs.uni-saarland.de

Philipp von Styp-Rekowsky is a PhD student at CISPA/Saarland University. His research focuses on mobile systems security with a strong emphasis on application layer solutions for Android. styp-rekowsky@cs.uni-saarland.de

Boxify is the first concept for full-fledged app sandboxing on stock Android. Building on app virtualization and process-based privilege separation, **Boxify** eliminates the necessity to modify the code of monitored apps or the device's firmware and thereby overcomes the existing legal concerns and deployment problems of prior approaches. As such, **Boxify** is a powerful tool for realizing privacy-protecting solutions on Android. In this article, we explain the **Boxify** concept and illustrate how it can benefit users in protecting their privacy on Android.

Background

Smart devices, like smartphones and tablets, in conjunction with the plethora of available apps, are very convenient companions in our daily tasks; they are our social hub to stay in touch with our friends and colleagues, help us organize our day, have replaced our digital cameras, and are our online banking portal or navigation system, among many more uses. From a privacy perspective, however, the data protection mechanisms in place on those platforms do not do justice to the rich functionality and data wealth that those platforms offer to apps. Although all popular platforms (like Apple's iOS and Google's Android) support permissions—that is user-granted privileges an app must hold to access user data and system resources [5]—permissions have been shown to be futile in creating more transparency of app behavior for users and in effectively protecting the users' privacy [3, 8, 9, 10, 13].

First, permissions do not communicate how apps are *actually* (ab)using their granted privileges but only what apps could *potentially* do. For instance, let's consider an on-demand transportation app that requests access to the user's SMS—access also commonly requested by banking trojans to intercept TAN (transaction authentication number) messages—how can the user distinguish whether this access has benign or malicious intent?

Second, permissions are too coarsely defined, are not conditional, and hence cause apps to be overprivileged—a direct contradiction of the principle of least privilege that permissions were originally intended to realize. Picking up the example of the transportation app, it is reasonable why such an app would need access to *some* of the user's address book data (e.g., a contact's address to quickly choose a destination) but not *all* data (e.g., why share the contact's email addresses and phone numbers?). And why should the app be able to always access those data and not only in the context of selecting a transportation destination?

Third, permissions apply to application sandboxes as a whole, including all third party libs included in the application, such as analytics and advertisement libraries. Currently, it is opaque to the user which security principal (app or lib) is levering the app's privileges. This entangled trust relationship between user, app developer, and included libs can be abused by third party code. In fact, ad libs on Android have demonstrated dubious behavior [4, 6] that actively exploits their host app's privileges to violate the user's privacy by, for example, exfiltrating private information or even dynamically loading untrusted code.

Lastly, even when the user is well informed about the apps' behavior (e.g., through app descriptions, reviews, or developer Web sites), she has very limited means to adjust the permissions to her own privacy preferences. Only very few selected permissions can be dynamically revoked by the user—on several platform versions of iOS and on Android 6—but many other privacy-critical permissions cannot be revoked or fine-grained data filtering enabled.

Full-Fledged App Sandboxing on Stock Android

What is required to improve on this situation and to shift the balance of power in favor of the users is an application sandbox controlled by the user and capable of enforcing user-defined privacy policies by reliably monitoring any interaction between apps and the Android system. The sandbox solution, additionally, must be easy to install on stock Android (e.g., as an app) and must refrain from modifying the underlying platform, that is, no jailbreak/rooting or reinstallation of the operating system, which entails unlocking the device and, usually, data loss and which forms a technical barrier for most end users.

With **Boxify** [1] we present a novel concept for Android app sandboxing that fills this gap. **Boxify** is based on app virtualization that provides full-fledged app sandboxing on stock Android devices. **Boxify** provides secure access control on apps without the need for firmware alterations, root privileges, or modifications of confined apps. The key idea of our approach is to encapsulate untrusted apps in a restricted execution environment within the context of another, trusted sandbox application. By introducing a novel app virtualization environment that intercepts all interactions between the app and the system, **Boxify** is able to enforce established and new privacy-protecting policies on third party apps.

Additionally, the virtualization environment is carefully crafted to be transparent to the encapsulated app in order to keep the app agnostic about the sandbox and retain compatibility to the regular Android execution environment. By leveraging on-board security features of stock Android, we ensure that the kernel securely and automatically isolates at process-level the sandbox app from the untrusted processes to prevent compromise of the policy enforcement code.

Boxify is realized as a regular app that can be deployed on stock Android versions higher than v4.1 and causes only a negligible performance penalty for confined apps (1.6–4.8% in benchmark apps). To the best of our knowledge, **Boxify** is the first solution to introduce application virtualization to stock Android.

In the remainder of this article, we outline the **Boxify** concept and how it can benefit users to efficiently and securely protect their privacy. For a full technical description of our solution, we refer the interested reader to our USENIX Security '15 conference paper [1].

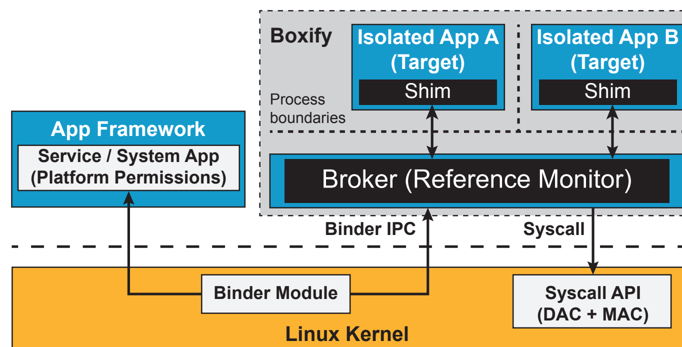


Figure 1: Architecture overview of **Boxify**

Boxify Architecture

Boxify sandboxes Android apps by dynamically loading and executing untrusted apps in one of its own processes. Thus, the untrusted application is not executed by the Android system itself but runs completely encapsulated within the runtime environment that **Boxify** provides (see Figure 1). This approach eliminates the need to modify the code of the untrusted application and works without altering the underlying OS, hence facilitating easy deployment by end users.

Boxify leverages the security provided by an on-board security feature of stock Android, called *isolated processes*, in order to isolate the untrusted code running within the context of **Boxify** by executing such code in a completely de-privileged process that has no permissions, very limited file system access, and highly constrained inter-application communication. However, Android apps are tightly integrated within the Android application framework (e.g., for application lifecycle management). With the restrictions of an isolated process in place, encapsulated apps are rendered dysfunctional.

Thus, the key challenge for **Boxify** essentially shifts from constraining the capabilities of the untrusted app to now gradually permitting I/O operations in a controlled manner in order to securely reintegrate the isolated app into the software stack. This reintegration is subject to privacy policies that govern very precisely to which functionality and data the untrusted app regains access (e.g., privacy-relevant services like location or filtering of data—like address book entries). To this end, **Boxify** creates two primary entities that run at different levels of privilege: a privileged controller process known as the **Broker** and one or more isolated processes called the **Target**(s).

Target

The **Target** hosts all untrusted code that will run inside the sandbox (see Figure 2). Each **Target** consists of a shim (**SandboxService**) that is able to dynamically load other Android

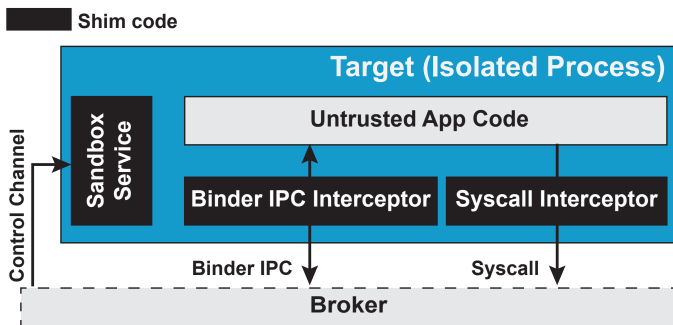


Figure 2: Components of a **Target** process

applications and execute them inside the isolated process. For the encapsulated app to interact with the system, the shim sets up interceptors that interpose system and middleware API calls.

All interaction between the app and the Android application framework occurs via IPC, and the Binder IPC Interceptor redirects all calls from the app to the application framework and other apps to the **Broker**. The IPC Interceptor does so quite efficiently by replacing all references to a central IPC service registry (**ServiceManager**) in the memory of the **Target** process with references to the IPC interface of the **Broker** process. Consequently, all calls directed to the **ServiceManager** are redirected to the **Broker** process instead, which acts as a proxy to the application framework and ensures that all subsequent interactions by the untrusted app with requested Android services are redirected to the **Broker** as well.

For system call interception, we rely on a technique called `libc` hooking. Android is a Linux-based software stack and ships with a custom C library, called *Bionic*, that acts as a wrapper around the underlying Linux kernel's system call interface. Bionic is used by default by Android user-space processes, including application processes, to issue system calls to the kernel. By hooking Bionic's `libc`, **Boxify** can efficiently intercept calls to `libc` functions and redirect these calls to a service client running in the **Target** process. This client forwards the function calls via IPC to a custom service component running in the **Broker**.

It is important to notice that both interceptors, IPC and system calls, do **not** form a security boundary but establish a compatibility layer when the code inside the sandbox needs to perform otherwise restricted I/O by forwarding the calls to the **Broker**.

Broker

The **Broker** is the main **Boxify** application process and acts as a mandatory, bi-directional proxy for all interactions between a **Target** and the system. On the one hand, the **Broker** relays all I/O operations of the **Target** that require privileges beyond the ones of the isolated process. Thus, if the encapsulated app bypasses the **Broker**, the app is limited to the extremely confined privilege set of its isolated process environment (*fail-safe defaults*). As a consequence, the **Broker** is an ideal control-flow location in our **Boxify** design to implement a policy enforcement point (*reference monitor*). To protect the **Broker** (and hence reference monitor) from malicious app code, the **Broker** runs in a separate process under a different UID than the isolated processes. This establishes a strong security boundary between the reference monitor and the untrusted code. On the other hand, the **Broker** dispatches IPC calls initiated by the system (e.g., basic lifecycle operations) to the correct **Target**.

The **Broker** is organized into three layers (see Figure 3): the **API Layer** abstracts from the concrete characteristics of the Android service IPC interfaces to provide compatibility across different Android versions. To this end, the **API Layer** bridges the semantic gap between the raw IPC transactions forwarded by the **Target** and the application framework semantics of the other layers in the **Broker** by transforming the raw Binder IPC parcels back into their high-level Java objects representation. This also facilitates the definition of more convenient and meaningful privacy policies.

The **Core Logic Layer** replicates a small subset of the functionality that Android's core system services provide. Further, this layer decides whether an Android API call is emulated using a replicated service (e.g., a mock location service) or forwarded to the pristine Android service (through the **Virtualization Layer**). The **Core Logic Layer** is therefore responsible for managing the IPC communication between different sandboxed apps (abstractly like an *"IPC switch"*). Furthermore, this layer implements the policy enforcement points for Binder IPC services and syscalls. We emulate the integration of enforcement points into pristine Android services by integrating these points into our mandatory service proxies in the **Core Logic Layer**. This allows us to instantiate security and privacy solutions (including the default Android permissions) from the area of OS security extensions, but at the application layer. Similarly, syscall policy enforcement points enforce system call policies on network and file-system operations.

The **Virtualization Layer** is responsible for translating the bi-directional communication between the Android application framework and the **Target**. This technique can be abstractly best described as an *IPC Network Address Translator*. Thus, the sandbox is transparent to the **Target,** and all interaction with the application framework from the **Target**'s perspective appears as in any regular, non-virtualized app. At the same time, the sandbox is completely opaque to the application framework, and sandboxed apps are hidden from the framework and other regular installed apps, which can only detect the presence of the **Boxify** app.

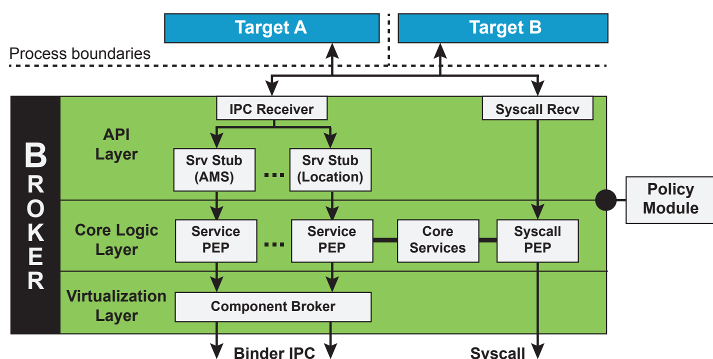


Figure 3: Architecture of the **Broker**

Use Cases

Boxify allows the instantiation of different security models from the literature on Android security extensions. In the following, we present three selected use cases on fine-grained permission control, separation of ad libraries, and domain isolation that have received attention before in the security community.

Fine-Grained Permission Control

The TISSA [14] OS security extension empowers users to flexibly control in a fine-grained manner which personal information will be accessible to applications. We reimplemented the TISSA functionality as an extension to the **Core Logic Layer** of the **Boxify Broker**. This brings TISSA's enforcement strategy to Android as an application layer-only solution that does not require the user to exchange or alter her device's firmware. To this end, we instrumented the mandatory proxies for core system services (e.g., location or telephony service) so that they can return a filtered or mock data set based on the user's privacy settings. Users can dynamically adjust their privacy preferences through a management UI added to **Boxify**. For instance, the transportation app from our motivating example could be restricted to the user's contacts' street addresses and barred from accessing their email addresses, phone numbers, etc. Similarly, access to the SMS data can be completely removed by returning only empty data sets to queries or be fine-grained, restricted to only SMS from whitelisted phone numbers (e.g., the transportation company's service numbers).

Separation of Advertisements

For monetarization of apps, app developers frequently bundle their apps with advertisement libraries. However, those libraries have been shown to exhibit very dubious and even dangerous behavior for the user's privacy [6]. To better protect the user from those unsafe practices, technical solutions [7, 11] for privilege separation have been brought forward that retrofit the Android middleware to isolate advertising libraries from their host apps and then subjugate them to a separate privacy policy.

In this spirit, we instantiate a similar solution on **Boxify** at application layer that extracts advertising libraries from apps, executes them in a separate **Target**, and reintegrates them with their host app through IPC-based inter-app communication via the **Boxify Core Logic Layer**. This is possible, since advertising libraries are by default only loosely coupled with their host application code. As a result, separate privacy policies can be applied to the ad lib sandbox on **Boxify** (e.g., preventing the ad lib from exfiltrating private information). The same technique for extracting advertising libraries from their host apps can even be applied to remove the advertising libraries in their entirety from apps (i.e., ad blocking).

Domain Isolation

Particularly for enterprise deployments, container solutions have been brought forward to separate business apps from other (untrusted) apps [2, 12]. We implemented a domain isolation solution based on **Boxify** by installing business apps into the sandbox environment. The **Core Logic Layer** of **Boxify** enables a controlled collaboration between enterprise apps, while at the same time isolating and hiding them from non-enterprise apps outside of **Boxify**.

To separate the enterprise data from the user's private data, we take advantage of the **Broker**'s ability to run separate instances of system services (e.g., address book, calendar) within the sandbox. Our **Core Logic Layer** selectively and transparently redirects data accesses by enterprise apps to the sandboxed counterparts of those providers, thus ensuring that the data is not written to publicly available resources that can be accessed by non-enterprise apps (e.g., the default address book or calendar of Android).

Alternatively, the above described domain isolation concept can be used to implement a privacy mode for end users, where untrusted apps are installed into a **Boxify** environment with system services that return empty (or fake) data sets, such as location, address book, etc. Thus, users can test untrusted apps in a safe environment without risking harm to their private data.

Security Discussion

Lastly, we identify different security shortcomings of **Boxify** and discuss potential future security primitives of stock Android that would benefit **Boxify** and defensively programmed apps in general.

Privilege Escalation

A malicious app could bypass the syscall and IPC interceptors, for instance, by statically linking `libc`. For IPC, this does not lead to a privilege escalation, since the application framework apps and services will refuse to cooperate with an isolated process. However, for syscalls, a malicious process has the entire kernel API as an attack vector and might escalate its privileges through a root or kernel exploit.

To remedy this situation, additional layers of security could be provided by the underlying kernel to further restrict untrusted processes, e.g., program tracing or `seccomp()`. This is common practice on other operating systems, and we expect such facilities to become available on future Android versions with newer kernels.

Violating Least-Privilege Principle

The **Broker** must hold the union set of all permissions required by the apps hosted by **Boxify** in order to successfully proxy calls to the Android API. Because it is hard to predict a reasonable set of permissions beforehand, the **Broker** usually holds all available permissions. This makes the **Broker** an attractive target for attacks. A very elegant solution to this problem would be a **Broker** that drops all unnecessary permissions. Unfortunately, Android does not (yet) provide a way to *selectively* drop permissions *at runtime*.

Red Pill

Even though **Boxify** is designed to be invisible to the sandboxed app, it cannot exclude the untrusted app from gathering information about its execution environment that allows the app to deduce that it is being sandboxed. A malicious app can use this knowledge to change its runtime behavior when being sandboxed and thus hide its true intentions, or it can refuse to run in a sandboxed environment. While this might lead to refused functionality, it cannot be used to escalate the app's privileges.

Conclusion

We presented the first application virtualization solution for the stock Android OS. By building on isolated processes to restrict privileges of untrusted apps and by introducing a novel app virtualization environment, we combine the strong security guarantees of OS security extensions with the deployability of application layer solutions. We implemented our solution as a regular Android app called **Boxify** and demonstrated its capability to enforce established security and privacy policies without incurring significant runtime performance overhead. To make **Boxify** more accessible to security engineers, future work will investigate programmable security APIs that allow instantiation of various use cases in the form of code modules rather than patches to **Boxify**.

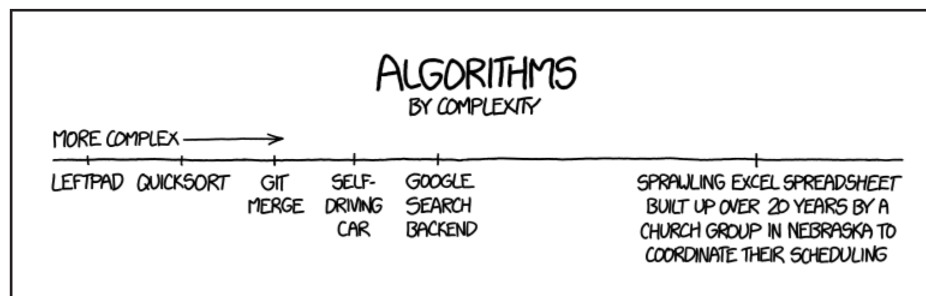
Availability

The IP and the corresponding patent of the Boxify technology are owned by the company Backes SRT, which plans to make a noncommercial version of Boxify for academic research available on their Web site (www.backes-srt.com).

Acknowledgments

This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy, and Accountability (CISPA).

XKCD



xkcd.com

References

[1] M. Backes, S. Bugiel, C. Hammer, O. Schranz, P. von Styp-Rekowsky, "Boxify: Full-Fledged App Sandboxing for Stock Android," in *Proceedings of the 24th USENIX Security Symposium (SEC '15),* 2015: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-backes.pdf.

[2] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and Lightweight Domain Isolation on Android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11),* 2011, pp. 51–62.

[3] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10),* 2010: https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Enck.pdf.

[4] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th USENIX Security Symposium (SEC '11),* 2011: http://www.enck.org/pubs/enck-sec11.pdf.

[5] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," in *IEEE Security and Privacy*, vol. 7, no. 1, 2009, pp. 50–57: http://css.csail.mit.edu/6.858/2015/readings/android.pdf.

[6] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe Exposure Analysis of Mobile In-App Advertisements," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC),* 2012: http://www4.ncsu.edu/~mcgrace/WISEC12_ADRISK.pdf.

[7] P. Pearce, A. Porter Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege Separation for Applications and Advertisers in Android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communication Security (ASIACCS),* 2012: https://www.eecs.berkeley.edu/~daw/papers/addroid-asiaccs12.pdf.

[8] A. Porter Felt, S. Egelman, and D. Wagner, "I've Got 99 Problems, but Vibration Ain't One: A Survey of Smartphone Users' Concerns," in *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM),* 2012: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-70.pdf.

[9] A. Porter Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android Permissions: User Attention, Comprehension, and Behavior," in *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS 2012),* 2012: https://blues.cs.berkeley.edu/wp-content/uploads/2014/07/a3-felt.pdf.

[10] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the Description-to-Permission Fidelity in Android Applications," in *Proceedings of the 21st ACM Conference on Computer and Communication Security (CCS '14),* 2014: http://pages.cs.wisc.edu/~vrastogi/static/papers/qrzczc14.pdf.

[11] S. Shekhar, M. Dietz, and D. Wallach, "Adsplit: Separating Smartphone Advertising from Applications," in *Proceedings of the 21st USENIX Security Symposium (SEC '12),* 2012: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final101.pdf.

[12] X. Wang, K. Sun, and Y. Wang, J. Jing, "DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS '15),* 2015: https://www.internetsociety.org/sites/default/files/02_5_1.pdf.

[13] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android Permissions Remystified: A Field Study on Contextual Integrity," in *Proceedings of the 24th USENIX Security Symposium (SEC '15),* 2015: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-wijesekera.pdf.

[14] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh, "Taming Information-Stealing Smartphone Applications (on Android)," in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST '11),* 2011: https://pdfs.semanticscholar.org/75d7/0671eee7ead26c5636fe5d1e00fef5d993b3.pdf.

Using OpenSCAP

MARTIN PREISLER

Martin Preisler works as a Software Engineer at Red Hat, Inc. He works in the Identity Management and Platform Security team, focusing on security compliance using Security Content Automation Protocol. He is the principal author of SCAP Workbench, a frequent contributor to OpenSCAP and SCAP Security Guide, and a contributor to the SCAP standard specifications. Outside of Red Hat, he likes to work on open source projects related to real-time 3D rendering and game development. mpreisle@redhat.com

Security best practices dictate that we do not run any software with known and exploitable vulnerabilities, but achieving this is difficult. While vulnerability databases do exist, they are not in formats useful for scanning file systems, much less for examining VM images and containers. I work on OpenSCAP, a tool that uses information extracted from the National Vulnerability Database [1] and security policies, and checks for vulnerabilities. oscap can also remediate, or suggest remediations, for configurations that don't meet established policies. In this article, I explain how OpenSCAP works, how to use both its GUI and command-line versions, and how you can use oscap to improve your site's security.

Ensuring proper configuration and no vulnerabilities in your production environment has become an essential part of proactive security. In the past it used to be possible to manually go over a single golden image and then deploy it en masse, but that has changed radically. Typical business deployments are now much larger than they used to be and are no longer run just using physical machines. Modern deployments are using virtual machines and containers and tend to deploy many different images. This brings new challenges to both vulnerability assessment and configuration management.

Finding Vulnerabilities

Let us say we have a deployed installation. How do we figure out whether it has any vulnerabilities? We could look at the National Vulnerability Database [1] and go over the vulnerabilities one by one, comparing affected versions with the versions of software we have installed. Unfortunately, there are two major issues with this approach. First of all, we would go crazy very soon, as this approach does not scale to even one machine let alone an infrastructure. Secondly, we would not get accurate results on most enterprise Linux operating systems such as Red Hat Enterprise Linux or SUSE Enterprise Linux. The vendors of those operating systems backport fixes for vulnerabilities into older versions of the software. This way they minimize the differences between consecutive versions of the operating systems, which is something their users really appreciate. On the other hand, this makes checking version ranges for CVEs more complex because the versions no longer match the upstream original versions.

So how do we deal with this? We need to get a vulnerability database with these backports recorded. Fortunately, vendors of enterprise operating systems are increasingly supplying a so-called "CVE feed" with corrected affected versions for each vulnerability. Still, going over them manually is a lost battle; we need an automated approach.

OpenSCAP can load the CVE feed and go over all vulnerabilities for you, detecting which vulnerabilities are in your systems.

Scanning a Physical Machine for Vulnerabilities

So how does OpenSCAP perform the scanning? First, OpenSCAP loads the given CVE database which has information about security advisories from the vendor. Then it goes over every CVE item in that database, checking the package and affected version ranges to see whether we have a version in that range and thus are affected. This works very well for official, signed packages from the vendors. The vulnerability check itself does not check checksums of the RPMs; instead most security policies check checksums of every package installed from an official source. The reasoning is that we need to check all the checksums anyway because attackers might have injected into any package with any vulnerability. To save time this is done in one go as part of the "verify RPM signatures" rule in security policies. The rule does something very similar to "rpm -Va"—it verifies that properties of installed files match package metadata. We will touch on security policies later. The criteria for determining whether we are vulnerable usually look like this:

```
<criteria operator="AND">
  <criterion comment="openssl is earlier than 0:1.0.1e-30
.el6_6.2" test_ref="oval:com.redhat.rhsa:tst:20141652019"/>
  <criterion comment="openssl is signed with Red Hat
redhatrelease2 key" test_ref="oval:com.redhat.rhsa:tst
:20140679006"/>
</criteria>
```

In the example above we are checking whether `openssl` is installed, and if it is, whether it is the Red Hat signed version and also whether it's an earlier version than the one that contains the fix for the specific vulnerability. In some cases the criteria get more complex because sometimes a vulnerability gets introduced in some version and then gets fixed in another. In this case we are checking that a package is installed, is signed by Red Hat, and is either greater than some version or earlier than another version.

Let us first show how to do a vulnerability scan on a physical machine. We will assume Red Hat Enterprise Linux 6 in our example. Each vendor publishes their CVE feed at a different location; in the case of Red Hat it is https://www.redhat.com/security/data/oval/. For Red Hat Enterprise Linux 6 specifically, we need to choose Red_Hat_Enterprise_Linux_6.xml in that directory.

```
# yum install openscap-utils
# wget
https://www.redhat.com/security/data/oval/Red_Hat
        _Enterprise_Linux_6.xml
# oscap oval eval --results results.xml --report report.html
        Red_Hat_Enterprise_Linux_6.xml
```

As oscap is executed we will see lines of each of the vulnerabilities being scanned. If the line says "false", that means we are not vulnerable. The output will look like the following:

```
Definition oval:com.redhat.rhsa:def:20160286: false
Definition oval:com.redhat.rhsa:def:20160258: false
…
```

After the scan finishes we can either look at `results.xml`, the machine readable results, or `report.html`, the human-readable HTML report. Covering the machine-readable results is outside the scope of this article. Let us instead discuss the HTML report. The report will contain several rows, one for each Red Hat Security Advisory that is being checked. The green rows are the rows we do not need to be concerned about; we are not vulnerable to the CVEs in them. The rows that are highlighted orange are the ones that our infrastructure is vulnerable to. It is important to realize that each RHSA can fix one or more CVEs, that there is no direct 1:1 mapping.

Suppose we have a vulnerability in the kernel in our infrastructure. What can we do about that? To fix the situation, we should get all the latest updates installed with `yum update`. Then we need to remove the vulnerable kernels to prevent them from being booted by accident. As long as there is at least one vulnerable kernel installed, OpenSCAP will report the vulnerability being in the infrastructure.

Scanning a Container for Vulnerabilities with oscap-docker

We could scan a container by installing the tools and security policies inside it and then running oscap. But that is impractical and goes against best practices of container deployment. Instead we want to scan containers from the host without affecting them.

There are two ways of scanning a container with OpenSCAP. Let us start with oscap-docker, which is a command-line tool wrapping the functionality of oscap. It has the same command-line arguments as oscap with the exception of the first two arguments—the mode of operation and the container or image ID. Before we can use it we need to install it; on Red Hat Enterprise Linux 7.2 it is part of the openscap-utils package.

After it is installed we can use it if we have root privileges. There are two subcommands: `container-cve` scans a running container, while `image-cve` scans a container image.

```
# oscap-docker container-cve $TARGET_ID
# oscap-docker image-cve $TARGET_ID
```

To start, we can scan a single container image: for example, the `rhel7` base image.

```
# docker pull rhel7
# oscap-docker image-cve rhel7
2016-02-25 12:07:58
URL:http://www.redhat.com/security/data/oval/com.redhat.rhsa
-all.xml.bz2 [1863765/1863765] -> "docker.6xTkgY/cve-oval.xml
.bz2" [1]
Definition oval:com.redhat.rhsa:def:20160286: false
Definition oval:com.redhat.rhsa:def:20160258: false
…
```

The lines ending with "false" are telling us that we are not vulnerable to CVEs listed in the respective Red Hat Security Advisories. If any of the lines end with "true" we are in trouble and need to update our image.

Scanning a Container for Vulnerabilities with Atomic

In case you are using Atomic for container management, you can use the atomic scan functionality instead. The advantage is that it is easier to use since it automatically manages the CVE feeds for the user. The tool makes sure you are using the right CVE feed and that it is up-to-date. If you are running Red Hat Enterprise Linux 7 and do not have the atomic command-line tool installed you need to run:

```
# yum install atomic
```

If you are on Atomic Host, the atomic command should already be available. After the atomic command is installed, you need the OpenSCAP-daemon to perform the scans. You can install it directly on the host and run it or you can download a super-privileged container (SPC) image that provides it. We will go with the SPC image in this section because it is a little bit simpler to set up. For the SPC image, we will be using the Fedora 23 OpenSCAP-daemon container image. There may be other images available in the future.

```
# atomic install openscap/openscap-daemon-f23
# atomic run openscap/openscap-daemon-f23
```

When the SPC is in place and running we can issue atomic scan commands.

Let us now look at an example of atomic scan. This time we will use a custom Red Hat Enterprise Linux 7.2 image that I created that actually has vulnerabilities. It will help us demonstrate features of the atomic scan.

```
# atomic scan 6c3a84d798dc
Container/Image     Cri     Imp     Med     Low
-----------------   ---     ---     ---     ---
6c3a84d798dc         0       0       2       0
```

As we can see, container image 6c3a84d798dc has two medium-severity vulnerabilities. How do we list them? We need to use the `--detail` argument.

```
# atomic scan --detail 6c3a84d798dc
6c3a84d798dc
 OS        : Red Hat Enterprise Linux Server release 7.2 (Maipo)
 Moderate : 2
   CVE      : RHSA-2016:0008: openssl security update (Moderate)
   CVE URL  : https://access.redhat.com/security/cve/CVE-2015-7575
   RHSA ID  : RHSA-2016:0008-00
   RHSA URL : https://rhn.redhat.com/errata/RHSA-2016-0008.html

   CVE      : RHSA-2916:0007: nss security update (Moderate)
   CVE URL  : https://access.redhat.com/security/cve/CVE-2015-7575
   RHSA ID  : RHSA-2016:0007-00
   RHSA URL : https://rhn.redhat.com/errata/RHSA-2016-0007.html
```

If we need to scan a container instead of an image, we just need to replace the ID with an ID of the container. Atomic scan also allows scanning all images, all containers, or both with a single command—`--images`, `--containers`, and `--all`, respectively.

Vulnerability Scanning for Virtual Machines

Scanning for vulnerabilities on virtual machines is technically very similar to scanning containers, but the commands are different. Instead of using oscap-docker, we can use oscap-vm to scan virtual machines. Keep in mind that the oscap-vm command is fairly new. It is available on Fedora but still not available on Red Hat Enterprise Linux 7 at the time of this writing. It is part of the openscap-utils package we have installed previously.

oscap-vm allows us to scan running or shut down virtual machines, or raw storage images. Let us look at an example:

```
# wget https://www.redhat.com/security/data/oval/Red_Hat
        _Enterprise_Linux_6.xml
# oscap-vm domain rhel6vm oval eval --results results.xml
        --report report.html Red_Hat_Enterprise_Linux_6.xml
```

Here, we are testing a Red Hat Enterprise Linux 6 virtual machine called "rhel6vm" running on the host.

Checking Configuration with OpenSCAP

So far we have only talked about vulnerabilities. We also need to make sure our infrastructure is set up in a secure way, that we have hardening in place. To do that we first need to choose a set of rules—a so-called security policy. A security policy is usually a list of rules in PDF or even printed out. Each rule usually has a description, rationale, identifiers, and some steps to check and fix the machines. The workflow with these security policies is that the auditors carry them in big binders and manually check the machines for compliance. This may be fine for small infrastructures, but it does not scale and is not cost effective.

Let us explore how to use OpenSCAP for fully automated security compliance. OpenSCAP is what has searched vulnerabilities in the first section of this article, but it was hidden under a few layers of abstraction. Now we need to interact with it more

directly so it makes sense to introduce it. OpenSCAP is an open-source implementation of SCAP 1.2, the standard for automated security compliance. You can read more about OpenSCAP at https://www.open-scap.org/. We will start by choosing a suitable security policy and profile. For the purposes of this article let us use PCI-DSS profile from SCAP Security Guide for Red Hat Enterprise Linux 7. SCAP Security Guide, or SSG, is another open-source project we will be using. SSG provides SCAP security policies for various products like Red Hat Enterprise Linux 6, 7, Fedora, CentOS, Firefox, and others. We will again start by scanning a physical machine before moving to containers. Let us log into the machine and install the necessary packages—scap-workbench and scap-security-guide. You might think that using such tools as SCAP Workbench, a graphical user interface, feels out of place in system administration. Keep in mind that SCAP Workbench lets you prepare the customized policies for a fully automated deployment in the future.

After installation has finished, we can start SCAP Workbench by clicking its icon in Applications → System Tools.

SCAP Workbench will start and ask us which content to select. Since we installed SCAP Security Guide in the previous step, we have the option to select `ssg-rhel7-ds.xml` in `/usr/share/xml/scap/ssg/content`. We will discuss how to scan using the command line only later in the article.

Once the content is loaded we will be presented with the main window of SCAP Workbench, which lists the rules that will be applied to the system. Let us select the PCI-DSS profile from the profile combo box.

After selecting the PCI-DSS profile we are all set to perform the initial scan. The only thing we need to do is click Scan, elevate privileges by typing the password, and wait a few minutes. On a default Red Hat Enterprise Linux 7.2 installation at the time of writing, the results were 31 passes and 43 fails. If we click Show Report we can see more details about our system.

At this point we can make customizations to the security policy by clicking "Customize." For example, we may want our infrastructure to be set up more strictly than PCI-DSS requires. In that case we can select additional rules to check and even increase minimum password length or other values. After we are done with the customization, we can choose File → Save as RPM, which gives us a package with our customized security policy ready-made to be deployed using Satellite 6.

