;login:

AUGUST 2015 VOL. 40, NO. 4

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Security and Sysadmin

usenix 40TH ANNIVERSARY

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

USENIX Security '15: 24th USENIX Security Symposium

August 12–14, 2015, Washington, D.C., USA
www.usenix.org/usenixsecurity15

Co-located with USENIX Security '15:

WOOT '15: 9th USENIX Workshop on Offensive Technologies
August 10–11, 2015
www.usenix.org/woot15

CSET '15: 8th Workshop on Cyber Security Experimentation and Test
August 10, 2015
www.usenix.org/cset15

FOCI '15: 5th USENIX Workshop on Free and Open Communications on the Internet
August 10, 2015
www.usenix.org/foci15

HealthTech '15: 2015 USENIX Summit on Information Technologies for Health
Safety, Security, Privacy, and Interoperability of Health Information Technologies
August 10, 2015
www.usenix.org/healthtech15

JETS '15: 2015 USENIX Journal of Election Technology and Systems Workshop
(*Formerly EVT/WOTE*)
August 11, 2015
www.usenix.org/jets15

HotSec '15: 2015 USENIX Summit on Hot Topics in Security
August 11, 2015
www.usenix.org/hotsec15

3GSE '15: 2015 USENIX Summit on Gaming, Games, and Gamification in Security Education
August 11, 2015
www.usenix.org/3gse15

LISA15

November 8–13, 2015, Washington, D.C., USA
www.usenix.org/lisa15

Co-located with LISA15:

UCMS '15: 2015 USENIX Container Management Summit
November 9, 2015
Submissions due September 5, 2015
www.usenix.org/ucms15

URES '15: 2015 USENIX Release Engineering Summit
November 13, 2015
Submissions due September 4, 2015
www.usenix.org/ures15

FAST '16: 14th USENIX Conference on File and Storage Technologies

February 22–25, 2016, Santa Clara, CA, USA
Submissions due September 21, 2015
www.usenix.org/fast16

NSDI '16: 13th USENIX Symposium on Networked Systems Design and Implementation

March 16–18, 2016, Santa Clara, CA, USA
Paper titles and abstracts due: September 17, 2015
www.usenix.org/nsdi16

USENIX ATC '16: 2016 USENIX Annual Technical Conference

June 22–24, 2016, Denver, CO, USA
Submissions due February 1, 2016
www.usenix.org/atc16

Co-located with USENIX ATC '16 and taking place June 20–21, 2016:

HotCloud '16: 8th USENIX Workshop on Hot Topics in Cloud Computing

HotStorage '16: 8th USENIX Workshop on Hot Topics in Storage and File Systems

USENIX Security '16: 25th USENIX Security Symposium

August 10–12, 2016, Austin, TX, USA

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

November 2–4, 2016, Savannah, GA, USA

Stay Connected...

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

;login:

AUGUST 2015 VOL. 40, NO. 4

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

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

www.usenix.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published 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.

EDITORIAL

Musings

RIK FARROW

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

I first encountered Sergey Bratus in a dingy stairwell in a Westin hotel in San Francisco. We were attending the 20th USENIX Security Symposium, and Sergey was a co-author of two WOOT papers. I could tell that an article pitch was coming and listened carefully as Sergey expounded on *weird machines*, unplanned-for VMs that exist in most code.

Sergey's student, James Oakley, had won the Best Student Paper award for showing how `gcc`'s exception handling format (DWARF) was rich enough to provide a complete execution environment. While this notion appeared a bit obscure to me, even as it was alarming that DWARF was exploitable, I still wondered just how big the impact was. I have paid attention to new exploits since I became interested in UNIX security in 1984, and couldn't recall any exploits that relied on this particular format.

Sergey, a short, rounded man with a graying comb-over, patiently explained to me that it wasn't just this example: weird machines could be found everywhere in code. And the more Sergey talked, the more I began to see the connection between the exploits I had studied over many years and what he explained in that dim and echoing stairwell.

Sergey, a Research Associate Professor at Dartmouth, has co-authored many papers and several *;login:* articles on this topic since that day. He approached me again this year (by email), asking me if I wanted to attend the LangSec workshop [1] happening as part of the IEEE Security and Privacy Workshops in May 2015. He also had another article idea, but I wanted something different: a clear description of the problems caused by weird machines, without resorting to insider jargon (like the term *weird machines*). Fortunately for us, Sergey, along with Meredith Patterson and Anna Shubina, did spend a lot of time writing an article for this issue. And I believe they've done a great job.

If I were to attempt to describe this issue as an elevator pitch (you have just 30 seconds), here's what I'd say. There is a programming issue that is the single cause of most exploits, and while it is possible in many cases to fix this problem, it has been ignored. This issue can be fixed by using programming techniques, many over 40 years old, that get ignored by programmers who write exploitable code instead. But there are cases where proper coding cannot help you, because the protocols involved are too complex by design. And some of those impossibly complex protocols include some of the foundations for the security of the Internet, like TLS and HTML5.

While fixing problems with input parsing, the appropriate place in any program, won't solve all security issues, this single type of fix would do more to improve the security of our computers, cars, smartphones, and devices than would any other change. In fact, any software-controlled device that accepts input beyond a simple on-and-off switch will *never* be secure without observance of the principles described in Sergey's article. Those principles are based on both research as well as years of observation into exploitable software, and the conversion to having parsers that can be proven to be correct will have more impact than anything else we could possible do to improve security today.

Our computers are flexible by design—that's what makes them so useful for doing a huge variety of tasks. If we expose our computers through the use of complex parsers or protocols to Turing-complete input languages, we must expect that our software, and our devices, can never be made secure. Attackers will continue to make our devices dance.

The Lineup

I've already provided an introduction to the first article, so let's consider the second. Sun et al. wrote a HotOS workshop paper about their research into unreliable operating systems. Their insight is that many exploits are brittle, and providing some randomness to the responses of the operating system to programs that aren't whitelisted will disrupt their behavior.

Zhuang et al. have built an environment that supports the collection of sensor data from smartphones. Their solution must overcome both privacy concerns and security issues involved in running software on strangers' phones.

I interviewed Marc Maiffret, a self-educated man who founded a successful security company at age 17 after a bit of a rough start. Marc has a unique viewpoint into the world of Microsoft security, having helped to prod Microsoft into a better security posture.

Mark D. Roth explains how Google uses an unreliability budget to provide more reliable services. This is a neat idea, one I first heard about during SREcon in 2014, and am happy that the unreliability budget has finally been clearly explained.

Andy Seely continues his series on managing with an article looking at the seven levers that can be used to help retain talented employees.

Gunawi et al. have shared their ongoing research into the causes of failures in distributed applications, such as HDFS and Cassandra. Some of the problems only appear at large scale, making them difficult to test, while others are more tractable.

David Beazley continues his two part series on concurrency in Python by explaining coroutines. Coroutines rely on application-level programming to provide a form of concurrency, using `yield`, but still have the Global Interpreter Lock to deal with.

David N. Blank-Edelman also has a second part in his own series about concurrency in Perl, using the Coro modules. Coro uses `cede` to yield control to other threads, and this can be done using semaphores, or by using other modules, like AnyEvent.

Dave Josephsen shares his experience in determining Key Performance Indicators (KPI), in particular, by choosing the latencies measured between the components of a service.

Dan Geer and HD Moore have taken a measured look at the number of IPv4 addresses that you can actually probe, and it appears that there are huge enclaves of devices that are hidden, generally by mobile broadband providers. There are also, of course, devices that we wish *were* hidden, provided mostly by cable companies.

Robert G. Ferrell muses about the future of quantum computing. Specifically, just how will we write scripts to manage systems where each test value can be both true and false at the same time.

Mark Lamourine has written two book reviews for this issue. His first covers a book on Swift, Apple's new language for apps. Mark takes a look at a book on programming in Python on the Raspberry Pi for his second review.

I started this column discussing a topic, input parsing, that is actually not as simple as I might have implied. I doubt that many programmers today have even heard of the Chomsky hierarchy of formal languages [2], first described by Noam Chomsky in 1956. And even if programmers are aware of this hierarchy, grasping the difference between context-free and context-sensitive grammars will be far beyond what we should expect of people writing Web applications in PHP or JavaScript.

But I certainly believe that computer scientists and members of industry who are responsible for protocols, such as HTML5, TLS, X.509, XML, and IPv6, should be aware of the implications of designs that require nondeterministic Turing machines, that is, ones that cannot be proven to be correct, to interpret them. When we base our technological future on systems that are insecure by design, we should not be surprised by that very lack of security that surfaces daily.

References

[1] Second Workshop in LangSec (Language Security): http://spw15.langsec.org/; first workshop: http://spw14.langsec.org/.

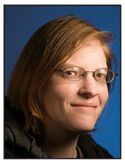
[2] The Chomsky Hierarchy: https://en.wikipedia.org/wiki/Chomsky_hierarchy.

SECURITY

The Bugs We Have to Kill

SERGEY BRATUS, MEREDITH L. PATTERSON, AND ANNA SHUBINA

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


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

Anna Shubina is a Research Associate at the Dartmouth Institute for Security, Technology, and Society and maintains the CRAWDAD.org repository of traces and data for all kinds of wireless and sensor network research. She was the operator of Dartmouth's Tor node when the Tor network had about 30 nodes total.
ashubina@cs.dartmouth.edu

The code that parses inputs is the first and often the only protection for the rest of a program from malicious inputs. No programmer can afford to verify every implied condition on every line of code—even if this were possible to implement without slowing execution to a crawl. The parser is the part that is supposed to create a world for the rest of the program where all these implied conditions are true and need not be explicitly checked at every turn. Sadly, this is exactly where most parsers fail, and the rest of the program fails with them. In this article, we explain why parsers continue to be such a problem, as well as point to potential solutions that can kill large classes of bugs.

To do so, we are going to look at the problem from the computer science theory angle. Parsers, being input-consuming machines, are quite close to the theory's classic computing models, each one an input-consuming machine: finite automata, pushdown automata, and Turing machines. The latter is our principal model of general-purpose programming, the computing model with the ultimate power and flexibility. Yet this high-end power and flexibility come with a high price, which Alan Turing demonstrated (and to whose proof we owe our very model of general-purpose programming): our inability to predict, by any general static analysis algorithm, how programs for it will execute.