What we have achieved above can be done using the command line only, with the exception of the customization.

```
# oscap xccdf eval --profile xccdf_org.ssgproject.content
      profile_pci-dss --results /tmp/results.xml --report
      /tmp/report.html /usr/share/xml/scap/ssg/content/
      ssg-rhel7-ds.xml
```

The snippet above will scan the local machine for compliance with PCI-DSS and will store results and report in `/tmp/results.xml` and `/tmp/report.html`, respectively.

Changing the Configuration to Be Compliant with OpenSCAP

If we want to change configuration of the machine to make more rules pass, we need to check the Remediate checkbox in SCAP Workbench and click Scan again. If remediation is enabled, SCAP Workbench will go over the rules figuring out which are passing and which are failing.

Then for each failing rule it will run a so-called remediation—code that automatically fixes the configuration—and then check the rule again. In case the rule is now passing, SCAP Workbench will declare the rule as *fixed*. If everything worked smoothly, our Red Hat Enterprise Linux 7 installation should report no failed rules.

If we cannot use the GUI, we can do the above using the command line only:

```
# oscap xccdf eval --profile xccdf_org.ssgproject.content
      _profile_pci-dss --remediate --results /tmp/results.xml
      --report /tmp/report.html /usr/share/xml/scap/ssg/content
      /ssg-rhel7-ds.xml
```

The important difference from scanning is the `--remediate` option. This instructs oscap to run remediation scripts on every failed check.

Keep in mind that automated remediations can be dangerous and cannot be undone! They can break some of the functionality of deployed infrastructure! We recommend testing remediations on nonproduction machines before deployment.

Very likely you are running a configuration management system, such as Puppet, Chef, or Ansible. In this case the remediations will still work but the configuration management systems may override them, putting your systems out of compliance! Instead of running the remediations, it may be more valuable to see their code and adapt the settings of the configuration systems accordingly. To generate a list of fixes instead of running them, run the following:

```
$ oscap xccdf generate fix --result-id xccdf_org.open-scap
      _testresult_xccdf_org.ssgproject.content_profile_pci
      _dss /tmp/results.xml
```

The `result-id` will be correct if you ran the PCI-DSS evaluation we have just discussed. In case you used a different profile, look into the `/tmp/results.xml` file, find the `<TestResult>` element, and use its id attribute.

The above will output a shell script into stdout with all the changes OpenSCAP would make to your system if you ran the remediation.

Container Security Compliance

Now we can scan the configuration of local and remote machines, but how do we deal with containers? We could scan them as a remote machine but that would require installing SSH and openscap-scanner inside, which is impractical. Instead, let us look at how to scan containers from the host. We will need the oscap-docker tool, which is part of the openscap-utils package.

We recommend running `oscap-docker --help` to explore its capabilities. It can operate in four different modes. We have already seen how it can scan containers for vulnerabilities, so we will skip over the image-cve and container-cve modes in this section. Instead we will scan a container image for security compliance.

```
# oscap-docker image $IMAGE_ID xccdf eval --profile
      xccdf_org.ssgproject.content_profile_common
      --results /tmp/results.xml --report /tmp/report.html
      /usr/share/xml/scap/ssg/content/ssg-rhel7-ds.xml
```

These command-line arguments should look familiar. Apart from the first two, we are using the same command-line arguments as with the oscap tool. Instead of using the PCI-DSS profile in this section, we will use a new profile called the common profile. It is less focused on the financial industry and contains rules checking common security practices instead.

With the command-line snippet above we are doing roughly the same thing as with SCAP Workbench and oscap earlier in this article; we have scanned a container image with the common profile with content coming from the SCAP Security Guide project for Red Hat Enterprise Linux 7. The results are stored in `/tmp/results.xml`, and the HTML report is stored in `/tmp/report.html`.

Virtual Machine Security Compliance

As is the case with vulnerability assessment, scanning for security compliance is very similar between containers and virtual machines. Instead of using the `oscap-docker` command, we need to use `oscap-vm`.

```
# oscap-vm domain rhel7vm xccdf eval --profile
      xccdf_org.ssgproject.content_profile_common --results
      /tmp/results.xml --report /tmp/report.html
      /usr/share/xml/scap/ssg/content/ssg-rhel7-ds.xml
```

The semantics are the same between `oscap-vm` and `oscap-docker`. The key difference is under the hood. `oscap-vm` uses guestmount to inspect virtual machines instead of the atomic mount mechanism inside `oscap-docker`.

Offline Evaluation Advantages and Limitations

What we have used in the previous two sections is called offline SCAP evaluation. When we are scanning local or remote machines, we are using the normal online evaluation. When scanning containers and virtual machines from the host, we are using the offline evaluation. The difference between offline and online evaluation is that the latter has access to all running processes and can do runtime checks. That means that it can, for example, ask systemd about information about a unit. Offline scanning does not have access to the running system; it is exploring the file system mounted somewhere in read-only mode.

This has advantages and disadvantages. One advantage of offline scanning is that we can scan a running container or virtual machine without any risk of affecting them. On the other hand, we cannot perform some of the runtime checks like checking which processes are running. We also cannot fix systems in the offline evaluation mode since the file systems are mounted read-only. That is not a limitation of the offline mode but rather its implementation in OpenSCAP. There is an outstanding feature request to fix this [2].

Conclusion

Using OpenSCAP helps businesses prevent vulnerabilities and insecure configuration settings in their infrastructure. In this article we have explored how to use OpenSCAP for physical machines as well as for virtual machines and containers. Mixing automated SCAP remediations with configuration management systems proved difficult but can be handled by going through the remediation steps and adapting configuration systems accordingly. Using SCAP for containers and virtual machines requires a new approach called offline evaluation. While limited and fairly new, it is proving useful for practical container and virtual machine evaluation.

Acknowledgments

I would like to thank Jan Černý, Rik Farrow, Yoana Ruseva, and anonymous reviewers for helping me with this article.

References

[1] National Vulnerability Database: https://nvd.nist.gov/.

[2] Feature request to add offline mode repair: https://fedorahosted.org/openscap/ticket/467.

Interview with Nick Weaver

RIK FARROW

Nick Weaver received a BA in astrophysics and computer science in 1995, and his PhD in computer science in 2003 from the University of California at Berkeley. Although his dissertation was on novel FPGA architectures, he was also highly interested in computer security, including postulating the possibility of very fast computer worms in 2001. In 2003, Nick joined ICSI, first as a postdoc and then as a staff researcher. His primary research focus is network security—notably, worms, botnets, and other Internet-scale attacks—and network measurement. Other interest areas have included both hardware acceleration and software parallelization of network intrusion detection, defenses for DNS resolvers, and tools for detecting ISP-introduced manipulations of a user's network connection. nweaver@icsi.berkeley.edu

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

I attended the first Enigma conference in January 2016 and was pleased by the quality of the talks as well as by the depth managed by speakers, who had just 20 minutes to make their points. While I had lots of favorite talks, such as those by Stefan Savage and Ron Rivest [1], I found myself wanting to dig a bit deeper into Nick Weaver's talk.

Nick talked about "The Golden Age of Bulk Surveillance," but in my mind his talk was really about what can be done with captured data. Nick walked the audience through the process of de-anonymization and just how easy it is when you have most traffic, particularly the metadata for the traffic [2].

Rik: Your PhD dissertation involved FPGAs, but somehow you got interested in TCP/IP and security at the network layer. Can you explain how that happened?

Nick: Well, during my dissertation I participated in the NIST Advanced Encryption Standard (AES) process, evaluating how easy the various five final candidates would map to high performance hardware.

But overall it was Code Red. When Code Red hit, the worldwide reaction was "13 hours, god that's fast." A friend of mine, Michael Constant, and I were sucking down sodas and going "13 hours, god that's slow. We're computer people, we should be able to do it faster than that."

So we started sketching out concepts that became the "Warhol Worm" concept: how to efficiently infect all the vulnerable machines in 15 minutes [3]. Once that happened, then you have to think about various automated defenses. So I came into security during the dawn of the "Worm Era," where high speed, broad malcode became a thing.

Rik: There really was something like the Warhol Worm, a worm that abused some Windows RPC over UDP, wasn't there?

Nick: SQL Slammer which targeted MS-SQL. This was how the UCSD/ICSI collaboration formed: Slammer hit, spread worldwide in ~10 minutes, and it was a rush analysis between Vern Paxson, Stuart Staniford, myself, Stefan Savage, Colleen Shannon, and David Moore at CAIDA to figure out just Whisky Tango Foxtrot happened [4].

It actually used a much simpler scheme than the one I proposed; it just sent infectious UDP packets at line rate. We've since seen a similar one with the "Witty" worm [5].

Rik: In your Enigma talk, you mentioned Bro [6]. Is that something you use in your work at ICSI?

Nick: I'm mostly just a Bro user. I've had a part in some high level and researchy stuff with it (e.g., hardware acceleration and parallelization), but for the most part I'm just a user who happens to work down the hall from the developers.

Interview with Nick Weaver

My primary focus at ICSI is a catch-all of network measurement and network security, including, in the past, malcode, packet injection, network monitoring, and network mapping. Additionally, "security@ICSI" is composed of not just system administrators but myself and other researchers, so we end up having to do our own incident response.

When the Snowden slides came out, I looked at them and went "Well, they pretty much do what we do."

Rik: So your Enigma presentation really comes out of a combination of the work you do and the Snowden slides?

Nick: Yep. I looked at the slides and saw basically what I do with more money.

If you discount my civil liberties streak, my abysmal management skills, and my blanket refusal to get a security clearance, I would make a great technical director for the NSA. If anything, what often frustrates me the most is where the NSA is inefficient or inelegant. If the NSA is going to try to spy on the rest of the world and annoy everyone else in the process, at least they should do a good (and less expensive) job!

Rik: It sounds like the way you work at ICSI has taught you how to put together metadata. You also put together a monitoring device that collects data but fits inside a lunchbox. Tell us about that.

Nick: I don't personally work at LBNL (Lawrence Berkeley, not Lawrence Livermore), but Vern does, and I'm fairly familiar with how they operate; I've been involved in multiple studies where I or someone else have some analysis that should be performed, and Vern then takes the analysis and runs it on LBNL's data.

Thus, for example, in "The Matter of Heartbleed," the LBNL bulk recording was used to verify that Heartbleed wasn't exploited by someone against LBNL before public disclosure.

The lunchbox is largely taking advantage of the IDS flow's scalability not just up (to 100 Gbps+ installations) but down: you can run on slower links with cheaper hardware. So the bulk of the work in making the lunchbox was simply deciding what additional analyses to run. One of my minor to-dos is to see how well I can get things to run on the latest Raspberry Pi 3.

In looking at the Snowden documentation, the big difference between what the NSA does on the network is focus on users, not just machines. Thus they have some specific metadata they want to extract to identify "who" rather than "what" is on the network, and most of the work in the lunchbox was focusing on adding these analyses and creating a snazzy Web interface: it was more work for me to figure out enough Bootstrap to make it look good than it was to figure out how, through passive analysis, to identify people in the network traffic.

Rik: Give us a paragraph about the hardware in your lunchbox.

Nick: The hardware is simply an Intel NUC (now two generations old) combined with a really nice DualComm Ethernet Tap/Switch. I admit I gilded the NUC (16 GB RAM and a 120 GB SSD), and you can buy cheaper taps (e.g., SharkTap) that could substantially reduce the cost.

Everything else is simply stuff needed to create an access point for people to connect to since I don't want to tap people who don't consent to be monitored.

References

[1] Enigma YouTube channel: https://www.youtube.com/channel/UCIdV7bE97mSPTH1mOi_yUrw.

[2] Nicholas Weaver, "The Golden Age of Bulk Surveillance": https://www.youtube.com/watch?v=zqnKdGnzoh0.

[3] Stuart Staniford, Vern Paxson, and Nicholas Weaver, "How to 0wn the Internet in Your Spare Time," in *Proceedings of the 11th USENIX Security Symposium (USENIX Security '02)*, 2002: http://www.icir.org/vern/papers/cdc-usenix-sec02/.

[4] Inside the Slammer Worm: http://www.icsi.berkeley.edu/pubs/networking/insidetheslammerworm03.pdf.

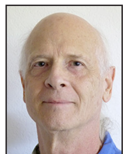
[5] Abhishek Kumar, Vern Paxson, and Nicholas Weaver, "Exploiting Underlying Structure for Detailed Reconstruction of an Internet-Scale Event," in *Proceedings of the Internet Measurement Conference 2005 (IMC 2005)*: http://www.icir.org/vern/papers/witty-imc05.pdf.

[6] The Bro Network Security Monitor: https://www.bro.org/.

Interview with Peter Gutmann

RIK FARROW

Peter Gutmann is a researcher in the Department of Computer Science at the University of Auckland working on design and analysis of cryptographic security architectures and security usability. He helped write the popular PGP encryption package, has authored a number of papers and RFCs on security and encryption, and is the author of the open source cryptlib security toolkit, *Cryptographic Security Architecture: Design and Verification* (Springer, 2003), and an upcoming book on security engineering. In his spare time he pokes holes in whatever security systems and mechanisms catch his attention and grumbles about the lack of consideration of human factors in designing security systems. pgut001@cs.auckland.ac.nz



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

I probably first met Peter Gutmann at a USENIX Security conference. I really don't remember which one, but Peter was also a friend of an acquaintance, which led to the usual social lunches during conferences.

Peter seemed like an "odd bird," and he's actually a Kiwi, a person from New Zealand, making that part true enough. But I always enjoyed talking with him.

I recently had read some of his postings about the failure of a very prominent crypto library (think Heartbleed) and thought now would be a good time for an interview, one that would allow me to ask some questions about programming and cryptography, and get answers that I felt most people would understand.

During the interview with Peter, conducted over email, the person I had asked to write about the AFL fuzzing tool dropped out, and Peter mentioned that he had given a talk about AFL at a Kiwi-con and was willing to write about AFL as well. As if this wasn't enough, I had asked Peter to review a book about iOS security well before any of this started (it takes months to get publishers to ship books to New Zealand), so Peter has almost as many pieces in this issue as I do.

Rik: Your first USENIX paper [1] had to do with remanence: that data written to hard drives and later erased or overwritten could often still be recovered. I think you were working for IBM at the time, and I imagine you had access to some unusual/sophisticated hardware in order to image partial tracks on disk drive platters.

Peter: Actually that was written some years before I was at IBM. I communicated via email with a few people who had worked in the area for general details on what was involved in reading disk tracks with MFMs and stories about deletion (and lack thereof) and heard a few interesting stories, but that sort of thing would have required access to pretty specialized hardware to do. By "specialized" I mean "not in your standard lab" but readily available to hard-drive manufacturers. That's how the sampling-scope drive read that I mention in the paper was done, something that was revived in 2011 to recover data off an old Cray-1 disk pack [2].

Rik: Did you work in New York for IBM? What did you do there?

Peter: I was at Watson Labs in Hawthorne (OK, "the IBM Thomas J. Watson Research Center," an offshoot of the main one in Yorktown Heights) as a visiting scientist working on my thesis. The idea is that IBM gets people in from all over to work there for a while with the hope that eventually they decide to stay. It was a fantastic place to work; if you ever had a question about something, there was a good chance that an expert on the topic was just a few doors down the hallway. Not to mention access to their extensive library, and people who had been in the industry for years (or decades). This was mostly before archives of paper journals and conference proceedings were scanned and put online (and some were never put online), so a lot of the background material in the thesis was gathered from there.

Rik: In other work, you programmed a library called cryptlib [3]. Why did you decide to create your own cryptographic library when others were available at the time?

Peter: cryptlib was from 1995, when what was around was mostly Eric Young's libdes. The underlying code goes back even further, to about 1992 or 1993 in the HP ACK archiver (I should have trademarked that name, given its reuse in HTTP 2), which offered digital signatures and public-key encryption of archives and assorted other things, and that was based on work on PGP 2 from even earlier.

Initially it was just something I did for fun, but then in about 1997 I was persuaded to license it commercially. Before that I'd refused to take money for it, which caused problems with some users because they needed a commercially supported product and not just open-source throw-it-over-the-wall. So it's dual-licensed under the Sleepycat license: you can use it as open source or commercially supported if you need that.

Rik: So you have firsthand experience with supporting a software package that has a dual open source/commercial license. Not many people do that on their own. What are the pros and cons?

Peter: There are pros and cons to being OSS and being commercial. OSS is obviously free, but it's also often throw-it-over-a-wall stuff; if you have a problem with it then you google Stack Overflow or post to a mailing list and hope that at some point someone volunteers to help you. Lots of commercial organizations can't work that way; for starters, since their work is security-related you need an NDA. Then you need commercial support with a guaranteed response time. If [the organization] has a problem, they want to get someone on the phone or on-site who can walk them through dealing with it; they want a standard commercial license (which often boils down to them knowing that there's someone they can sue if things go wrong); and so on. In broader terms, they need to deal with risk management: is there a commercial entity behind this? will they still be around in a year's time? will it be supported in 10 years' time? and again, in general, what do we do when things go wrong?

Having a commercial option has been really useful. With open source you do essentially just throw the code over the wall and sometimes you get feedback, but mostly you don't, so there's little opportunity to enhance the code based on users' needs because if someone has a problem you rarely hear about it; they either patch in a "fix" themselves or go elsewhere. Commercial users, since they have a guarantee of support, will come to you with issues, and so you can then use that to improve the code and work with them to solve the problem. Virtually all commercial users have agreed to having specific fixes put into the main, supported code base, because the last thing they want is to have to deal with a custom version for the rest of eternity. That's kind of weird really; in the OSS world, everyone seems to be happy to fork off their own special-snowflake version with their own code in it, while the commercial users want a single, stable, supported code base and are OK with sharing the code changes.

The much longer answer can be found at [4].

Rik: The Internet of Things, IoT, looks like it will include anything with a computer that is not a personal computer or device, or a server, and that is connected to the Internet. To my mind, that poses certain very real threats: to the security of the device itself and to the security of the data the device collects and transmits. You have been working with cryptography for over 20 years. Do you think that people can develop reasonable ways to handle cryptography on these low-powered devices? We still haven't solved key management on the larger devices we are already using.

Peter: We haven't even solved basic crypto on these devices. This is a bit of a pet peeve of mine. It's scary the number of times I've seen people on security mailing lists announce that in the future we'll all have infinite CPU and RAM and therefore can design arbitrarily baroque and complicated protocols and don't have to worry about resource constraints. Only last week someone pointed out that the just-appeared Raspberry Pi 3 has *X* resources and so we don't have to worry about resource-constrained embedded anymore. A Raspberry Pi of any kind, including the Pi Zero, isn't an embedded device, it's a PC. So is any smartphone, tablet, router, WiFi access point, and a long list of other items. Citing Moore's Law won't help because a significant portion of the market uses it to make things cheaper rather than faster, so that performance stays constant while cost goes down rather than the usual PC equilibrium of cost staying constant while performance goes up.

An embedded device is something like a Cortex M3, or more recently the M0, a 32-bit CPU perhaps clocked at a blazing 40 or even 70 MHz, with something like 256-kB flash and 32-kB RAM. Ten years ago the state of the art was a Cortex M3, while today it's a cheaper Cortex M3, and in 10 years' time it'll still be a cheaper M3 (or possibly an M0+++ by then). No standard security protocol will run on that. A week ago I got to review two proposed ISO standards for IoT in which a bunch of networking engineers tried to invent some sort of crypto mechanism that makes WEP look like a model of good design, because the obvious candidates TLS and SSH are far too bloated to work for them. It's not their fault; they're networking engineers and shouldn't be expected to have to do this, but the crypto community just assumes infinite resources and goes from there. So we've got a desperate need to secure IoT but no widely accepted standardized protocol that works for it.

This is already a problem with smart meters because regulators imposed requirements for certificate-based signed messaging and updates onto CPUs like TI MSP430s, Motorola ColdFires, and ARM Cortex-Ms, and some clocked as high as 16 MHz and with as much as 32 kB of RAM (for everything, not just the crypto). The solution with smart meters was to cut corners as much as possible in order to make things fit, skipping certificate

verification, assuming hardcoded public keys, and various other measures that are destined to become entertaining Black Hat or DEFCON presentations in the future.

That's a really short version of what could turn into a really long answer. I haven't even touched on the problem of dealing with the fact that these devices will need to work in rough environments where the hardware will experience faults, while the fashion seems to be to move towards extremely fault-prone algorithms like ECC crypto (where almost any kind of fault tends to result in the private key being leaked) and GCM-mode encryption, where a single fault, failing to increment the counter value or change the IV, results in a total, catastrophic loss of security. So that's been another ongoing project: since embedded devices are going to experience faults, how resistant can you make your crypto to random (or perhaps deliberate, maliciously induced) faults?

Rik: I noticed that cryptlib is written in C. Why not use a "safe" language, one with built-in safeguards against exploitation?

Peter: Whether C is safe or not depends on what you mean by "safe." In many high-assurance/safety-critical applications, C is regarded as safe, and languages like C++ and Java are unsafe, because with C you can establish pretty clear correspondence between the C code and the resulting binary, while with higher-level languages you can never really be sure what's going to happen.

The FAA, for example, one organization that really cares about safety, has spent about 10 years trying to develop guidelines for the safe use of OO languages (typically C++ as a follow-on from C), but is still having problems dealing with dynamic dispatch, multiple inheritance, polymorphism, overloading and method resolution, and other aspects of OO programming (see the DO-332 supplement to the DO-178C avionics software standard for more on this).

When you're operating in environments where you can't have recursion (you could run out of stack), you can't have dynamic memory allocation (it leads to nondeterministic behavior in the program), and you need to perform worst-case execution time (WCET) analysis on every routine to make sure it doesn't block, or stall, time-critical code; the less fancy high-level stuff your language has, the better.

Another thing about C is that the language is simple enough to have a huge amount of tool support available. I was recently re-reading Les Hatton's *Safer C*, a seminal book [5] from the mid-'90s, and even then he was comparing criticisms of C that were mostly based on K&R with the then-current analysis tools that went way beyond what the standard said in terms of code checking.

Things haven't stood still since then. You've now got incredibly sophisticated tools like Microsoft's PREfast that can treat C almost like Pascal or Ada. For example, they'll tell you that a variable that you've said has the range 0...1000 has been assigned a value of 1001 (or whatever). That's something that looks like C on reading, but which can be analyzed by the dev tools as if it had Pascal or Ada's type-checking.

The reasoning that some of these tools can apply is phenomenal. The clang analyzer, part of the LLVM compiler suite, can do things like tell you that if you enter this loop and take these code branches, then after going through five iterations this unexpected result (e.g., a pointer value being null) will occur (commercial tools like Coverity do this too, but in even greater detail). That's something that no human would be able to detect, and that testing probably wouldn't ever turn up either because you may need to go through 20 or 30 distinct steps to get to that point.

So a programming language is more than just something to translate into object code, it's the sum of the tools available that support it. Consider, for example, the use of the AFL fuzzer that I talk about in the AFL article on page 11 in this issue. That uses compiler-based instrumentation to detect memory issues with the address sanitizer ASAN, not just out-of-bounds accesses but other problems, like use of uninitialized memory and so on, and more instrumentation to do execution-path analysis to maximize code coverage by the fuzzer. Now imagine trying to do that with a JVM, where something outside your control (the virtual machine) is dealing with most of the stuff that's exposed in C. How would you fuzz that? You end up either missing a lot of the stuff that needs to be fuzzed, or fuzzing the JVM itself rather than the program you want to check.

References

[1] Peter Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," in *Proceedings of the Sixth USENIX Security Symposium,* 1996: https://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html.

[2] Recovering data from an ancient disk pack: http://www.chrisfenton.com/cray-1-digital-archeology/.

[3] Cryptlib: https://www.cs.auckland.ac.nz/~pgut001/cryptlib/.

[4] "Self-Sustaining Open Source Software Development," Conference for Unix, Linux and Open Source Professionals (AUUG2005), slide deck: https://www.cs.auckland.ac.nz/~pgut001/pubs/oss_development.pdf.

[5] Les Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems* (McGraw-Hill International Series in Software Engineering, 1995).

[6] Peter Gutmann's home page, with lots more pointers to slide decks and other materials: https://www.cs.auckland.ac.nz/~pgut001/.

FAST '17: 15th USENIX Conference on File and Storage Technologies

February 27–March 2, 2017 • Santa Clara, CA

Sponsored by USENIX, the Advanced Computing Systems Association

Important Dates

- Paper submissions due: **Tuesday, September 27, 2016, 9:00 p.m. PDT**
- Tutorial submissions due: **Tuesday, September 27, 2016, 9:00 p.m. PDT**
- Notification to authors: **Monday, December 12, 2016**
- Final paper files due: **Tuesday, January 31, 2017**

Conference Organizers

Program Co-Chairs

Geoff Kuenning, *Harvey Mudd College*
Carl Waldspurger, *CloudPhysics*

Program Committee

TBA

Steering Committee

Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
William J. Bolosky, *Microsoft Research*
Angela Demke Brown, *University of Toronto*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Carnegie Mellon University and Panasas, Inc.*
Casey Henderson, *USENIX Association*
Kimberly Keeton, *HP Labs*
Florentina Popovici, *Google*
Erik Riedel, *EMC*
Jiri Schindler, *SimpliVity*
Bianca Schroeder, *University of Toronto*
Margo Seltzer, *Harvard University and Oracle*
Keith A. Smith, *NetApp*
Eno Thereska, *Confluent and Imperial College London*
Ric Wheeler, *Red Hat*
Erez Zadok, *Stony Brook University*
Yuanyuan Zhou, *University of California, San Diego*

Overview

The 15th USENIX Conference on File and Storage Technologies (FAST '17) brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. The program committee will interpret "storage systems" broadly; everything from low-level storage devices to information management is of interest. The conference will consist of technical presentations including refereed papers, Work-in-Progress (WiP) reports, poster sessions, and tutorials.

FAST accepts both full-length and short papers. Both types of submissions are reviewed to the same standards and differ primarily in the scope of the ideas expressed. Short papers are limited to half the space of full-length papers. The program committee will not accept a full paper on the condition that it is cut down to fit in the short paper page limit, nor will it invite short papers to be extended to full length. Submissions will be considered only in the category in which they are submitted.

Topics

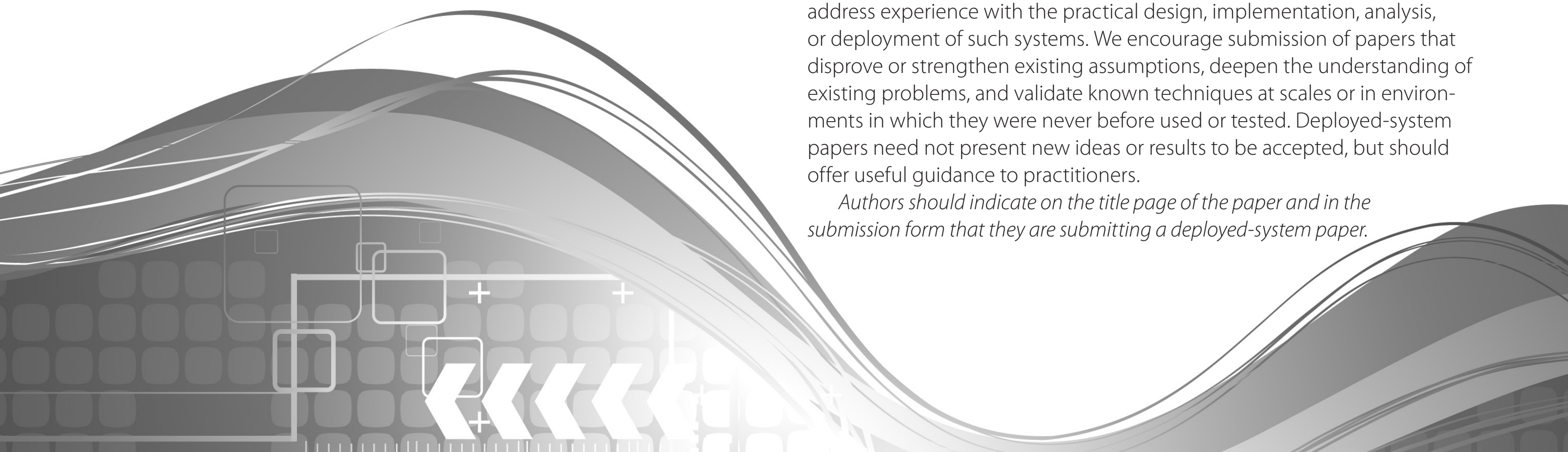
Topics of interest include but are not limited to:

- Archival storage systems
- Auditing and provenance
- Caching, replication, and consistency
- Cloud storage
- Data deduplication
- Database storage
- Distributed storage (wide-area, grid, peer-to-peer)
- Empirical evaluation of storage systems
- Experience with deployed systems
- File system design
- High-performance file systems
- Key-value and NoSQL storage
- Memory-only storage systems
- Mobile, personal, and home storage
- Parallel I/O and storage systems
- Power-aware storage architectures
- RAID and erasure coding
- Reliability, availability, and disaster tolerance
- Search and data retrieval
- Solid state storage technologies and uses (e.g., flash, byte-addressable NVM)
- Storage management
- Storage networking
- Storage performance and QoS
- Storage security
- The challenges of big data and data sciences

New in 2017! Deployed Systems

In addition to papers that describe original research, FAST '17 also solicits papers that describe large-scale, operational systems. Such papers should address experience with the practical design, implementation, analysis, or deployment of such systems. We encourage submission of papers that disprove or strengthen existing assumptions, deepen the understanding of existing problems, and validate known techniques at scales or in environments in which they were never before used or tested. Deployed-system papers need not present new ideas or results to be accepted, but should offer useful guidance to practitioners.

Authors should indicate on the title page of the paper and in the submission form that they are submitting a deployed-system paper.



Submission Instructions

Please submit full and short paper submissions (no extended abstracts) by 9:00 p.m. PDT on September 27, 2016, in PDF format via the Web submission form on the FAST '17 Web site, www.usenix.org/fast17/cfp. Do not email submissions.

- The complete submission must be no longer than 12 pages for full papers and 6 pages for short papers, excluding references. The program committee will value conciseness, so if an idea can be expressed in fewer pages than the limit, please do so. Supplemental material may be appended to the paper without limit; however the reviewers are not required to read such material or consider it in making their decision. Any material that should be considered to properly judge the paper for acceptance or rejection is not supplemental and will apply to the page limit. Papers should be typeset on U.S. letter-sized pages in two-column format in 10-point Times Roman type on 12-point leading (single-spaced), with the text block being no more than 6.5" wide by 9" deep. Labels, captions, and other text in figures, graphs, and tables must use reasonable font sizes that, as printed, do not require extra magnification to be legible. Because references do not count against the page limit, they should not be set in a smaller font. **Submissions that violate any of these restrictions will not be reviewed.** The limits will be interpreted strictly. No extensions will be given for reformatting.
- Templates and sample first pages (two-column format) for Microsoft Word and LaTeX are available on the USENIX templates page, www.usenix.org/templates-conference-papers.
- **Authors must not be identified in the submissions, either explicitly or by implication.** When it is necessary to cite your own work, cite it as if it were written by a third party. Do not say "reference removed for blind review." Any supplemental material must also be anonymized.
- 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.
- If you are uncertain whether your submission meets USENIX's guidelines, please contact the program co-chairs, fast17chairs@usenix.org, or the USENIX office, submissionspolicy@usenix.org.
- Papers accompanied by nondisclosure agreement forms will not be considered.

Short papers present a complete and evaluated idea that does not need 12 pages to be appreciated. Short papers are not workshop papers or work-in-progress papers. The idea in a short paper needs to be formulated concisely and evaluated, and conclusions need to be drawn from it, just like in a full-length paper.

The program committee and external reviewers will judge papers on technical merit, significance, relevance, and presentation. A good research paper will demonstrate that the authors:

- are attacking a significant problem,
- have devised an interesting, compelling solution,
- have demonstrated the practicality and benefits of the solution,
- have drawn appropriate conclusions using sound experimental methods,
- have clearly described what they have done, and
- have clearly articulated the advances beyond previous work.

A good deployed-system paper will demonstrate that the authors:

- are describing an operational system that is of wide interest,
- have addressed the practicality of the system in more than one real-world environment, especially at large scale,
- have clearly explained the implementation of the system,
- have discussed practical problems encountered in production, and
- have carefully evaluated the system with good statistical techniques.

Moreover, program committee members, USENIX, and the reading community generally value a paper more highly if it clearly defines and is accompanied by assets not previously available. These assets may include traces, original data, source code, or tools developed as part of the submitted work.

Blind reviewing of all papers will be done by the program committee, assisted by outside referees when necessary. Each accepted paper will be shepherded through an editorial review process by a member of the program committee.

Authors will be notified of paper acceptance or rejection no later than Monday, December 12, 2016. If your paper is accepted and you need an invitation letter to apply for a visa to attend the conference, please contact conference@usenix.org as soon as possible. (Visa applications can take at least 30 working days to process.) Please identify yourself as a presenter and include your mailing address in your email.

All papers will be available online to registered attendees no earlier than Tuesday, January 31, 2017. 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 first day of the main conference, February 28, 2017. Accepted submissions will be treated as confidential prior to publication on the USENIX FAST '17 Web site; rejected submissions will be permanently treated as confidential.

By submitting a paper, you agree that at least one of the authors will attend the conference to present it. If the conference registration fee will pose a hardship for the presenter of the accepted paper, please contact conference@usenix.org.

If you need a bigger testbed for the work that you will submit to FAST '17, see PRObE at www.nmc-probe.org.

Best Paper Awards

Awards will be given for the best paper(s) at the conference. A small, selected set of papers will be forwarded for publication in ACM *Transactions on Storage* (TOS) via a fast-path editorial process. Both full and short papers will be considered.

Test of Time Award

We will award a FAST paper from a conference at least 10 years earlier with the "Test of Time" award in recognition of its lasting impact on the field.

Work-in-Progress Reports and Poster Sessions

The FAST technical sessions will include a slot for short Work-in-Progress (WiP) reports presenting preliminary results and opinion statements. We are particularly interested in presentations of student work and topics that will provoke informative debate. While WiP proposals will be evaluated for appropriateness, they are not peer reviewed in the same sense that papers are. We will also hold poster sessions each evening. WiP submissions will automatically be considered for a poster slot, and authors of all accepted full papers will be asked to present a poster on their paper. Other poster submissions are very welcome. Please see the Call for Posters and WiPs, which will be available soon, for submission information.

Birds-of-a-Feather Sessions

Birds-of-a-Feather sessions (BoFs) are informal gatherings held in the evenings and organized by attendees interested in a particular topic. BoFs may be scheduled in advance by emailing the Conference Department at bofs@usenix.org. BoFs may also be scheduled at the conference.

Tutorial Sessions

Tutorial sessions will be held on February 27, 2017. Please submit tutorial proposals to fasttutorials@usenix.org by 9:00 p.m. PDT on September 27, 2016.

Registration Materials

Complete program and registration information will be available in December 2016 on the conference Web site.

Rev. 2/21/16

STORAGE

Serving Data to the Lunatic Fringe

The Evolution of HPC Storage

JOHN BENT, BRAD SETTLEMYER, AND GARY GRIDER

John Bent. Making supercomputers superer for over a decade. At Seagate Government Solutions. john.bent@seagategov.com.
John Bent.

Brad Settlemyer is a Storage Systems Researcher and Systems Programmer specializing in high performance computing. He works as a research scientist in Los Alamos National Laboratory's Systems Integration group. He has published papers on emerging storage systems, long distance data movement, network modeling, and storage system algorithms. bws@lanl.gov

As Division Leader of the High Performance Computing (HPC) Division at Los Alamos National Laboratory, Gary Grider is responsible for all aspects of high performance computing technologies and deployment at Los Alamos. Gary is also the US Department of Energy Exascale Storage, I/O, and Data Management National Co-Coordinator. Gary has 30 active patents/applications in the data storage area and has been working in HPC and HPC-related storage since 1984. ggrider@lanl.gov

Before the advent of Big Data, the largest storage systems in the world were found almost exclusively within high performance computing centers such as those found at US Department of Energy national laboratories. However, these systems are now dwarfed by large datacenters such as those run by Google and Amazon. Although HPC storage systems are no longer the largest in terms of total capacity, they do exhibit the largest degree of concurrent write access to shared data. In this article, we will explain why HPC applications must necessarily exhibit this degree of concurrency and the unique HPC storage architectures required to support them.

Computing for Scientific Discovery

High performance computing (HPC) has radically altered how the scientific method is used to aid in scientific discovery and has enabled the development of scientific theories that were previously unimaginable. Difficult to observe phenomena, such as galaxy collisions and quantum particle interactions, are now routinely simulated on the world's largest supercomputers, and large-scale scientific simulation has dramatically decreased the time between hypothesis and experimental analysis. As scientists increasingly use simulation for discovery in emerging fields such as climatology and nuclear fusion, demand is driving the growth of HPC platforms capable of supporting ever-increasing levels of fidelity and accuracy. Extreme-scale HPC platforms (i.e., supercomputers), such as Oak Ridge National Laboratory's Titan or Los Alamos National Laboratory's Trinity, incorporate tens of thousands of processors, memory modules, and storage devices into a single system to better support simulation science. Researchers at universities and national laboratories are continuously striving to develop algorithms to fully utilize these increasingly powerful and complex supercomputers.

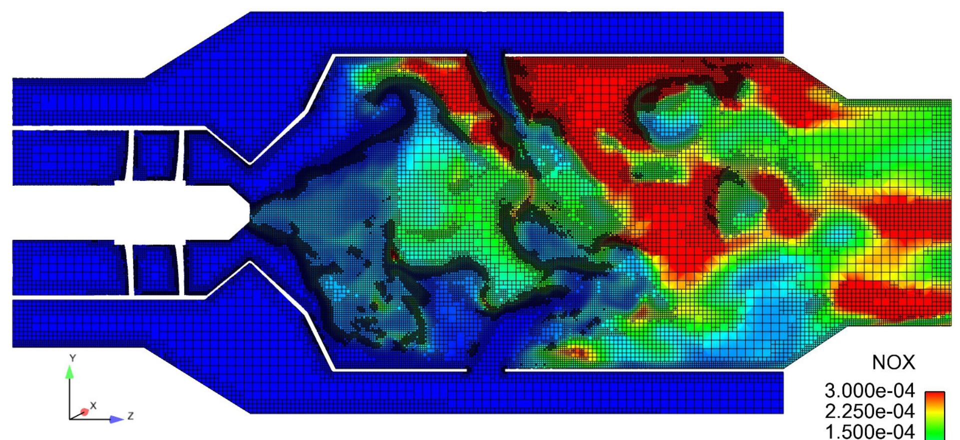


Figure 1: Adaptive Mesh Refinement. An example of adaptive mesh refinement for a two-dimensional grid in which the most turbulent areas of the mesh have been highly refined (from https://convergecfd.com/applications/gas-turbines/).

A simulation is typically performed by decomposing a physical region of interest into a collection of cells called a mesh and then calculating how the properties of the elements within each cell change over time. The mesh cells are distributed across a set of processes running across the many compute nodes within the supercomputer. Contiguous regions of cells are assigned to each process, and processes must frequently communicate to exchange boundary conditions between neighboring cells split across processes. Although replicated cells, called ghost cells, are sometimes used to reduce the frequency of communication, processes still typically exchange messages dozens of times per second.

Additional communication occurs during a *load-leveling* phase. Complex areas of the mesh that contain a large number of different types of elements are more difficult to simulate with high fidelity. Accordingly, simulations will often subdivide these areas into smaller cells as shown in Figure 1. This process of *adaptive mesh refinement* causes work imbalance as some processes within the parallel application will suddenly be responsible for a larger number of cells than their siblings. Therefore the simulation will rebalance the assignment of cells to processes following these refinements.

Due to the frequency of communication, the processes must run in parallel and avoid performance deviations to minimize the time spent waiting on messages. This method, *tightly coupled bulk synchronous computation,* is a primary differentiator between HPC and Big Data analytics.

Another primary differentiator is that the memory of each process is constantly overwritten as the properties of the mesh are updated. The total amount of this distributed memory has grown rapidly. For example, an astrophysics simulation on LANL's Trinity system may require up to 1.5 PB of RAM to represent regions of space with sufficient detail. Memory is one of the most precious resources in a supercomputer; most large-scale simulations expand to use all available memory. The large memory requirements coupled with the large amount of time required to simulate complex physical interactions leads to a problem for the users of large-scale computing systems.

The Need for Checkpointing

How can one ensure the successful completion of a simulation that takes days of calculation using tens of thousands of tightly coupled computers with petabytes of constantly overwritten memory?

The answer to that question has been checkpoint-restart. Storing the program state into a reliable storage system allows a failed simulation to restart from the most recently stored state.

Seemingly a trivial problem in the abstract, checkpoint-restart in practice is highly challenging because the actual writes in a checkpoint are extremely chaotic. One, the amount of data stored by each process is unlikely to match any meaningful block size in the storage system and is thus unaligned. Two, the writes are to shared data sets and thus incur either metadata or data bottlenecks [3, 8]. Three, the writes are *bursty* in that they all occur concurrently during the application checkpoint phase following a large period of storage idleness during the application compute phase. Four, the required bandwidth is very high; supercomputer designers face pressure to ensure 90% efficiency such that the checkpoint-restart of massive amounts of memory must complete quickly enough that no more than 10% of supercomputer lifetime is used.

Although many techniques have been developed to reduce this chaos [4], they are typically not available in practice. Incremental checkpointing reduces the size of the checkpoint but does not help when the memories are constantly overwritten. Uncoordinated checkpointing reduces burstiness but is not amenable to bulk synchronous computation. Two-phase I/O improves performance by reorganizing chaotic writes into larger aligned writes. Checkpointing into neighbor memory improves performance by eliminating media latencies. However, neither of these latter two is possible when all of available memory is used by the application.

Thus, HPC storage workloads, lacking common ground with read-intensive cloud workloads or IOPS-intensive enterprise workloads, have led to the creation of *parallel file systems*, such as BeeGFS, Ceph, GPFS, Lustre, OrangeFS, and PanFS, designed to handle bursty and chaotic checkpointing.

Storage for Scientific Discovery

From the teraflop to the petaflop era, the basic supercomputer architecture was remarkably consistent, and parallel file systems were the primary building block of its storage architecture. Successive supercomputers were largely copied from the same blueprint because the speed of processors and the capacities of memory and disk all grew proportionally to each other following Moore's Law. Horizontal arrows in Table 1 show how these basic

FLOPS / RAM	⇐⇒
RAM / core	⇓
MTTF per component	⇐⇒
MTTI per application	⇓
Impact of performance deviations	⇑
Drive spindles for capacity	⇐⇒
Drive spindles for bandwidth	⇑
Tape cassettes for capacity	⇐⇒
Tape drives for bandwidth	⇑
Storage clients / servers	⇑

Table 1: Supercomputer Trends Affecting Storage. Horizontal lines do not necessarily indicate no growth in absolute numbers but rather that the trend follows the relative overall growth in the machine.

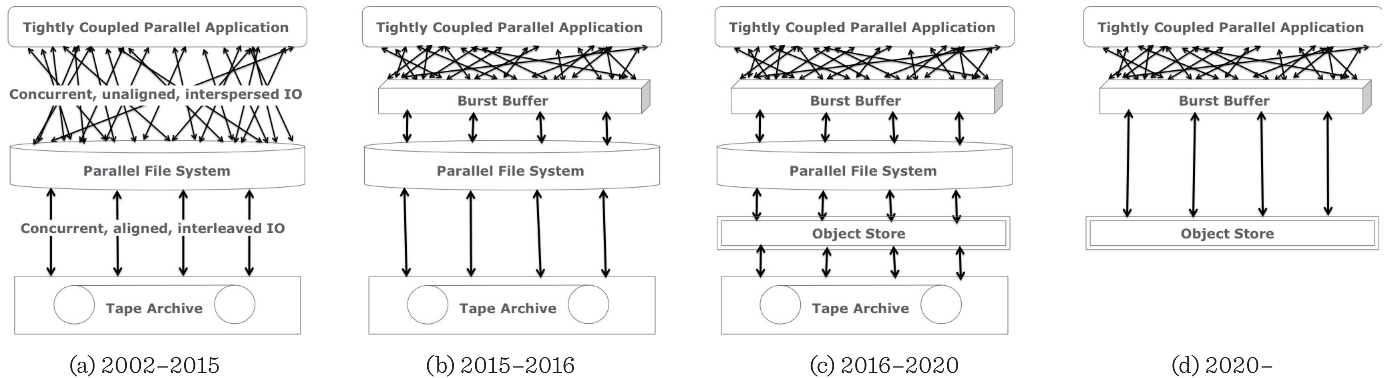


Figure 2: From 2 to 4 and back again. Static for over a decade, the HPC storage stack has now entered a period of rapid change.

architectural elements scaled proportionally. However, recent inflection points, shown with the vertical arrows, require a redesign of both the supercomputer and its storage architecture.

Era: 2002–2015. The storage architecture for the teraflop and petaflop eras is shown in Figure 2a. Large tightly coupled applications ran on large clusters of mostly homogeneous components. Forward progress, despite inevitable application interruptions, was ensured via checkpoint-restart. Tape-based archives stored cold data permanently, and disk-based parallel file systems satisfied the storage requirements for hot data.

These storage requirements are easily quantifiable for each supercomputer. An optimal checkpoint frequency is derived from the MTTI. The checkpoint size is typically no larger than 80% of the memory of the system, and HPC sites typically desire an efficiency of at least 90%. Given a checkpoint frequency and a checkpoint size, the required checkpoint bandwidth must be sufficient such that the total time spent checkpointing (and restarting and recomputing any lost work) is less than 10% of a system's operational time. The storage system also includes a capacity requirement derived from the needs of the system users and typically results in a capacity requirement between 30 and 40 times the size of the system memory.

As an example, imagine an exascale computer with 32 PB of memory and a checkpoint frequency of one hour. Ignoring restart and recompute, a checkpoint can take no more than six minutes, and therefore the required storage bandwidth must be at least 72 TB/s. The required storage capacity must be at least 960 PB.

An important characteristic of supercomputers within this era was that the minimum number of disks required for capacity was larger than the minimum number of disks required for bandwidth. This ensured that the simple model of a storage tier for performance and a second tier for capacity was economically optimal throughout this era.

During this era, total transistor counts continued to double approximately every 24 months. However, in the second half of this era, they did so *horizontally* by adding more compute nodes with larger core counts as opposed to merely increasing transistors within cores. This has had important implications which affect the storage stack: (1) although component reliability is little changed, large-scale systems built with increasing numbers of components are experiencing much shorter mean times to failure; (2) the amount of private memory available to each process is decreasing; (3) the number of processes participating in checkpoint-restart is growing; and (4) since larger supercomputers are more sensitive to performance deviations across their components, programming models which rely on bulk synchronous computing are less efficient.

Although petascale systems such as LANL's Roadrunner and ORNL's Jaguar began to stress the existing storage model in 2008, it survived until 2015.

Era: 2015–2016. In this era the inflection point in Table 1 noting the growth in the number of disk spindles required to meet checkpoint bandwidth requirements becomes a limitation. Until 2013, the system capacity requirement ensured sufficient disks to simultaneously satisfy the performance requirement. However, as disks have gotten relatively slower (as compared to their growth in capacity), a larger number of spindles is required to meet the bandwidth demand. A flash-only storage system could easily satisfy performance but would be prohibitively expensive for capacity: thus the introduction of a small performance tier situated between the application and the disk-based parallel file system as shown in Figure 2b. This tier is commonly referred to as a *burst buffer* [1, 7] because it is provisioned with enough bandwidth to meet the temporary bandwidth requirement but not sufficient capacity to meet the overall system demands. The bursty nature of the checkpoint-restart workload allows sufficient time to *drain* the data to a larger-capacity disk-based system. Initial burst buffer systems have been built on TACC's Wrangler, NERSC's Cori, and LANL's Trinity supercomputers.

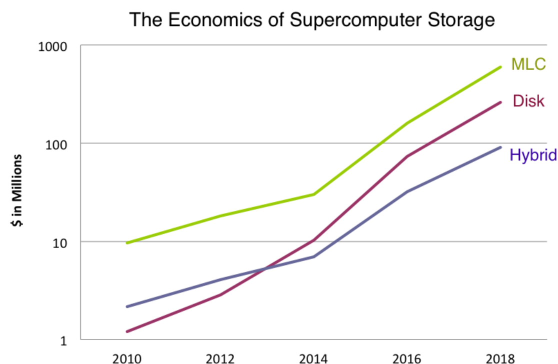


Figure 3: HPC Storage System Costs. This graph shows the media costs required to satisfy the checkpoint bandwidth and capacity requirement for 90% forward progression. We can see that in 2013, storage systems composed of one media type (disks or flash) are not as economical as a storage system that uses flash to meet bandwidth requirements and disks to meet capacity requirements.

Era: 2016–2020. A similar shift to that which motivated burst buffers now has occurred in the economic ratio between tape and disk. Although tape's primary requirement is to satisfy long-term capacity, it also has a performance requirement which is becoming increasingly expensive due to unique characteristics of tape. Specifically, tape capacity and performance are purchased separately, whereas, for disk and flash, they are purchased together; for example, a typical disk might provide a TB of capacity, 100 MBs of bandwidth, and 100 IOPS. Conversely, tape capacity is purchased in inexpensive cassettes, bandwidth in expensive drives, and IOPS in very expensive robots to connect the two. Analysis has shown that a hybrid disk-tape system was the correct economic choice for archival HPC storage as early as 2015 [6], as shown in Figure 2c. The emergence of efficient erasure coded object storage software has enabled the development of MarFS [5], the first instance of a disk-based archival storage system for HPC data that can provide the minimal bandwidth and extreme levels of data protection required for long-term data retention. For reference, the four storage tiers of LANL's 2016 Trinity supercomputer are shown in Table 2.

For future supercomputers of this era, we also expect that the physical location of the flash memory burst buffers will change. Instead of being in dedicated external nodes accessible to all of the compute nodes, flash memory will begin appearing within the supercomputer (e.g., node-local storage). Despite this change, this era will continue to be defined by the presence of a parallel file system.

Era: 2020 Onward. The four storage tiers shown in Figure 2c are an unnecessary burden on system integrators, administrators, and users. Therefore, as shown in Figure 2d, we predict a return to two-tier storage architectures for HPC systems. Also in this era, we predict the emergence of new storage interfaces to better utilize node-local storage.

Return to Two Tiers. Parallel file systems were created to handle chaotic, complex, and unpredictable streams of I/O as shown in Figure 2a. Figure 2b shows that burst buffers now absorb that chaos, leaving parallel file systems serving only orderly streams of I/O issued by system utilities. When burst buffers were first introduced, the possibility that these orderly streams might go directly to the tape archive was discussed. And while this was feasible for writes, reads would have experienced unacceptable latency. However, the move to disk-based object stores as shown in Figure 2c does provide sufficient performance for both reads and writes issued from the burst buffer. Thus, a separate parallel file system is no longer needed. The top two storage tiers in Figure 2c will combine, and the burst buffer will subsume the parallel file system.

Similarly, the bottom two tiers will merge. The erasure codes used in object storage present a much richer approach to reliability and data durability than modern tape archives. As the costs of an object store fall below those of a tape archive [6], the role of tape will increasingly be filled by the disk-based object store.

Early prototypes of a return to a two-tier HPC storage system are EMC's eponymous 2 Tiers™ and CMU's BatchFS [9]. Both expose a file system interface to the application, store hot data in a burst buffer, and can store cold data in an object store.

New Storage Interfaces. Although the burst buffer will subsume the parallel file system, over time it will decreasingly resemble current systems. Many bottlenecks within parallel file systems are due to the legacy POSIX semantics that are typically unnecessary for HPC applications. That vendors have provided POSIX is not surprising given that parallel file systems are also used by enterprise (i.e., non-HPC) customers who require more rigorous semantics.

However, as burst buffers will begin appearing as local storage within each compute node, these bottlenecks will finally become intractable. Maintaining POSIX semantics for a global

	Size	Bandwidth	Lifetime
Memory	2.1 PB	1–2 PB/s	milliseconds
Burst Buffer	3.7 PB	4–6 TB/s	hours
Parallel FS	78 PB	1–2 TB/s	weeks
Object Store	30 PB	100–300 GB/s	months
Tape Archive	50+ PB	10 GB/s	forever

Table 2: The Four Tiers of Trinity Storage. LANL's Trinity supercomputer uses Cray's DataWarp as a burst buffer, Lustre as a parallel file system, MarFS as a file system interface over an object store, and HPSS for the tape archive. The sizes above reflect the sizes at installation; the object store and the tape archive will grow over the lifetime of the machine.

namespace across distributed local burst buffers destroys the low latency of node-local storage since data access requires higher latency communications with a remote centralized metadata service. Thus, emerging systems like DeltaFS [10] and Intel's DAOS-M employ relaxed semantics to allow HPC applications with less rigorous storage requirements to achieve higher performance. We expect that these, or similar storage services, will increasingly appear as user-space libraries utilizing OS-bypass for low-latency access to local storage with eventual namespace reconciliation as first developed in Coda.

Further, the increased sensitivity to system noise is diminishing the efficiency of bulk synchronous computing models. Accordingly, programming models such as Stanford's Legion and asynchronous MPI will become attractive alternatives, but they will require asynchronous checkpointing models, such as uncoordinated checkpointing using message logging [4] or asynchronous transactions [2] with relaxed consistency semantics.

Conclusion

Unique properties of scientific simulations require unique storage architectures. Static for over a decade, HPC storage systems have now entered a period of rapid development. The opportunity for innovation in HPC storage is currently immense. We urge interested readers to join us in building new storage technologies for exascale computing.

Acknowledgments

A portion of this work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the US Department of Energy contract DE-FC02-06ER25750 and a CRADA between LANL and EMC. The publication has been assigned the LANL identifier LA-UR-16-21697.

References

[1] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-Free Co-Processing on a Prototype Exascale Storage Stack," in *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST),* April 2012, pp. 1–5.

[2] J. Bent, B. Settlemyer, H. Bao, S. Faibish, J. Sauer, and J. Zhang, "BAD Check: Bulk Asynchronous Distributed Checkpointing and IO," in *Proceedings of the Petascale Data Storage Workshop at SC15 (PDSW15),* Nov. 2015, pp. 19–24.

[3] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-File Access in Parallel File Systems," in *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS 2009),* May 2009, pp. 1–11.

[4] K. B. Ferreira, "Keeping Checkpointing Viable for Exascale Systems," PhD dissertation, University of New Mexico, Albuquerque, 2011, ISBN: 978-1-267-28351-1.

[5] G. Grider et al., "MarFS—A Scalable Near-Posix Metadata File System with Cloud Based Object Backend," in *Proceedings of the 10th Parallel Data Storage Workshop (PDSW '15),* Abstract—Work-in-Progress, 2015: http://www.pdsw.org/pdsw15/wips/wip-lamb.pdf.

[6] J. Inman, G. Grider, and H. B. Chen, "Cost of Tape Versus Disk for Archival Storage," in *Proceedings of the IEEE 7th International Conference on Cloud Computing (CLOUD),* June 2014, pp. 208–215.

[7] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the Role of Burst Buffers in Leadership-Class Storage Systems," in *Proceedings of the 2012 IEEE Conference on Massive Data Storage,* 2012: https://www.mcs.anl.gov/papers/P2070-0312.pdf.

[8] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield, "Zest: Checkpoint Storage System for Large Supercomputers," in *Proceedings of the 3rd Parallel Data Storage Workshop (PDSW '08),* Nov. 2008, pp. 1–5.

[9] Q. Zheng, K. Ren, and G. Gibson, "BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers," in *Proceedings of the 9th Parallel Data Storage Workshop (PDSW '14),* 2014, pp. 1–6: http://dx.doi.org/10.1109/PDSW.2014.7.

[10] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale File Systems Scale Better without Dedicated Servers," in *Proceedings of the 10th Parallel Data Storage Workshop (PDSW '15),* ACM, 2015, pp. 1–6: http://doi.acm.org/10.1145/2834976.2834984.

REGISTER TODAY!

2016 USENIX Annual Technical Conference

JUNE 22–24, 2016 • DENVER, CO

www.usenix.org/atc16

USENIX ATC '16 brings leading systems researchers together for cutting-edge systems research and unlimited opportunities to gain insight into a variety of must-know topics, including virtualization, system administration, cloud computing, security, and networking.

Co-located with USENIX ATC '16:

SOUPS 2016

Twelfth Symposium on Usable Privacy and Security

JUNE 22–24, 2016

www.usenix.org/soups2016

SOUPS 2016 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 '16

8th USENIX Workshop on Hot Topics in Cloud Computing

JUNE 20–21, 2016

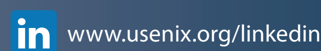
Researchers and practitioners at HotCloud '16 share their perspectives, report on recent developments, discuss research in progress, and identify new/emerging "hot" trends in cloud computing technologies.

HotStorage '16

8th USENIX Workshop on Hot Topics in Storage and File Systems

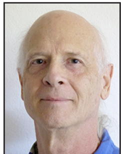
JUNE 20–21, 2016

HotStorage '16 is an ideal forum for leading storage systems researchers to exchange ideas and discuss the design, implementation, management, and evaluation of these systems.



Linux FAST Summit '16 Summary

RIK FARROW



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

Following FAST '16, 26 people met for the 2016 USENIX Research in Linux File and Storage Technologies Summit (Linux FAST '16) to discuss file systems, storage, and the Linux kernel. I've learned that the reason behind these discussions is to help people get their changes into the kernel, or at least to help people understand what the process is like. Ted Ts'o (Google) has pointed out at past Linux FAST workshops that there are already over 120 file systems in Linux, and getting new ones added is not going to be easy.

We began the workshop by going around the room and sharing our names, affiliations, and our reason for attending this year. Often, people attend because they just presented a file system, or code that extends an existing one, at the FAST workshop and want to understand how to get their code added to the Linux kernel. Others attend because they are working on new devices or need support for doing things in file systems that haven't been done before. We had all these interests this year. But let's start with the perennial issue: getting your code accepted into the Linux kernel.

Advice on how to do this was sprinkled throughout the workshop, and rather than mimic that distribution, I thought I would try to condense it into something more coherent.

Also, before going any further, I want to mention there was also a BSD File System workshop on the same day as FAST '16 tutorials. I couldn't attend because of the tutorials, but Kirk McKusick shared a link to the agenda, which also includes some notes [1] about what was covered. I did drop by and counted 31 people in attendance, seated lecture style. When I visited later, the meeting had broken up into working groups.

Going Upstream

Getting your code added to the Linux kernel, maintained by Linus, means having your code accepted *upstream*. All distros start with the upstream code and then add what distinguishes their distro from others. Also, getting new code into the kernel is definitely trying to swim upstream—it's not easy.

Ted Ts'o, the first US contributor to the Linux kernel, and someone who has attended every Linux FAST I've been to, always starts by suggesting you join, not the kernel mailing list, but a storage-specific mailing list appropriate to the type of code you've developed. There are lists (http://vger.kernel.org/vger-lists.html) for the block devices, SCSI, block caches, btrfs, and even Ceph, and all of them much more focused than the generic kernel mailing list. There is also a device mapper mailing list on a separate server (dm-devel@redhat.com).

This time, Ted Ts'o also suggested posting early code to the appropriate list. Ted's rationale for doing this is to quickly learn whether your approach is workable or not and whether there are problems before you have spent a lot of time working on a solution.

There was also some discussion about which kernel version to use. Erez Zadok (Stony Brook University) pointed out that it sometimes takes years for a graduate student to complete a

project, but Ric Wheeler (Red Hat) explained that there is one stable kernel per year (approximately), and working with a stable kernel is best.

Another big issue with getting a new file system accepted upstream is having someone to support the code in the future. You can't just toss your code over the wall—someone must continue to fix bugs and adapt the code as other kernel code changes. That someone should have industry support: in other words, work for a company that wants the code maintained. While you might wonder how common that is, Linux FAST always has a number of people who work for companies doing this. Linux FAST '16 had kernel developers working for Facebook, Google, Intel, HP Enterprise, Huawei, Micron, Red Hat, and Seagate, and four of those companies had multiple coders present. Industry-supported kernel hackers outnumbered students and faculty at this year's Linux FAST.

Ric Wheeler noted that there is another Linux file system and storage workshop that is more appropriate for industry users, rather than academics and developers, called Vault [2]. Unlike Linux FAST, where the goal is discussion, Vault has keynotes and talks.

I was sitting next to Jian Xu, a PhD student from the University of California, San Diego, who had presented a paper about a new file system for NVRAM (NOVA [3]). While NOVA is much faster than F2FS, the currently used flash file system for Linux-based devices, Ted pointed out that ext2 is actually faster on flash than F2FS for key-value stores but that F2FS does better in SQLite benchmarks. I suggested that Xu try to find industry backing—some company with a device that would benefit from using NOVA over F2FS. Ted had also suggested finding some "economic actor" that would support his project.

Shingled Magnetic Recording

SMR was a big topic in the past and continued to be this year. Ted Ts'o announced that he had been working with someone at CMU on host-aware SMR drive management, leading to some discussions throughout the afternoon. SMR disk drives get added capacity by eliminating the space between most tracks on a platter. Adrian Palmer (Seagate) pointed out that a single zone, or writeable area, on a typical SMR drive is 256 MB, whereas in conventional drives, the write unit is a sector, or 4 kB. In ext4, the largest block size is 128 MB, half the zone size in SMR.

Current SMR drives are device managed, which means that the SMR drives you can buy today hide the fact that they are SMR: they behave like drives with small sectors by handling all writes as sequential writes to a zone. That implies that SMR drives perform block mapping behind the scenes, and must also perform garbage collection, dealing with the holes created in zones when files are deleted or truncated. These activities are hidden from users (and the operating system), except when they cause unexpected latency. I overheard someone say that, when using an SMR drive, they could finish a benchmark in 30 seconds or 12 minutes, depending on the internal state of the drive. Revealing the internal state of drives was discussed to some extent during Linux FAST and was the topic of Eric Brewer's (Google) keynote at FAST '16 [4].

Ted Ts'o and people at Carnegie Mellon University have been working with Seagate on host-aware SMR drives, which are still self-managed but accept input from the host operating system to optimize performance. Peter Desnoyers (Northeastern University) and an associate have been working with Western Digital on the same problem but are using WD drives. Shaun Tancheff is a software engineer, consulting for Seagate, working on the problem from the manufacturer's side. Shaun asked for flags that can be included with requests to host-aware and host-managed SMR drives. Jens Axboe (Facebook) said that it is possible to add modifier flags to SCSI requests. Andreas Dilger (Intel) mentioned that he has been using some bits in the stream ID, but Jens Axboe said that he was not opposed to adding flags to the file struct for doing this.

There is another type of SMR drive, host-managed. The people at Linux FAST who I've suggested were working on host-aware drives may actually have been working on host-managed drives. They probably can't confirm that, however, because of the conditions they work under (NDAs). Host-managed drives take control of the SMR drive, the exact opposite of a device-managed drive. Host-managed drives must always write at the write-pointer, the furthest point in an SMR zone that has recently been written. Having the file system or deeper kernel layers manage an SMR drive means more than having to be aware of the 256 MB zones: the OS must also handle block mapping, copying blocks to better locations, as well as handling garbage collection. In some ways, SMR requires software very like the Flash Translation Layer (FTL) found in SSDs.

Adrian Palmer brought up the issue of out-of-order writes being a problem when working with SMR drives. Drive manufacturers have been making the assumption that the write-queue will be ordered (logical block addresses in either monotonically increasing or decreasing order). In practice, they had seen non-ordered writes. John Grove (Micron) also shared interest in having a mode where block I/O ordering is guaranteed through the block stack. Jens Axboe took the concerns seriously and suggested that people propose solutions. Jens also pointed out that in a multi-queue environment, ordering would practically require that all order-dependent I/Os go through one queue.

Trim

The `trim` command was created as a method of telling SSDs that certain logical blocks were no longer in use—the file system had

deleted the files containing those blocks, or truncated a file, also releasing blocks. `Trim` means an SSD can change the mappings of blocks to unused, and in theory this could help SSD performance by helping to reduce garbage collection overhead.

Initially, the `trim` command could not be queued: any commands in a queue for a drive would have to complete before the `trim` command could be issued. Later versions of the standard (ATA 3.1) allowed `trim` commands to queue.

Ted Ts'o pointed out that there are a number of SSDs that have been blacklisted by kernel developers because of data corruption issues when the `trim` command was used in Linux. See [5] for a list of blacklisted drives.

`Trim` also affects file systems and drivers for SMR drives, as SMR drives also need to perform garbage collection, dealing with freed space, and `trim` offers a method of communicating to a drive which logical blocks have designated as unused.

Tim Feldman (Seagate) opened the discussion of `trim` by mentioning that Seagate works with T10 and T13 standards bodies, which affect both stream IDs and `trim` for both flash and disk drives. Tim also suggested that some internal states of drives, which are actually intelligent devices, could be communicated back to the kernel: for example, the failure of a single head or other health characteristics. Ric Wheeler said that it would be useful to know when a drive has a non-volatile (NV) cache enabled, and Tim answered that this is well-defined in standards, but in practice, results may not be correct. Andreas Dilger said the standards consider this optional, and Tim agreed that NV cache state should be exposed.

`Trim` for a drive-managed SMR drive could change the write pointers, but Shaun pointed out that the problem is how to share this information with block device drivers. Hannes Reinecke (SUSE) had posted some code for supporting host-managed SMR drives [6], and his post was mentioned in the context of `trim` for SMR.

Jens Axboe mentioned that he is working on patches that support sharing information about timing/delays, write-stream IDs for flash devices, to reduce write amplification and some latency improvement. A lot of this work has been on the standards side so they can support it in the kernel. At this point, they can push a million-plus I/Os through the kernel.

Non-Volatile Memory

The Intel Micron 3D XPoint NVRAM was on lots of people's minds. About 1000 times faster than flash, and very likely arranged in cache-line-sized blocks (64–128 bytes) instead of kilobyte- or megabyte-addressable blocks, 3D XPoint will first appear on the memory or PCIe busses. And unlike flash, which could conveniently be treated as if it were a disk device, 3D XPoint needs to be treated more like a persistent form of DRAM. While not as fast as DRAM, NVRAM-like 3D XPoint will be much denser than DRAM, allegedly allowing a server to have as much as 6 TB of fast persistence storage. For HPC, this means that burst buffers (see Bent et al.'s HPC storage article in this issue) would go away, to be replaced with CPU-board storage for checkpointing.

Suparna Bhattacharya (HP Enterprise) asked whether 3D XPoint would appear as a storage device or be more like memory. Dan Williams (Intel) replied that today it appears that 3D XPoint will first appear as memory. When you read from 3D XPoint, lines get loaded into the appropriate CPU cache, and when you flush, lines should be flushed back. The current way of mapping a file into memory using `mmap()` will likely be extended to work with 3D XPoint and similar devices. Dan said that while some people want more control over cache behavior, he doesn't believe that they should be able to do this: the CPU is in the best position to make decisions about the cache. But fsyncing an mmapped file should result in the portions of the file in cache being copied to non-volatile storage, as happens with fsyncing data back to a disk. Dan says that the decisions on how to handle this have not completed, and perhaps fsyncing the device should force a cache flush.

Dan also introduced DAX/DMA into persistent memory, the biggest ticket item for persistent memory. DAX was developed for NVRAM, like 3D XPoint, but looks to have other applications as well. While mmap() memory maps files, DAX provides a pointer right into memory, and will be useful not just for NVRAM, but also in file systems like Jens Axboe's and ext4 (but not btrfs), where being able to overwrite a section of a file is useful. With DAX, you write, then commit, and once you commit the process blocks until the cache has been successfully flushed. DAX sounds like it will solve some of the problems people have with reworking mmap() to work with NVRAM.

BetrFS

Several people from Stony Brook University, including some of the authors of the Best Paper Award-winning "Optimizing Every Operation in a Write-Optimized File System" [8] were present. Rob Johnson (Stony Brook University) said that the primary reason for staying for Linux FAST was to learn more about getting their file system into the kernel. Rob said that the core of their optimizations (B-epsilon trees [9]) was part of a commercial product, and it was likely that someone from their crew would be hired to work on that product. That would mean that someone would be paid to maintain any changes to the kernel to support BetrFS.

Ceph

There were also several people present from Ceph, a distributed file storage product. While Ceph is a user-level overlay, currently used for block and object store, there appeared to be things that the Ceph folks would like to see in the kernel, such as a having

a key-value store there. Greg Farnum (Red Hat) seemed more interested in having access to unwritten file extents in user space. `fallocate()` won't expose unwritten blocks, because that's a security issue, but in the case of Ceph, being able to have more control over where Ceph writes its data and metadata would help them improve performance [10]. The key-value store is less interesting, as a transactional store would be more useful. The BetrFS crew also expressed some interest in transactional storage, leading to objections from Ted Ts'o.