Yet most of our parsers are just a layer on top of this totally flexible computing model. It is not surprising, then, that without carefully limiting our parsers' design and code to much simpler models, we are left unable to prove these input-consuming machines secure. This is a powerful argument for making parsers and their input formats and protocols simpler, so that securing them does not require having to solve undecidable problems!

Parsers, Parsers Everywhere

To quote Koprowski and Binsztok [1]:

> Parsing is of major interest in computer science. Classically discovered by students as the first step in compilation, parsing is present in almost every program which performs data-manipulation. For instance, the Web is built on parsers. The HyperText Transfer Protocol (HTTP) is a parsed dialog between the client, or browser, and the server. This protocol transfers pages in HyperText Markup Language (HTML), which is also parsed by the browser. When running web-applications, browsers interpret JavaScript programs which, again, begins with parsing. Data exchange between browser(s) and server(s) uses languages or formats like XML and JSON. Even inside the server, several components (for instance the trio made of the HTTP server Apache, the PHP interpreter and the MySQL database) often manipulate programs and data dynamically; all require parsers.

So do the lower layers of the network stack down to the IP and the link layer protocols, and also other OS parts such as the USB drivers (and even the hardware: turning PHY layer symbol streams into frames is parsing, too! [2]). For all of these core protocols, we add, their parsers have had a long history of failures, resulting in an Internet where any site, program, or system that receives untrusted input can be presumed compromised.

While we may believe in special programmers who write so-called critical software with the care and precision the rest of our tribe lacks, where are these secret coding schools training such ninjas? And if these programmers are so few and far between, can we really expect them to scale? Neither collective insanity nor collective negligence are comfortable to contemplate, but so we must as our reliance on software grows.

Perhaps we don't care nearly enough. After all, every C programmer experiences thousands of segfaults while learning the language and sees that the world doesn't collapse, nor does the computer suddenly become hostile. It certainly is annoying when programs crash, but it's easy enough to restart them—with an automatic watchdog, if need be. Indeed, few of us suspect how often embedded software in our devices gets restarted.

This habituation to crashes doesn't serve us well. It forms a false perception that "bugs are just bugs," and systems that engineer around them rather than fix them can be trustworthy, except in rare and exotic cases. But, in fact, this is where the common programming intuition lets us down badly.

A segfault is a would-be corruption of memory or state, an unexpected, out-of-type memory reference that got caught. It is eminently observable and doesn't result in much computation beyond the error. Therefore, it's easy to assume the same thing about any memory corruption—unless one is familiar with just how complete a programming environment a simple memory corruption can create for an attacker, and how far and wide beyond its expected execution paths a program can run after a memory corruption.

It's natural for programmers to view the executable binary generated from their programs through the prism of their source code. In that view, functions do not get jumped into sideways, nor are they called from locations other than their explicit call sites; variables retain their values unless assigned to by name or by reference; assembly instructions cannot spring into existence unless somehow implied by the code's semantics; and so on.

As attackers know, all of these expectations are false. In the gap between these expectations and the actual reality of binary execution at runtime, entire modes of programming sprang up. Around 2000, hacker researchers demonstrated that if one manages to overflow the program stack with what looks like a sequence of stack frames, one can construct arbitrary programs that will successfully execute in the corrupted process [3]. In 2007, an academic paper by Hovav Shacham [4] made this understanding precise by proving that a typical process is in fact a Turing-complete environment for such programming.

However, this kind of bare-boned exploit programming likely still feels too exotic for most programmers. Its power can only be experienced through practicing it, and most of us have our hands too full with the programming we need to do to pick up another, weirder kind of programming. So we'll need to approach it with a different set of intuitions, which are closer to the classic computer science than to hacking (although, as we will see, here hacking comes very close to the very foundations of computer science).

When Programs Crash, Where Do Their Proofs of Correctness Go?

C. A. R. Hoare developed the beginnings of the axiomatic proofs-of-correctness theory for programs in 1968. Owing to this theory, we see programs and their modules, functions, and constructs such as loops in terms of preconditions and postconditions, and chain these for proofs. Whenever such a chain can be constructed for the entire program, starting with its individual operations and statements, and the initial precondition is the atomic "True" (i.e., there are no additional preconditions), we say that we have proven the program's correctness (no matter what the inputs or the state of the rest of the world). Although few programmers actually end up proving their programs, generations of programmers have been taught to think of their loops and branches in terms of preconditions and postconditions. We intuitively understand the $P\,\{Q\}\,R$ notation even if we don't use it explicitly. That is, given preconditions P and code Q, postconditions R are assured.

But do we stop to think what happens when instead of P our code Q gets some $P' \neq P$? What will code Q be able to compute in that case? How far would possible conditions R' in $P'\,\{Q\}\,R'$ vary? Our intuition, based on axiomatic programming, does not tell us that—while an exploiter's intuition is all about it.

Some of our best theoretic means for achieving predictable code behavior, such as Proof-Carrying Code (PCC) and programming language safety guarantees, are of little help against the divergence in preconditions. For PCC, we can only be sure of what the code does if it's run within its specification [5]; otherwise, the proofs it carries do not preclude it from entering an unexpected "weird" state. The language-based guarantees rigorously proven on the source code can be broken either by the language's runtime implementation [6] or by compiler optimizations [7].

For parsers and the code that receives parsed input data, this question is even simpler: What happens when the inputs that hit the parser are invalid and unexpected? What will the parser itself compute then? If allowed through to the rest of the code, what effects will the inputs transformed by the parser have on it? Clearly, if the parser was supposed to reject the data and didn't, assumed preconditions to subsequent code on its path will not hold. The runtime world then belongs to whoever can predict the computational effects of violated preconditions, *even when the code is proven correct*.

It gets worse. Suppose we have a program that implements a simple finite state machine that responds to an input language. What happens when this code is fed inputs not in this language? Will the program still behave like a finite state machine, or will it present a much richer programming model to the attacker able to feed it custom-crafted inputs?

Accidentally Turing-Complete

The answer is almost certainly "yes." Software and even firmware intended as automata with limited, specialized purposes have been shown to actually play the role of a universally programmable Turing machine to attacker's inputs, which, for all their syntactic peculiarity, acted as programs for these machines. These inputs didn't even need to be malformed; either buffer or integer overflow bugs were similarly not a necessity.

For example, the standard ELF relocation code provided by the Linux dynamic linker and present in any dynamically linked process is driven by the relocation metadata present in every ELF executable. This code is meant to patch up the addresses in code that is loaded into a different address range than it was linked for—as a means of ASLR protection, for example, or simply because a previously loaded library already occupies part of the original address range—but it is capable of much more. In fact, craftily prepared well-formed metadata entries can make it carry out any computation at all, as if that code were a virtual machine and the relocation entries its bytecode [8]! This code was never meant nor written for such generality, but it can achieve it nevertheless [9].

What we think of as hardware is not far behind. For example, we trust the isolation of our processes to the x86 MMU, and we imagine it as a fairly simple mechanism that sets up our page tables on exec(), manages them on context switch, and translates every memory reference. Clearly, in this translation a finite automaton is involved, but in fact the MMU features are so rich that the configuration tables it interprets can be used to program anything—any Turing-complete computation [10]! Again, the MMU's logic was designed for a specific purpose, and great effort is spent on validating its correctness—but it turns out that it can do so much more than intended, with no bugs involved. Due to its feature-richness, merely well-formed crafted inputs suffice.

In short, computer security appears to have its very own parallel to Arthur Clarke's observation that "Any sufficiently advanced technology is indistinguishable from magic," namely, "Any sufficiently complex input format is indistinguishable from bytecode; the code receiving it is indistinguishable from a virtual machine."

The latter observation, of course, accords very well with the exploiters' everyday experiences. So long as the inputs are complex enough, and the software is correspondingly complex, there will be crashes, and some of these crashes will lead to full control of the receiving software.

The trick is putting these observations together and realizing what goes wrong. In full accordance with Clarke's laws, exploit developers lead in this exploration, because "The only way of discovering the limits of the possible is to venture a little way past them into the impossible." Indeed, in the programmers' mental models of their environments, exploits are supposed to be the impossible—and yet they exist.

The irony of these models is that the computational model of the general purpose computing, the Turing machine, was a proof of unsolvability, the impossibility of programming certain tasks due to the richness of the platform itself. The simplest of these is a particular kind of static analysis, a general algorithm for deciding statically whether a program would halt. The difficulty of this problem is by no means a fluke: according to Rice's Theorem, general algorithms for deciding other "non-trivial" properties of programs are in the same boat. This is not to say that static analysis of programs is hopeless but, rather, that it is hard, and this hardness is a matter of natural law that would not just yield to cleverness or extravagantly massive investment. As Geoffrey Pullum put it in his "Scooping the Loop Snooper" [11]:

> *No general procedure for bug checks will do.*
> Now, I won't just assert that, I'll prove it to you.
> I will prove that although you might work till you drop,
> you cannot tell if computation will stop.
>
> ...
>
> You can never find general mechanical means
> for predicting the acts of computing machines;
> it's something that cannot be done. So we users
> must find our own bugs. Our computers are losers!

This puts paid to the hope of exhaustively automating static security analysis for the kind of code that we most often write and use. Yet it is Turing's insights and his model of computing—an answer to Hilbert's tenth problem—that form the basis for most computers we know. Our software is just a layer on top of this totally flexible computer, and unless this software presents very simple parsers, that software is also likely to be totally flexible and cannot be proven to be secure—unless we programmers take great care to not use the full extent of this power and flexibility, and purposefully keep ourselves to simpler models that can be proven and verified.

Can We Verify Our Way Out of This Mess?

Maybe. First, we need to define the problem in a way that program verification tools can help. Then we need to pick a simple enough model of what parsing is and stick to it in our implementation.

Thus the long answer is, verification of parsers will help only if we co-design data formats and code that parses them. Parsers must create the preconditions for the rest of the proof; thus they should be the simplest machines possible, to ease effective verification. If you think this is a solved problem, it isn't. Quoting again from Koprowski and Binsztok,

> In the recent article about CompCert, an impressive project formally verifying a compiler for a large subset of C, the introduction starts with a question "Can you trust your compiler?" Nevertheless, the formal verification starts on the level of the [Abstract Syntax Tree] and does not concern the parser. Can you trust your parser?

So how simple should "simple" be?

Be Simple and Definite about What You Receive!

When software gets exploited by inputs—its execution takes a path it was never meant to take because of consuming the input data—it is obvious that the data is driving it to do so. But, in fact, although it may be less obvious, the data is driving the software even when it executes as expected. "The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program." [12]

This means that we should look at the data itself as a program—and model the parser code consuming it as an automaton driven by it. Then, so long as we keep this automaton simple, we can prove and verify its behavior on all possible inputs. We have the mathematics for it and a hierarchy of such automata by simplicity and power.

For example, consider a regular expression. We think of them as implemented by finite automata we can draw with circles and arrows, and emulate their execution by moving a coin from one circle to another along the arrow marked with the character we consume from the input string [13]. But then the string is what drives this automaton from state to state; it's the program for the automaton. The same is true for pushdown automata. It is obvious for a Turing machine: whatever goes on the tape is the program and is the input at the same time.

Regular expressions seem to be everyone's favorite way of validating inputs in scripting languages. This can be just right or can go horribly wrong, depending on the language of inputs one is trying to validate. Matching a regular language of inputs, one that consists of all strings matched by a regex anchored at the start and at the end of the string, would be just right. Of course, such languages work best for the data structures with no or limited nesting; for those like HTML or JSON that allow arbitrary nesting of their elements, it can go horribly wrong [14]. Validating arbitrarily nested HTML with regexes is a classic mistake, made by both novice Web developers and the designers of anti-XSS protections in IE 8 [15]. The mathematical reason for this world of XSS fail is simple: such languages are context-free or context-sensitive, and require at least a pushdown automaton to match them.

The purpose of the parser as a protector of the rest of the code is to match the correct inputs and drop the incorrect ones (without getting exploited itself, obviously). So we need to start by defining the language of the valid inputs, and then write the parser as the consuming automaton of the type we can validate. Usually this means keeping the input language regular or context-free, and using a regex (a finite state machine) or a pushdown automaton, respectively. We've seen how to safely approach what the parser consumes—but what about its outputs?

Types to the Rescue

To verify parsers, we need to first write their specifications. It's easy to say that parsers must consume strings, any strings, and reject those that are invalid or unexpected. But how can we describe what parsers must produce? What kinds of assumptions on input that passes the parser would be helpful for both ordinary programmers and the proof engineers seeking to verify their code?

This question goes back to the foundations of type theory. For example, the plight of the programmer who must rely on assumptions assured by the previous code was the subject of James Morris, Jr.'s "Types Are Not Sets" in 1973: "[The programmer] could begin each operation with a well-formedness check, but in many cases the cost would exceed that of the useful processing." Just as relevant to the programmers today as it was then!

The job of the parser then becomes clear once we see it from the type-theory angle. The parser eliminates strings; it introduces other objects of types that have to do with the program's semantics. The rest of the program assumes that these objects are well-typed; the parser is their constructor that builds them from the strings it consumes.

Parser bugs, then, generally come in two flavors: the parser code, instead of rejecting an invalid input, provides an attacker with a virtual execution engine for exploits, or the objects it constructs are not the type expected by the rest of the code. The former often occurs whenever the parser allocates and copies memory based on a value in user input it has neither fully parsed nor checked for consistency. Various integer overflows in X.509 and other ASN.1-based formats are examples of the latter: instead of the syntactically correctly encoded Bignum unbounded integer, the parser creates a bounded platform-default Fixnum [16]. So it is with Apache and Nginx chunked-encoding vulnerabilities, discussed later.

Format Foibles, Protocol Peeves

Exploiter intuition has long singled out certain syntactic features as the breeding grounds for parser vulnerabilities. Given the choice between constant-length and variable-length fields, the exploiter's money would be on the latter; several length fields that must agree for the message to be valid up the ante. A typical memory corruption scenario with such protocols involves copying some elements of input into buffers sized and dynamically allocated based on values supplied in the same input—and lying, to cause a buffer overflow. Although it's easy to blame such bugs on the implementers' negligence, it's undeniably the syntactic complexity of the underlying protocols that makes an implementer's mistake both more likely to happen and harder to catch.

Generally speaking, the more context a parser must keep to correctly parse the next element of the message, the more likely it is to get it wrong; the more complex the relationship between already parsed syntax elements and the remaining ones, the more likely an unchecked, unwarranted assumption is to slip through. Looking at the problem through the program proof lens, we can see the rapid accumulation of preconditions in context-sensitive protocols. However, the Internet—with its scale such that if a coding error can be made, it will be made—has a much more direct way of steering us towards regular and context-free formats.

In the Internet Protocol's early days, the variable-length IP options tacked on behind the constant-width IP header fields were considered essential. These days, their mere presence in a packet is enough for many firewall configurations to regard the packet as suspicious or to drop it outright. This happened for a good reason: IP option parsing bugs have plagued 1990s stacks (including firewalls like Raptor CVE-1999-0905 and Gauntlet CVE-1999-0683, which they caused to freeze or crash), made a few impressive appearances such as CVE-2005-0048 in the 2000s, and recently resurfaced as the "Darwin Nuke" kernel panic CVE-2015-1102 in Mac OS 10.10.2. Accordingly, the Internet de facto converged on the simpler constant-width IP header, a regular language—not by standard, but by a "rough consensus of firewalls."

Of course, any gains from this subsetting of IPv4 have been offset by the advent of IPv6 with its chains of variable-length Extension Headers, including nestable fragmentation headers. While concerned ASes filter and drop up to 40%(!) of certain kinds of IPv6 packets, newer RFCs call for limiting the allowed variations in header order and combinations [17]. This subsetting-by-firewall of IPv6 to a simpler grammar will likely continue.

The situation with the core trust infrastructure of the Internet, the X.509 PKI standard, is hardly more encouraging than that of IPv6. The wide variety of ways to represent basic data types such as integers and strings allowed under the ASN.1 Basic Encoding Rules (BER) makes parsing X.509 certificates and related data something of a guessing game as to what other implementations might mean; the "PKI Layer Cake" effort [15] revealed over 20 ways that different SSL/TLS implementations could interpret the same data in the certificate—including the Common Name! Thus a CA granting a certificate signing request for what looks like an innocuous domain could in fact create a certificate seen by the browsers as that of a different, high-value domain name. This abundance of differences is not surprising, since establishing equivalence of parsers is in fact a problem that becomes undecidable beyond a certain syntactic complexity, which X.509 significantly exceeds. Given the choice between ASN.1-based formats, the simpler DER and other encoding rules that fix respective canonical ways to represent each data type should be definitely preferred over BER, but syntactic complexity is the dark energy of the Internet: once created, it never goes away.

Speaking of SSL/TLS, the past year has been rich in famous SSL/TLS parser bugs. It wasn't just the infamous Heartbleed CVE-2014-0160; the GnuTLS Hello bug CVE-2014-3466 and Microsoft's Secure Channel bugs under CVE-2014-6321 demonstrate that the misery of complex input syntax really loves company.

While XML-based document formats are a definite improvement over the older binary ones, allowing a simple context-free subset to represent tree-like documents with recursively nested objects, the full XML specification still strays far enough from syntactic simplicity. Not surprisingly, the same elements, such as XML entities that introduce context-sensitivity to XML serve as a major source of its over 600 associated CVEs. By contrast, a simpler JSON, whose syntax would be context-free except for the requirement that its dictionary keys be unique, scores only about 60 CVEs; anecdotally, JSON parsers seem to be ahead of the game.

However, the Web has offset the simplicity that it promised in formats by an enormous explosion of computational power exposed to attacker inputs. Ubiquitous JavaScript ensures that the document one's client renders may have absolutely nothing to do with what one receives, precluding any kind of meaningful static analysis before rendering; instead of separating benign sheep from the malicious goats, the client has to put its trust into its sandbox being inescapable. And if this weren't bad enough, the combination of HTML5 and CSS in modern browsers already gives rise to programming models strong enough to exfiltrate one's passwords [18]. One may hope that such computations are accidental, but the demonstration that HTML and CSS3 are actually Turing-complete [19] leaves little hope that they will remain exotic or can be easily contained.

Chances are that we may need to rethink both the data formats and the computation models of the Internet before the mass of unwanted computation forces us into walled gardens of servers and peers somehow "trusted" not to poison our software.

Where Are We Now?

Decades of frustration have taught us to not roll our own crypto libraries. Although legacy crypto libraries are still complex and hard to use, new and simpler ones are just now emerging, like NaCl [20]. The Iron Age of crypto may be finally dawning on us.

With parsing, it's arguably worse. We are still in the Stone Age of parsing, despite a promising glint of Bronze and Iron here and there, or even an occasional laser beam. All across production programming, "rolling your own parser for speed" still reigns rather than raising skeptical eyebrows. Parser generators exist, but aren't seen as a vital necessity for input-handling code in either office document applications, messaging protocols, network stacks, or elsewhere; in short, their use cases are deemed limited rather than universal. Verified parsers are extremely rare; a majority of parsers are pwned-by-design, not least those we use in our cryptography.

One can continue blaming developers who don't "program securely" or fail to "validate inputs" (and some still do). However, a closer look at the nature of parser exploitation suggests this may be blaming the victim. Syntactically complex, context-sensitive protocols may in fact require the programmer to solve undecidable problems to create secure programs, an impossible feat.

As with all other kinds of engineering, the way forward lies in understanding which problems are impossible and which are merely hard, and not confusing the two. After all, every kind of engineering in the physical world works around its own impossibilities: conservation of energy and momentum, laws of thermodynamics, quantum-scale indeterminacy effects, and so on. Yet how sure can we be that random software engineers would so readily name the hard natural-law limits of their trade as physical engineers would?

It would be naive to expect that software engineering has no such limitations. Indeed, computability theory and complexity theory bring them to light. Nowhere do these limitations manifest themselves so cruelly as in our inability to predict computation. This inability is what we colloquially know as insecurity: we cannot trust our computers to stick to the computations we expect in the presence of inputs we don't control.