Ted had two concerns: first, that an application would crash during a transaction, leaving the transaction orphaned, and second, that an application might be greedy and spool up so much data into one transaction that the transaction would dominate the log (and work that could currently be done). Rob Johnson said they would be happy to have limits on the log, and timeouts could handle the crashing during a transaction issue. Greg Farnum wrote that Ceph doesn't really *need* a POSIX file system but wants a transactional key-value store that runs in kernel space. Listening to this discussion, I thought such changes seem currently unlikely. But big changes have occurred, such as the discontinuation of ext3 in recent kernels and some distros now making btrfs the default file system.

References

[1] FreeBSD NewStorage Technologies Summit 2016: https://wiki.freebsd.org/201602StorageSummit/NewStorageTechnologies#Agenda.

[2] Vault: http://www.linuxfoundation.org/news-media/announcements/2014/08/linux-foundation-launches-new-conference-vault-address-growing.

[3] Jian Xu and Steven Swanson, "NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16),* 2016: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[4] Eric Brewer, "Spinning Disks and Their Cloudy Future," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16),* slides and audio: https://www.usenix.org/conference/fast16/technical-sessions/presentation/brewer.

[5] `Trim` Shortcomings: https://en.wikipedia.org/wiki/Trim_(computing)#SCSI.

[6] ZBC host-managed device support, SCSI mailing list: https://lwn.net/Articles/653187/.

[7] DAX: https://lwn.net/Articles/618064/.

[8] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter, "Optimizing Every Operation in a Write-Optimized File System, in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16),* 2016: https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan.

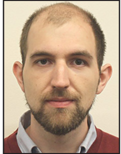
[9] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan, "An Introduction to Bϵ-trees and Write-Optimization." *;login:*, vol. 40, no. 5, October 2015: https://www.usenix.org/publications/login/oct15/bender.

[10] See Ts'o's comment on `fallocate()` after Linux Fast: http://marc.info/?l=linux-api&m=145704481128395&w=2.

SYSADMIN

Improve Your Multi-Homed Servers with Policy Routing

JONATHON ANDERSON



Jonathon Anderson has been an HPC Sysadmin since 2006 and believes that everything would be a lot easier if we just spent more time figuring out the correct way to do things. He's currently serving as HPC Engineer at the University of Colorado and hopes to stick around Boulder for a long time to come.
jonathon.anderson@colorado.edu

Traditional IP routing systems route packets by comparing the destination address against a predefined list of routes to each available subnet; but when multiple potential routes exist between two hosts on a network, the preferred route may be dependent on context that cannot be inferred from the destination alone. The Linux kernel, together with the iproute2 suite [1], supports the definition of multiple routing tables [2] and a routing policy database [3] to select the preferred routing table dynamically. This additional expressiveness can be used to avoid multiple routing pitfalls, including asymmetric routes and performance bottlenecks from suboptimal route selection.

Background

The CU-Boulder Research Computing environment spans three datacenters, each with its own set of special-purpose networks. A traditionally routed host simultaneously connected to two or more of these networks compounds network complexity by making only one interface (the default gateway) generally available across network routes. Some cases can be addressed by defining static routes, but even this leads to asymmetric routing that is at best confusing and at worst a performance bottleneck.

Over the past few months we've been transitioning our hosts from a single-table routing configuration to a policy-driven, multi-table routing configuration. The end result is full bi-directional connectivity between any two interfaces in the network, irrespective of underlying topology or a host's default route. This has reduced the apparent complexity in our network by allowing the host and network to Do the Right Thing™ automatically, unconstrained by an otherwise static route map.

Linux policy routing has become an essential addition to host configuration in the University of Colorado Boulder "Science Network." It's so useful, in fact, that I'm surprised a basic routing policy isn't provided by default for multi-homed servers.

The Problem with Traditional Routing

The simplest Linux host routing scenario is a system with a single network interface.

```
# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
       pfifo_fast state UP qlen 1000
    link/ether 00:50:56:88:56:1f brd ff:ff:ff:ff:ff:ff
    inet 10.225.160.38/24 brd 10.225.160.255 scope global dynamic ens192
       valid_lft 60184sec preferred_lft 60184sec
```

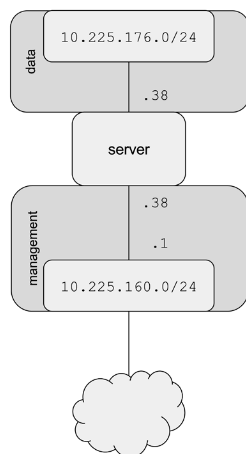


Figure 1: A simple dual-homed server with a traditional default route

Such a typically configured network with a single uplink has a single default route in addition to its link-local route.

```
# ip route list
default via 10.225.160.1 dev ens192
10.225.160.0/24 dev ens192  proto kernel  scope link  src
10.225.160.38
```

Traffic to hosts on 10.225.160.0/24 is delivered directly, while traffic to any other network is forwarded to 10.225.160.1.

A dual-homed host adds a second network interface and a second link-local route, but the original default route remains (see Figure 1).

```
# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
     valid_lft forever preferred_lft forever
2: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
        pfifo_fast state UP qlen 1000
   link/ether 00:50:56:88:56:1f brd ff:ff:ff:ff:ff:ff
   inet 10.225.160.38/24 brd 10.225.160.255 scope global
dynamic ens192
   valid_lft 86174sec preferred_lft 86174sec
3: ens224: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
        pfifo_fast state UP qlen 1000
   link/ether 00:50:56:88:44:18 brd ff:ff:ff:ff:ff:ff
   inet 10.225.176.38/24 brd 10.225.176.255 scope global dynamic
ens224
     valid_lft 69193sec preferred_lft 69193sec

# ip route list
default via 10.225.160.1 dev ens192
```

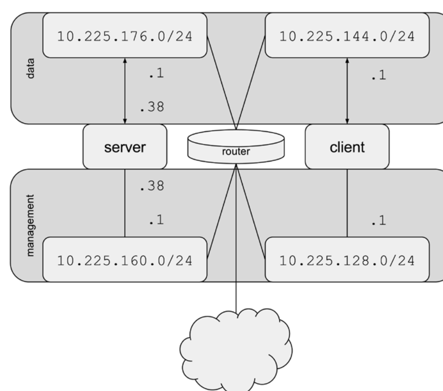


Figure 2: A server and a client, with static routes between their data interfaces

```
10.225.160.0/24 dev ens192  proto kernel  scope link  src
10.225.160.38
10.225.176.0/24 dev ens224  proto kernel  scope link  src
10.225.176.38
```

The new link-local route provides access to hosts on 10.225.176.0/24 and is sufficient for a private network connecting a small cluster of hosts. In fact, this is the configuration that we started with in our Research Computing environment: .160.0/24 is a low-performance "management" network, while .176.0/24 is a high-performance "data" network.

In a more complex network, however, link-local routes quickly become insufficient. In the CU Science Network, for example, each datacenter is considered a discrete network zone with its own set of "management" and "data" networks. For hosts in different network zones to communicate, a static route must be defined in each direction to direct performance-sensitive traffic across the high-performance network route (see Figure 2).

```
server # ip route add 10.225.144.0/24 via 10.225.176.1
client # ip route add 10.225.176.0/24 via 10.225.144.0
```

Although managing these static routes can be tedious, they do sufficiently define connectivity between the relevant network pairs: "data" interfaces route traffic to each other via high-performance networks, while "management" interfaces route traffic to each other via low-performance networks. Other networks (e.g., the Internet) can only communicate with the hosts on their default routes; but this limitation may be acceptable for some scenarios.

Even this approach is insufficient, however, to allow traffic between "management" and "data" interfaces. This is particularly problematic when a client host is not equipped with a symmetric set of network interfaces (see Figure 3). Such a client may only have a "management" interface but should still

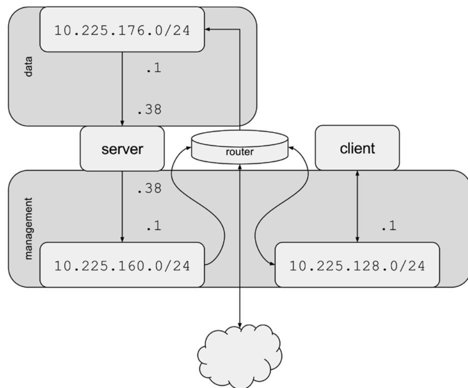


Figure 3: In a traditional routing configuration, the server would try to respond to the client via its default route, even if the request arrived on its data interface.

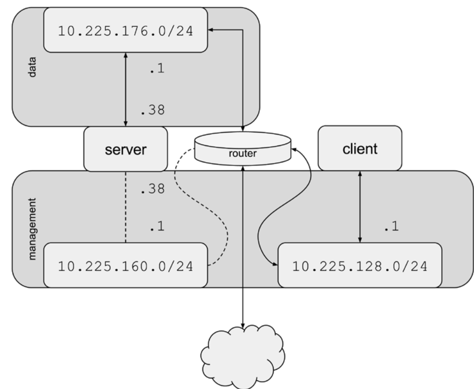


Figure 4: Routing policy allows the server to respond using its data interface for any request that arrives on its data interface, even if it has a different default route.

communicate with the server's high-performance interface for certain types of traffic. (For example, a dual-homed NFS server should direct all NFS traffic over its high-performance "data" network, even when being accessed by a client that itself only has a low-performance "management" interface.) By default, the Linux `rp_filter` [4] blocks this traffic, as the server's response to the client targets a different route than the incoming request; but even if `rp_filter` is disabled, this asymmetric route limits the server's aggregate network bandwidth to that of its lower-performing interface.

The server's default route could be moved to the "data" interface—in some scenarios, this may even be preferable—but this only displaces the issue: clients may then be unable to communicate with the server on its "management" interface, which may be preferred for certain types of traffic. In Research Computing, for example, we prefer that administrative access and monitoring not compete with IPC and file system traffic.

Routing Policy Rules

Traditional IP routing systems route incoming packets based solely on the the intended destination; but the Linux iproute2 stack supports route selection based on additional packet metadata, including the packet source. Multiple discrete routing tables, similar to the virtual routing and forwarding (VRF) support found in dedicated routing appliances [5], define contextual routes, and a routing policy selects the appropriate routing table dynamically based on a list of rules.

In the following example, there are three different routing contexts to consider. The first of these—the "main" routing table—defines the routes to use when the server initiates communication.

```
server # ip route list table main
10.225.144.0/24 via 10.225.176.1 dev ens224
default via 10.225.160.1 dev ens192
10.225.160.0/24 dev ens192  proto kernel  scope link  src
10.225.160.38
10.225.176.0/24 dev ens224  proto kernel  scope link  src
10.225.176.38
```

A separate routing table defines routes to use when responding to traffic on the "management" interface. Since this table is concerned only with the default route's interface in isolation, it simply reiterates the default route.

```
server # ip route add default via 10.225.160.1 table 1
server # ip route list table 1
default via 10.225.160.1 dev ens192
```

Similarly, the last routing table defines routes to use when responding to traffic on the "data" interface. This table defines a *different* default route: all such traffic should route via the "data" interface.

```
server # ip route add default via 10.225.176.1 table 2
server # ip route list table 2
default via 10.225.176.1 dev ens224
```

With these three routing tables defined, the last step is to define routing policy to select the correct routing table based on the packet to be routed. Responses from the "management" address should use table 1, and responses from the "data" address should use table 2. All other traffic, including server-initiated traffic that has no outbound address assigned yet, uses the "main" table automatically.

```
server # ip rule add from 10.225.160.38 table 1
server # ip rule add from 10.225.176.38 table 2
server # ip rule list
0:  from all lookup local
32764:  from 10.225.176.38 lookup 2
32765:  from 10.225.160.38 lookup 1
32766:  from all lookup main
32767:  from all lookup default
```

With this routing policy in place, a single-homed client (or, in fact, *any* client on the network) may communicate with both the server's "data" and "management" interfaces independently and successfully and the bi-directional traffic routes consistently via the appropriate network (see Figure 4).

Persisting the Configuration

This custom routing policy can be persisted in the Red Hat "ifcfg" network configuration system by creating interface-specific `route-` and `rule-` files.

```
# cat /etc/sysconfig/network-scripts/route-ens192
default via 10.225.160.1 dev ens192
default via 10.225.160.1 dev ens192 table mgt

# cat /etc/sysconfig/network-scripts/route-ens224
10.225.144.0/24 via 10.225.176.1 dev ens224
default via 10.225.176.1 dev ens224 table data

# cat /etc/sysconfig/network-scripts/rule-ens192
from 10.225.160.38 table mgt

# cat /etc/sysconfig/network-scripts/rule-ens224
from 10.225.176.38 table data
```

The symbolic names `mgt` and `data` used in these examples are translated to routing table numbers as defined in the `/etc/iproute2/rt_tables` file.

```
# echo "1 mgt" >>/etc/iproute2/rt_tables
# echo "2 data" >>/etc/iproute2/rt_tables
```

Once the configuration is in place, activate it by restarting the network service (e.g., `systemctl restart network`). You may also be able to achieve the same effect using `ifdown` and `ifup` on individual interfaces.

Red Hat's support for routing rule configuration has a confusing regression that merits specific mention. Red Hat (and its derivatives) has historically used a `network` init script and subscripts to configure and manage network interfaces, and these scripts support the aforementioned `rule-` configuration files. Red Hat Enterprise Linux 6 introduced NetworkManager, a persistent daemon with additional functionality; however, NetworkManager did not support `rule-` files until version 1.0, released as part of RHEL 7.1 [6]. If you're currently using NetworkManager, but wish to define routing policy in `rule-` files, you'll need to either disable NetworkManager entirely or exempt specific interfaces from NetworkManager by specifying `NM_CONTROLLED=no` in the relevant `ifcfg-` files.

In a Debian-based distribution, these routes and rules can be persisted using `post-up` directives in `/etc/network/interfaces`.

Further Improvements

We're still in the process of deploying this policy-based routing configuration in our Research Computing environment, and, as we do, we discover more cases where previously complex network requirements and special-cases are abstracted away by this relatively uniform configuration. We're simultaneously evaluating other potential changes, including the possibility of running a dynamic routing protocol (such as OSPF) on our multi-homed hosts, or of configuring every network connection as a simultaneous default route for failover. In any case, this experience has encouraged us to take a second look at our network configuration to reevaluate what we had previously thought were inherent limitations of the stack itself.

References

[1] Iproute2: http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2.

[2] Routing tables: http://linux-ip.net/html/routing-tables.html.

[3] Policy-based routing: http://linux-ip.net/html/routing-rpdb.html.

[4] rp-filter How To: http://tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.kernel.rpf.html.

[5] Virtual routing and forwarding: http://www.cisco.com/c/en/us/td/docs/net_mgmt/active_network_abstraction/3-7/reference/guide/ANARefGuide37/vrf.html.

[6] Red Hat on policy-based routing persistence: https://access.redhat.com/solutions/288823.

MongoDB Database Administration

MIHALIS TSOUKALOS

Mihalis Tsoukalos is a UNIX System Administrator, a programmer (UNIX and iOS), a DBA, a mathematician, and a technical writer. You can reach him at http://www.mtsoukalos.eu/ and @mactsouk. mactsouk@gmail.com

MongoDB [1, 2] is a document-oriented NoSQL database that has become quite popular. In this article, I will show you how to perform various administrative tasks, after setting up a dummy collection that will be used as an example. You will learn how to create and drop collections, use indexes, and convert a MongoDB database from using the MMAPv1 storage engine to using WiredTiger. I will also talk about mtools, which is a convenient set of Python scripts for processing MongoDB log files.

Table 1 introduces you to the MongoDB terminology compared to the well-known Relational Database terminology.

SQL Term	MongoDB Term
Database	Database
Table	Collection
Index	Index
Row	BSON document
Column	BSON field
Primary key	_id field
Group	Aggregation
Join	Embedding and linking

Table 1: MongoDB and RDBMS terminology

For this article, I used MongoDB version 3.2.1 [3] running on Mac OS X; however, most of the presented commands will also work on the older 2.6 version. MongoDB was installed on Mac OS X using the Homebrew [5] package. You will most certainly find a ready-to-install package for your operating system, but you can also get precompiled MongoDB binaries from [4].

A bit of warning before continuing with the rest of the article. MongoDB neither monitors disk space nor displays any warning messages related to disk space, so it is up to the system administrator to deal with such issues. The only occasion where you will hear MongoDB complaining about disk space is when there is no disk space left!

Basic DBA Commands

Most of the tasks presented in this article will be performed from the Mongo shell, which starts by executing the `mongo` command. The name of the MongoDB server process is `mongod`. First, you should run the following JavaScript code from the MongoDB shell in order to add some data on your database and have something to experiment with:

```
> use login
switched to db login
> for (var i=300; i<100000; i++)
{ db.myData.insert({x:i, y:2*i}); }
WriteResult({ "nInserted" : 1 })
> db.myData.count();
100000
```

The first command switches to the "login" database—if the database does not exist, it will be automatically created. It then uses a JavaScript for loop so that it can insert 100,000 documents to the myData collection, which will also be created if needed. The last command shows how you can find out the total number of records that exist in a collection.

As you can see, you do not have to specifically create a collection or its fields (keys). If you try to insert a document to a collection that does not exist, MongoDB will automatically create the collection. Additionally, if you try to insert a document that has a different set of keys to an existing collection, MongoDB will create it without any complaints. This means that small typographical errors cannot be detected very easily.

If you wish to delete the entire "myData" collection and start with an empty one, you should use the drop() method:

```
> db.myData.drop();
true
```

As saving data on a database takes disk space, it is good to know how to delete an entire database. The following command deletes the entire "login" database, including its data files:

```
> use login
switched to db login
> db.runCommand( { dropDatabase: 1 } )
{ "dropped" : "login", "ok" : 1 }
```

Should you wish to view the list of the available databases on the MongoDB server you are connected to, you can execute the following command from the MongoDB shell:

```
> show databases
LXF    0.031GB
local  0.078GB
login  0.063GB
test   0.031GB
```

After you select a database, you can see its available collections as follows:

```
> show tables
myData
system.indexes
```

The system.indexes collection contains information about the indexes of a database. However, it should not be accessed directly as if it was a regular collection but with the help of the getIndexes() function.

You can manually start a MongoDB server process from the command line as follows:

```
$ mongod --fork --logpath a.log --smallfiles --oplogSize 50 --port
  27101 --dbpath w1 --replSet w --logappend
```

The --port parameter defines the port number that the MongoDB server will listen to, the --dbpath value defines the folder that will contain the database files, the value of the --logpath parameter shows the file that will hold the log messages, and the --fork parameter tells the operating system that the process will run in the background without a terminal. The --replset parameter defines the name of the replica set and should only be included when you want to define a replica set. The --logappend parameter tells the MongoDB process to append to the log file instead of overwriting it.

By default, MongoDB does not require users to log in to connect from the local machine, which means that anyone who has access to a machine can do whatever she wants with the entire MongoDB server. In order to enable authorization you must use the --auth when running mongod or use the security.authorization setting in the configuration file of MongoDB.

Converting a Database from MMAPv1 to WiredTiger

MongoDB currently supports two Storage Engines: MMAPv1 and WiredTiger [6]; the good thing is that all commands related to database administration are the same regardless of the storage engine used.

WiredTiger is an open source project that was built separately from MongoDB and is also used by other databases. Apart from the performance gains, its main advantage is that it supports document-level locking, allowing you to lock only the document you are currently processing instead of locking the entire collection your document belongs to. Starting with MongoDB version 3.2, WiredTiger is the default storage engine whereas previous MongoDB versions used MMAPv1. This section will show you how to convert a MongoDB database from the MMAPv1 to the WiredTiger storage engine.

The data directory of a MongoDB database that uses WiredTiger looks like the following:

```
$ ll data/
total 272
-rw-r--r--  1 mtsouk  staff    49 Feb 13 14:12 WiredTiger
-rw-r--r--  1 mtsouk  staff    21 Feb 13 14:12 WiredTiger.lock
-rw-r--r--  1 mtsouk  staff   918 Feb 13 14:14 WiredTiger.turtle
-rw-r--r--  1 mtsouk  staff 40960 Feb 13 14:14 WiredTiger.wt
-rw-r--r--  1 mtsouk  staff  4096 Feb 13 14:12 WiredTigerLAS.wt
-rw-r--r--  1 mtsouk  staff 16384 Feb 13 14:13 _mdb_catalog.wt
-rw-r--r--  1 mtsouk  staff 16384 Feb 13 14:13
  collection-0-7818407182795123090.wt
-rw-r--r--  1 mtsouk  staff  4096 Feb 13 14:21
  collection-2-7818407182795123090.wt
drwxr-xr-x  4 mtsouk  staff   136 Feb 13 14:21 diagnostic.data
-rw-r--r--  1 mtsouk  staff 16384 Feb 13 14:13 index-1
-7818407182795123090.wt
-rw-r--r--  1 mtsouk  staff  4096 Feb 13 14:21 index-3
-7818407182795123090.wt
drwxr-xr-x  5 mtsouk  staff   170 Feb 13 14:12 journal
-rw-r--r--  1 mtsouk  staff     5 Feb 13 14:12 mongod.lock
-rw-r--r--  1 mtsouk  staff 16384 Feb 13 14:14 sizeStorer.wt
-rw-r--r--  1 mtsouk  staff    95 Feb 13 14:12 storage.bson
```

The filenames in the data directory show whether you are using WiredTiger or not. However, you can find the storage engine of your MongoDB server from the shell by executing the following command:

```
> db.serverStatus().storageEngine
{ "name" : "mmapv1", "supportsCommittedReads" : false }
```

Alternatively, you can execute the following command, which gives the same information in a different format:

```
> db.serverStatus().storageEngine.name
mmapv1
```

So the previous database uses MMAPv1. Using the same commands on a database that uses WiredTiger produces the following output:

```
> db.serverStatus().storageEngine
{ "name" : "wiredTiger", "supportsCommittedReads" : true }
> db.serverStatus().storageEngine.name
wiredTiger
```

Executing the "`db.serverStatus().wiredTiger`" command on a database that uses WiredTiger produces a large amount of output with information about various WiredTiger parameters and useful statistics, including buffer sizes, number of update, insert, remove, search calls, cache data, connection data, etc.

Converting a MongoDB 3.0.x or newer database that uses MMAPv1 to one using WiredTiger is a relatively easy process. You will first need to back up your data, delete existing data files, change the configuration file of MongoDB in order to make it use WiredTiger, and then import your backup data to MongoDB. Starting MongoDB with an empty data directory makes MongoDB generate all necessary files, which makes our job much easier. For Mac OS X, the required steps and commands are the following:

```
$ mongo
MongoDB shell version: 3.2.1
connecting to: test
> db.serverStatus().storageEngine.name
mmapv1
> use login
switched to db login
> db.myData.count()
100000
<Control-C>
$ mongodump -d login -c myData
2016-02-13T18:40:00.114+0200 writing login.myData to
2016-02-13T18:40:00.367+0200 done dumping login.myData
       (100000 documents)
$ launchctl unload ~/Library/LaunchAgents/homebrew.mxcl.
       mongodb.plist
$ rm -rf /usr/local/var/mongodb
$ mkdir /usr/local/var/mongodb
$ cp /usr/local/etc/mongod.conf{,.old}
$ vi /usr/local/etc/mongod.conf
$ diff /usr/local/etc/mongod.conf /usr/local/etc/mongod.conf.old
7d6
<   engine: "wiredTiger"
$ launchctl load ~/Library/LaunchAgents/homebrew.mxcl.
       mongodb.plist
$ mongorestore
$ mongo
MongoDB shell version: 3.2.1
connecting to: test
> db.serverStatus().storageEngine.name
wiredTiger
> use login
switched to db login
> db.myData.count()
100000
```

The two `count()` commands verify that all documents have been successfully imported. As you can understand from the output of the `diff` command, you just need to add a single line in the MongoDB configuration file to make MongoDB use WiredTiger. The `mongodump` command creates a directory named "`dump`" inside the current directory. If you execute the `mongorestore` command from the same directory you ran `mongodump`, then `mongorestore` will automatically find and use the "`dump`" directory.

About Log Files

The default location of the log files of a HomeBrew [5] MongoDB installation on a Mac OS X system as defined in the `/usr/local/etc/mongod.conf` file is `/usr/local/var/log/mongodb/mongo.log`. Data files are kept inside `/usr/local/var/mongodb`. If you choose not to use `mongod.conf`, you will have to define all required parameters from the command line. On a usual Linux installation, you can find `mongodb.conf` inside `/etc`, and `mongodb.log` inside `/var/log/mongodb`, whereas the data directory is usually located at `/var/lib/mongodb/`.

If you try to start a MongoDB server without a proper log file directory, you are going to get the following error message:

```
2015-11-29T12:01:54.349+0200 F CONTROL  Failed global
initialization:
FileNotOpen Failed to open "/Users/mtsouk/Downloads/./aPath
/a.log"
```

You are also going to a get a similar error message if the data directory is missing:

```
2015-11-29T12:02:42.547+0200 I STORAGE  [initandlisten]
exception in
initAndListen: 29 Data directory ./myData not found.,
terminating
2015-11-29T12:02:42.548+0200 I CONTROL  [initandlisten]
dbexit:  rc: 100
```

Dropping an entire database and deleting its data files produces the following kind of log messages:

```
2016-02-13T11:04:29.593+0200 I COMMAND  [conn25]
dropDatabase
      green starting
2016-02-13T11:04:29.714+0200 I JOURNAL  [conn25]
journalCleanup...
2016-02-13T11:04:29.714+0200 I JOURNAL  [conn25]
removeJournalFiles
2016-02-13T11:04:29.718+0200 I JOURNAL  [conn25]
journalCleanup...
2016-02-13T11:04:29.718+0200 I JOURNAL  [conn25]
removeJournalFiles
2016-02-13T11:04:29.720+0200 I COMMAND  [conn25]
dropDatabase
      green finished
2016-02-13T11:04:29.720+0200 I COMMAND  [conn25] command
green
command: dropDatabase { dropDatabase: 1.0 } keyUpdates:0
writeConflicts:0 numYields:0 reslen:41 locks:{ Global:
{ acquireCount:
{ r: 2, w: 1, W: 1 } }, MMAPV1Journal: { acquireCount: { w: 4 } },
Database: { acquireCount: { W: 1 } } } protocol:op_command 126ms
```

Therefore, a command like the following would show the databases that were dropped from your MongoDB installation:

```
$ grep -w dropDatabase /usr/local/var/log/mongodb/mongo.log |
grep -w finished | awk {'print $6'}
```

Looking at log files is an important task of a DBA, so the next section presents mtools, a set of Python scripts that deal with MongoDB log files.

The mtools Set of Scripts

mtools [7] is a set of Python scripts that can help you parse, filter, and visualize MongoDB log files. The mtools set includes mloginfo, mlogfilter, mplotqueries, mlogvis, mgenerate, and mlaunch. Please note that at the moment, mtools is not compatible with Python 3. The mloginfo script displays information about the data of a log file in a format similar to the following:

```
$ mloginfo /usr/local/var/log/mongodb/mongo.log
     source: /usr/local/var/log/mongodb/mongo.log
       host: iMac.local:27017
      start: 2015 Sep 12 19:21:04.358
        end: 2016 Feb 14 23:10:16.192
date format: iso8601-local
     length: 5118
     binary: mongod
    version: 3.0.6 -> 3.0.7 -> 3.2.0 -> 3.2.1
    storage: wiredTiger
```

As you can see, mloginfo shows the MongoDB versions that generated the log messages as well as the storage engine and the time period of the log entries.

The mlogfilter script is a log file parser that can be used for extracting information out of busy MongoDB log files. Think of it as an advanced `grep` utility for MongoDB log files. The mlaunch script lets you create MongoDB test environments on your local machine quickly. The mplotqueries script is used for visualizing MongoDB log files. The mlogvis script is similar to the mplotqueries script, but instead of creating a graphics file, its output is an interactive HTML page on a Web browser. Last, the mgenerate script produces pseudo-random data for populating MongoDB databases with sample data.

The first thing you should learn is about the installation of the mtools package, which can be done as follows:

```
$ pip install mtools
...
Successfully built mtools psutil
Installing collected packages: psutil, mtools
Successfully installed mtools-1.1.9 psutil-3.4.2
```

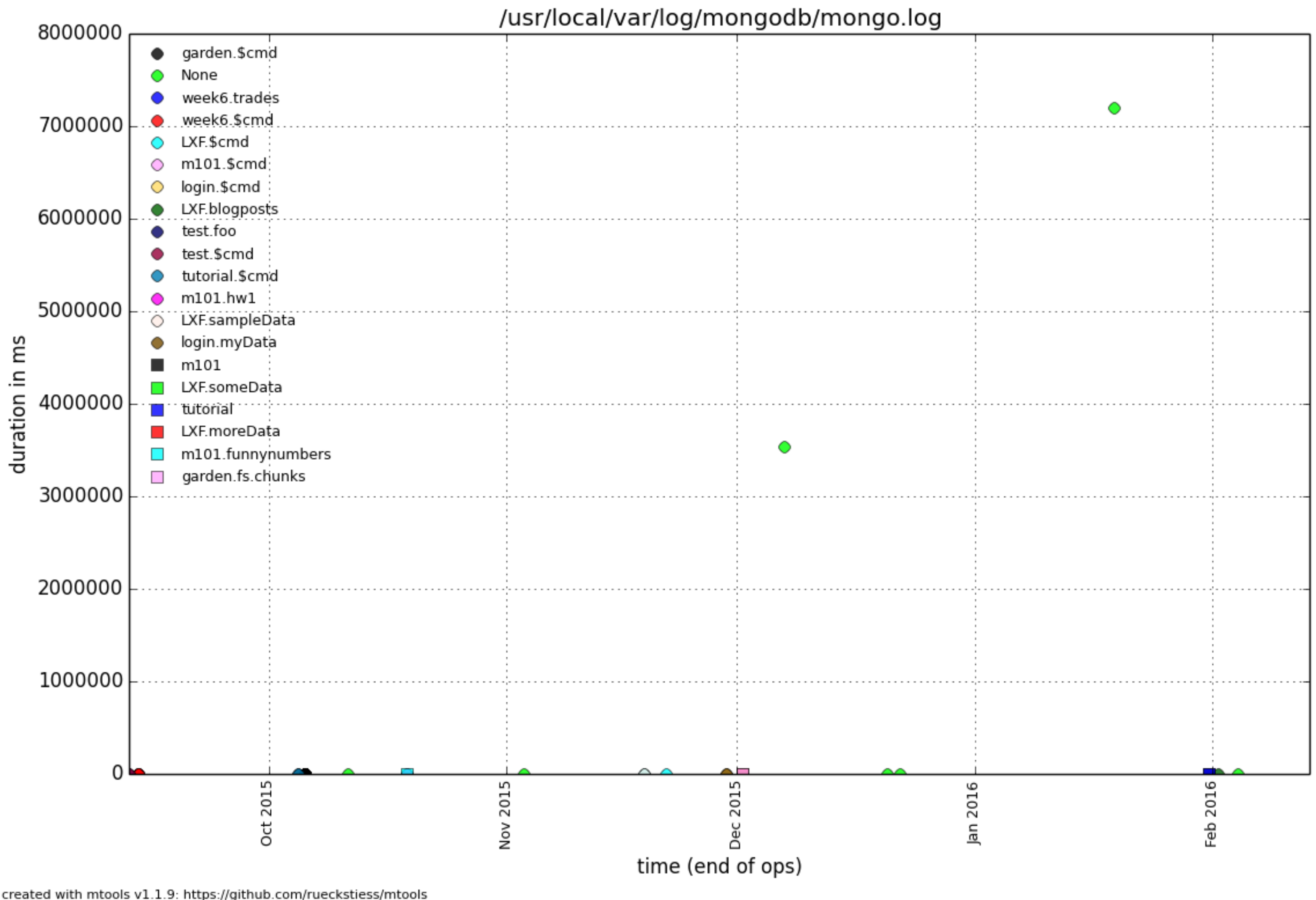


Figure 1: The output of the mplotqueries Python script on a log file from my local MongoDB server

For ease of installation, it is recommended that you use the `pip` utility to install mtools when possible; otherwise visit the mtools installation page [8] for more information.

Now that you have installed mtools, it is time to use mplotqueries to get information from a MongoDB log file and plot it on screen. All you have to do is execute the following command:

```
$ mplotqueries /usr/local/var/log/mongodb/mongo.log
      --output-file login.png
```

The mplotqueries tool reads the specified log file and summarizes the information based on the name of the collection used. It then prints the duration of each operation on a timeline. Figure 1 shows the output mplotqueries produced using the MongoDB log file found on my Mac. As you can see, most operations ran almost instantly. This is a very handy way of overseeing your MongoDB data that can also run as a cron job. If you do not use the `--output-file` option, an interactive output is automatically going to be displayed on your screen.

Figure 2 shows the interactive output generated by mlogvis on the same log file as before.

Generally speaking, both mlogvis and mplotqueries are very handy for detecting outliers. If you think certain operations on a collection are running slow for some reason, you can use mlogfilter to look into them:

```
$ mlogfilter /usr/local/var/log/mongodb/mongo.log --word login
```

The previous command returns log entries that contain the "login" keyword. One of them is the following:

```
015-11-29T15:33:11.226+0200 I COMMAND [conn46] command
login.$cmd
command: insert { insert: "myData", documents: [ { _id:
ObjectId('565afe9781f59f422de5bd05'), x: 0.0, y: 0.0 } ], ordered:
true } keyUpdates:0 writeConflicts:0 numYields:0 reslen:40 locks:{
Global: { acquireCount: { r: 2, w: 2 } }, MMAPV1Journal: {
```

Figure 2: The output of the mlogvis Python script on a log file from my local MongoDB server

```
acquireCount: { w: 8 }, acquireWaitCount: { w: 1 },
timeAcquiringMicros: { w: 2058 } },
Database: { acquireCount:
{ w: 1, W: 1 } }, Collection: { acquireCount:
{ W: 1 } },
Metadata: { acquireCount: { W: 4 } } }
199ms
```

What the previous log entry says is that it took MongoDB 199 ms to execute an insert operation in the "myData" collection of the "`login`" database. If you want to speed up an insert operation, you might need to upgrade your hardware; however, if such operations do not happen very often, you should not be concerned.

If you want to find out all possible options for each tool, you can execute it with the `--help` option. I think that the mtools set of Python scripts is a useful tool to add to your arsenal.