Building a Secure(r) Parser

We know the execution models for consuming inputs in which we can predict computation and protect it: these tend to be regular or context-free. We also know that context-sensitive and richer input languages harbor undecidable problems. As usual, the cure for an impossibility revealed by science is more science. In the case of parsers, we are lucky: we already have the mathematical models and the rough split of tasks into the possible and the impossible.

Our programming must follow these models and stay within the safe protocol designs that do not pose undecidable problems as requirements for "securing" them—that is, being able to automate testing of their implementations and reasoning about the possible courses their computations can take. For all the seeming flexibility and extensibility benefits of more complex protocols—and, respectively, more powerful computation models—building on them is like building on quicksand.

There is an important caveat for parsers explicitly hand-coded as finite automata, however: it should be clear from the code what kind of valid input any given part of it expects, and what syntactic construct it is responsible for parsing. For example, Nginx implements its parser of HTTP headers as a large hand-coded automaton (2300+ lines of C code, 57 switch statements, 272 single-character case statements). In 2013, it was found to incorrectly parse the chunk lengths in the HTTP chunked encoding (CVE-2013-2028), producing negative (signed) integers for large hexadecimal chunk lengths—exactly the same issue that was discovered for Apache in 2002 (CVE-2002-3092). It took over 11 years to find that bug in Nginx—and if you try looking through Nginx's ngx_http_parse.c to find where the chunk length is actually parsed, you will see why.

If the expected valid input is not intelligible from the code, finding bugs in it can take forever. In our experience, parsers whose code resembles the grammar of their expected inputs tend to do best. The Parser Combinator style of programming makes writing such code easy—and, although it was developed in functional languages such as Scala and Haskell, it's quite possible to use it in C/C++ and other languages as well. The Hammer parser construction kit is meant to demonstrate this; it requires no background in functional languages to use [21].

Help Me, Verifiable Parser, You Are My Only Hope!

When we look to the future of computers, what can we expect? Almost all kinds of programs will need to handle remote, untrusted inputs. The trend to connect everything and anything seems unstoppable; the "Internet of Things" and "cloud computing" (i.e., running trusted components of programs on remote systems) may only be its first wave.

Visions of self-driving cars, smart homes, and computerized medicine project from the current state of computing power, but not from its current trustworthiness. The only sustainable way to achieve these visions without an exploding attack surface is to make sure that all these programs exposed to hostile inputs can't be trivially exploited or disrupted by them. And if encrypted tunnels seem to be an answer, consider just how vulnerable the code base of our cryptographic infrastructure is to non-cryptographic attacks related to mere parsing of padding, PKCS message formats, and X.509 certificates.

The only hope for a secure connected future is software that can hold its own against the maliciously crafted inputs, without crutches such as firewalls, application proxies, antiviruses, and so on. This software will need to apply solid computation theory principles to what it accepts, and will accept only what it can validate. Once accepted, input can be turned into data types that will provide the rest of the software code with unambiguous preconditions. And although eliminating all bugs is provably impossible, the future should at least be free of the parser bugs on both input and output—the bugs we need to kill to build computers we can finally trust.

References

[1] Adam Koprowski and Henri Binsztok, "TRX: A Formally Verified Parser Interpreter," LMCS 2011.

[2] Travis Goodspeed, "Phantom Boundaries and Cross-Layer Illusions in 802.15.4 Digital Radio," First LangSec IEEE S&P Workshop, 2014: http://spw14.langsec.org/papers/8th-of-a-nybble.pdf.

[3] Gerardo Richarte, "Re: Future of Buffer Overflows," October 2000, Bugtraq: http://seclists.org/bugtraq/2000/Nov/32; Nergal, "Advanced Return-into-lib(c) Exploits: The PaX Case Study," *Phrack* 58:4, 2001; see also Sergey Bratus et al., "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation," USENIX *;login:*, December 2011, for further history.

[4] Hovav Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," ACM CCS 2007.

[5] Julien Vanegue, "The Weird Machines in Proof-Carrying Code," First LangSec IEEE S&P Workshop, 2014: http://spw14.langsec.org/papers/jvanegue-pcc-wms.pdf.

[6] Eric Jaeger, Olivier Levillain, and Pierre Chifflier, "Mind Your Language(s): A Discussion about Languages and Security," First LangSec IEEE S&P Workshop, 2014: http://spw14.langsec.org/papers/MindYourLanguages.pdf.

[7] Vijay D'Silva, Mathias Payer, Dawn Song, "The Correctness-Security Gap in Compiler Optimization," Second LangSec IEEE S&P Workshop, 2015: http://spw15.langsec.org/papers/dsilva-gap.pdf.

[8] Shapiro et al., "'Weird Machines' in ELF: A Spotlight on the Underappreciated Metadata," USENIX WOOT 2013: http://www.cs.dartmouth.edu/~sergey/wm/woot13-shapiro.pdf.

[9] Mach-O and PE formats have comparable properties.

[10] Bangert et al., "The Page-Fault Weird Machine: Lessons in Instruction-less Computation," USENIX WOOT 2013: http://www.cs.dartmouth.edu/~sergey/wm/woot13-bangert.pdf.

[11] http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html.

[12] Taylor Hornby, quoted in Dan Geer, "Dark Matter: Driven by Data," Second LangSec IEEE S&P Workshop, 2015: http://spw15.langsec.org/geer.langsec.21v15.txt.

[13] This page's links explain how regex works in general, and particularly in Perl: http://perl.plover.com/Regex/.

[14] "Parsing HTML the Cthulhu Way": http://blog.codinghorror.com/parsing-html-the-cthulhu-way/, http://blog.codinghorror.com/content/images/2014/Apr/stack-overflow-regex-zalgo.png.

[15] http://p42.us/ie8xss/; see also Eduardo Vela Nava, David Lindsay, "Abusing Internet Explorer 8's XSS Filters": http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf.

[16] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure": https://www.cosic.esat.kuleuven.be/publications/article-1432.pdf.

[17] See, e.g., F. Gont et al., "Observations on IPv6 EH Filtering in the Real World," 2015: https://tools.ietf.org/html/draft-gont-v6ops-ipv6-ehs-in-real-world-02.

[18] M. Heiderich et al., "Scriptless Attacks: Stealing the Pie without Touching the Sill," CCS 2012: https://www.hgi.rub.de/media/emma/veroeffentlichungen/2012/08/16/scriptlessAttacks-ccs2012.pdf.

[19] Eli Fox-Epstein, "Stupid Machines": https://github.com/elitheeli/stupid-machines.

[20] NaCl: http://nacl.cr.yp.to/.

[21] Hammer parser: https://github.com/UpstandingHackers/hammer. Also check out the Hammer Primer: https://github.com/sergeybratus/HammerPrimer for a gentle introduction.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Do you know about the USENIX Open Access Policy?

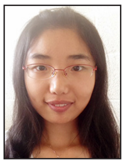
USENIX is the first computing association to offer free and open access to all of our conference proceedings and audio and video recordings. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your financial support plays a major role in making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

Please help to us to sustain and grow our open access program. Donate to the USENIX Annual Fund, renew your membership, and ask your colleagues to join or renew today.

www.usenix.org/annual-fund

The Case for Unpredictability and Deception as OS Features

RUIMIN SUN, MATT BISHOP, NATALIE C. EBNER, DANIELA OLIVEIRA, AND DONALD E. PORTER



Ruimin Sun is a first year PhD student in the Department of Electrical and Computer Engineering at the University of Florida. Her research interest lies in operating system security and software vulnerabilities. She's under the direction of Dr. Daniela Oliveira. gracesrm@ufl.edu

Matt Bishop is a Professor in the Department of Computer Science at the University of California, Davis. He does research in many areas of computer security, including data sanitization, vulnerabilities analysis, attribution, the insider problem, and computer security education. mabishop@ucdavis.edu

Natalie C. Ebner is an Assistant Professor in the Department of Psychology and adjunct faculty in the Department of Aging and Geriatric Research at the University of Florida in Gainesville, Florida. Her research adopts an aging perspective on affect and cognition. She conducts experimental research using a multi-methods approach that integrates introspective, behavioral, and neurobiological data. natalie.ebner@ufl.edu

The conventional wisdom is that OS APIs should behave predictably, facilitating software development. From a system security perspective, this predictability creates a disproportionate advantage for attackers. Could making OSes behave unpredictably create a disproportionate advantage for system defenders, significantly increasing the effort required to create malware and launch attacks without too much inconvenience for "good" software? This article explores the potential benefits and challenges of unpredictable and deceptive OS behavior, including preliminary measurements of the relative robustness of malware and production software to unpredictable behavior. We describe Chameleon, an ongoing project to implement OS behavior on a spectrum of unpredictability and deceptiveness.

Introduction

The art of deception has been successfully used in warfare for thousands of years. Strategists such as Sun Tzu, Julius Caesar, and Napoleon Bonaparte advocated the use of unpredictability and deception in conflicts as a way to confuse and stall the enemy, sap their morale, and decrease their maneuverability. A "holy grail" for system security is to put system defenders in a situation with more options than the attacker.

Unfortunately, current systems are in the exact opposite situation. System defenses generally do not adapt well to new conditions, whereas motivated attackers have effectively unlimited time and resources to find and exploit weaknesses in computer systems.

This situation is rooted in the fact that *predictability* is a first-class system design goal. Predictability simplifies application engineering and usability issues, such as compatibility among different versions of the system. The downside of predictability is a computer system monoculture [1], where vulnerabilities become reliably exploitable on all systems of the same type. With so few operating system kernels, libc implementations, or language runtimes deployed in practice, any predictable exploit applies to a significant fraction of computers in the world.

The Need for Unpredictability

At the system level, approaches to unpredictability generally involve limited randomness. For example, address space layout randomization (ASLR) randomizes the placement of pages of a program in memory during execution. An attack relying on a buffer overflow causing a branch to a library function or gadget will fail, as the address of that target will vary among instances of an operating system. But this randomization is often insufficient. In a recent paper, Bittau et al. [2] demonstrated how, even without specific knowledge of the address space layout randomization (ASLR) scheme of a Web server, an attacker can quickly identify and exploit portions of the address space that are insufficiently random.