Indexes

Analyzing a query is a very good way to find out why a query runs slow as well as how your query is executed. This can happen with the help of the `explain()` method. The interesting part from the `explain("executionStats")` command that displays how a query is executed is the following:

```
> db.myData.find({ "x": { $gt: 99990}
}).explain("executionStats")
...

"executionStats" : {
"executionSuccess" : true,
"nReturned" : 9,
"executionTimeMillis" : 46,
"totalKeysExamined" : 0,
"totalDocsExamined" : 100000,
"executionStages" : {
"stage" : "COLLSCAN",
"filter" : {
"x" : {
"$gt" : 99990
}
},
"nReturned" : 9,
"executionTimeMillisEstimate" : 0,
...
"docsExamined" : 100000
...
```

The explain command shows that the execution plan chosen does a full collection scan (COLLSCAN)—in other words, it searches all documents in the requested collection, which is not very efficient. As you can see from the values of both "totalDocsExamined" and "docsExamined," 100,000 documents were accessed in order to return nine documents, as indicated by the value of "nReturned".

This time I will define an index and then execute the previous query and show a part of its execution plan. Please note that as of MongoDB 3.0, the ensureIndex() command that used to create an index is deprecated; you should use createIndex() instead. The index for the "x" key will be created as follows:

```
> db.myData.createIndex({"x":1})
{
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 1,
"numIndexesAfter" : 2,
"ok" : 1
}
```

You can verify that the index you just created is there with the help of the getIndexes() function that reads the system.indexes collection:

```
> db.myData.getIndexes()
[
{
"v" : 1,
"key" : {
"_id" : 1
},
"name" : "_id_",
"ns" : "login.myData"
},
{
"v" : 1,
"key" : {
"x" : 1
},
"name" : "x_1",
"ns" : "login.myData"
}
]
```

As you can see, the index for the "x" key is named "x_1". Please note that MongoDB automatically creates an index for the _id field for every collection.

By executing exactly the same explain() command, you can verify the usefulness of the index. The interesting part of the output is the following:

```
...
"executionStats" : {
"executionSuccess" : true,
"nReturned" : 9,
"executionTimeMillis" : 1,
"totalKeysExamined" : 9,
"totalDocsExamined" : 9,
...
"inputStage" : {
"stage" : "IXSCAN",
"nReturned" : 9,
"executionTimeMillisEstimate" : 0,
...
```

This query returned the results by scanning the index keys (IXSCAN); therefore, it is significantly faster than the previous query. As you can also see, the select query accessed only nine documents this time, as indicated by the value of "totalDocsExamined" in order to return nine documents, which is perfect!

Summary

This article is far from complete as no single article can cover all aspects of MongoDB administration. For example, Replication and Sharding were not covered at all. However, the commands and knowledge presented will help you start working effectively with a MongoDB database and perform many administrative tasks.

References

[1] MongoDB site: https://www.mongodb.org/.

[2] Kristina Chodorow, *MongoDB: The Definitive Guide*, 2nd edition (O'Reilly Media, 2013).

[3] Kyle Banker, Peter Bakkum, Shaun Verch, Douglas Garrett and Tim Hawkins, *MongoDB in Action*, 2nd edition (Manning Publications, 2016).

[4] Download MongoDB: https://www.mongodb.org/downloads.

[5] HomeBrew: http://brew.sh/.

[6] WiredTiger: http://www.wiredtiger.com/.

[7] mtools: https://github.com/rueckstiess/mtools.

[8] mtools installation: https://github.com/rueckstiess/mtools/blob/master/INSTALL.md.

Save the Date!

12th USENIX Symposium on Operating Systems Design and Implementation

November 2–4, 2016 • Savannah, GA

Join us in Savannah, GA, November 2–4, 2016, for the **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)**. The Symposium brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software.

Co-located with OSDI '16 on Tuesday, November 1:

- **Diversity '16:** 2016 Workshop on Supporting Diversity in Systems Research
- **INFLOW '16:** 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads

The full program and registration will be available in August.

www.usenix.org/osdi16

Linux at 25

PETER H. SALUS

Peter H. Salus is the author of *A Quarter Century of UNIX* (1994), *Casting the Net* (1995), and *The Daemon, the Gnu and the Penguin* (2008). peter@pedant.com

In June 1991, at the USENIX conference in Nashville, BSD NET-2 was announced. Two months later, on August 25, Linus Torvalds announced his new operating system on comp.os.minix. Today, Android, Google's version of Linux, is used on over two billion smartphones and other appliances. In this article, I provide some history about the early years of Linux.

Linus was born into the Swedish minority of Finland (about 5% of the five million Finns). He was a "math guy" throughout his schooling. Early on, he "inherited" a Commodore VIC-20 (released in June 1980) from his grandfather; in 1987 he spent his savings on a Sinclair QL (released in January 1984, the "Quantum Leap," with a Motorola 68008 running at 7.5 MHz and 128 kB of RAM, was intended for small businesses and the serious hobbyist). It ran Q-DOS, and it was what got Linus involved:

> One of the things I hated about the QL was that it had a read-only operating system. You couldn't change things …
>
> I bought a new assembler … and an editor.… Both … worked fine, but they were on the microdrives and couldn't be put on the EEPROM. So I wrote my own editor and assembler and used them for all my programming. Both were written in assembly language, which is incredibly stupid by today's standards. [1]

But look for a moment at Linus' posting of August 25, 1991: comp.os.minix:

> I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

Linus had ported bash (1.08) and gcc (1.40). More would come. In the fall of 1990, the University of Helsinki had installed its first UNIX machine, a MicroVAX running Ultrix. But Linus was "eager to work with Unix by experimenting with what I was learning in Andrew Tanenbaum's book" ([1], p. 53) and read all 700-odd pages of *Operating Systems* [Prentice-Hall, 1987]. The book "lived on my bed." *Operating Systems* came with the code for Minix, Tanenbaum's UNIX clone.

One of the things that struck Linus about UNIX was its openness. Another was its simplicity. And then came a bolt from the blue: in early 1991, Lars Wirzenius dragged Linus to the Polytechnic University of Helsinki to hear Richard Stallman. "I don't remember much about the talk," Linus says. "But I guess something from his speech must have sunk in. After all, I later ended up using the GPL for Linux."

On January 5, 1991, Linus got his father to drive to a "mom and pop" computer store, where he had ordered a no-name 4-Meg, 33 MHz, 386 box. He was 21. The box came with DOS, but Linus wanted Minix and ordered it. It took a month for the parcel to find its way to Finland, but it arrived and Linus fed the 16 diskettes to the machine. And then he began "playing" with it. The first thing he wrote was a terminal emulator: "That's how Linux got started. With my test programs turning into a terminal emulator."

Because Linus was truly dependent upon the Internet and (specifically) the comp.os.minix newsgroup, we can date events far more accurately than in earlier decades.

We know that Linus' first posting to comp.os.minix, asking about the POSIX standard, was July 3, 1991. And we can see his posting about "doing a (free) operating system (just a hobby, won't be big and professional like gnu) … This has been brewing since april …" of August 25, 1991.

There was a reasonable expression of interest. We thus know that Linus put what we would now call Linux 0.01 up on the University of Helsinki ftp site on September 17, 1991. "No more than one or two people ever checked it out," he said.

The following January there was discernible growth in the Linux community, leading (I think) to the online debate about kernels begun by Andy Tanenbaum on January 29, 1992. Although I don't want to go into detail, the debate began with Andy stating that microkernels were better than monolithic kernels, and that Linux was therefore already obsolete.

It is more important, in my opinion, that in the spring of 1992, Orest Zborowski ported X-windows to Linux and that, thanks to the Internet and to Usenet, the work of a hobbyist in Finland could be picked up elsewhere in Europe, in Australia, and in the US.

Also in 1992, Rémy Card wrote the extended file system for Linux, the first to use the virtual file system, modeled after the one in BSD. Later, Card went on to write ext2, which moved further away from the limitations of the Minix file system and was more like the BSD fast file system.

The number of Linux users continued to grow, as did the versions of the software: Linux .01 was 63 KB compressed. Only a few weeks later, on October 5, Linus posted .02; on December 19, v.11 was posted; and on January 5, 1992, v.12—108 KB compressed—appeared. On March 7, there was v.95 and on May 25, 1992, v.96 showed up, with support for X and taking up 174 KB compressed.

Ted Ts'o was the first North American Linux user. "There was fairly strong social cohesion," he told me. "Linux was the first big project to succeed in a distributed fashion."

The Birth of Distros

Following Linus' postings of 1991, there soon were what we have come to call "distributions." And, rather than utilizing ftp, they came on CD-ROM.

The first of these was Adam Richter's Yggdrasil. In the Old Norse Edda, Yggdrasil is the "world ash," from a branch of which Odin/Wotan made his spear. Yggdrasil alpha was released on December 8, 1992, and was called LGX: Linux/GNU/X—the three components of the system.

John Gilmore, Michael Tiemann, and David Henkel-Wallace formed Cygnus in 1989. Richter spoke to Michael Tiemann about setting up a business but was "definitely uninterested in joining forces with Cygnus."

Yggdrasil beta was released the next year. Richter's press release read:

> The Yggdrasil beta release is the first UNIX(R) clone to include multimedia facilities as part of its base configuration. The beta release also includes X-windows, networking … an easy installation mechanism, and the ability to run directly from the CD-ROM.

The beta was priced at $50; the production release was $99.

SuSE was also formed in 1992 as a consulting group. SuSE was originally S.u.S.E.—"Software-und-System-Entwicklung," or Software and System Development—but did not release a Linux distribution for several years. The next distribution—and the oldest still in existence—was Patrick Volkerding's Slackware, released July 16, 1993, soon after he graduated from Minnesota State University Moorhead. Slackware, in turn, was the basis for SuSE's release "Linux 1.0" of SLS/Slackware in 1994. SLS was "Softlanding Linux System," Peter MacDonald's 1992 distribution, on which parts of Slackware were based. SuSE later integrated Florian La Roche's Jurix distribution, resulting in a unique distribution: SuSE 4.2 (1996).

The next year, Mark Bolzern was trying to sell a UNIX database from Multisoft, a German company. He encountered difficulties because it was relatively expensive to set up the UNIX system. Then he came across Gnu/Linux and realized that he now had a real solution. He convinced Multisoft to port Flagship (the db) to Linux, and "that was the first commercial product released on Linux," Bolzern said.

"People were always having trouble installing Linux," he continued, "and then Flagship wouldn't run right because something had changed." Bolzern decided that what was needed was a release that wouldn't change for a year, so he "picked a specific distribution of Slackware" and "the name Linux Pro." Soon he was selling more Linux than Flagship: "we're talking hundreds per month."

And when Red Hat came out, Bolzern picked that up.

Marc Ewing had set up Red Hat in 1993. He said: "I started Red Hat to produce a development tool I thought the world needed. Linux was just becoming available and I used [it] as my development platform. However, I soon found that I was spending more time managing my Linux box than I was developing my software, and I concluded that what the world really needed was a good Linux distribution."

In 1993, Bob Young was working for Vernon Computer Rentals. He told me: "I knew the writing was on the wall for my future with that company [VCR]." He continued:

> Red Hat the company was legally incorporated in March of 1993 in Connecticut under the name ACC Corp., Inc. It changed its name to Red Hat Software, Inc. in early 1996, and changed its name a last time to simply Red Hat, Inc. just before going public in June of 1999.
>
> ACC Corp., Inc. bought the assets, including all copyrights and trademarks (none were registered at the time) relating to Marc Ewing's sole proprietorship business venture, in January 1993. Marc's Red Hat project was not incorporated but was run out of Marc's personal checking account. Marc received shares in ACC Corp, Inc. in return for the Red Hat name and assets.

In 1995 Red Hat packaged Linux, some utilities, and initial support for $50. Also in 1995, Bryan Sparks (with funding from Ray Noorda, former CEO of Novell) founded Caldera, and the Apache Software Foundation released Apache, which would become the most widespread Web server. But Red Hat soon became the most popular Linux release. This was unexpected: Linus had said that he expected Caldera to be the top supplier, because it was "kind of a step beyond" in that it was targeting the office market. "I think what's interesting about Caldera is they based their stuff on Red Hat and then they added a commercial kind of approach."

When Red Hat became a "success," Bob Young and family moved from Connecticut to North Carolina (Ewing lived in Durham).

ACC, Young's company, had sold Linux/UNIX software and books. Young had been introduced to the UNIX world in 1988, when he was with Vernon Leasing and Rentals, and began publishing New York UNIX as well as catalog sales. This led to his being the founding editor of *Linux Journal*, a post he held for two issues in 1994, before "splitting the losses" with Phil Hughes, the publisher of *LJ*.

On November 5, 1993, Linus spoke at the NLUUG (Netherlands UNIX Users' Group).

On March 12, 1994, Linus released Linux 1.0, basically v0.99, patch level 157. It was the first stable kernel distribution.

I don't want to go into extensive detail here, but I think that there are a number of important points to be made:

1. The birth, growth, and development of Linux were totally unorganized.
2. It was geographically well-distributed.
3. It was conducted via the Internet.

In the summer of 1995, I was approached by Lisa Bloch, then the Executive Director of the Free Software Foundation (FSF), as to the feasibility of a conference on "Freely Redistributable Software." I was enthusiastic but had my qualms about profitability. Richard Stallman, at our meeting, was quite understanding: FSF would bankroll the affair, but he hoped we could turn a small profit.

Lisa and I put together a committee (Bob Chassell, Chris Demetriou, John Gilmore, Kirk McKusick, Rich Morin, Eric Raymond, and Vernor Vinge) and we posted a Call for Papers on several newsgroups.

Thanks to "Maddog" (Jon Hall), Linus agreed to be a keynote speaker, and Stallman was the other. We had a day of tutorials and two days of papers (February 3–5, 1996, at the Cambridge Center Marriott). Half a dozen distributions were represented, and everything ran smoothly. By the end, I was a nervous wreck, and the FSF ended up making a tiny profit.

Debian Linux was created by Ian Murdock (Debian = Debbie + Ian), who officially founded the "Project" on August 16, 1993. From November 1994 to November 1995, the Debian Project was sponsored by the FSF.

In November 1995, Infomagic released an experimental version of Debian which was only partially in ELF format as "Debian 1.0." On December 11, Debian and Infomagic jointly announced that this release "was screwed." Bruce Perens, who had succeeded Murdock as "leader," said that the data placed on the 5-CD set would most likely not even boot.

The real result was that the "real" release, Buzz, was 1.1 (June 17, 1996), with 474 packages. Bruce was employed by Pixar and so all Debian releases are named after characters in *Toy Story* (1995):

- 1.2 Rex, December 12, 1996 (848 packages)
- 1.3 Bo, June 5, 1997 (974 packages)
- 2.0 Hamm, July 24, 1998 ("over 1500 packages")
- 2.1 Slink, March 9, 1999 ("about 2250 packages")
- 2.2 Potato, August 15, 2000 ("more than 3900 binary packages")
- 3.0 Woody, July 19, 2002 (8500 binary packages)
- 3.1 Sarge, June 6, 2005 (15,400 packages)
- 4.0 Etch (obsolete)
- 5.0 Lenny (obsolete)
- 6.0 Squeeze (obsolete)
- 7 Wheezy (obsolete)
- 8 Jessie (current stable release)
- 9 Stretch

Buzz fit on one CD, Slink went to two, and Sarge is on 14 CDs in the official set. It was released fully translated into over 30 languages and contains a new debian-installer. Slink had also introduced ports to the Alpha and Sparc. In 1999, Debian also began a Hurd port.

Although Debian carried the burden of being tough to install for several years, Sarge changed that. The new installer with automatic hardware detection was quite remarkable. That's why I've reduced the detail over the next decade.

At this point, I'd like to introduce Mandrake, a Linux distribution based on Red Hat 5.1 and KDE. KDE was a joke on CDE (Common Desktop Environment), begun by Matthias Estrich in Tuebingen in 1996. Mandrake was created by Gael Duval, a graduate of Caen University, in July 1998. From 1998 to early 2004, Mandrake was reasonably successful, notable for its high degree of internationalization as well as for the variety of chips it would run on. However, in February 2004 Mandrakesoft lost a suit filed by the Hearst Syndicate, which claimed invasion of their trademarked "Mandrake the Magician." Starting with 10.0, there was a minor name change. Then, in April 2005, Mandrakesoft announced that there was a merger with Conectiva and that the new name would be Mandriva.

Joseph Cheek founded Redmond Linux in 2000. In 2001 it merged with DeepLinux. In January 2002, the company was renamed Lycoris, and its Desktop/LX was based on Caldera's Workstation 3.1. In June 2005, Lycoris was acquired by Mandriva.

It might be a full-time job to track all the distributions and their origins. For instance, Kanotix is a Debian derivative. It is also a Knoppix derivative, as it is a live CD, and it is solid as a rock.

Knoppix was created by Klaus Knopper, a freelance IT/Linux consultant. It has achieved popularity because it is easily run from the CD, without installation, and because it can be readily employed to fix corrupted file systems, etc. It was the first Linux on a live CD.

The last time I attempted a tally of Linux distributions, about ten years ago, there were well over 100; over a dozen might be considered popular. But then the world changed. Android lurched onto the scene. Initially developed by Android, Inc., which Google bought in 2005, Android was unveiled in 2007.

Just look how far that (free) operating system has come:

- 2013: Android claims 75% of the smartphone market share, in terms of the number of phones shipped (a total of 1.859 billion in 2015).
- 2014: Ubuntu claims 22 million users.
- 2015: Version 4.0 of the Linux kernel is released.

Statista projects 2.8 billion smartphone users worldwide in 2016. Over 2 billion of them employ Android.

Happy 25th birthday, Linux, and thank you, Linus!

Further details of the birth and development of Linux may be found in my book *The Daemon, the Gnu, and the Penguin* (Reed Media Services, 2008).

Reference

[1] L. Torvalds, *Just for Fun* (HarperCollins, 2001), pp. 45, 53.

;login: 2016 Publishing Schedule

;login: has changed from a bimonthly to a quarterly schedule, with four issues per year. Below is the publishing schedule for the rest of 2016.

Issue	Article Drafts Due	Final Articles Due	Columns Due	Proofs to Authors	Issue Mailing Date
Fall	June 6	June 13	June 27	August 1	September 1
Winter	September 6	September 13	September 20	October 24	November 26

COLUMNS

Precious Memory

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

Most of the Python code that I write isn't part of an exotic framework or huge application. Instead, it's usually related to a mundane data analysis task involving a CSV file. It isn't glamorous, but Python is an effective tool at getting the job done without too much fuss. When working on such problems, I prefer to not worry too much about low-level details (I just want the final answer). However, if you use Python for manipulating a lot of data, you may find that your scripts use a large amount of memory. In this article, I'm going to peek under the covers of how memory gets used in a Python program and explore options for using it more efficiently. I'll also look at some techniques for exploring and measuring the memory use of your programs. Disclosure: Python 3 is assumed for all of the examples, but the underlying principles apply equally to Python 2.

Reading a Large CSV File

My Chicago office is located along a major bus route, the trusty #22 that will take me down the road to Wrigley Field if I want to avoid work during the summer. It tends to be a pretty busy route, but just how busy? Chicago, being a data-friendly city, has historical bus ridership data posted online [1]. You can download it as a CSV file. If you do, you'll get a 13.8 MB file with 676,476 lines of data that give you the ridership of every bus route in the city on every day of the year going back to the year 2001. It looks like this:

```
route,date,daytype,rides
3,01/01/2001,U,7354
4,01/01/2001,U,9288
6,01/01/2001,U,6048
8,01/01/2001,U,6309
9,01/01/2001,U,11207
...
```

By modern standards, a 13.8 MB CSV file isn't so large. Thus, I'm inclined to grab it using Python's `csv` module. Problem solved:

```
>>> import csv
>>> with open('cta.csv') as f:
...     rows = list(csv.DictReader(f))
...
>>> len(rows)
676476
>>> rows[0]
{'date': '01/01/2001', 'route': '3', 'rides': '7354', 'daytype': 'U'}
>>>
```

Now let's tabulate the ridership totals across all of the bus routes using the `collections` module:

```
>>> from collections import Counter
>>> ride_counts = Counter()
>>> for row in rows:
...     ride_counts[row['route']] += int(row['rides'])
...
>>> ride_counts['22']
104039097
>>>
```

While we're at it, why don't we find out the five most common bus routes.

```
>>> ride_counts.most_common(5)
[('79', 153736884), ('9', 138645554), ('49', 113908939),
 ('4', 111154851), ('66', 110746972)]
>>>
```

Great. Before you quit, however, go look at the memory use of the Python interpreter in your system process viewer—you'll find that it's using nearly 300 MB of RAM (maybe more). Yikes! For a 13.8 MB input file, that sure seems like a lot—almost as much as some of the minimally useful apps on my phone. The horror.

Measuring Memory Use

Measuring the memory use of a Python program in a portable way was not an entirely easy task until somewhat recently. Yes, you could always go view the Python process in the system task viewer, but there were no standard library modules to help you out. This changed somewhat in Python 2.6 with the addition of the `sys.getsizeof()` function. It lets you determine the size in bytes of individual objects. For example:

```
>>> import sys
>>> a = 42
>>> sys.getsizeof(a)
28
>>> b = 'hello world'
>>> sys.getsizeof(b)
60
>>>
```

Unfortunately, the usefulness of `sys.getsizeof()` is a bit limited. For containers such as lists and dicts, it only reports the size of the container itself, not the cumulative sizes of the items contained inside. It's subtle, but you can see this yourself if you look carefully at this example where the combined size of two items in a list is smaller than the reported size of the list itself:

```
>>> a = 'hello'
>>> b = 'world'
>>> items = [a, b]
>>> sys.getsizeof(a)
54
>>> sys.getsizeof(b)
54
>>> sys.getsizeof(items)   # Notice size is less than combined
                           # a, b size
80
>>>
```

Containers also present complications in determining an accurate use. For example, the same object might appear more than once such as in a list of `[a, a, b, b]`. Also, Python tends to aggressively share immutable values under the covers. So it's not a simple case where you can just add up the byte totals for all of the items in a container and get an accurate figure. Instead you'd need to gather information on all unique objects using their object IDs like this:

```
>>> items = [a, a, b, b]
>>> unique_items = { id(item): sys.getsizeof(item) for item
      in items }
>>> total_size = sys.getsizeof(items) + sum(unique_items.
      values())
204
>>>
```

If you had deeply nested data structures, you'd have to take further steps to recursively traverse the entire data structure. Needless to say, it gets ugly. Just to illustrate, here's how you would measure the memory usage of the list holding all of that bus data.

```
>>> unique_objects = { id(rows): rows }
>>> unique_objects.update((id(row), row) for row in rows)
>>> unique_objects.update((id(val), val) for row in rows for
      val in row.values())
>>> sum(sys.getsizeof(val) for val in unique_objects.values())
308977196
>>>
```

Starting in Python 3.4, you can obtain global memory statistics using the `tracemalloc` module [2]. This module allows you to selectively monitor the memory use of Python and have it record memory allocations. It's not so useful for small measurements, but you can use it in a script:

```
import tracemalloc
import csv

def read_data(filename):
    with open(filename) as f:
        return list(csv.DictReader(f))

tracemalloc.start()
rows = read_data('cta.csv')
print(len(rows), 'Rows')
print('Current: %d, Peak %d' % tracemalloc.get_traced_memory())
```

If I run this on my machine with Python 3.5, I get the following output:

```
676476 Rows
Current: 308979047, Peak 309009543
```

The reported memory use is ever so slightly higher than what was calculated directly with `sys.getsizeof()`, but basically the two figures agree.

Exploring Common Data Structure Choices

Given the large memory footprint associated with reading this file, you might consider other choices for representing a simple record, such as a list, tuple, or class instance. Here are several different functions that read the data in different forms:

```
import csv

def read_data_as_dicts(filename):
    with open(filename) as f:
        return list(csv.DictReader(f))

def read_data_as_lists(filename):
    with open(filename) as f:
        rows = csv.reader(f)
        headers = next(rows)
        return list(rows)

def read_data_as_tuples(filename):
    with open(filename) as f:
        rows = csv.reader(f)
        headers = next(rows)
        return [tuple(row) for row in rows]

class RideData(object):
    def __init__(self, route, date, daytype, rides):
        self.route = route
        self.date = date
        self.daytype = daytype
        self.rides = rides

def read_data_as_instances(filename):
    with open(filename) as f:
        rows = csv.reader(f)
        headers = next(rows)
        return [ RideData(*row) for row in rows ]
```

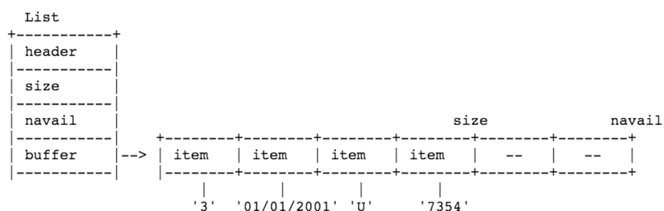
If you run and measure these different functions using `tracemalloc`, you will get memory use as follows:

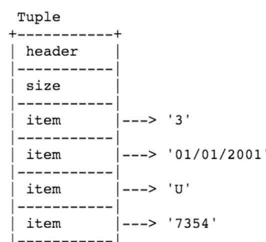
Record Type	Memory Use (MB)
Dict	294.7
List	170.8
Tuple	160.1
Instance	268.5

In these results, you find that tuples provide the most efficient storage. This shouldn't be a surprise, but there are still some subtle aspects to the results. For example, what explains the 10 MB gap between tuples and lists? On the surface it doesn't seem like there would be much difference between the two structures given that they're both "list like." We can investigate with `sys.getsizeof()`:

```
>>> a = ('3', '01/01/2001', 'U', '7354')
>>> b = ['3', '01/01/2001', 'U', '7354']
>>> import sys
>>> sys.getsizeof(a)
80
>>> sys.getsizeof(b)
96
>>>
```

Here, we find that there is a 16-byte difference in storage between a list and tuple. Added up across the 676,476 rows of data, that amounts to about 10 MB of storage. The 16-byte difference is due to the fact that lists are a little more complicated than they might first seem. For one, since lists are mutable, their size can change as elements are added or removed. To manage this, lists internally contain a memory pointer to a resizable memory buffer where items are stored. Tuples, being immutable, don't have to handle resizing. Thus, the items in a tuple can be stored directly at the end of the underlying tuple structure. Lists also overallocate their internal storage so as to make repeated `append()` operations faster (this is to minimize a potentially expensive memory reallocation each time a new element is added). For example, a list containing only five items might actually have room to store eight items without asking for more space. To manage this, lists maintain an extra counter of how much total space is available in addition to a counter that records the actual number of elements used. Here is a diagram that illustrates the difference in the memory layout of a tuple versus a list:





The header portion contains some bookkeeping information, including the object's type and the reference count used in memory management. This is the same for all objects. The 16-byte difference in tuple/list storage is explained by the presence of an extra memory pointer (buffer) and counter (navail) on lists. Depending on the amount of unused space, lists might even be a bit larger.

Another surprising result is the efficiency of instances over dictionaries—especially if you happen to know that instances are actually built using dictionaries. For example:

```
>>> r = RideData('3', '01/01/2001', 'U', '7354')
>>> r.__dict__
{ 'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': '7354'
}
>>>
```

Thus, what explains the 26 MB advantage of using instances over dictionaries? As it turns out, this is also another memory optimization. When creating a lot of instances, Python makes an assumption that the dictionaries for all of the instances will probably contain the exact same set of keys. It makes sense—all objects are initialized in `__init__()` and are likely to have an identical underlying structure. Python exploits this and creates what's known as a key-sharing dictionary as described in PEP 412 [3]. In a nutshell, the keys for the instance data are split off from the normal dictionary and stored in a shared structure. It makes for a slightly smaller dictionary structure. You can investigate:

```
>>> c = { 'route': '3', 'date': '01/01/2001', 'daytype': 'U',
'rides': '7354' }
>>> sys.getsizeof(c)
288
>>> d = RideData('3', '01/01/2001', 'U', '7354')
>>> d
<__main__.RideData object at 0x101ad4f60>
>>> d.__dict__
{ 'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': '7354'
}
>>> sys.getsizeof(d.__dict__)  # Size of instance dict
192
>>>
```

Here, you see that the instance dictionary is quite a bit smaller than a normal dictionary. However, you can't forget that instances also contain some state, including the class and reference count:

```
>>> sys.getsizeof(d)   # Size of the instance structure
56
>>>
```

So, in this example, you'll find that an instance requires 56 bytes of storage plus the storage required for the instance dictionary. Added together, you find that an instance requires 248 bytes vs. 288 bytes for a normal dictionary. Multiplied by the 676,476 records, you get a savings of about 26 MB.

Named Tuples

Tuples are efficient, but one downside is that they often lead to code where you do a lot of ugly indexing. For example:

```
>>> rows = read_data_as_tuples('cta.csv')
>>> from collections import Counter
>>> ride_counts = Counter()
>>> for row in rows:
...     ride_counts[row[0]] += int(row[3])
...
>>>
```

You can clean this up using the `namedtuple()` function to define a class. For example:

```
from collections import namedtuple
RideTuple = namedtuple('RideTuple', ['route','date','daytype',
'rides'])
```

The `namedtuple()` function performs a neat trick using properties that produces a class roughly equivalent to this:

```
class RideTuple(tuple):
    __slots__ = ()       # Explained in next section
    @property
    def route(self):
        return self[0]
    @property
    def date(self):
        return self[1]
    @property
    def daytype(self):
        return self[2]
    @property
    def rides(self):
        return self[3]
```

In this class, properties have been added to pull attributes from a specific tuple index. This gives you nice access to those values via the dot (.) operator. For example:

```
>>> r = RideTuple('3','01/01/2001','U','7354')
>>> r.route
'3'
>>> r.date
'01/01/2001'
>>> r[0]
'3'
```

```
>>> r[1]
'01/01/2001'
>>>
```

Named tuples also offer a cautionary tale of measuring Python's memory use—namely, that you can't always trust it to tell you the truth! For example, suppose you measure the memory of a single named tuple versus a tuple:

```
>>> a = ('3', '01/01/2001', 'U', 7354')
>>> b = RideTuple('3','01/01/2001','U','7354')
>>> sys.getsizeof(a)
80
>>> sys.getsizeof(b)
80
>>>
```

Here, you will find that the memory is identical. That looks good. However, if you run two versions of code under `tracemalloc`, you'll find that they have different behavior.

Record Type	Memory Use (MB)
Tuple	160.1
Named tuple	165.3

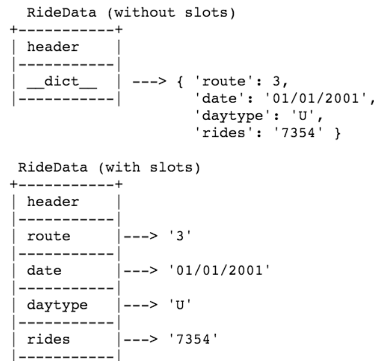
For reasons unknown, named tuples allocate an extra machine word (8 bytes on a 64-bit machine) for each instance. Added up over the 676,476 rows of our data set, that amounts to an extra 5 MB. If there's any takeaway, the results of `sys.getsizeof()` are not always to be trusted. If you must know, objects self-report their size using a special method `__sizeof__()` which could be implemented incorrectly. If you really care about accuracy, it's a good idea to measure memory use a few different ways.

Slots

A somewhat lesser known technique for saving memory is to define a class with a `__slots__` specifier like this:

```
class RideData(object):
    __slots__ = ('route', 'date', 'daytype', 'rides')
    def __init__(self, route, date, daytype, rides):
        self.route = route
        self.date = date
        self.daytype = daytype
        self.rides = rides
```

Normally, instances are represented by a dictionary. However, if you use slots, you're giving a hint about how many attributes will be stored. Python uses this to eliminate the instance dictionary and rearrange the storage of attributes into something that looks a lot like a tuple. Here is a diagram showing how instances are stored with and without slots:



Remarkably, a class that uses slots is even slightly more efficient than one using a tuple. For example, if you run a test under `tracemalloc`, you'll get these results:

Record Type	Memory Use (MB)
Tuple	160.1
Instance with slots	155.0

The savings is due to the fact that unlike a tuple, instances don't support indexing of attributes (e.g., `r[n]`). Thus, it is not necessary for a size to be stored on a per-instance basis. The attributes are merely loaded and stored from a hardwired position known in advance. The exact mechanism is almost exactly the same as the attribute properties defined on a named tuple.

Using the Appropriate Datatypes

In our example, we were being lazy and storing the numeric ride data as a string (e.g., `'7354'`) instead of as an integer (7354). However, strings are not the most efficient representation. Let's explore:

```
>>> a = '7354'
>>> b = 7354
>>> c = 7354.0
>>> sys.getsizeof(a)
53
>>> sys.getsizeof(b)
28
>>> sys.getsizeof(c)
24
>>>
```

As you can see, storing the number as an integer saves 25 bytes. However, storing the value as a floating point number saves a bit more. Integers require more space because they are allowed to grow to arbitrary magnitude. To handle this, they must not only store the integer value, but some additional sizing information. Floats don't need this.

By changing just one column of the data to a float, we save about 18 MB of memory. So being smart about what you store makes a difference.

Value Sharing

Under the covers, Python memory management is based on memory pointers. For example, suppose you make a list and "copy" it to another variable:

```
>>> a = [1,2,3]
>>> b = a
>>>
```

This didn't actually make a copy of the list. Instead, the names "a" and "b" both refer to the same object. If you change the list, it's reflected in both variables.

```
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> b
[1, 2, 3, 4]
>>>
```

The `id()` function will give you the object identity, a unique integer value. You can use this to see that `a` and `b` in the above example are the same object.

```
>>> id(a)
56623488
>>> id(b)
56623488
>>>
```

Now, how to use this? When reading certain kinds of data sets, you might encounter a lot of repetition. To illustrate, let's grab the bus data again.

```
>>> f = open('cta.csv')
>>> rows = list(csv.DictReader(f))
>>> unique_routes = set(row['route'] for row in rows)
>>> len(unique_routes)
182
>>> route_ids = set(id(row['route']) for row in rows)
>>> len(route_ids)
634285
>>>
```

What you're seeing here is that the data contains only 182 unique values for the "route" field, yet those values are stored in 634,285 unique objects. It's a bit odd that there aren't 676,476 unique values corresponding to the length of the entire data set. As it turns out, Python caches objects representing all single-letter ASCII strings. Thus routes 1–9 get special treatment. You can verify this:

```
>>> route_ids = set(id(row['route']) for row in rows if
len(row['route'])==1)
>>> len(route_ids)
9
>>>
```

Perhaps you can take a similar caching strategy for reusing the rest of the values. Here is a simple function that caches strings:

```
def cache(value, _values = {}):
    if value not in _values:
        _values[value] = value
    return _values[value]
```

Next, you can apply the cache function to selected values during instance creation. For example:

```
class RideData(object):
    __slots__ = ['route','date','daytype','rides']
    def __init__(self, route, date, daytype, rides):
        self.route = cache(route)
        self.date = cache(date)
        self.daytype = daytype
        self.rides = float(rides)
```

Making this change, the storage required for our example data is reduced down to about 68 MB—not too bad considering it started out at over 300 MB.

Changing Your Orientation

So far, we have worked to represent the data as a list of records—varying the representation of each record. However, another approach is to turn everything sideways and represent the data as a collection of columns. For example, suppose you read the data using this function:

```
def read_data_as_columns(filename):
    route = []
    date = []
    daytype = []
    rides = []
    with open(filename) as f:
        for row in csv.DictReader(f):
            route.append(cache(row['route']))
            date.append(cache(row['date']))
            daytype.append(row['daytype'])
            rides.append(float(row['rides']))

    return {
        'route': route,
        'date': date,
        'daytype': daytype,
        'rides': rides
        }
```

Making this change reduces the memory use to about 38 MB. However, it also shatters your head as working with the resulting data is wacky. Instead of getting a single list of records, you get four lists representing each column. For example:

```
>>> columns = read_data_as_columns('cta.csv')
>>> len(columns)
4
>>> columns['route'][0]
'3'
>>> columns['date'][0]
'01/01/2001'
>>>
```

Yes, you can work with the data like this, but doing so might require a bit of ingenuity and increase your job security. You would probably be better off using a third party library such as Pandas, which also stores its data in a column form [4]. This brings us to the last important point about memory. Third party libraries often rely on C extensions and code outside of Python that can't be measured accurately using the tools described here. For example, you can try this experiment with Pandas:

```
>>> import pandas
>>> import tracemalloc
>>> tracemalloc.start()
>>> data = pandas.read_csv('cta.csv')
>>> tracemalloc.get_traced_memory()
(433375, 471219)
>>> import sys
>>> sys.getsizeof(data)
135868754
>>>
```

Pandas is efficient, but it's not so efficient that it's storing all of the data in only 430 KB. Nor is the reported size of the `data` variable 135 MB. A look in the task viewer shows Python actually using about 56 MB of memory. Bottom line: if you're using certain kinds of Python extensions, the memory profiling tools described here might not work.

If You Liked It, You Should Have Put a Generator on It

In the end, maybe it's best to ask yourself if you actually need to read all of the data at once. Perhaps a generator function can do the trick:

```
import csv
from collections import Counter
def read_data(filename):
    with open(filename) as f:
        rows = csv.DictReader(f)
        for row in rows:
            yield { **row, 'rides':int(row['rides']) }

ride_counts = Counter()
for row in read_data('cta.csv'):
    ride_counts[row['route']] += row['rides']
```

If you run this version under `tracemalloc`, you'll find that it tabulates all of the data and uses only 36K of memory. Yes, generators are your friend.

Final Thoughts

This article has looked at a variety of issues surrounding Python memory use. There are probably a few important takeaways. First, there are some built-in tools such as `sys.getsizeof()` and the `tracemalloc` that you can use to investigate the memory use of your program. They're not always reliable, but when used in combination, you can often get a pretty good idea of what's happening. Second, there are a variety of ways in which you can represent data to reduce the memory footprint. For example, using `__slots__` in a class definition. Small details, such as your choice of low-level data representation and value sharing with caching, can also make a big impact. Last but not least, different data organizations (e.g., rows vs. columns) can be important.

References

[1] CTA-Ridership Data: https://data.cityofchicago.org/Transportation/CTA-Ridership-Bus-Routes-Daily-Totals-by-Route/jyb9-n7fm.

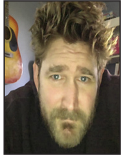
[2] `tracemalloc` module: https://docs.python.org/3/library/tracemalloc.

[3] PEP 0412 -- Key-Sharing Dictionary: https://www.python.org/dev/peps/pep-0412/.

[4] Pandas: pandas.pydata.org.

iVoyeur
Go Instrument Some Stuff

DAVE JOSEPHSEN



Dave Josephsen is the sometime book-authoring developer evangelist at Librato.com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

I hate talking about programming languages. Have you heard of Alcibiades? He was, well, I guess you could say he was the frat boy of ancient Greece. The original Bro. There's this pretty funny story about him where he was wrestling this other guy (I forget his name), and Alcibiades, feeling that he was about to lose, bit him on the ear. Yes, exactly like Mike Tyson.

Now biting your wrestling partner was, in ancient Greece, every bit as frowned upon as it is today. It is not, suffice to say, a statutorily valid attack in wrestling among gentlepersons. And Alcibiades' opponent didn't have any qualms about letting him know; his quote (according to Plutarch, who wasn't there) was:

"Alcibiades! You bite! Like a woman!"

Setting aside for the moment the Grecian misogyny—for which I apologize on their behalf (as a male, not as a Grecian)—I think Alcibiades' opponent was expressing a few overlapping emotions here. There was, of course, the outrage at having been bitten (especially in the course of, no doubt, well-executed by-the-book wrestling on his part). And then there was the surprise at having been bitten by Alcibiades, who was (despite ample evidence to the contrary) always assumed to be a stand-up bro by those who hung out with him. And, finally, there was the shaming component of the accusation, the part where he called Alcibiades out by comparing him to the so-called "weaker sex."

I'm sure everyone in attendance thought this was a pretty slick burn. I can almost hear the room explode with the ancient Grecian analog of "Oh *snap*!" And I mean he deserved it, right? Surely everyone could agree that biting was not only illegal, but also un-bro-like, which put Alcibiades clearly in violation of not only the wrestling rule book, but also, and probably more importantly, bro-code—or whatever you want to call that unwritten collection of etiquette particular to those people in that place. The former violation merely lost him the match, but it was the latter that made him worthy of disdain. But *then* Alcibiades replies:

"Nay. I bite like a *lion*!"

He did this sort of thing all the time; just running roughshod over the rules and undermining anyone who called him on it. It was basically his thing, and he did it so confoundingly well that he *always* got away with it. He just didn't consider defying convention something to be ashamed of and was therefore immune to this sort of politesse-rooted shaming.

Programming Languages

When I talk about programming languages, I always wind up feeling just exactly like the guy Alcibiades bit must have felt, which is to say: pretty sure what I just said was technically correct, but no longer convinced that it matters, and therefore confused about my place in the universe.

I'm not possessed of a Herculean-strength intellect, and I struggle to learn these languages just like the guy Alcibiades bit no doubt worked hard to master wrestling. Everyone assured us both it was the right and proper thing to do (it says so right there in the introduction sec-

tion of the O'Reilly book you wrote about the awesome new language you designed). And like the guy Alcibiades bit, I maintain this assumption that we, the community of people who struggled to learn Ruby or Python or Java or wrestling or whatever, have an understanding about what it means to be "good" or "bad" when we go about it. About the merits of this or that programming philosophy. About what constitutes an acceptable degree of inefficiency. About what is and is not secure.

But really, we don't have anything like that understanding. And bite by bite, I'm slowly beginning to realize that I will never have whatever equates to moral high-ground with respect to programming languages. That in fact maybe there never was such a thing, it just looked that way because my world was so small. There will never be *that* language that everyone who uses programming languages can finally maintain at least a begrudging respect for. Maybe that's a good thing, but it also means I am forever doomed to happily enter into excited conversation about this or that thing we're building only to be bitten on the ear over the language we chose to build it in.

I tire of this—this stupefied grasping at my bloody ear—and it's making me gun-shy. I've wanted to write this article for months, but I keep on balking because I know we're going to have to talk about languages. Well, at least one language, and worse, I'm going to have to *pick* it. I'm going to have to, once again, admit to liking a programming language, or at least admit to using one. My ear hurts just thinking about it.

Instrumenting Golang

Oh well. Let's get this show on the road. I want to talk about instrumenting the programs you create. And I'm going to do it in Golang, so deal with it.

When I say instrument, I'm talking about actually placing code inside the things you write that is designed to either time an interaction or quantify how often something occurs. It's easier if I just show you.

To that end I've written a Web service. It's pretty typical of the sort of thing I do when I wear my Ops hat these days: a simple program that listens for HTTP GET requests on port 80 and exposes some bit of operational knowledge to whatever happens to be asking. This one responds with an "answer." Here's what an answer looks like:

```
type Answer struct {
  Type  string
  Desc  string
  Get   func(index int) string
  Rand  func() string
  DB    []string
}
```

Even if you don't speak Go, this should be pretty obvious: an answer is a data structure that consists of a type; a description; two functions, one for getting specific answers and another for getting random answers; and an array of strings where we keep all of our responses.

I won't have space to paste all of the code for this project here, but you can clone it from GitHub (https://github.com/djosephsen/answers), which means you can also `go get` it with `go get github.com/djosephsen/answers`. You'll find the source under the src directory in your `GOPATH`.

Since one likes to be modular about these things, the code is designed so that we can come along later and add new types of "answer" and register them into a global index of answers we can give. If you look in the root directory, you'll see that it comes with two types of answer modules, one that provides answers to the question "Why did the chicken cross the road?" and one that provides answers to the question "Knock-knock. Who's there?"

In `main.go`, you'll see that after we go about registering the available modules into the global index of answers with `initAnswers()`, we use the `net/http` module to register two Handler functions with `net/http` and then start listening on port 8080.

```
func main() {
  initAnswers()
  metrics.Connect()
  http.HandleFunc("/", helpHandler)
  http.HandleFunc("/get/", getHandler)
  http.ListenAndServe(":8080", nil)
}
```

Now, ignoring `metrics.Connect()` for a moment, I think you can kind of see where this is going. If a user gets '/', we respond with a help menu, but if you ask for something that begins with '/get/', we launch `getHandler()`.

The help handler traverses all the answers we know about and prints back a list of URLs of the form `/get/answer.Type` followed by the `answer.Desc` that corresponds to the Type. We can see what it looks like when we use curl to ask for '/'.

```
(osapi) [dave@otokami][librato-vagrant]  [master|+ 1]
-> curl localhost:8080
Welcome to the answer service:
Valid answers:
/get/chicken :: Answers to why did the chicken cross the road?
/get/knocknock :: Knock Knock jokes1
```

Then when we ask for "/get/chicken" the getHandler function fires, and we get a random answer from the chicken module via that module's `Rand()` function. We can see what it looks like from `curl`:

```
(osapi) [dave@otokami][librato-vagrant]  [master↓.2+ 1]
-> curl localhost:8080/get/chicken
because the road crossed him
```

If you write Go, this code should be pretty familiar. It's a classic pattern for using `net/http` to write Web services in Go. In fact it began life as a copy/paste straight out of the `net/http` documentation. I'm not going to delve into the answer modules, because to instrument this application we don't need to leave `main.go`.

What we want to do is time and quantify our calls to the various handlers this program has now, as well as any that we might add in the future. In other words, we want to know how often `get/chicken` is called, and we want to know how long `get/chicken` takes to do what it does. And we don't just want that for `get/chicken`, we also want it for the help handler, `get/knocknock`, and any other answer modules we might add in the future. And we're lazy so we don't want to add new instrumentation every time we add another answer module.

And there's one more problem that you'll already be aware of if you're in the habit of timing function calls. Sometimes, aberrant measurements occur, such as calls to `get/chicken` that take three seconds because of bogons (possibly in the monitoring code) that we'll never be able to effectively track down. So we're going to want to generate percentiles for our timing measurements. We want to put extremely aberrant measurements in perspective. Is it one time in a million, or is it one time in a hundred?

What I'm very intentionally describing here is the use-case for which StatsD [1] was invented. We use it all over the place at Librato, and if you write services like these you probably should too unless you have something better. Specifically, we run Statsite [2] (a StatsD clone written in C (because native StatsD is a NodeJS daemon (lol programming languages))) on every instance we bring up. That way we can emit metrics directly from each instance to our metrics backend rather than risking packet-loss over the wire to a centralized StatsD instance (StatsD is a UDP protocol).

In this project, I'm using Etsy's `statsd` client, which is imported by my code in `metrics/metrics.go`. It provides a decent set of primitives but doesn't really give us what I'd call a Go-idiomatic means (lol programming languages) of implementing what we want here, so I have a few functions in `metrics.go` to help the client out. Let's take a look at my `Time()`:

```
func Time(name string, start time.Time) {
  if client == nil {
    return
  }
  now := time.Now()
  duration := now.Sub(start)
  if duration > 5000*time.Millisecond {
    fmt.Printf("Latent measurement for %q: %s", name, duration)
  }
  milliseconds := int64(duration / time.Millisecond)
  toStatsd(func() {
    // record the duration
    client.Timing(name, milliseconds)
    // also record a count
    client.UpdateStats([]string{fmt.Sprintf("%s.count", name)},
                                     1, 1)
  })
}
```

You'll find this function in several of our Go projects at Librato, and it's pretty clever. It takes a start-time and the name of the metric we want to show up in the metrics backend. It then computes the difference between the given start time and now and sends that duration into `statsd`. But wait, you're wondering, how does that time the actual function invocation? Stand by, that's the clever part. But before I get to that, you'll notice that it sends the timing to StatsD by way of a local `toStatsd()` function, which in turn takes a function with no arguments as its argument. Let's look at `toStatsd()`:

```
func toStatsd(fn func()) {
  start := time.Now()
  fn()
  duration := time.Now().Sub(start)
  if duration > 250*time.Millisecond {
    fmt.Printf("Statsd time took %s", duration)
  }
}
```

This was probably more along the lines of what you expected to see in `Time()`. This function captures the before time, runs the given function, and then captures the after time. If you look back up at `Time()`, you'll notice that we're passing `toStatsd()` an anonymous function that sends in the actual timing metric and increments a counter. So really `toStatsd()` is just a wrapper to time how long it takes the `statsd` client itself to do its thing. We're actually measuring how much latency `statsd` itself incurs.

Now let's bring this full circle by taking a look at `getHandler()` in `main.go` to see how we use `metrics.Time()`:

```
func getHandler(w http.ResponseWriter, r *http.Request) {
  answerType := r.URL.Path[len("/get/"):]
  defer metrics.Time("answer.handler."+answerType, time.Now())
  fmt.Fprintf(w, "%s\n",a.Answers[answerType].Rand())
}
```

Ah hah, the plot thickens. We're calling `metrics.Time` in a `defer` statement. If you don't program in Go, the `defer` statement is used to postpone the execution of a given function until just before the parent function returns. The interesting part about

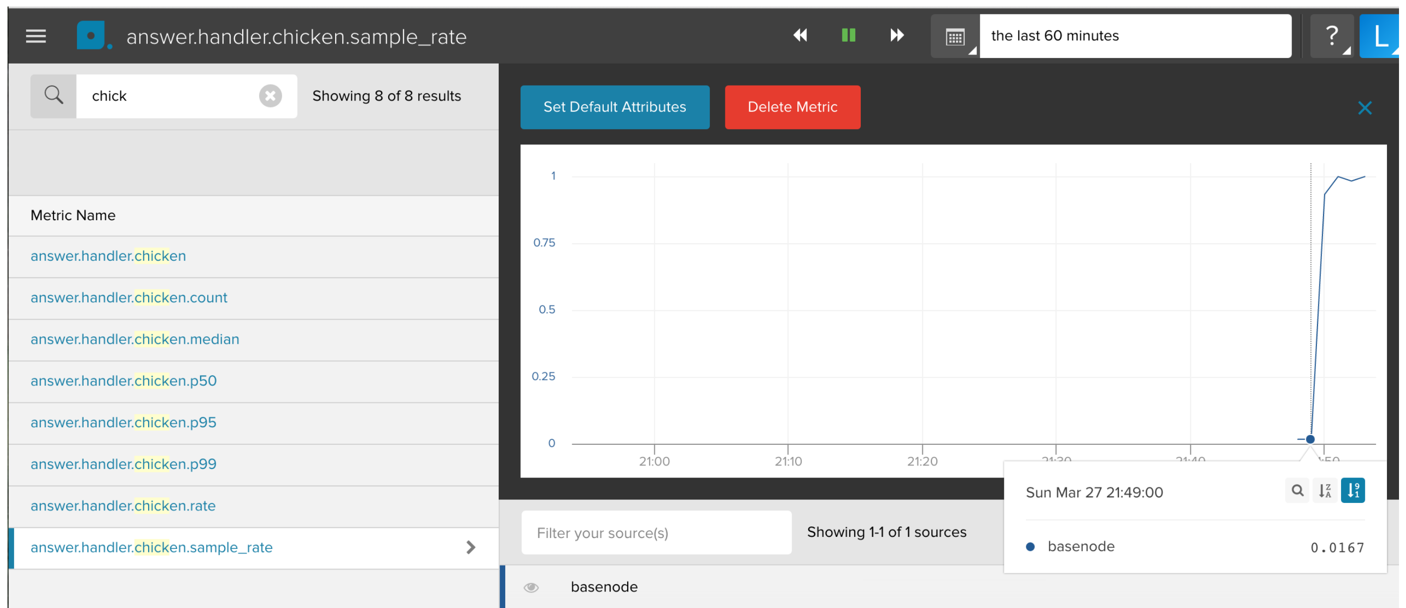


Figure 1: Look at all the lovely data!

this is that `defer` evaluates our functions *arguments* immediately, so when we pass `time.Now()` as the second argument to `metrics.Time`, that's evaluated immediately. That's how we capture our "before" time. Then `defer` takes care of executing `metrics.Time()` just before the function returns (after our answer module has done its thing), and as we've already seen `metrics.Time` captures its own after-time.

This gives us a single line of code we can inject at the beginning of any function in Go to get timing data as well as a count and rate of that function's invocation. The percentiles come automatically from StatsD as you can see in Figure 1.

So aside from `metrics/metrics.go`, which is completely reusable and modular/importable if desired, I've only added three lines of code to fully instrument every handler invocation this application has and any that might be added in the future, and one of those three lines was `metrics.Connect()`, which opens a socket to the local StatsD daemon.

I don't care how much you hate Golang, that's pretty cool, right? And, I mean, look at the graphs, *everybody* likes graphs … right?

Ow. My ear!

References

[1] StatsD: https://github.com/etsy/statsd.

[2] Statsite: https://github.com/armon/statsite.

Practical Perl Tools

Perl to the Music

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'.
dnblankedelman@gmail.com

Given all of the APIs and their Perl interactions we've discussed in this column, it is a little surprising it has taken me this long to get around to one of my favorite subjects: music. I started to pay closer attention to the APIs of the streaming music services right around the time one of my favorites (Rdio) was shuttered. I had amassed a pretty large collection of artists and albums I wanted to replicate on another service and was concerned about extracting the info from their service before it closed down. Luckily, the engineers at Rdio were equally concerned for their users and did a superb job of providing each user with an export of their data. But that started me down the path of wondering just what I could do in Perl to interact with my music data.

Several of the major streaming services have a decent API. Here's the rundown as of this writing:

- Spotify: https://developer.spotify.com
- Rhapsody: https://developer.rhapsody.com
- Deezer: http://developers.deezer.com/api
- Pandora: (only official partners can use it)
- Tidal: none
- Guevara: none
- Apple Music: none
- Google Play Music: none (really, Google? No API?)

Some of the services above have "unofficial APIs" where a random developer has reverse-engineered how the service works and published code that uses that information. We're not going to touch any of those APIs with a very large pole for any number of reasons, most of which I bet you can guess. Instead, in this column we'll pick the top one of the list above, Spotify, and dive into it. Spotify has a particularly mature API. Note: for some of these examples, you will need a Spotify account (and indeed, may need to be a subscriber).

Do I Know Who You Are?

The Spotify API distinguishes between authorized requests and public, non-authorized requests. The former is less rate-limited than the latter and (depending on the kind of authorization) also allows for querying of more sensitive data. But we can still get some good stuff from the API using public requests, so let's start there before we get into the auth game.

Given all of the past columns on APIs, I expect no gasps of astonishment when I say that the API is REST-based and that the results returned are in JSON format. The one Spotify API-specific piece we haven't really seen before (but is actually pretty common) is that Spotify has a unique resource identifier and ID for pointing to a specific object in their system. So, for example, here in their docs list:

```
spotify:track:6rqhFgbbKwnb9MLmUQDhG6
```

is a Spotify URI and

```
6rqhFgbbKwnb9MLmUQDhG6
```

is a Spotify ID. The URI includes what kind of thing is being referenced, the ID just simply provides which specific thing (i.e., that track) is being referenced. And in case you were curious, that URI in their doc points to the first track of an iconic album. I'll let you paste it in to the Spotify client to see just which one.

Dither, Dither, Dither

Right about now in the column the hero has a small crisis where he frets about which Perl module/approach he should use. Should he build something up using the minimalist but elegant modules that only do one basic thing really well (like performing an HTTP request or parsing JSON)? Should he instead use the all-singing, all-dancing REST request module that does both of these things and four other things besides? Perhaps he should use the module specifically made for this Web service. Or maybe show all three? Decisions, decisions.

It may shorten this column, but let's go right for the purpose-built module this time: WebService::Spotify. I'll explain this decision in more depth (complete with a dash of dithering) when we come back to authenticated/authorized requests. This module appears to be the most up-to-date (there is a module available called WWW::Spotify, but it calls the deprecated Spotify API). The difference between WebService::Spotify and something more lightweight becomes apparent when you install it. Because it is using Mouse (the smaller version of the modern object-oriented framework called Moose), it requires a whole slew of dependencies. cpanminus or CPANPLUS (discussed in a previous column) will handle this for you, but it can still be a bit disconcerting to watch the module names scroll by when you install it.

Once installed, using the module for non-authorized API calls is super simple:

```
use WebService::Spotify;

my $s = WebService::Spotify->new;

my $r = $s->search( 'chloe', type => 'artist' );

foreach my $artist ( @{ $r->{artists}->{items} } ) {
   print "$artist->{name} ($artist->{uri})\n";
}

print "total=$r->{artists}->{total}\n";
print "previous=$r->{artists}->{previous}\n";
print "next=$r->{artists}->{next}";
print "\nlimit=$r->{artists}->{limit}\n";
print "offset=$r->{artists}->{offset}\n";
```

Here's the output when I run it (which we will explain in a moment):

```
Chloe Angelides (spotify:artist:79A4RmgwxYGIDkqQDUHLXK)
Chloe (spotify:artist:71P0UphzXd95FKPipXjtE0)
Chloë (spotify:artist:2pCYsqZMqjA345dkjNXEct)
Chloe Martini (spotify:artist:6vhgsnZ2dLDaLDog3pqP2d)
Chloe Agnew (spotify:artist:34sL9HI0U50t8u0IQMZeze)
Chløë Black (spotify:artist:0Ifnpfl0VEmRGxCKaCYPX4)
Chloe (spotify:artist:2hg0g48H7GvAlTzkt3z5Vo)
Chloe Kaul (spotify:artist:35BBadnzA39iYkbQWL0r3p)
Chlöe Howl (spotify:artist:1hvPdvTeY6McdTvN4DyKGe)
Chloe Dolandis (spotify:artist:2SfamMWWSDbMcGpSua06o4)
total=340
previous=
next=https://api.spotify.com/v1/search?query=chloe&offset
=10&limit=10&type=artist
limit=10
offset=0
```

The code creates a new object, executes a search, and then prints key parts of the response. The response comes back in JSON form that is then parsed into Perl data structures. Here are some excerpts from a dump of that data structure:

```
0  HASH(0x7fe70ca94dc8)
   'artists' => HASH(0x7fe70ca94b88)
      'href' => 'https://api.spotify.com/v1/search?
query=chloe&offset=0&limit=10&type=artist'
      'items' => ARRAY(0x7fe70b003718)
...
         1  HASH(0x7fe70e129500)
           'external_urls' => HASH(0x7fe70e1157c8)
              'spotify' => 'https://open.spotify.com/artist/
                  71P0UphzXd95FKPipXjtE0'
           'followers' => HASH(0x7fe70e115138)
              'href' => undef
              'total' => 38
           'genres' => ARRAY(0x7fe70e129a40)
               empty array
           'href' => 'https://api.spotify.com/v1/artists/
                  71P0UphzXd95FKPipXjtE0'
           'id' => '71P0UphzXd95FKPipXjtE0'
           'images' => ARRAY(0x7fe70e0d6780)
              0  HASH(0x7fe70e0d67c8)
                 'height' => 640
                 'url' => 'https://i.scdn.co/image/20620033bdf1
                   c86e83cb3f18f97172aa89ee6eca'
                 'width' => 640
              1  HASH(0x7fe70e42e388)
                 'height' => 300
                 'url' => 'https://i.scdn.co/image/5c0a9f8cb3bb
                   f55e15c305720d6033090c04c136'
                 'width' => 300
              2  HASH(0x7fe70a7646e0)
                 'height' => 64
                 'url' => 'https://i.scdn.co/image/767603633a02
```

```
            6e73551c6c98d366b8cf08a1adec'
          'width' => 64
      'name' => 'Chloe'
      'popularity' => 39
      'type' => 'artist'
      'uri' => 'spotify:artist:71P0UphzXd95FKPipXjtE0'
...
    'limit' => 10
  'next' => 'https://api.spotify.com/v1/search?query=chloe
       &offset=10&limit=10&type=artist'
  'offset' => 0
  'previous' => undef
  'total' => 331
```

We get back a list of artists, each with their own sub-data structure (a list of hashes). Each artist that is returned has a number of fields. Our script prints out just the name and the URL, but you can see there's lots of interesting stuff here, including pointers to images for the artist. Already we are having some fun.

In addition to the list of artists, we also get back some data about the query itself, which I thought was so important to discuss, I print it explicitly in the script. This data helps us paginate through the large quantity of info in Spotify's database as needed. The "total" field tells us there are actually 331 artists with this name, but by default the module asks the API to limit the output to sending back 10 records at a time. This query started at the beginning of the data set (an offset of 0), and there is nothing before it (previous is "undef"). There is, however, the URI we should be querying to get the next 10 records (which includes an offset of 10, and an explicit limit of 10).

If we weren't using the WebService::Spotify module, we would call that URI to get the next set of 10. To do this explicitly with the module, we could add an "offset" parameter to the search() method like this:

```
my $r = $s->search( 'chloe', type => 'artist', offset => 10);
```

but WebService::Spotify makes it even easier by providing a next() method that consumes a results object and "does the right thing," as in:

```
my $s = WebService::Spotify->new;

my $r = $s->search( 'chloe', type => 'artist' );

while ( defined $r->{artists}->{next} ) {
   print_artists($r);
   $r = $s->next($r->{artists});
}

sub print_artists {
   my $r = shift;
   foreach my $artist ( @{ $r->{artists}->{items} } ) {
      print "$artist->{name} ($artist->{uri})\n";
   }
}
```

This code pages through the data using next() to pull the next result set if there is any.

So now that we've seen how to query for artists with a particular name, what can we do with that artist's info? As a start, with the artist ID, we can look up that artist's albums and her or his top tracks in a particular country:

```
# same search as before, let's pick the second result
my $artist = $r->{artists}->{items}->[1];
my $albums = $s->artist_albums( $artist->{id} );
my $top_tracks = $s->artist_top_tracks( $artist->{id},
                              'country' => 'US' );
```

Here's a small excerpt from the $albums data structure:

```
0  HASH(0x7fea614a86a8)
   'href' => 'https://api.spotify.com/v1/artists/
      71P0UphzXd95FKPipXjtE0/albums?
      offset=0&limit=20&album_type=single,album,compilation
,appears_on,ep'
   'items' => ARRAY(0x7fea614a8780)
      0  HASH(0x7fea613279d8)
         'album_type' => 'album'
         'available_markets' => ARRAY(0x7fea61268880)
            0  'AR'
            1  'AU'
            2  'AT'
            3  'BE'
            4  'BO'
            5  'BR'
            6  'BG'
            7  'CA'
            8  'CL'
            9  'CO'
            10  'CR'
...
            58  'ID'
         'external_urls' => HASH(0x7fea614a94c8)
            'spotify' => 'https://open.spotify.com/album
                  /3nVaq2gmNw8Z7k7rgVB961'
         'href' => 'https://api.spotify.com/v1/albums/3nVaq
                  2gmNw8Z7k7rgVB961'
         'id' => '3nVaq2gmNw8Z7k7rgVB961'
         'images' => ARRAY(0x7fea61bd6768)
            0  HASH(0x7fea614a9438)
               'height' => 640
               'url' => 'https://i.scdn.co/image/d64f75bc4f
                     b474478b904b7e4058bf369d2373e1'
               'width' => 640
            1  HASH(0x7fea61bd68a0)
               'height' => 300
               'url' => 'https://i.scdn.co/image/d2e5b264f4
                     87ac3bf1e3c32d48b1296247e99455'
               'width' => 300
            2  HASH(0x7fea61bd6f00)
               'height' => 64
```

```
          'url' => 'https://i.scdn.co/image/c20d9975b9
              253255d879c9a10d8f3a8deab077e5'
          'width' => 64
      'name' => 'Only Everyone'
      'type' => 'album'
      'uri' => 'spotify:album:3nVaq2gmNw8Z7k7rgVB961'
```

We've received info about an album called "Only Everyone." The response tells us it is available in 58 markets. Album covers are available at the specified URLs. Another interesting field is "album_type." If we had wanted to, we could have narrowed down the type of album we were seeking (for example, if we wanted to see all of the singles available from this artist). To do that, we'd add an extra parameter to the query, as in:

```
my $albums = $s->artist_albums( $artist->{id},
              'album_type' => 'single' );
```

The result of our artist_top_tracks() method call is equally fun. In our code, we've asked what the top tracks are for that artist in the US market (remember the list of markets in the previous output?). Here's an excerpt from what we get back:

```
     'tracks' => ARRAY(0x7f841614be30)
       0  HASH(0x7f841486e0c0)
         'album' => HASH(0x7f84131f7978)
           'album_type' => 'album'
           'available_markets' => ARRAY(0x7f841227ea98)
 'AR'

 ...
             58  'ID'
           'id' => '5mwk4GspWSXQHIkZGGdnhm'
           'name' => 'Boys and Girls Soundtrack'
           'type' => 'album'
           'uri' => 'spotify:album:5mwk4GspWSXQHIkZGGdnhm'
         'artists' => ARRAY(0x7f841250d5a8)
           0  HASH(0x7f841347d2e0)
             'id' => '71P0UphzXd95FKPipXjtE0'
             'name' => 'Chloe'
             'type' => 'artist'
             'uri' => 'spotify:artist:71P0UphzXd95FKPipXjtE0'
         'available_markets' => ARRAY(0x7f841347d5b0)
           0  'AR'
 ...
           58  'ID'
         'disc_number' => 1
         'duration_ms' => 203800
         'explicit' => JSON::PP::Boolean=SCALAR(0x7f8412196d08)
           -> 0
         'external_ids' => HASH(0x7f841347c8a8)
           'isrc' => 'USAK10000397'
         'id' => '600tF3aqxRjwJ0tdjxEwzY'
         'name' => 'Get You Off my Mind'
         'popularity' => 15
         'preview_url' => 'https://p.scdn.co/mp3-preview/4d99d5
                 6fad8d2b31eb9fb7fa5c9a603fa23adb16'
         'track_number' => 14
         'type' => 'track'
         'uri' => 'spotify:track:600tF3aqxRjwJ0tdjxEwzY'
```

I've chopped a bunch of the fields of the data structure just to save space but left enough so that you can see that for each track, you get its name, the album it was on (and full info on that album), markets available, the artist info for that track, what disc it is (for multi-disc sets), what track it is on that disk, how long the actual track is, and even a URL to a 30-second MP3 preview of the track. Feel free to check out the preview, although I wish to insert a caveat that this example wasn't picked for its artistic merit.

There are a few more API calls we can make without authorization listed in the WebService::Spotify doc (for example, to return the tracks found on an album). I hope you have a sense of how you might build a more sophisticated script to do things like search for all of the albums and tracks by an artist.

What We Do in Private

There's a lot of fun that can be had using non-authorized API calls. I could probably end the column right here and you would have enough to play with for a long time. But one important part of these streaming music services is the ability to manipulate the service's music in various ways. Spotify is all about the playlist, so as the last item in this column, we're going to take a brief look at how to work with them from the API. In particular, we are going to look at how we retrieve our users' private playlists.

The very mention of users and ownership should perk up your ears because it means we get to get back into the authentication and authorization business. Spotify uses a method we discussed in detail a few columns back, namely OAuth2, to allow a user to delegate the privileges to a script/app to perform operations on your behalf. And this brings us back to the reason why using WebService::Spotify had such appeal. Yes, we certainly could have used LWP::Authen::OAuth2 as we did in that previous column, and I suspect it would work, but I was also perfectly pleased to use the OAuth2 support built into WebService::Spotify instead. There are good arguments for going either route, so I would say you should follow your own best judgment on this call.

Let's take a look at some code that uses OAuth2 behind the scenes to extract the contents of one of my private playlists. In order to use this code, I first had to register for an application at https://developer.spotify.com/my-applications/#!/applications. I gave the application a name, a description, and a redirect URI (I used "http://localhost:8888/callback"; more on that in a moment). Upon creation, the application was assigned a client ID and a client secret. Let's see the code and then we'll take it apart:

```
use WebService::Spotify;
use WebService::Spotify::Util;
```

```
my $username = "yourusername";

$ENV{SPOTIFY_CLIENT_ID}     = 'YOUR_CLIENT_ID_HERE';
$ENV{SPOTIFY_CLIENT_SECRET} = 'YOUR_CLIENT_SECRET_HERE';
$ENV{SPOTIFY_REDIRECT_URI}  = 'http://localhost:8888/
callback';

my $token = WebService::Spotify::Util::prompt_for_user_
       token($username);

my $s = WebService::Spotify->new( auth => $token );

my $playlist = $s->user_playlists($username)->{items}->[2];

# note, as of this writing, the module had a bug that
# causes the use of 'fields' to fail. It may be fixed
# by the time you read this.
#
# If not, line 133 of lib/WebService/Spotify.pm should read:
# return $self->get("users/$user_id/$method", fields =>
$fields);
my $tracks = $s->user_playlist(
    $username,
    'playlist_id' => $playlist->{id},
    'fields'      => 'tracks.items(track(name,album(name),
                 artists(name)))',
);

print "Playlist: $playlist->{name}\n";

foreach my $t ( @{ $tracks->{tracks}->{items} } ) {
    print "Track: $t->{track}->{name}\n";
    print "Album: $t->{track}->{album}->{name}\n";
    print "Artists: $t->{track}->{artists}->[0]->{name}\n";
    print "\n";
}
```

The first interesting part of the code is the "prompt_for_user_token" call. Here we are asking the module to do the necessary OAuth2 dance to get us a token that can be presented to the service by WebService::Spotify to show we have authorization.