Daniela Oliveira is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Florida. Her main research interest is interdisciplinary computer security, where she employs successful ideas from other fields to make computer systems more secure. Her current research interests include employing biology and warfare strategies to protect operating systems. She is also interested in understanding the nature of software vulnerabilities and social engineering attacks. daniela@ece.ufl.edu

Donald E. Porter is an Assistant Professor of computer science at Stony Brook University in Stony Brook, New York. His research aims to improve computer system efficiency and security. In addition to work in system security, recent projects have developed lightweight guest operating systems for virtual environments as well as efficient data structures for caching and persistent storage. porter@cs.stonybrook.edu

Although fixes to ASLR may mitigate this specific attack, this attack shows that *variation without unpredictability is not enough.* Unpredictability by half-measure leaves sufficient residual certainty that allows adversaries to craft reliable attacks even across multiple, differently randomized instances of the system.

Strategies for less predictable operating systems are constrained by concerns for efficiency and reliability. Yet consider what "efficient" and "reliable" mean for an operating system. An operating system's job is to manage tasks that the system is authorized to run, where "authorized" means "in conformance with a security policy." For *un*authorized tasks, such as those an attacker would execute to exploit vulnerabilities or otherwise misuse a system, the operating system should be as inefficient and unreliable as possible. So for "good" users and uses, the operating system should work predictably, but for "bad" users or uses, the system should be unpredictable. The latter case challenges efficiency and reliability. An extension is a *spectrum* of predictability, where the less that actions conform to the security policy, the more unpredictable the results of those actions should be.

Software Diversity

One specific, limited form of unpredictability is diversity. The intent of diversity is independence, which means that multiple instances yield the same result but in such a way that the *only* common factor is the inputs. Most fault-tolerant system designs require sufficient software diversity that faults are independent and can be masked by voting or Byzantine protocols. In practice, the barrier to implementing multiple, complete, monolithic OSes has been insurmountable.

One insight of this work is that diversifying the system implementation becomes easier as more of the system is moved to user space. Several research systems have demonstrated the value of pushing more system-level functionality into user-level libraries, such as moving I/O into user space for higher performance [3] or to reduce virtualization overheads for a single application [4]. Our vision is to mix-and-match different implementations of different components, such that one can run many instances of an application, such as a Web server, and only a few instances will share the same combinations of vulnerabilities. When the implementation effort is smaller and well defined, a small group of developers could easily generate dozens of functional implementations of each subsystem.

Application robustness can also be improved when system-level diversity is incorporated into the development and testing process. Even within POSIX, mature, portable software packages already handle considerable variations in system call behavior. Most of this maturity is the product of labor-intensive testing and bug reports across many platforms over a long period. Rather than require a software developer to manually test the software on multiple platforms, a spectrum-behavior OS would allow developers to more easily test software robustness, running the same test suite against different operating system behaviors.

Consistent versus Inconsistent Deception

Deception has been used in cyberdefense to a limited extent, primarily via *consistent deception* strategies, such as honeypots or honeynets. Consistent deception strategies make the deceiver's system appear as indistinguishable as possible from a production system. This means the deceptive system is just as predictable as the system it is impersonating. The idea of *inconsistent deception* [5], on the other hand, forgoes the need to project a false reality and instead creates an environment laden with inconsistencies designed to keep the attacker from figuring out characteristics of the real system. So long as the attacker is confused and fails to learn anything of value, the deception is successful, even more so if the attacker desists.

Iago attacks [6] are a good example of how inconsistent deception might work in practice. An Iago attack occurs when an untrusted system attacks a trusted program by returning system call results that the trusted program cannot robustly guard against, ultimately causing the trusted program to violate its security goals. We believe similar techniques can be employed for active system defense.

Unpredictability on Malware

We performed a case study on common malware, showing that malware can be quite sensitive to relatively minor misbehavior by the operating system. We used `ptrace` to alter the information returned by system calls invoked by a keylogger and botnet, introducing unpredictable behavior into their execution. In these cases, the malware ran without crashing, but some I/O were corrupted. Most I/O corruptions were within the specification of the network or potential storage failure modes; a robust application would detect most issues with end-to-end checks such as checksums or, in other cases, checks designed to shield against a malicious OS, such as MAC checks on an encrypted socket.

We selected candidate system calls for spectrum behavior based on analysis of system call behavior of benign processes and malware. We compared the system call patterns of 39 benign applications from SourceForge to 86 malware samples for Linux, including 17 back doors, 20 general exploits, 24 Trojan horses, and 25 viruses. We found that malware invokes a system call set that is smaller than benign software: approximately 50 different system calls.

In selecting strategies for spectrum behavior, our aim is to perturb system calls that harm malware, yet allow benign code to run. We found that a few system calls are critical to process start-up and execution, and cannot be easily varied; most other cases lead to non-fatal deviations. For instance, decreasing the length of a `write()` will cause a keylogger to lose keystrokes, silencing a `send()` might cause a process sending an email to fail, and extending the time of a `nanosleep()` will just slow down a process. We try to balance risks to benign processes with harm to malware through an experimentally determined *unpredictability threshold*, which bounds the amount of unexpected variation in system call behavior.

We studied the following strategies for spectrum behavior:

Strategy 1: Silence the system call. We immediately return a fabricated value upon system call invocation. This strategy only succeeds when subsequent system calls are not highly dependent on the silenced action. For example, this strategy worked for `read()` and `write()` but not on `open()`, where a subsequent `read()` or `write()` would fail.

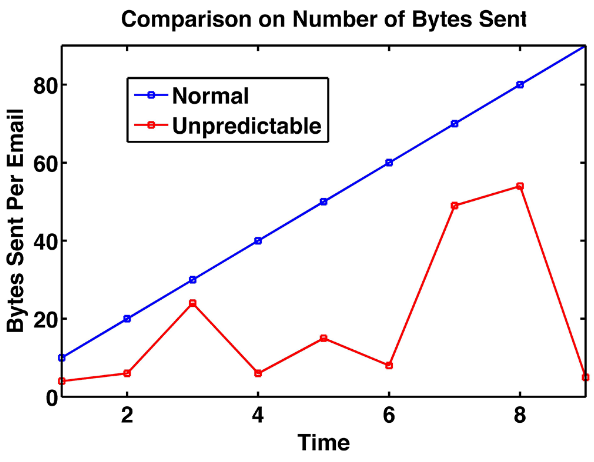


Figure 1: Comparison of email bytes sent from bots in normal and unpredictable environments

Strategy 2: Change buffer bytes. We randomly change some bytes or shorten the length of a buffer passed to a system call, such as `read()`, `write()`, `send()`, and `recv()`.

This strategy corrupts execution of some scripts, and it can frustrate attempts to read or exfiltrate sensitive data.

Strategy 3: Add more wait time. The goal of this strategy is to slow down a questionable process, such as rate-limiting network attacks. We randomly increase the time a `nanosleep()` call yields the CPU.

Strategy 4: Change file offset. This approach simulates file corruption by randomly changing the offset in a file descriptor between `read()`s and `write()`s.

We first applied unpredictability to the Linux Keylogger (LKL, http://sourceforge.net/projects/lkl/), a user-space keylogger, using strategies 1, 2, and 4. The keylogger not only lost valid keystrokes but also had some noise data added to the log file.

Next we applied unpredictability to the BotNET (http://sourceforge.net/projects/botnet/) malware, which is mainly a communication library for the IRC protocol that was refined to add spam and SYN-flood capabilities. We used the IRC client platform `irssi` to configure the botnet architecture with a bot herder, bots, and victims. The unpredictable strategies were applied to one of the bots.

We first tested commands that successfully reached the bot, such as `adduser`, `deluser`, `list`, `access`, `memo`, `sendmail`, and `part`. The bot reads commands one byte at a time, and one lost byte will cause a command to fail. Randomly silencing a subset of `read()` system calls in our unpredictable environment results in losing 40% of the commands from the bot herder.

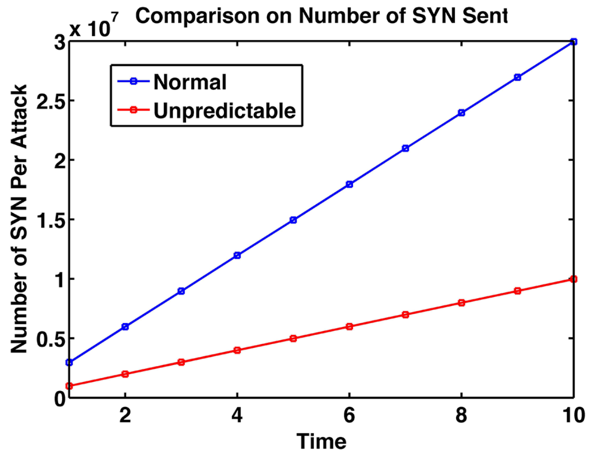


Figure 2: Comparison of SYN-flood attacks in normal and unpredictable environments. Unpredictability can increase the DDoS resource requirements.

We measured the impact of the unpredictable environment on the ability of the bot to send spam emails, shown in Figure 1. In the normal environment, nine emails varying in length from 10 to 90 bytes were successfully sent. In the unpredictable environment, only partial random bytes were sent out by arbitrarily reducing the buffer size of `send()` in the bot process. In the case of a spam bot, truncated emails will streamline the filtering process, not only for automatic filters, but also for the end users.

We also performed a SYN-flood attack to analyze the effectiveness of the unpredictable environment in mitigating DDoS attacks. In a standard environment, one client can bring down a server in one minute with SYN packets. When we set the unpredictability threshold to 70% and applied strategies 1 and 3, the rate of SYN packets arriving at the victim server decreased (Figure 2), requiring two additional bots to achieve the same outcome.

Preliminary tests with Thunderbird, Firefox, and Skype running in the unpredictable environment showed that these applications can run normally most of the time, occasionally showing warnings, and with some functionality temporarily unavailable.

A challenge is to dial this behavior in to minimize harm to benign, but not whitelisted, applications while frustrating potentially malicious code.

Spectrum-Behavior OS

We are building Chameleon, an operating system combining inconsistent and consistent deception with software diversity for active defense of computer systems and herd protection. Chameleon provides three distinct environments for process execution (Figure 3): (1) a diverse environment for whitelisted processes, (2) an unpredictable environment for unknown or suspicious processes (inconsistent deception), and (3) a consistently deceptive environment for malicious processes. Our

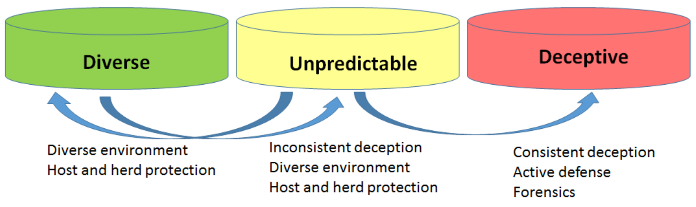


Figure 3: Chameleon can transition processes among three operating modes: *diverse*, to protect benign software; *unpredictable*, to disturb unknown software; and *deceptive*, to analyze likely malware.

HotOS '15 paper [7] provides a longer discussion of these issues, as well as a more extensive discussion of prior work on unpredictability and deception as tools for system security.

Known benign or whitelisted processes run in the *diverse* operating system environment, where the implementation of the program APIs are randomized to reduce instances with the same combinations of vulnerable code. In some sense, the diverse environment combines ASLR and other known randomization techniques with N-version programming [8], except that Chameleon doesn't run the versions in parallel but, rather, diversifies them across processes. Our insight is that a modular library OS design makes the effort of manual diversification more tractable. Rather than require multiple complete OS implementations, the Chameleon design modularizes the Graphene library OS [4], and components are reimplemented at finer granularity and possibly in higher-productivity languages. The power of this design is that mixing and matching pieces of N implementations multiplies the diversity by the granularity of the pieces.

Unknown processes run in the *unpredictable* environment, where a subset of the system calls are modified or silenced. Unpredictability is primarily implemented at the system call table or library OS platform abstraction layer. The execution of processes in this environment is unpredictable as they can lose some I/O data and functionality.

A malicious process in the unpredictable environment will have difficulty accomplishing its tasks, as some system call options used to exploit OS vulnerabilities might not be available, some sensitive data being collected from and transferred to the system might get lost, and network connectivity with remote malicious hosts is not guaranteed.

Unpredictability raises the bar for large-scale attacks. An attacker might notice the hostile environment, but its unpredictable nature will leave her with few options, one of them being system exit, which from the host perspective is a winning outcome.

Processes identified as malicious run in a *deceptive* environment, where a subset of the system calls are modified to deceive an adversary with a consistent but false appearance, while forensic data is collected and forwarded to response teams such as CERT. This environment will be sandboxed, files will be honeyfiles, and external connections will be intercepted and logged.

Chameleon can adjust its behavior over the lifetime of a process. Its design includes a dynamic, machine-learning-based process categorization module that observes behavior of unknown processes, and compares them to training sets of known good and malicious code. Based on its behavior, a process can migrate across environments.

What About the Computer User?

Sacrificing predictability will introduce new, but tractable, research questions, especially around usability. For example, a user who installs a new game with a potential Trojan horse will be tempted to simply whitelist the game if it isn't playable. We believe unpredictability can be adjusted dynamically to avoid interfering with desirable behavior, potentially with user feedback.

We envision Chameleon's architecture adopted in desktop computers for end users. This will allow a common group of whitelisted applications such as browsers or office software to run unperturbed and a suspicious application to be quarantined by Chameleon.

For example, consider Bob, 72, living in a retirement community in Florida. Bob is not computer savvy and tends to click links from spear-phishing emails, which might install malware in his computer. This malware will engage in later attacks compromising other machines and performing DoS attacks in critical infrastructure. Bob never notices malware running in his computer because the malware becomes active only after 1 a.m.

With Chameleon, Bob continues to browse for news, work on documents from his community homeowner association, or Skype with family without problems; these applications are whitelisted, running in the diverse environment. The diverse environment protects whitelisted applications by reducing the likelihood of their being exploited. Further, if Bob downloads a game that also includes a botnet, the unpredictable environment may cause the game to seem poorly designed, the visual images showing some glitches here and there, but Bob's credentials will be safe. Further, the botnet, which Bob will never notice, will fail to operate as the attacker wishes.

Part of the evaluation of Chameleon's success or failure will include usability studies. Our hypothesis is that Chameleon can strike a long-sought balance that preserves usability for desirable uses but thwarts significantly more compromises without frustrating users to the point of disabling the security measure.

Conclusions

Today's systems are designed to be predictable, and this predictability benefits attackers more than software developers or cybersecurity defenders. This leads us to have the worst of both worlds: rather simple attacks work, and both research and industry are moving towards models of mutual distrust between applications and the operating system [9, 10].

If applications will trust the operating system less in the future, why not leverage this as a way to make malware and attacks harder to write? If successful, sacrificing predictable behavior can finally give systems an edge over one of the primary sources of computer compromises: malware installed by unwitting users.

Acknowledgments

We thank the anonymous HotOS reviewers, Nick Nikiforakis, Michalis Polychronakis, and Chia-Che Tsai for insightful comments on earlier drafts of this paper.

This research is supported in part by NSF grants CNS-1149730, SES-1450624, CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, CNS-1408695. It is also supported by grants OCI-1246061 and DUE-1344369.

References

[1] S. Forrest, A. Somayaji, and D. Ackley, "Building Diverse Computer Systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS VI),* 1997.

[2] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking Blind," *in 2014 IEEE Symposium on Security and Privacy (SP),* May 2014, pp. 227–242.

[3] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System Is the Control Plane," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '14),* 2014, pp. 1–16.

[4] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-Process Applications," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys),* 2014, pp. 9:1–9:14.

[5] V. Neagoe and M. Bishop, "Inconsistency in Deception for Defense," in *New Security Paradigms Workshop (NSPW),* 2007, pp. 31–38.

[6] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API Is a Bad Untrusted RPC Interface," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2013, pp. 253–264.

[7] R. Sun, D. E. Porter, D. Oliveira, and M. Bishop, "The Case for Less Predictable Operating System Behavior," in *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS XV),* 2015.

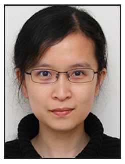
[8] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Digest of the Eighth Annual International Symposium on Fault-Tolerant Computing,* 1978, pp. 3–9.

[9] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Workshop of Hardware and Architectural Support for Security and Privacy (HASP),* 2013.

[10] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14),* 2014, pp. 267–283.

Privacy-Preserving Experimentation with Sensibility Testbed

YANYAN ZHUANG, ALBERT RAFETSEDER, AND JUSTIN CAPPOS


Yanyan Zhuang is a Research Professor at the Computer Science and Engineering Department at New York University. Her current research interests include fault diagnosis of distributed systems and the design and implementation of network testbeds for mobile devices. yyzh@nyu.edu

Albert Rafetseder is a Research Professor at the Computer Science and Engineering Department at New York University. He is the current technical lead for the Seattle Testbed project and takes an interest in large-scale application-layer network measurements. albert.rafetseder@univie.ac.at

Justin Cappos is an Assistant Professor in the Computer Science and Engineering Department at New York University. Justin's research philosophy focuses on improving real-world systems, often by addressing issues that arise in practical deployments. jcappos@nyu.edu

Recent privacy breaches and security break-ins of mobile systems have raised concerns about using mobile devices like smartphones and tablets [1]. As a result, many users are aware that running apps on their smartphones can increase privacy risks. On the other hand, the data from the enormous number of smartphones, if used properly, can be of tremendous value to the research community. Is there a way to safely do research on these devices without rendering them vulnerable? We explain about how our project may help both researchers and volunteers.

Ever wondered what science could achieve if any researcher can get data from other people's smartphones? Imagine that we would simply write a few lines of code, fire it up on a number of strangers' phones, and within minutes we would know where the dead spots of our mobile data plans are. We could also have a zero-cost navigation system when no GPS or any other location services are available; we could achieve this by establishing a Bluetooth connection with a neighboring device and get the location data from it. If we constantly monitor accelerometer data on mobile devices, we can detect vibrations within the frequency and intensity range of seismic waves, and assist distributed earthquake detection. These all sound fantastic, except that who would let us get data off their devices? Our friends and family would probably trust us. Other people? Not so much.

The privacy and security challenges on mobile devices have increased dramatically over the years. Although having apps post tweets to a user's Twitter account without asking for permission is seriously off-putting [3], hacked apps that let criminals break into an individual's bank account are clearly detrimental [4]. Running code to collect data from smartphones is much more complex than it sounds. We not only need to ensure the security of a device so that the code that does the data collection cannot damage or hack into the device, but we also must protect the privacy of a device owner so that the code cannot eavesdrop on phone conversations, steal passwords, etc.

We introduce Sensibility Testbed [5], a smartphone testbed that allows researchers to run code and perform measurements on others' smartphones for research purposes. It ensures the security of user-owned devices and the privacy of user-generated data. The usage model of Sensibility Testbed is unique in that it manages how device owners make their devices accessible to different research communities without putting their devices at risk. Meanwhile, it offers technical resources that allow researchers to collect data from remote mobile devices without impairing the device owner's privacy. As an added bonus, different research groups can pick, choose, share, and reuse each others' user base.

Yet Another Testbed?

While the rich set of sensors on mobile devices can provide useful data sets for research, today's security and privacy issues have created many obstacles to collecting data and sharing them among mobile devices. Note that in this work, sensors are broadly defined as the hardware components that can record phenomena about the physical world, such as WiFi/

cellular network, GPS location, movement acceleration, etc. The challenges of collecting sensor data are twofold: first, sensor data from mobile devices can reveal device owners' personal information and result in privacy breaches; second, potential bugs, sometimes inadvertent ones, in a research experiment can damage end users' personal devices and cause security issues. Collecting data using untrusted programs poses significant challenges for both device owners and bystanders. To run code safely on a stranger's smartphone without revealing this stranger's privacy sounds like a fantasy, or at least, mission impossible.