When this line of the code runs, it spits out a URL that you need to paste into a browser and then prompts for the URL that will be returned to us. Spotify shows you a standard OAuth2 authorization request screen. If you authorize the request, the service attempts to redirect to the redirect URI we supplied during the application creation stage but adds on a few parameters. These parameters include a representation of a token we will need later.

Here's where I have taken a less than elegant shortcut. Unless you are doing something interesting on the machine running the browser (localhost), chances are you don't have a server running on port 8888 to handle this redirect. As a result, the browser displays a "Sorry, I can't go to that URL" error page. That's just fine, we don't actually need to complete the redirect, we just need the URL being used for that redirect. We can just copy the URL it attempted from the browser's URL field and paste it to the prompt from our running script so it can continue. I suppose if we wanted to be cooler we could indeed spin up a tiny server (e.g., using something like Mojolicious) to catch the redirect and print out the URL, but that level of coolness is not required for this particular application.

Now that we've done the OAuth2 auth dance, the code in this example uses the appropriate token when creating the WebService::Spotify object. WebService::Spotify will make sure to handle getting the right token to Spotify during the rest of the transactions with the service. It is also smart enough to cache the token (and ask for renewals if necessary), so we won't have to go through that initial authorization step again.

The first thing we do with our newfound authorization is request one of the playlist records from the list of playlists our user owns. I'm picking the third playlist in my account just because I know it is short and hence useful for this example. You could easily write code that iterates through all of them.

Now to get the tracks for this playlist—we ask for the contents of my user's playlist with the ID retrieved from the playlist record. To be fancy, we're adding an additional parameter to our request that limits the fields returned from our query. That field says "tracks.items(track(name,album(name),artists(name)))", which means "given all of the items in a playlist record, pull out the tracks, and from within the tracks pull out the track name, the name of the album that track comes from, and the artists' name." We could have left this parameter off and just grabbed the fields from the large response we get back, but this is a bit more efficient, and the returned object is easier to look at in a debugger. We iterate through the tracks in the playlist, printing out these items. Here's what the output looks like for my shortest playlist:

```
Playlist: Pre-class
Track: I Zimbra - 2005 Remastered Version
Album: Fear Of Music (Deluxe Version)
Artists: Talking Heads

Track: Default
Album: Default
Artists: Django Django
```

Working with all of this data is done just the way we saw in the previous sections. Creating (user_playlist_create) and manipulating playlists (user_playlist_add_tracks) is supported by WebService::Spotify as well. Both are straightforward given what we already know about playlist IDs and track URIs.

I hope you have fun experimenting with this API. Take care, and I'll see you next time.

What's New in Go 1.6—Vendoring

KELSEY HIGHTOWER

Kelsey Hightower has worn every hat possible throughout his career in tech, and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he is not slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration and distributed systems.
kelsey.hightower@gmail.com

Go 1.6 was released in Q1 of 2016 and introduced support for Linux on MIPS and Android, HTTP2 support in the standard library, and some amazing improvements to the garbage collector (GC), which reduced latency and improved performance for the most demanding applications written in the language. Improvements to the standard library and runtime normally get the most attention leading up to a new release. However, with the release of Go 1.6 it's all about dependency management, which takes on one of the biggest pain points in the Go community.

Before you can really appreciate the impact of the Go 1.6 release, and the work around improving dependency management, we need to review how we got here.

Manual Dependency Management

Until recently, managing dependencies in Go required pulling external libraries into your GOPATH and attempting to build your application. To make things easy, Go ships with the go tool, which automates fetching dependencies and putting things in the right place. In the early days one could argue this was enough to get by. There was no real pressure to focus on tracking versions of your dependencies, largely because everything was relatively new and had a single version.

Let's take a look at managing dependencies for a simple application called hashpass—which prints a bcrypt hash for a given password.

First we need to create a directory to hold the hashpass source code:

```
$ mkdir -p $GOPATH/src/github.com/kelseyhightower/hashpass
$ cd $GOPATH/src/github.com/kelseyhightower/hashpass
```

Now save the hashpass source code to a file named main.go:

```
package main

import (
   "fmt"
   "log"
   "syscall"

   "golang.org/x/crypto/bcrypt"
   "golang.org/x/crypto/ssh/terminal"
)

func main() {
   fmt.Println("Password:")
   password, err := terminal.ReadPassword(syscall.Stdin)
   if err != nil {
      log.Fatal(err)
```

```
    }
    passwordHash, err := bcrypt.GenerateFromPassword
(password, 12)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(string(passwordHash))
}
```

With the hashpass source code in place it's time to compile a binary:

```
$ go build .

main.go:8:5: cannot find package "golang.org/x/crypto
/bcrypt" in any of:
    /usr/local/go/src/golang.org/x/crypto/bcrypt (from $GOROOT)
    /Users/khightower/go/src/golang.org/x/crypto/bcrypt
(from $GOPATH)
main.go:9:5: cannot find package "golang.org/x/crypto/ssh
/terminal" in any of:
    /usr/local/go/src/golang.org/x/crypto/ssh/terminal
(from $GOROOT)
    /Users/khightower/go/src/golang.org/x/crypto/ssh/terminal
(from $GOPATH)
```

Fail.

The build did not work, but what happened? The error message is telling us that we are missing a few dependencies required to build hashpass. Recall the import block at the top of the main.go source file.

```
import (
    "fmt"
    "log"
    "syscall"

    "golang.org/x/crypto/bcrypt"
    "golang.org/x/crypto/ssh/terminal"
)
```

The first three imports—fmt, log, and syscall—can be found in the standard library installed as part of the Go distribution. The next two libraries are external dependencies that must be fetched and installed into our GOPATH before we can use them.

Use the go tool to fetch both external dependencies:

```
$ go get golang.org/x/crypto/bcrypt
$ go get golang.org/x/crypto/ssh/terminal
```

Notice we are not fetching a specific version of our external dependencies. Yeah, you can see where this is going; stay with me. With our external dependencies in place, try building hashpass again, but this time use the -v flag to print each of the dependencies as they are being compiled:

```
$ go -v build .
golang.org/x/crypto/blowfish
golang.org/x/crypto/ssh/terminal
golang.org/x/crypto/bcrypt
github.com/kelseyhightower/hashpass
```

Success! We now have the hashpass binary in our current directory. Run the hashpass binary and enter a (fake) password to get a bcrypt hash:

```
$ ./hashpass
Password:
$2a$12$bD51ZjG//
NWrbo5dYWSFeeppvJwZazRBWqLBh4afnPOpUQSg3yAMy...
```

Managing dependencies this way works for simple projects with few external dependencies, but there are many hidden gotchas here. We are not tracking each version of our dependencies, which means other people will have a hard time reproducing our build. The version of our dependencies is based on when we fetched them, not specific versions we declared ahead of time. This is the problem the Go community has been trying to solve for nearly five years.

Third Party Dependency Management Tools

Given the challenge of manually managing dependencies, many third party tools started to appear, Godep being the most popular. Godep helps track which version of a dependency your project is using and optionally includes those dependency source trees in the same repository as your code through a process called vendoring.

Let's see Godep in action. Install Godep using the go get command:

```
$ go get github.com/tools/godep
```

Remove the existing hashpass external dependencies from your GOPATH:

```
$ rm -rf $GOPATH/src/golang.org/x/
```

Now we are ready to use Godep to manage the hashpass external dependencies. Let's start from the hashpass source code directory:

```
$ cd $GOPATH/src/github.com/kelseyhightower/hashpass
```

Use the godep get command to fetch the hashpass external dependencies.

```
$ godep get .
Fetching https://golang.org/x/crypto/bcrypt?go-get=1
Fetching https://golang.org/x/crypto?go-get=1
Fetching https://golang.org/x/crypto/ssh/terminal?go-get=1
...
```

Use the godep save command to record the version of each dependency in use and copy their source code into your repository:

```
$ godep save
```

The godep save command creates a working directory named Godeps. Take a moment to explore it.

```
$ ls Godeps/
Godeps.json    Readme    _workspace
```

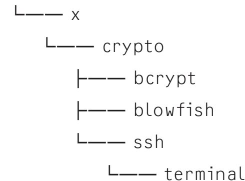
The Godeps.json file is used to record the versions of our dependencies:

```
$ cat Godeps/Godeps.json
{
 "ImportPath": "github.com/kelseyhightower/hashpass",
 "GoVersion": "go1.5",
 "GodepVersion": "v62",
 "Deps": [
  {
   "ImportPath": "golang.org/x/crypto/bcrypt",
   "Rev": "b8a0f4bb4040f8d884435cff35b9691e362cf00c"
  },
  {
   "ImportPath": "golang.org/x/crypto/blowfish",
   "Rev": "b8a0f4bb4040f8d884435cff35b9691e362cf00c"
  },
  {
   "ImportPath": "golang.org/x/crypto/ssh/terminal",
   "Rev": "b8a0f4bb4040f8d884435cff35b9691e362cf00c"
  }
 ]
}
```

Not what you are accustomed to seeing, right? The Godeps.json file is not tracking semantic version numbers like 1.0.0 or 2.2.1. This is where the Go community differs from many others. In the Go community we track the entire source tree that we depend on and do not rely on version numbers—version numbers can be reused and can point to later versions of a source tree, a security nightmare waiting to happen.

Another thing that’s not so obvious at first glance, Godeps copies all of our dependencies into our source tree under the Godeps /_workspace directory:

```
$ tree -d Godeps/_workspace/
Godeps/_workspace/
 └── src
     └── golang.org
```



```
         └── x
             └── crypto
                 ├── bcrypt
                 ├── blowfish
                 └── ssh
                     └── terminal

8 directories
```

The Godeps/_workspace directory mirrors part of the GOPATH we depend on for building our application. The idea here is to check in the entire Godeps directory and ensure our dependencies live next to our code. The Godeps/_workspace directory is ignored by the go build tool because it starts with an underscore and will require the godep command as a wrapper around the go build command.

```
$ godep go build -v .
```

The godep command ensures the Godeps/_workspace directory is included at the front of the GOPATH, which causes our vendored dependencies to take priority during the build process.

The main drawback to using Godep to manage dependencies was that Godep was non-standard and incompatible with the go tool—go get will ignore the Godeps/_workspace directory and force users to check out your entire project to build your application.

Vendoring

Vendoring makes it easier to deliver reproducible builds and reduces reliance on remote code repositories hosting your dependencies—it also prevents your development team from scrambling when GitHub goes down in the middle of a release: fun times!

Godep proved that managing dependencies can be done with the right tools, but required help from the core Go project to reach its full potential. That help arrived in Go 1.6; a snippet from the release notes:

- Go 1.6 includes support for using local copies of external dependencies to satisfy imports of those dependencies, often referred to as vendoring.
- Code below a directory named “vendor” is importable only by code in the directory tree rooted at the parent of “vendor,” and only using an import path that omits the prefix up to and including the vendor element.

In a nutshell Go 1.6 will search a vendor directory for external dependencies, but you’ll still need tools to copy them there.

Newer versions of Godeps will detect that you're using Go 1.6 and copy your dependencies into the vendor directory.

```
$ cd $GOPATH/src/github.com/kelseyhightower/hashpass
$ rm -rf Godep
$ godep save
```

Godep creates a vendor directory and copies the hashpass external dependences into it:

```
$ tree -d vendor/
vendor/
└── golang.org
    └── x
        └── crypto
            ├── bcrypt
            ├── blowfish
            └── ssh
                └── terminal

7 directories
```

As an added bonus, Godeps continues to write the Godeps /Godeps.json file, so you can track exactly where your dependencies came from. At this point we can rebuild hashpass without any wrapper tools thanks to support for the vendor directory.

```
$ go build -v
github.com/kelseyhightower/hashpass/vendor/golang.org/x
/crypto/blowfish
github.com/kelseyhightower/hashpass/vendor/golang.org/x
/crypto/ssh/terminal
github.com/kelseyhightower/hashpass/vendor/golang.org/x
/crypto/bcrypt
github.com/kelseyhightower/hashpass
```

Notice all the hashpass dependencies are being pulled in from the local vendor directory. Feel free to take the rest of the day off, you've earned it!

Summary

Since the release of Go 1.0, the Go community has been on a long journey to get a handle on dependency management. Over the years the community has stepped in to help define what the right dependency management solution looks like for Go and, as a result, gave way to the idea of vendoring and reproducible builds. Now, with the release of Go 1.6, the community has a standard we can rely on for many years to come.

For Good Measure

Five Years of Listening to Security Operations

DAN GEER

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

Mukul Pareek, a colleague at a market maker bank, and I have run the Index of Cyber Security for five years [1]. This article is a kind of compendium of what the Index has shown over those five years, but before I get to that I will discuss how we got to where we are.

The only purpose that makes security metrics worthy of pursuit is that of decision support, where the question being studied is more one of trajectory than exactly measured position. None of the indices I'll discuss are attempts at science, although those that are in science (or philosophy) will also want measurement of some sort to backstop their theorizing. We are in this because the scale of the task compared to the scale of our tools demands force multiplication—no game play improves without a way to keep score.

Early in the present author's career, a meeting was held inside a major bank. The CISO, a recent unwilling promotion from Internal Audit, was caustic even by the standards of NYC finance. He began his comments precisely thus:

> Are you security people so #$%&* stupid that you can't tell me:
>
> - How secure am I?
> - Am I better off than I was this time last year?
> - Am I spending the right amount of money?
> - How do I compare to my peers?
> - What risk transfer options do I have?

Twenty-five years later, those questions remain germane. The first, "How secure am I?" is unanswerable; the second, "Am I better off than I was this time last year?" is straightforward given diligence and stable definitions of terms; the third, "Am I spending the right amount of money?" is evaluable in a cost-effectiveness regime, although not in a cost-benefit regime; the fourth, "How do I compare to my peers?" can only be known directly via open information or indirectly via consultants; and the fifth, "What risk transfer options do I have?" is about to get very interesting as clouds take on more risk and re-insurers begin pricing exercises in earnest.

The argument for an index is that when measurement is hard, process consistency is your friend. If we can find one or a few measures that can be tracked over time, those measures, those base numbers do not have to be guaranteed correct—so long as any one series is wrong with some sort of consistency, its wrongness doesn't change the inferences drawn from it. In our kind of work, it is the shape of the trendline that matters. Decisions are supported when we know what direction something is going.

As an example, for some years the National Vulnerability Database has published a daily number called the "Workload Index" [2], which is a weighted sum of current vulnerabilities in the NVD. To quote from NIST:

> [The 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
>
> [In other words, t]he 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.

Ten years of the NVD Workload Index is shown in Figure 1.

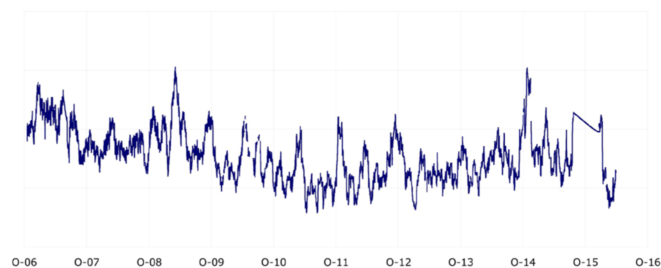


Figure 1: Ten years of the NVD Workload Index

The NVD Workload Index encourages a particular inference: that the arrival rate of new vulnerabilities approximates a random process. Graphing the Workload Index in the aggregate and comparing that to a Gaussian bell curve shows a fair congruence with some right-skew and a bit of kurtosis, as seen in Figure 2.

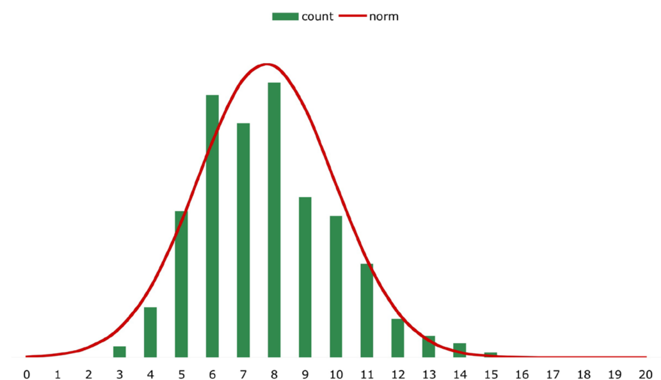


Figure 2: Daily workload number by prevalence

Looking at other methods of binning the Index values, Figure 3 shows some strong variation year over year,

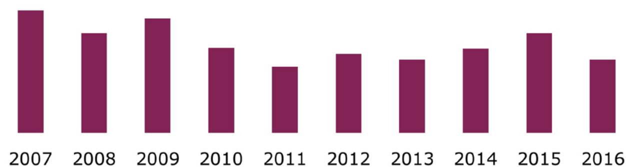


Figure 3: NVD workload year by year

Figure 4 shows seeming seasonality,

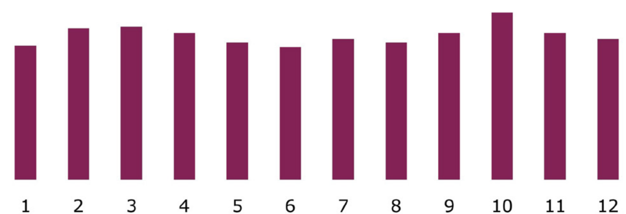


Figure 4: NVD workload month by month

and Figure 5 shows a pretty clear implication of work week.

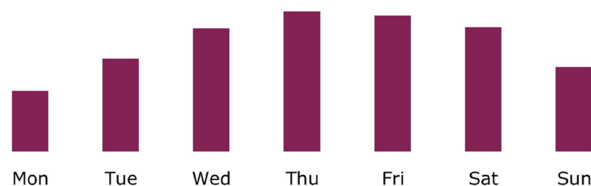


Figure 5: NVD workload day by day

In short, the NVD workload is a straightforward example of an index. I would argue that to be a useful index there has to be something to measure that, once measured, might help one to make some decisions. I would also argue that to be believable, there has to be some transparency as to methods—especially regarding the parameters of sampling—and a believable willingness to carry out a relatively unexciting routine indefinitely. Thank you, NIST, for your long-term diligence in this and so many other things.

Security Pressure Index

Before Pareek and I began the Index of Cyber Security, I had tried various indices before. A different colleague, Dan Conway, and I put together what we called the "Security Pressure Index," meaning an estimate of the time rate of change in the pressure on security professionals. With indices, seeking generality usually means that you want more than one input. We settled on four: we got a measure of phishing from the Anti-Phishing

Working Group, a measure of spam from Commtouch, a measure of data loss from the Dataloss Database, and that measure of workload from NIST. Together, these four yielded the Security Pressure Index as shown in Figure 6.

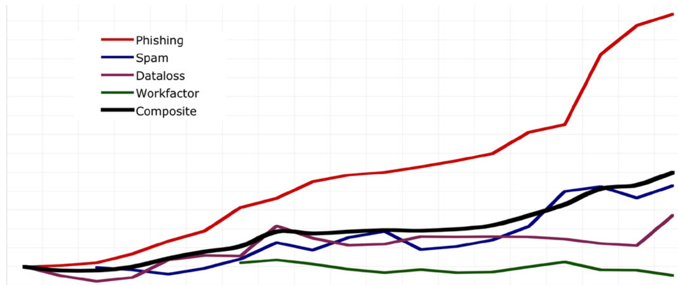


Figure 6: Five years of the Security Pressure Index

To be clear, in each case we were mooching off other people's work, which is rarely polite and leaves you no recourse should, say, one of those sources change its numbering scheme, change its publication schedule, or change anything at all without telling you. We thanked all four sources in print every month, but what we were doing was, in so many words, predictably unsustainable. After five years, we called it quits for the SPI. Close, perhaps, but no cigar.

0wned Price Index

So what next to try? With the aphorism "Good artists copy, great artists steal" in mind, Conway and I ripped off PNC Financial's long-running "Christmas Price Index" [3]. The XPI, as it is called, calculates the price of buying all the gifts described in the song "The Twelve Days of Christmas" such that you might know what your true love's affection is going to cost you. In our case, we put together a price index for stolen data and similar illicit digital goods. To get a little attention, we called it the "0wned Price Index," and after nailing down a variety of stolen goods for which market price information paralleled the XPI, we amalgamated them in a way that could, after a fashion, actually

verified PayPal keys	12	90.00	7.50
CCVs	22	71.50	3.25
Unix roots	30	75.00	2.50
FTP hacks	36	216.00	6.00
full identities	40	200.00	5.00
fresh emails	42	210.00	5.00
rich bank accounts	42	31,500.00	750.00
Windows boxes	40	1.20	0.03
US passports	36	28,800.00	800.00
Gbyte DDoS	30	4,500.00	150.00
Dell Preferred	22	1,320.00	60.00
Juniper router	12	150.00	12.50
		67,133.70	

Figure 7: 0wned Price Index for Christmas 2009

be sung to the tune of "The Twelve Days of Christmas" [4]. We published this for three years (see Figure 7).

And then we stopped. The reason we stopped was a kind of progress. The market price data we relied upon came from eavesdropping on so-called carder forums and the like—places where stolen data was sold at auction. But those sources of information dried up once law enforcement infiltrated them and began making arrests. After that, to get auction pricing you had to be a market participant, but Conway and I were not ready to be market participants. Repeating myself, if you rely on data sources you do not control, then what you are doing is inherently temporary.

Index of Cyber Security

Which leads me to the main event for this column. Based on the experience(s) described above and just general knowledge of the field, Pareek and I put together the Index of Cyber Security, which turned five years old in April 2016. The first lesson, that it is better to source your own data if you expect to be in the game for the long haul, means we have to ask our own questions, not just graze in other people's pastures during their growing season.

Another lesson is that, even yet and perhaps forever, as a field we will not be able to agree on precise terminology. Yes, we can all agree that "vulnerability," say, is a term in general use, but as to a fully precise definition, universally held—that's not coming. That, in turn, means that if you ask, "How many vulnerabilities are there?" the answers you get will be biased by the definitions of the individuals answering. This is not completely serious, but terminological confusion substantially interferes with reproducibility of survey results.

As a central point, survey research is vulnerable to idiot respondents. If you are looking for generalizability, you administer your survey to as large a population as you can afford and you pick the people replying either by randomization or by selection. If you randomize, you gain some immunity to idiot respondents. The well known Consumer Confidence Index [5] (CCI) is based on 5000 random phone calls a month, thus washing out the idiot fraction, at least so long as that fraction is not growing. The CCI is run consistently, and many financial instruments factor in the new value of it as soon as it is issued. It is a forward-looking indicator.

If generalizability to the public at large is not a goal, then you administer your survey to a vetted population where there is no idiot fraction. But by selecting your respondents, your results are conditional on the methodology of your selection process. The well-known Purchasing Managers' Index (PMI) [6] picks its respondents carefully and has many fewer of them, but because the PMI respondents are selected for what they know, this is a feature not a bug. The PMI is a weighted sum of five variables, in this case production level, new orders, supplier deliveries,

inventories, and employment level. Like the CCI, the PMI is run consistently, and many financial instruments factor in the new value of it as soon as it is issued. It is a forward-looking indicator.

So Pareek and I looked at both the Consumer Confidence Index and the Purchasing Managers' Index for inspiration. Both of them ask subjective questions about the opinion of the respondent. The CCI wants opinions that are representative of the population at large, so they take the randomization route. The PMI wants opinion to be knowledge-based, so they take the vetted respondents route. Pareek and I decided that we would follow the PMI approach, that is, to have as respondents people who actually know something.

But what is it they are supposed to know? We decided that if the Index of Cyber Security was to be a forward-looking indicator, then we had to have as respondents people who are on the front lines, people with operational responsibility for cybersecurity. We do not want people whose knowledge of current cybersecurity is academic, or based on police power, or the result of having memoranda passed up the management chain to them. We wanted people who were doing cybersecurity, not people who had knowledge that didn't come from actual daily practice.

This means that we rely on a certain kind of expert, and the Index of Cyber Security is an amalgamated subjective opinion on the state of play as understood by people who are actually in the game, per se. When I say "subjective" it is because we do not have solid, unarguable measures of security. In fact, that we don't is precisely why we are doing the Index—when you don't have unarguable measures, the next best thing is the collected wisdom of experts. And note that I said "experts"—we neither know nor care what a respondent's official position is in some organizational entity; we care about experts wherever we find them. So it may be that some handful of experts work for the same employer, and some employers will have no experts present at all. So be it; we are not collecting insights into the Fortune 500—we are collecting experts.

Because every term we might use has, as I mentioned before, some degree of ambiguity as a term, we cannot just ask, "How bad is malware?" Asking "How bad is malware?" requires a precise, shared definition of "malware" and a malware thermometer that reads "78" or the like. So what then do you do?

What we do is ask a series of 25 questions, and the questions are the same every month. All of the questions read like this:

Since a month ago, the threat of insider attack has

- Gotten Better
- Gotten Worse
- Gotten a Lot Better
- Gotten a Lot Worse
- Remained Unchanged

We ask 25 questions like that.

There are two things to note at this point: one, you may recognize the response set as a Likert scale. Likert scales are standard practice in survey-based research. They are always symmetric with an odd number of options so that the central option is considered neutral. The score for a question is a weight assigned to each of the alternatives.

The main point here is that each question is of the form "Since a month ago," meaning that what we are looking for is change, not valuation. That is far easier to estimate reproducibly than estimating a number in an absolute range. The rest of the question, "the threat of insider attack has gotten," does not require everyone to agree on what insider or attack means. We do not have to train our respondents to use this or that word precisely in one way that might differ from how they usually use it. All we need is for the individual respondent to have a mental definition of the word or phrase that is reasonably stable. If your definition of, say, "malware" and mine are subtly different, we can still say whether the pressure from it has gotten better or worse.

In other words, the Likert scale's symmetry avoids biasing the respondent in one direction or the other. Additionally, by asking about the trend of a characteristic rather than the value of some measurement of that characteristic, the respondent is relieved of having to conform to either some official definition or to a scaling mechanism they did not invent. Instead, they can use their own definition and don't need numbers.

Because each question is of the same form, the Index of Cyber Security is then calculated by counting how many "Gotten Better" answers, how many "Gotten Worse" answers, etc.,—one count for each Likert category. Those counts are combined in a weighted sum:

Much Better	Better	Unchanged	Worse	Much Worse
-20%	-7.50%	0	7.50%	20%

Being a measure of risk, the ICS is bounded on the low side but not on the high side, hence the directionality of the weightings. In other words, the ICS rises as perceived risk rises. An example:

Much Better	Better	Unchanged	Worse	Much Worse
6	58	614	150	15
-20%	-7.50%	0	7.50%	20%

Multiplying it out and dividing by the sum of the above, we get 0.010235. Exponentiating that gets a multiplier to apply to last month's ICS to get this month's, i.e., 1.010287, or an increase in the ICS of a tiny bit over 1%.

We do this calculation not only in the aggregate so as to derive the Index of Cyber Security value, but also on a question-by-

question basis so as to watch trends in specific risks. These trendlines by question we refer to as sub-indices, and they are part of a detailed monthly report that only respondents get.

And, yes, we occasionally replace one question with a new one. To maintain continuity of the ICS as a whole, we apply a correction factor done in precisely the same way that any financial index such as the Dow Jones Industrial Average does when it replaces one stock with another.

Perhaps you did not need to know all that, but our point is that the way the ICS is calculated is 100% conventional and entirely boring. We want "boring" because whatever our results, we want them to never be thought of as an artifact of some new method we cooked up on the spot. Much as amateurs should rarely create their own crypto algorithms, amateurs should rarely create their own analytic regimes.

So this is the scheme—a largely fixed set of Likert-valued questions, a vetted respondent base, a trade of data for data, and a commitment to a long run. This is information sharing at its best.

What we have learned so far: our respondents believe that risk in the aggregate is and has been rising almost inexorably, but which of the 25 components of the ICS is changing the most each month varies over time—a lot—as seen in Figure 8,

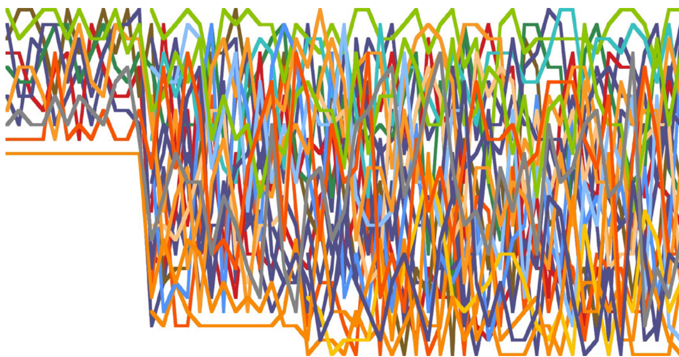


Figure 8: Rank order % change across sub-indices month by month

which can also be seen looking at the trailing four-month volatility of the sub-indices in Figure 9.

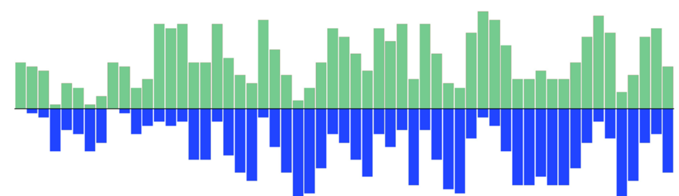


Figure 9: Trailing four-month volatility up versus down

Another way of looking at dispersion of risk across questions is that for 14 of the 58 months seen in Figure 10, at least one question reached its lowest value, and in 14 of those months at least one question reached its highest value.

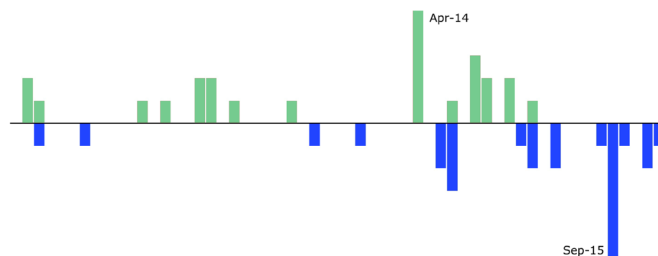


Figure 10: Trailing four-month volatility up versus down, highest and lowest values

In five of those months, both one question's highest and another question's lowest values were set, and in 32 of those months, no question reached its most extreme value. In April 2014, 20% of the questions returned their highest values ever for rate of change. In September 2015, 25% of the questions returned their lowest values ever for rate of change. "Why?" is hard to guess.

Let me be clear that we are not trying to do science here. If your purpose in building a model is to come to a definitive conclusion about causality, about how nature works, then you are saying that the inputs to your model and the coefficients that calibrate their influence within your model are what matters. Parsimony in the sense of Occam's Razor is your judge, or, as Saint-Exupéry put it, "You know you have achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away."

By contrast, when your purpose in building a model is to enable control of some process or other, then you will not mind if your input variables are correlated or redundant—their correlation and their redundancy are not an issue if your goal is to direct action rather than to explain causality. A goal of understanding causality in its full elegance leads to $F = ma$ or $E = mc^2$. A goal of control leads to econometric models with thousands of input variables, each of whose individual contribution is neither clear nor relevant.

That said, if you look month by month you see that some questions are perceived to indicate more risk than others. Ranking the magnitude of individual risks over a 58-month period gives us:

Risk	Number of times in the top three for the month
Counterparty	52
Media & public perception	40
Hacktivist/Activist	30

If you rank not by risk score but by which risk had the biggest jump (volatility) that month, then you find

Risk	Number of times in the top three for the month
Media & public perception	42
Phishing/Social engineering	35
Counterparty	16

With the usual caveats about correlating too many things at once, if you put all 25 current questions into a correlation matrix, then some do appear to be in lock step.

Correlation	Risk Pairs
0.971	Effect desired: Data theft Weapons: Phishing/Social engineering
0.958	Effect desired: Data theft Attackers: Criminals
0.946	Overall: Media & public perception Weapons: Phishing/Social engineering
0.940	Weapons: Phishing/Social engineering Attackers: Criminals
0.931	Overall: Media & public perception Effect desired: Data theft

Given that array, one could argue that there is really only one risk between all of those: the risk of data theft by criminals using social engineering so that you look stupid in the newspaper.

Of course, one thing that we wish we had done from the get-go was to record the dates of important security events, whether that is in the newspaper, the laboratory, or the underworld. We didn't, and we're not going to start now. But when you look at all the variation, we do often want to say, "Where did that come from?" We can't answer that, so we won't make believe we can; to do so given our methods would be pure speculation [7].

We also compute for each risk and overall a diffusion index and do it the same way as diffusion indices are done in finance. Diffusion indices are a symmetric construct; they are just the sum of all the indicators in a basket of indicators that are going in one direction plus half of those that are static. As the ICS is a risk index, we report what percentage of responses are either "Worse" or "A Lot Worse" plus half the responses that are "Neutral." For January of this year, the top three were

Risk	Diffusion Index
Phishing/Social engineering	69%
Criminals	63%
Customization to target	62%

Of the 25 risks, five of them had diffusion indices of 50% or less. The other 20 were above 50%.

One final thing; each month, in addition to the standing set of 25 questions, we ask a question of the month. Once in a while these are suggested. Most of the time Pareek and I think them up. In 2015, Questions of the Month covered encryption, safe harbor, ransomware, IPv6, affordability, change management, CEO involvement, regulation, worst case scenarios, security metrics, and offensive dominance.

Sometimes we will repeat a question. For example, in September of 2012 we asked, "What percentage of the security products you are running now would you still run if you were starting from scratch?" In January of this year we asked that question again. Compiling the answers, we found in September of 2012 that 35.5% of the products then installed would not be reinstalled should the respondents be in a position to start fresh. Call that buyers' remorse. In January of this year, we found that buyers' remorse had swelled from 35.5% to 51.9%. I don't have figures for the number of cybersecurity products available for sale month by month, but it is surely greater now than it was three and a half years ago. I can tell you from where I work that the number of cybersecurity startups has never been greater; a spokesman for Kleiner-Perkins says that they are tracking over 1100 cybersecurity startups now in some part of the funding game. Is a rising level of buyers' remorse a sign that better tools are on offer or that unmitigable risk is getting worse? It's a puzzle.