You are probably wondering, aren't there plenty of network testbeds out there, and don't they already resolve these issues? The problem with existing testbeds is that they do not yet have a systematic way to protect device owners' security and privacy. To lower potential risks, many smartphone-related testbeds choose to recruit participants from a trusted group, such as students, colleagues, and friends. For example, PhoneLab (https://www.phone-lab.org/) provides a platform for people to run Android apps on their participants' smartphones and log data. PhoneLab recruits participants by giving them an Android device and data plan for free, in exchange for a commitment to use the phone as their personal device for six months or longer. This approach cannot solve the privacy issue. With such a usage model, the researchers from different research groups are not able to test their hypothesis at a world-wide scale or reuse each others' user base. For example, a researcher who uses PhoneLab at the University of Buffalo cannot share the same user base with Community Seismic Network [6] at Caltech, and vice versa.

Sensibility Testbed is different in several aspects. First, in our testbed, device owners participate as volunteers, and researchers request these devices through our server. This server, which is called a clearinghouse, mediates remote device access but does not store any personal data. As a result, Sensibility Testbed relieves researchers from recruiting participants and allows different groups to share all the devices used in the testbed. Additionally, Sensibility Testbed does not require researchers to write full-fledged Android apps to perform experiments. Instead, it provides an easy-to-use Python-like programming interface. Last but not least, using Sensibility Testbed, device owners do not need to trust the researchers who run code on their devices. As you will see later in this article, Sensibility Testbed provides a secure, sandboxed environment for anyone to run experiment code on Android devices. This can effectively prevent potential security and privacy breaches.

Yet Another Testbed!

The Sandbox

Researchers run experiments in Sensibility Testbed by writing code for a restricted, Python-based sandbox. This is the same security-reviewed Repy (Restricted Python) sandbox [2] used

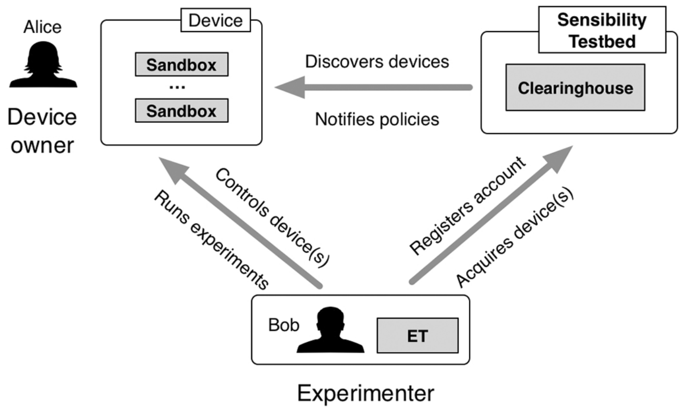


Figure 1: Sensibility Testbed architecture

in the Seattle testbed (https://seattle.poly.edu/). This sandbox has been deployed on the Seattle testbed over the last six years. Our experience has shown that the risk of it being faulty is very low. The Repy sandbox is restricted in that its API limits what a sandboxed program can do: reading from and writing to the file system can only occur in a per-experiment directory; sending and receiving data via the network interface cannot exceed a configured rate; CPU, memory, and battery consumption cannot exceed a limit, etc. Therefore, the sandbox isolates the program from the rest of the device. More importantly, the sandbox allows us to interject code to implement privacy policies and control what happens with the data gathered on the device. You will read more on how to add privacy policies a bit later.

Interacting Parties

In Sensibility Testbed, there are three categories of interacting parties: mobile devices owned by ordinary people, with our app installed; a clearinghouse server that discovers and configures participating devices; and researchers wanting to run experiments on mobile devices (see Figure 1). These three parties interact as follows. Mobile devices provide resources and data for researchers to use in their experiments. As mentioned above, researchers' code runs in a sandbox on a remote device that isolates the code from the rest of the device host system. Meanwhile, our clearinghouse server helps researchers acquire and manage devices, and enforces policies specified by the researcher's institutional review board (IRB), thus protecting device owners' personal information. Finally, researchers use their local machines to initiate and control experiments in Sensibility Testbed. They use an experiment tool (ET) to deploy and run experiments in sandboxes on remote devices that are acquired through the clearinghouse.

How Does It Work?

To get a sense of the technical details, let's walk through two scenarios: (1) a smartphone owner, Alice, participates in the testbed, and (2) a researcher, Bob, runs code on Sensibility Testbed using Alice's smartphone, among other devices. Specifically, Bob wants to know the cellular service quality in major cities. As such, he needs location information of individual devices, their cellular service provider, network type (3G, 4G, LTE, etc.), and signal strength. Note that the exact nature of Bob's experiment, be it collecting data, performing computation, etc., is not critical at this point due to code containment by our sandbox.

When Alice decides to participate in Sensibility Testbed, she first goes to the Google Play Store to download our Sensibility Testbed app [7]. The app contains sandboxes for researchers to run experiments on Alice's device, and a user interface for Alice to start and stop the app. When the app is started, Alice's device can be discovered by the clearinghouse. To keep track of Alice's device, the clearinghouse uses a database that stores her device's unique public cryptographic key that is generated during installation. This key is not associated with Alice's or her device's identity, but only the installation on the device. If Alice ever uninstalls the Sensibility app, this key is deleted, which effectively "unlinks" her device from any metadata stored on the clearinghouse. Instead of uninstalling, Alice may also choose to opt out of individual experiments.

To run code on Sensibility Testbed, Bob provides a detailed experiment description to our clearinghouse. Before Bob can request a device, his experiment needs to be approved for human-subjects compliance by his IRB (or equivalent). The IRB at Bob's institution specifies what data can be accessed by a research experiment, at which granularity or frequency of such data can be accessed, and so on. For example, Bob's experiment can (1) read location information from devices at the granularity of a city; (2) read accurate cellular signal strength and network type, but no information about cell IDs should be accessed; and (3) get location and cellular network data updates every ten minutes. Bob submits an appropriate experiment description for these requirements, which the clearinghouse codifies into policies that are later enforced on remote mobile devices.

Note that Bob cannot request access to all sensors at any rate even if his IRB approves such a policy. The Sensibility Testbed's IRB allows access to sensors in a way that is low risk, whose access can be pre-approved with the researcher's local IRB. However, we do not provide unfettered access to all sensors. Access to sensors of higher risk needs to go through the Sensibility Testbed's IRB, in addition to the researcher's IRB. However, for most cases, we expect that researchers need only go through their local IRB to get the sensor access they need for their experiments.

Bob next obtains an experiment account and requests a number of devices from our clearinghouse. The clearinghouse looks up available devices, finds Alice's phone is available (among others), assigns it to Bob's experiment account, and instructs the sandbox on her device to apply data access policies for Bob's experiment: for policy 1 above, the sandbox blurs the location information returned from Alice's phone down to the coordinates of the nearest city; for policy 2, the sandbox blocks the access to cell IDs; for policy 3, the sandbox limits the rate of GPS location and cellular network queries from Bob's experiment to one every ten minutes. Bob then uses the experiment tool (ET) on his local computer to access Alice's device and do experiments. After collecting the data he needs, Bob can either use ET to download data from the remote devices from time to time, set up his own server to store all the data, or use a data store service we provide (Sensevis: https://sensibilitytestbed.github.io/sensevis/).

If Bob stores data at his own server, he must use protective measures to ensure that the data sent from the mobile devices is properly encrypted and that the server storage cannot be tampered with by any other parties. For example, Bob needs to register his server by providing the server's certificate and URL to our clearinghouse. The clearinghouse then instructs the devices accessible to Bob that all the sensor data collected should be sent to this server. The sandboxes on these devices then issue HTTPS POST using the server's certificate, and send encrypted data to Bob's server. After the data is collected, how to store the data securely is mandated by Bob's IRB.

Implementing Policies

To understand how policies are implemented, we need to start with the Sensibility Testbed app. The app on Alice's device contains a native Android portion, and our Repy sandbox. When Alice starts the app, the native code initializes a Python interpreter, launches the Repy sandbox, and starts the communication between the device and our clearinghouse. The sandbox's restricted, secure API provides calls to file system, networking, threading functions, and so on. Therefore, Bob's code can read files, send data through the network, etc. from Alice's device. However, the original Repy sandbox does not include calls specific to mobile devices, such as GPS location, WiFi network, Bluetooth, accelerometer, cellular network, etc.

To obtain smartphone-specific data, we first implemented our sensor API using native code in the Sensibility Testbed Android app. The Repy sandbox then uses RPC to invoke the corresponding Android code, and returns the data from native code to a sandboxed program. The Repy sandbox thus defines the sensor API as a set of higher level calls, such as `get_location()`, `get_wifi()`, `get_accelerometer()`, and so on. Our Wiki page [8] hosts the current ever-growing list. As such, the original Repy interface and the added sensor API together provide the complete "OS level" sandbox kernel on a mobile device, as shown in Figure 2.

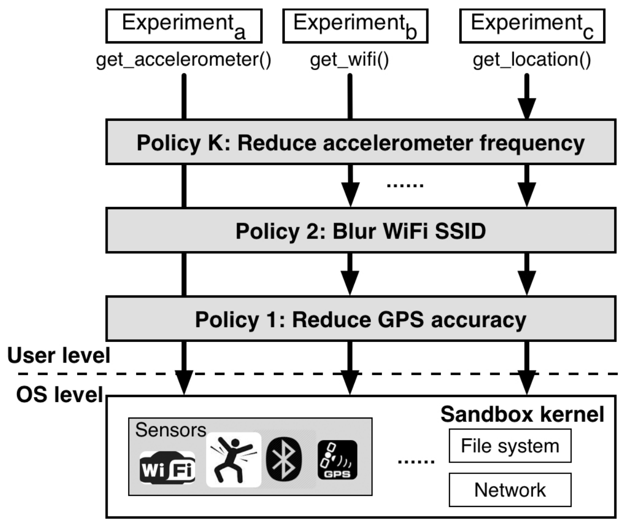


Figure 2: Sensibility Testbed blur policies

Finally, this sandbox kernel determines how policies are implemented by affecting API calls. It can interpose on a call and modify the data returned, or control how frequently a call can be made over time. As mentioned above, Bob provided his IRB policies through our clearinghouse. So before Bob runs his experiment, the clearinghouse instructs the sandbox on Alice's device to restrict sensor access in accordance with these IRB policies. Using the `get_location()` call as an example, when Bob's code requests location data from Alice's device, the Repy sandbox first invokes the location-related Android code. As the location data is returned, Bob's IRB policy indicates that the returned location coordinates should be blurred to the nearest city to Alice's device, instead of her actual location. As a result, the sandbox returns an approximate location to Bob's program. Furthermore, as Bob's IRB policy disallows collecting information about cell tower IDs, the access to cell IDs is blocked entirely on Alice's device. Similarly, other information like WiFi SSID can be blurred to a hashed string, the frequency to access an accelerometer can be restricted to prevent inferring passwords from the movement and tilt of the device, and so on. As shown in Figure 2, different policies can be stacked together as a set of filters for different sensors before a sandboxed program can access the sensor data.