Conclusion

This seems a good place to stop insofar as it is surely possible to just keep doing exploratory data analysis for pages more. But that isn't actually what I have been doing. What I've been doing is talking about a different kind of information sharing, bottom up, as it were. All the talk about information sharing always seems to mean something top down, something where those with more power or better eyes or an enforceable structural advantage share a portion of their information trove with the worthy below them. I am not making fun of that; it is a time-proven technique and it is policy across the board. It comes out of the idea of "need to know," and need to know is a protective mechanism in so many things. Yet it seems to us that once upon a time any one of us could start from nothing and, by diligence, come to know just all that was necessary for cybersecurity. That is clearly less true than it once was. The technical knowledge base has both deepened and broadened, deepened in that sense of an accumulating welter of obscure interdependencies, and broadened in that sense of cybersecurity becoming an issue wherever networks go.

That affects need to know in ways we have only barely acknowledged. Sure, the Federal government, or any Western government, can grant security clearances to the CISOs of every

market maker bank, or any other institution that matters to them, so as to share classified information.

But, for Pareek and myself, the argument for official channels is unsatisfactory and insufficient. We don't mind them, but cyber-security in its complexity just doesn't seem to us to be headed for some sort of denouement when all will become clear at taxpayer expense. We are doing the Index of Cyber Security the way we are on the grounds that (1) you can't know what's going on unless you are on the playing field yourself, and (2) that there is no way to tell if the risks you are seeing are specific to you without comparing your risks to those of other people in your position elsewhere.

In the fullness of time, we may add other things to our repertoire, but we are expecting to keep doing the ICS for the indefinite future. We invite you to participate. The respondent's workload is insignificant, the shared data cannot be gotten elsewhere, and we are doing everything we know to do to make it possible for respondents to be frank without concern to being quoted in any way. To take part in this project, see the Contact page under reference [1].

References

[1] Index of Cyber Security: cybersecurityindex.org.

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

[3] PNC Christmas Price Index: www.pncchristmaspriceindex.com/cpi.

[4] D. Geer and D. Conway, "What We Got for Christmas," *IEEE Security & Privacy*, January 2008: geer.tinho.net/ieee/ieee.sp.geer.0801.pdf.

[5] Conference Board Consumer Confidence Index, issued at 10 a.m., Eastern Time, on the last Tuesday of every month: www.conference-board.org/data/consumerconfidence.cfm.

[6] Institute for Supply Management, Purchasing Managers' Index, issued at 10 a.m., Eastern Time, on the first business day of every month: www.instituteforsupplymanagement.org/ismreport/mfgrob.cfm.

[7] If you want to read the best talk ever given on speculation, the late Michael Crichton nailed it in 2002 with "Why Speculate?" archived at geer.tinho.net/crichton.why.speculate.txt.

Thanks to Our USENIX Supporters

USENIX Patrons
Facebook Google Microsoft Research NetApp VMware

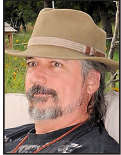
USENIX Benefactors
ADMIN magazine Hewlett-Packard *Linux Pro Magazine* Symantec

USENIX Partners
Booking.com CanStockPhoto

Open Access Publishing Partner
PeerJ

/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

The first time I heard the term "Fuzzy Logic" I thought it referred to the way cats plead with you to pet them and then park themselves just outside your reach. Rather than binary absolutes, fuzzy logic admits the existence of "degrees of truth" that acknowledge the ambiguous nature of nature. Life, in other words, is a multiple-multiple examination wherein the answer often turns out to be "all of the above." One commonly cited example is when you toss a ball to someone. You don't calculate exact values for all the variables constituting the trajectory; you just throw the dang thing in the general direction of your target and hope it doesn't break Mrs. Anderson's front window again. Fuzzy logic is a pandemic that has infected people and institutions worldwide and across historical epochs.

The poster children for fuzzy thinking are of course politics, politicians, and the election thereof. Every two years the American people throw rationality to the wind and vote for the candidate with the best clothes, facial features, and PR team. The reason we do this is that any given politician's stance on issues is dependent on who's asking, whether the month ends in "y," and the phase of the moon. Not that their stance matters, anyway, because what they promise or assert during the run-up to the election often has no bearing on what they accomplish—or more often fail to accomplish—once in office. An American presidential election is a giant game of chess where the only piece left on the board by mid-November is a king.

Another area where fuzzy logic rears its head is consumerism. Admit it: when some gadget or other cool thing you have no actual need for rings your bells, the fuzz-synthesis system swings into high gear. In the time it takes to type your PayPal password you will have come up with at least two solid rationalizations for why you simply have no option but to hit the "Checkout" button. It matters little that you already own three shower TVs; this one is 4K. There is absolutely no point in living if you can't watch *Dancing with the Stars* in 4K while you lather up your pits: am I right?

Fuzziness experiences many manifestations in and around the home—the answer to the perennial query "what's for dinner," for example. Open just about any fridge in America (I suspect a great many other nations, as well) and with not a lot of effort you can probably uncover a half-dozen textbook examples of fuzzy. Nor are household chores immune from fuzziness. Anyone who uses a clothes dryer will know what I mean. I'm convinced that dryers are quantum gateways, in fact.

When I put a pair of socks in a dryer, they assume a superpositional state wherein they exist simultaneously as both a pair and a single sock. Opening the dryer door collapses the waveform. In my case it usually collapses in "single sock" mode. The probability should be 50% per pair for any given load, but it isn't. I can only presume that something in my laundry room environment biases this result.

It seems to me that one of the prime uses for fuzzy logic in the upcoming years will be in the Internet of Things. I would go so far as to suggest, in fact, that we rename it the "Internet of Fuzz" in honor of that relationship. After all, it takes a dedicated fuzzy thinker to connect one's lights, thermostat, security system, fish tank, doorknob warmer, microwave, welcome mat, garage door opener, medicine cabinet, pet door, and seal-a-meal to the notoriously insecure global mishmash we call the Internet. A hacker taking control of all these things might not be able to cause irreparable harm, but one could well come home to a frozen house with dead fish, a garage full of raccoons, an exceptionally high utility bill, and a pissed-off SWAT team camped on the front lawn.

Fuzzy thinking is also evident in many legislative actions, from Supreme Court rulings right down to your local city council's ponderous pronouncements. Legislative gems running the gamut from authorization for police to bite dogs to calm them down to forbidding the sale of both toothpaste and a brush for application of such to the same person on Sunday demonstrate beyond a reasonable doubt that the rule makers of our great nation subsist on a rich diet of tasty, tasty fuzz.

The private sector evinces no shortages in the fuzz department, either. Companies search high and low, night and day, for fuzzy staff. This results in some quite puzzling products, and even more puzzling warning labels. To be fair, some of these labels owe their existence more to the anticipated fuzziness of the consumer than that of the label creators themselves. My favorites include instructions not to operate a motor vehicle included on bottles of pills intended for babies and dogs, admonitions against inserting people into washing machines, and a warning not to swallow wire coat hangers. Because that's a widespread health hazard.

Returning to our own occupational neighborhood, the security practices of some computer-related concerns are so heavily fuzzed it's hard to see through to the content. Password generation algorithms for security products that do not allow special characters: fuzzy. SSL sessions that accept self-signed certificates: fuzzy. Vendors who quietly upload "security updates" to your device that significantly increase the vulnerability of said device to remote exploits: ridiculously fuzzy.

Now, you might argue that none of what I've laid out above is really "fuzzy logic." I would respond by making for the door while distracting you with an ad for tablets boasting 4K resolution that promise to download entire HD movies that haven't even been released yet in the time it takes to enter your character-only password. The door in question can be closed, open, or somewhere in between, but as long as I can squeeze through, I don't care. I still carry my old Fuzzbuster.

Book Reviews

BOOKS

MARK LAMOURINE, PETER GUTMANN, AND RIK FARROW

Thinking Security: Stopping Next Year's Hackers

Steven M. Bellovin
Addison Wesley, 2016, 382 pages
ISBN 978-0-13-427754-7
Reviewed by Mark Lamourine

It turns out that Steve Bellovin and I have a very similar taste for science fiction. It's common for authors to include an epigraph at the beginning of each chapter to provide a hint at the topic or even some humor. Bellovin's choices come from some of my favorite authors: Lewis Carroll, Poul Anderson, E. E. "Doc" Smith, and Larry Niven among others. He uses many quotes from a recent novelist, Charles Stross. Each of these authors writes about a proposed future or alternate world. They also write about the human implications of their new framework. Stross in particular writes about a near future, extrapolating on current trends in technology and the threats they pose as well as the promise. Bellovin wants us to do the same thing.

In *Firewalls and Internet Security*, Bellovin's previous book on firewalls, he and co-authors William Cheswick and Aviel D. Rubin wrote about a specific known threat and a particular technology to address it. In *Thinking Security* he gives the reader a broad survey of how things stand today, the technologies that protect our systems, and how they can fail us, and he offers some hints about how we might plan for tomorrow.

The narrative arc is what you would expect: understand the problem; survey the technology, uncovering strengths and weaknesses; and consider how to use the strengths to mitigate the dangers. In each chapter, he clearly lays out the topic and cites examples of ways in which each technology has failed. At the end of each chapter he gives a brief summary analysis and conclusions. The book is broken into five sections, which broadly are: the problem statement; the list of technologies considered; a survey of human factors; a set of architectural case-studies; and, finally, a discussion and guidelines for understanding, planning, and pitching good security practices to corporate management.

In the preface, Bellovin calls this book a graduate-level introduction to computer systems security. It is aimed at working system administrators and other security-related IT professionals. He assumes a working level of understanding of how computer systems operate and constantly underpins his text with references to articles and papers that illustrate his topic, with the understanding that readers will follow up if they are not already familiar with the material. Maintaining computer security is an active pursuit.

His motivation in writing this book is a recognition that, in large part, the computer industry is spending resources in ways that do not really improve the security of our systems. This misallocation of effort is understandable but not necessary. His book is a call to arms for system administrators to become versed in the ways in which they can be effective and to make a clear case for good security design and practice over bad. He also wants to impress IT managers to understand and to trust the front-line workers and to support them in their efforts to educate those who make the decisions. Good security is all of our responsibility.

Essential Scrum

Kenneth Rubin
Pearson Education, 2013, 452 pages
ISBN 978-0-13-704329-3
Reviewed by Mark Lamourine

One of the major tenets at the root of all Agilish software development processes is to eliminate unnecessary "ceremony." The idea is that many of the meetings and memos that were the backbone of business processes from the '90s and before have lost their meaning and become largely empty rituals. They consume time without actually conveying information or improving coordination.

But, humans being humans, we love our rituals. People find a truly free-form "just do what you need to" process disconcerting or uncomfortable. If there are no prescribed activities, people create them. It's just my opinion, but I think that's one of the reasons for the popularity of the Scrum method.

Scrum is, in my experience, the most well-defined of the Agile process methods.

In *Essential Scrum*, Rubin presents scrum with the same precision and structure that Scrum offers to the family of Agile methods. His book is a proper reference, presenting the reader with the goals, concepts, and the process of Scrum for software development.

Rubin's approach and tone will suit the business reader and coach. There are no whimsically drawn characters and banter-filled dialogs, which are often used to try to soften the process of learning a new software process framework.

After presenting the conflict that most traditional software development processes create between rigid long-term planning and interrupt-driven priorities (which are the reality of modern

software development), Rubin introduces the core device of Scrum: the sprint. He devotes an entire chapter to the concept and purpose of the sprint: to break the work being addressed into manageable chunks and set achievable goals and deadlines. He uses that as the center around which the rest of the Scrum concepts are based.

Three detailed sections follow on how to define the work to be done, prioritize it, and then plan for development and delivery over time. First, Rubin introduces user stories and the ideas of backlog, technical debt, cadence, and velocity. He shows how to define, think about, discuss, and finally agree on a plan and a schedule for work that all of the participants can meet.

In the second of these sections, Rubin explores in detail the roles of the people who participate in the development process, identifying them by their interests in the product and their responsibilities.

This is where Rubin circles back and devotes the final five chapters to the sprint process. He guides the reader through the phases, from planning through execution to retrospective. He closes by noting that Scrum, like most of business, is an endless process, but by providing a set of markers in time, it allows the participants to see and recognize their real accomplishments and keep their eye on the the larger goals of the project.

I usually read and review books with an individual reader in mind, but it is really difficult to gauge how effective a book like this would be for an individual. Books on software development processes are about interactions and communications, and these are not going to be immediately applicable for a reader in isolation no matter how motivated he or she is. For a person who is joining a team already using Scrum this book may be some help. Where it will excel is as a manual for a team lead who is familiar with the process but needs a touchstone to stay grounded while coaching others. Read once, return often, make mistakes, and learn.

Kanban in Action
Marcus Hammarberg and Joakim Sunden
Manning Publications, 2014, 330 pages
ISBN 978-1-617291-05-0
Reviewed by Mark Lamourine

I have mixed feelings sometimes about the "in Action" theme of this series of books. I like books that are either reference or tutorial, and sometimes these books try to span the two types without really reaching the goals of either. Kanban, though, fits the "in Action" slot perfectly.

Hammarberg and Sunden begin by introducing the process itself, then laying down the philosophy that guides the process. They don't shy away from discussing the pitfalls and mistakes that new kanban participants can make. Most significantly, they finish with a section on teaching kanban, closing the loop with the reader.

Kanban is a group process. It centers around the kanban board and the cards, which represent tasks, but the heart of kanban is the process and the culture it builds. Kanban requires practice and diligence until the process becomes comfortable and innate. Then it will no longer seem like something imposed, but rather a natural way of thinking when organizing and managing work for a group.

Whenever I can, I like to review both the dead-tree and ebook versions of books. Personally, I like the experience of paper, but I know others who prefer searchable media. Often there are significant differences in the presentation, especially in the graphics and the code sample rendering on tablets and phones. This is a case where both are effective, but the differences remain.

The graphics in the ebook are clean and full color. This is especially important when the discussion is how to use color to convey information on the board. This is a significant loss in the black-and-white paper copy. With its compactness and searchability, the ebook version would be my choice for a coach or group leader learning and teaching kanban. The paper will be on my shelf to scan and lend.

The authors do a good job of presenting the practice and theory of kanban. They address those learning kanban as team members and leaders, and include frequent sidebars to coaches. I do wish they'd included a section devoted to tips and guidelines for coaches. In my experience, coaching can be a full-time job, and it requires a constant awareness of both the topic under discussion and the "meta topic": keeping the discussion moving and on message. Everything I could want is there, I just wish I could find it in one place.

I said I generally like books that are either tutorial or reference. I think *Kanban in Action* actually will serve both purposes whether you're a member of a team or a new team lead or coach. I'm going to keep both the paper book and the ebook handy.

iOS Application Security: The Definitive Guide for Hackers and Developers
David Thiel
No Starch Press, 2015, 296 pages
ISBN: 978-1-59-327601-0
Reviewed by Peter Gutmann

This book begins with a good, solid backgrounder on iOS development, debugging, and testing that covers the first hundred-odd pages, which was useful for me as a non-iOS developer but which is something that I get the feeling the target audience should know already. It's in Part III, which covers the security aspects of the iOS API, that things get interesting.

For example, there have been a number of studies done on Android that revealed the widespread misuse of TLS, something that's typically done in order to make it easy (or at least easier) to use, but which also renders it totally insecure. The book goes to some lengths to tell developers both how to detect signs of this misuse in other apps and libraries and how to avoid doing it themselves, either by ensuring that the certificate checking is done right or, better, by using certificate pinning in which only specific certificates are trusted rather than anything that turns up signed by a commercial CA. The author's background in security research and pen-testing really comes through here in that he's seen the things that can go wrong and makes a point of addressing these specific issues, rather than just paraphrasing the API documentation.

The rest of the book continues in this manner, providing lots of information and advice to augment the standard documentation on various security-relevant areas, including numerous notes on informal workarounds for issues that developers have discovered over time. In that sense it's a bit like an iOS-security-oriented subset of Stack Overflow, providing all sorts of useful advice to developers that isn't covered in standard documentation.

The next section of the book contains a quick overview of non-iOS-specific issues like buffer and integer overflows, XSS, SQL and XML injection, and so on, standard OWASP issues that are also covered extensively elsewhere. While it's good to at least mention these issues here, given what a hugely complex topic this is and how difficult it is to address in the limited space that's available, it would have been useful to refer readers to more comprehensive coverage like the OWASP Top Ten or CERT's secure coding guides, or for non-free sources, something like *The Art of Software Security Assessment*.

Finally, the book concludes with sections on using (and not misusing) Apple-specific mechanisms like the keychain and dealing with privacy issues around user tracking and unique IDs, extending Apple's not-always-up-to-date-or-completely-accurate documentation in order to give developers best-practice advice on how to get things right.

In summary, this is a book that every iOS developer needs to read and then act on. The next time you see an app that leaks private data everywhere, is vulnerable to a whole host of injection attacks, and uses crypto like it's 1995, ask them why they didn't consult this book before shipping.

The Car Hacker's Handbook: A Guide for the Penetration Tester

Craig Smith
No Starch Press, 2016, 278 pages
ISBN: 978-1-59327-703-1

Reviewed by Rik Farrow

Ever since I got to work with Ian Foster and Karl Koscher on their CAN Bus article (and hear their WOOT '15 presentation), I've found myself wanting to know more about car hacking. Foster and Koscher were working at UCSD while the Jeep hackers, Charlie Miller and Chris Valasek, were devoting a significant chunk of their lives to hacking the Jeep. I wondered whether it was possible to get started in this field, or at least to exercise my curiosity about my own car.

Smith's *Handbook* does a very good job of helping you understand your car's (or a target car's) networking and computing environment. Smith starts out with a simplified description of penetration testing, then heads into his area of expertise: car networks. I was surprised (but shouldn't have been) that there are multiple networks in cars, but pleased to learn that the CAN Bus is the most common and certainly the best documented one. And the Linux kernel has had support for devices that interface to the CAN Bus for many years. Smith spends an entire chapter on explaining how to use Linux tools for communicating with a CAN Bus, as well as another chapter about setting up a testbed environment so you can learn more without risking the device you use to commute to work.

Smith is best when he is describing buses and Electronic Control Units (ECUs) but is not strong when it comes to disassembling binaries. He does provide pointers to tools, hints on how to identify the type of CPU, and so on, but I think his strong point is really on the hardware and communication protocol sides of things.

Even if you aren't interested in becoming a car penetration tester, but you do want to know more about the collection of computers you routinely drive, you would do well to buy and read 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

USENIX Board of Directors

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

PRESIDENT
Brian Noble, *University of Michigan*
noble@usenix.org

VICE PRESIDENT
John Arrasjid, *EMC*
johna@usenix.org

SECRETARY
Carolyn Rowland, *National Institute of Standards and Technology*
carolyn@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

Daniel V. Klein, *Google*
dan.klein@usenix.org

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

EXECUTIVE DIRECTOR
Casey Henderson
casey@usenix.org

Results of the Election for the USENIX Board of Directors, 2016–2018

The newly elected Board will assume office at the conclusion of the June Board of Directors meeting, which will take place immediately before the 2016 USENIX Annual Technical Conference in Denver, CO, June 22-24, 2016.

PRESIDENT
Carolyn Rowland, *National Institute of Standards and Technology (NIST)*

VICE PRESIDENT
Hakim Weatherspoon, *Cornell University*

SECRETARY
Michael Bailey, *University of Illinois at Urbana-Champaign*

TREASURER
Kurt Opsahl, *Electronic Frontier Foundation*

DIRECTORS
Cat Allman, *Google*

David N. Blank-Edelman, *Apcera*

Angela Demke Brown, *University of Toronto*

Daniel V. Klein, *Google*

Notice of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held on Tuesday, June 21, in Denver, CO, during the 2016 USENIX Annual Technical Conference.

What USENIX Means to Me

by John Yani Arrasjid, USENIX Vice President

My involvement with USENIX began almost 30 years ago while working on my computer science degree at SUNY Buffalo. I attended my first USENIX conference in San Diego, CA, and was hooked from then on. Before USENIX started hosting the narrower-focused events that we have today, they ran two technical conferences a year. Because my first job was working in the languages team at AT&T, many of my coworkers and leaders participated in USENIX events. I had my start as a UNIX hacker, learning as much as I could and even contributing code back to the BSD 4.3 release.

Attending a USENIX event allowed me to interact with my peers, industry luminaries, and potential team members. I remember at a USENIX conference in San Diego having dinner with Jeff Forys, Van Jacobson, Gretchen Phillips, and others, when Van borrowed a pen and a napkin and started writing. Much later I learned that at the dinner he had had an inspiration for his TCP header prediction algorithm. Many great things have had their start at USENIX conferences!

I have always found the “hallway track” to be a big benefit of attending conferences. I remember finishing BoFs late in the evening, followed by listening to stories by others in the community like Rob Pike, John Quarterman, and David Blank-Edelman, many of whom became friends over the years.

As my career evolved from programming to systems design and system administration, I continued to follow USENIX conferences, relying heavily on the tutorial program. Many thanks to Dan Klein in those early days for helping me get my footing and for eventually mentoring me on delivering USENIX tutorials on virtualization and cloud technologies, which I did for many years.

USENIX has played a very important role in my work and in enabling me to find new opportunities. I feel strongly enough about this that I have worked on three books on virtualization and the cloud in the Short Topics in System Administration series. Seeing the value that USENIX brings to its communities, my coauthors and I have agreed to give all proceeds from the books and the tutorials I taught at LISA back to USENIX.

Several years ago I made a further commitment to the sustainability of USENIX and have been an acting board member for three terms, serving on several USENIX and conference committees and serving as program co-chair for the LISA16 conference.

I will not be returning in my current role as USENIX Board of Directors Vice President for the next term but hope to return in the future. Thank you to those supporting my work and to USENIX for allowing me to participate and help shape USENIX.

My focus this year will be on making LISA16 a success with my co-chair Matt Simmons as we develop a program with Mike Ciavarella, Patrick Cable, Ben Cotton, Lee Damon, and Chris St. Pierre, with additional input from past chairs Carolyn Rowland and David Blank-Edelman. As always, I see teamwork in this and other conferences, with everyone contributing towards a successful conference!

USENIX continues to deliver on quality at its conferences by allowing a vendor-agnostic approach. USENIX’s support of open access to papers for researchers and the community and its active approach to achieving diversity in speakers and attendees have been immensely valuable.

I know that time will continue to provide new opportunities for USENIX to take a leadership role in the communities that it supports, including newer offerings such as SREcon and the Enigma conference on security threats and novel attacks.

I continue to be excited about attending USENIX events and especially about working with the USENIX staff, who have given their utmost to provide the quality that USENIX attendees have come to expect. I see USENIX staff, who ensure that members and conference attendees have a great experience, as USENIX’s crown jewel! I encourage others to support USENIX activities through membership. Those fees are part of what allows USENIX to put on great events and are the basis of the support for open access and low student fees.

I plan to continue learning through the conferences and other activities that USENIX presents. I encourage you to also contribute as a volunteer and to donate to the USENIX areas you can support, such as the Open Access hosting of papers and the USENIX student grant program.

Thank you.

John Yani Arrasjid, VCDX-001
USENIX Association Vice President
Sr. Consultant Technologist, EMC Office of the CTO

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

March 27–29, 2017 • Boston, MA

Sponsored by USENIX, the Advanced Computing Systems Association

Important Dates

- Paper titles and abstracts due: **September 14, 2016**
- Full paper submissions due: **September 21, 2016**
- Notification to authors: **December 5, 2016**
- Final paper files due: **February 23, 2017**

Conference Organizers

Program Co-Chairs

Aditya Akella, *University of Wisconsin–Madison*
Jon Howell, *Google*

Program Committee

Sharad Agarwal, *Microsoft*
Tom Anderson, *University of Washington*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Anirudh Badam, *Microsoft*
Mahesh Balakrishnan, *Yale University*
Fabian Bustamante, *Northwestern University*
Ranveer Chandra, *Microsoft*
David Choffnes, *Northeastern University*
Romit Roy Choudhury, *University of Illinois at Urbana–Champaign*
Mosharaf Chowdhury, *University of Michigan*
Mike Dahlin, *Google*
Anja Feldmann, *Technische Universität Berlin*
Rodrigo Fonseca, *Brown University*
Nate Foster, *Cornell University*
Deepak Ganesan, *University of Massachusetts Amherst*
Phillipa Gill, *Stony Brook University*
Srikanth Kandula, *Microsoft*
Teemu Koponen, *Styra*
Sanjeev Kumar, *Uber*
Swarun Kumar, *Carnegie Mellon University*
Wyatt Lloyd, *University of Southern California*
Boon Thau Loo, *University of Pennsylvania*
Jacob Lorch, *Microsoft*
Ratul Mahajan, *Microsoft*
Dahlia Malkhi, *VMware*
Dave Maltz, *Microsoft*
Z. Morley Mao, *University of Michigan*
Michael Mitzenmacher, *Harvard University*
Jason Nieh, *Columbia University*
George Porter, *University of California, San Diego*
Luigi Rizzo, *University of Pisa*
Srini Seshan, *Carnegie Mellon University*
Anees Shaikh, *Google*
Ankit Singla, *ETH Zurich*
Robbert van Renesse, *Cornell University*
Geoff Voelker, *University of California, San Diego*
David Wetherall, *Google*
Adam Wierman, *California Institute of Technology*
John Wilkes, *Google*
Minlan Yu, *University of Southern California*
Heather Zheng, *University of California, Santa Barbara*
Lin Zhong, *Rice University*

Steering Committee

Katerina Argyraki, *EPFL*
Paul Barham, *Google*
Nick Feamster, *Georgia Institute of Technology*
Casey Henderson, *USENIX Association*
Arvind Krishnamurthy, *University of Washington*
Jeff Mogul, *Google*
Brian Noble, *University of Michigan*
Timothy Roscoe, *ETH Zurich*
Alex C. Snoeren, *University of California, San Diego*

Overview

NSDI focuses on the design principles, implementation, and practical evaluation of networked and distributed systems. Our goal is to bring together researchers from across the networking and systems community to foster a broad approach to addressing overlapping research challenges.

NSDI provides a high-quality, single-track forum for presenting results and discussing ideas that further the knowledge and understanding of the networked systems community as a whole, continue a significant research dialog, or push the architectural boundaries of network services.

Topics

We solicit papers describing original and previously unpublished research. Specific topics of interest include but are not limited to:

- Highly available and reliable networked systems
- Security and privacy of networked systems
- Distributed storage, caching, and query processing
- Energy-efficient computing in networked systems
- Cloud/multi-tenant systems
- Mobile and embedded/sensor applications and systems
- Wireless networked systems
- Network measurements, workload, and topology characterization systems
- Self-organizing, autonomous, and federated networked systems
- Managing, debugging, and diagnosing problems in networked systems
- Virtualization and resource management for networked systems and clusters
- Systems aspects of networking hardware
- Experience with deployed/operational networked systems
- Communication and computing over big data on a networked system
- Practical aspects of network economics
- An innovative solution for a significant problem involving networked systems

Operational Systems Track

In addition to papers that describe original research, NSDI '17 also solicits papers that describe the design, implementation, analysis, and experience with large-scale, operational systems and networks. We encourage submission of papers that disprove or strengthen existing assumptions, deepen the understanding of existing problems, and validate known techniques at scales or environments in which they were never used or tested before. Such operational papers need not present new ideas or results to be accepted.

Authors should indicate on the title page of the paper and in the submission form that they are submitting to this track.

What to Submit

NSDI '17 is double-blind, meaning that authors should make a good faith effort to anonymize papers. This is new for NSDI in 2017. As an author, you should not identify yourself in the paper either explicitly or by implication (e.g., through the references or acknowledgments). However, only non-destructive anonymization is required. For example, system names may be left un-anonymized, if the system name is important for a reviewer to be able to evaluate the work. For example, a paper on experiences with the design of .NET should not be re-written to be about "an anonymous but widely used commercial distributed systems platform."

Additionally, please take the following steps when preparing your submission:

- Remove authors' names and affiliations from the title page.
- Remove acknowledgment of identifying names and funding sources.
- Use care in naming your files. Source file names, e.g., Joe.Smith.dvi, are often embedded in the final output as readily accessible comments.
- Use care in referring to related work, particularly your own. Do not omit references to provide anonymity, as this leaves the reviewer unable to grasp the context. Instead, a good solution is to reference your past work in the third person, just as you would any other piece of related work.
- If you need to reference another submission at NSDI '17 on a related topic, reference it as follows: "A related paper describes the design and implementation of our compiler [Anonymous 2017]." with the corresponding citation: "[Anonymous 2017] Under submission. Details omitted for double-blind reviewing."
- Work that extends an author's previous workshop paper is welcome, but authors should (a) acknowledge their own previous workshop publications with an anonymous citation and (b) explain the differences between the NSDI submission and the prior workshop paper.
- If you cite anonymous work, you must also send the deanonymized reference(s) to the PC chair in a separate email.
- Blinding is intended to not be a great burden. If blinding your paper seems too burdensome, please contact the program co-chairs and discuss your specific situation.

Submissions must be no longer than 12 pages, including footnotes, figures, and tables. Submissions may include as many additional pages as needed for references and for supplementary material in appendices. The paper should stand alone without the supplementary material, but authors may use this space for content that may be of interest to some readers but is peripheral to the main technical contributions of the paper. Note that members of the program committee are free to not read this material when reviewing the paper.

Submissions must be in two-column format, using 10-point type on 12-point (single-spaced) leading, with a maximum text block of 6.5" wide x 9" deep, with .25" inter-column space, formatted for 8.5" x 11" paper. Papers not meeting these criteria will be rejected without review, and no deadline extensions will be granted for reformatting. Pages should be numbered, and figures and tables should be legible when printed without requiring magnification. Authors may use color in their figures, but the figures should be readable when printed in black and white. All papers must be submitted via the Web submission form linked from the Call for Papers Web site, www.usenix.org/nsdi17/cfp.

Submissions will be judged on originality, significance, interest, clarity, relevance, and correctness.

Policies

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.

Previous publication at a workshop is acceptable as long as the NSDI submission includes substantial new material. See remarks above about how to cite and contrast with a workshop paper.

Authors uncertain whether their submission meets USENIX's guidelines should contact the Program Co-Chairs, nsdi17chairs@usenix.org.

Papers accompanied by nondisclosure agreement forms will not be considered. All submissions will be treated as confidential prior to publication on the USENIX NSDI '17 web site; rejected submissions will be permanently treated as confidential.

Ethical Considerations

Papers describing experiments with users or user data (e.g., network traffic, passwords, social network information), should follow the basic principles of ethical research, e.g., beneficence (maximizing the benefits to an individual or to society while minimizing harm to the individual), minimal risk (appropriateness of the risk versus benefit ratio), voluntary consent, respect for privacy, and limited deception. When appropriate, authors are encouraged to include a subsection describing these issues. Authors may want to consult the Menlo Report at www.caida.org/publications/papers/2012/menlo_report_actual_formatted/ for further information on ethical principles, or the Allman/Paxson IMC '07 paper at conferences.sigcomm.org/imc/2007/papers/imc76.pdf for guidance on ethical data sharing.

Authors must, as part of the submission process, attest that their work complies with all applicable ethical standards of their home institution(s), including, but not limited to privacy policies and policies on experiments involving humans. Note that submitting research for approval by one's institution's ethics review body is necessary, but not sufficient—in cases where the PC has concerns about the ethics of the work in a submission, the PC will have its own discussion of the ethics of that work. The PC's review process may examine the ethical soundness of the paper just as it examines the technical soundness.

Processes for Accepted Papers

If your paper is accepted and you need an invitation letter to apply for a visa to attend the conference, please contact conference@usenix.org as soon as possible. (Visa applications can take at least 30 working days to process.) Please identify yourself as a presenter and include your mailing address in your email.

Accepted papers may be shepherded through an editorial review process by a member of the Program Committee. Based on initial feedback from the Program Committee, authors of shepherded papers will submit an editorial revision of their paper to their Program Committee shepherd. The shepherd will review the paper and give the author additional comments. All authors, shepherded or not, will upload their final file to the submissions system by the camera ready date for the conference Proceedings.

All papers will be available online to registered attendees before the conference. 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 first day of the conference.

Best Paper Awards

Awards will be given for the best paper(s) at the conference.

Community Award

To encourage broader code and data sharing within the NSDI community, the conference will also present a "Community Award" for the best paper whose code and/or data set is made publicly available by the final papers deadline, February 23, 2017. Authors who would like their paper to be considered for this award will have the opportunity to tag their paper during the submission process.

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.

REAL SOLUTIONS FOR REAL NETWORKS

FREE DVD CentOS 7(1511) Rockstor Tuning MySQL

2 FREE DISTROS!

ADMIN Network & Security

APR/MAY 2016

Tuning MySQL

Speed up your MySQL database

6 SERVER DISTROS FOR SMALL BUSINESS

Dive into DevOps with Otto
Automating development and deployment

Get to know the Linux Storage Stack

What's new in PostgreSQL 9.5

Integrate Office 365 into hybrid environments

Container Corner
• Container solutions, old and new
• Docker monitoring

Backup Tools
rdiff-backup and rsnapshot

Schedulix
Enterprise job scheduling system

• Unix • Solaris

FREE CD or DVD in Every Issue!

Each issue delivers technical solutions to the real-world problems you face every day.

Learn the latest techniques for better:

- network security
- system management
- troubleshooting
- performance tuning
- virtualization
- cloud computing

on Windows, Linux, Solaris, and popular varieties of Unix.

6 issues per year!

ORDER ONLINE AT: shop.linuxnewmedia.com

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

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

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

25TH USENIX SECURITY SYMPOSIUM

AUGUST 10–12, 2016 • AUSTIN, TX

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

The following co-located events will take place immediatey before the symposium:

WOOT '16: 10th USENIX Workshop on Offensive Technologies
August 8–9

CSET '16: 9th Workshop on Cyber Security Experimentation and Test
August 8

FOCI '16: 6th USENIX Workshop on Free and Open Communications on the Internet
August 8

ASE '16: 2016 USENIX Workshop on Advances in Security Education
August 9

HotSec '16: 2016 USENIX Summit on Hot Topics in Security
August 9

Register by July 18 and Save!
www.usenix.org/sec16