Testbed Status

At this stage, multiple groups have experimented with Sensibility Testbed on their local phones, while we finalize outside use via our internal IRB and clearinghouse. We have also hosted two successful hack-a-thon-styled workshops with IEEE Sensors Applications Symposium [9] in 2014 and 2015. At these events, about two dozen participants from diverse universities and backgrounds worked in teams to build an application of their choice. Despite having no background in the platform and only a few hours to work, the participants in seven teams built applications that they demoed at the conference. The applications they developed varied from building automation using Bluetooth, to auto-device power saving that shuts down unnecessary network interfaces.

Sensibility Testbed provides a focal point for smartphone-based research. We believe this will bring benefit to researchers, as we make their experiment prototype faster, the remote control and management of devices easier, and running experiment code more secure. This will also benefit device owners in the long run, as researchers identify opportunities for improving current network protocols or systems, and implement or evaluate new services, algorithms, and research ideas. We believe Sensibility Testbed will bring more opportunities to research and encourage innovation from the general research community.

Sensibility Testbed is an open source research project, and we invite you to participate too! All of our source code, including the Android app, sandbox, clearinghouse, the experiment tool, etc., can be found on GitHub [10]. By installing the Sensibility Testbed application, you can become an important part of research discoveries that benefit science and technology.

References

[1] "A Warning About Those Free Smartphone Apps": http://abcnews.go.com/Technology/warning-free-smartphone-apps/story?id=30484903.

[2] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, T. Anderson, "Retaining Sandbox Containment Despite Bugs in Privileged Memory-Safe Code," *ACM Conference on Computer and Communications Security (CCS '10).*

[3] "App Auto-Tweets False Piracy Accusations": http://yro.slashdot.org/story/12/11/13/2249203/app-auto-tweets-false-piracy-accusations.

[4] "Hackers Are Draining Bank Accounts via the Starbucks App": http://money.cnn.com/2015/05/13/technology/hackers-starbucks-app/index.html.

[5] Sensibility Testbed: https://sensibilitytestbed.com/.

[6] Community Seismic Network: http://csn.caltech.edu/.

[7] Google Play Store, Sensibility Testbed: https://play.google.com/store/apps/details?id=com.sensibility_testbed.

[8] Sensibility Testbed, Using Sensors: https://sensibilitytestbed.com/projects/project/wiki/sensors.

[9] IEEE, Sensors Applications Symposium: http://sensorapps.org/.

[10] GitHub, Sensibility Testbed: https://github.com/SensibilityTestbed.

Interview with Marc Maiffret

RIK FARROW

After being raided by the FBI at age 17, Marc Maiffret started his first security software company, eEye Digital Security, pioneering early research into critical Microsoft vulnerabilities. As an entrepreneur, Maiffret created one of the first vulnerability management products as well as one of the first Web application firewall products—both of which have been deployed worldwide and won numerous awards. Maiffret has also been a leader in next-generation malware prevention while serving as Chief Security Architect for FireEye. Maiffret served as Chief Technology Officer of the privilege and vulnerability management firm BeyondTrust after its acquisition of eEye Digital Security. In 2015, Maiffret left BeyondTrust to pursue a new but unannounced venture. Maiffret has testified before Congress, published an op-ed in the *New York Times,* and is an avid speaker and advocate for improving security.

Quite appropriately, I first "met" Marc Maiffret online. We were both participants in a security mailing list, and I was struck by Marc's youthful exuberance. I could tell that Marc was on a mission, and that mission appeared to be to embarrass Microsoft into improving its security practices.

What I didn't know at first was Marc started his first business at 17. I'd certainly noticed the rough edges in his online postings but had little idea just how young he was or how he had become an expert in Windows security through self-education and experimentation.

Over 17 years later, I decided to ask Marc more about what he had been doing before we met, his part in some security drama (Code Red), and his various business adventures. I also wanted to get Marc's impression of the state of Windows security today.

Rik: When did you start learning about computers?

Marc: My path to learning about computers really started first with an interest in phone phreaking. I had a friend who introduced me to the world of phone phreaking in 6th or 7th grade. I always had a curiosity about how different things worked, and the telephone system seemed like this infinite world to explore and learn from. Wanting to learn more about phone phreaking led to needing to get on BBS systems, and that was my gateway to eventually getting more into computers, hacking, Internet, etc.

We didn't have enough money for a computer at home, so in the beginning I learned what I could from computers at the school library or the office where my mom worked; the owner was nice enough to let me use a system sometimes after school. That same business owner eventually gave me an old computer from their office, and that is when things really started to move quickly for me in learning about early hacking and related topics.

I had a turbulent home life growing up that can be summarized by my deciding to run away from home for almost a year when I was 16, moving entirely across the U.S. to live with friends in the hacking and research group Rhino9, my stepdad eventually dying from a drug overdose, and so on. Not to understate it all, but it was a variety of things, plus my natural curiosity about how things worked, that drove me deeply into learning as much as I could about computers as a means of escaping my then reality.

Rik: You started a business, eEye, back in 1998. What led you to develop software to help secure Windows systems, back when Windows was really awful?

Marc: When I got back home after running away from California to Florida, and a few places in between, I was 17. I did not want to go back to high school and wanted to start working in computers. My mother gave me three months to find a job and support myself or it was back to school. Within a few weeks I had my first job, and then a couple of months later all the hacking I had been doing over the previous few years caught up to me when the FBI raided my family's home. This was a great wakeup call for me to try to figure out what I was going to do with my life. At the time, I had been writing a lot of free security tools for Windows and researching various software vulnerabilities. This led to the creation of my first company, eEye Digital Security, and the vulnerability assessment product Retina. This was all around the 1998–1999 time frame.

Within a few short years, eEye and Retina had a lot of business success. More importantly, though, we were a part of pioneering a lot of the early research into Microsoft-related vulnerabilities. We also were pushing aggressively for companies like Microsoft to treat security as a technical problem instead of a marketing one.

People coming into IT security today would not recognize the Microsoft of the early 2000s. At eEye, we did not just find numerous critical vulnerabilities within Microsoft software, but rather exerted as much pressure as possible to get them to change their culture and behavior, making it as painful as possible for them while we helped to get vulnerabilities fixed to protect customers. This was a sometimes difficult balancing act which led to fun encounters, like the then head of Microsoft security response calling on the phone to curse me out. There were many people doing great security research back then, and all of this led to Microsoft evolving in positive ways. eEye had a very special role in that process, not just through research but also by having customers we could help leverage to put pressure on Microsoft.

For example, a large reason why Patch Tuesday was created was because of customers being outraged and exhausted by having to deploy critical patches for remote system vulnerabilities one after another on a random basis. A lot of people do not know that behind-the-scenes we were doing things like accumulating critical vulnerabilities that we would report to Microsoft one at a time. As soon as they patched one, within hours we would send them another, and another, to keep pressure on until something broke their poor software development behaviors.

That something eventually came in the form of Bill Gates' Trustworthy Computing memo, in large part spawned by the efforts of eEye and many others and, of course, the fallout from things like Code Red and other widespread malware/worm attacks at the time.

Rik: Tell us about Code Red.

Marc: Code Red was a Microsoft computer worm discovered by Ryan Permeh and me while we were at eEye. Code Red leveraged a vulnerability within Microsoft's IIS Web server that Riley Hassell, also at eEye, had discovered. Ryan and I were actually hanging out on a Friday after work drinking beers at his place when an IT guy emailed mentioning that their IIS Web server was acting weird, connecting to other systems. Now this is in 2001, a very different world in IT and security. You can actually find archives on IT mailing lists where people were experiencing IIS Web server crashes for a good week or two before Ryan and I made this discovery. After getting a packet capture of some IIS traffic, we started looking to see what might be going on and determined, in fact, that someone had developed a worm to automatically propagate to IIS Web servers via the vulnerability Riley had discovered.

Ryan and I worked over the weekend to write up a technical analysis of the worm and eventually posted our analysis online late Sunday or early Monday morning. We didn't think much of it at the time as Code Red was one of the first of its kind. By Monday afternoon, the whole thing had taken on a life of its own, and by the end of the week we had done everything from talked to folks in the White House situation room to the head of marketing for Pepsi, the company behind Code Red Mountain Dew, which we had named the worm after. While the worm was actually easy to manage in the end, due to its propagation method, it affected a lot of systems and was certainly a wakeup call for Microsoft.

Rik: What did you do after you left eEye?

Marc: eEye was always more than simply a business to me and to a lot of the employees there, particularly those working in research: guys like Ryan Permeh, Barnaby Jack, Yuji Ukai, Derek Soeder, Riley Hassell and too many others to list. We wanted to make a great product in Retina, but also we were a part of the early days of the security industry and were hackers trying to find our place in this world. Living in Southern California, I find conversations with old skateboarders who rode the wave of evolution from their hobby to a business to be more relatable in understanding just how special what we were all a part of was, as opposed to some person who is new to IT security these days.

When people have been in this industry all of five minutes, it is easy to think security is terrible and hasn't made much progress, when in reality a great deal of progress has been made. When I catch up with my old colleagues and we reflect on the wild ride we were a part of, it is not so much about what place eEye holds in that history but rather about hoping people understand that the evolution in security and improvements in companies like Microsoft has not happened naturally; instead they've happened because a research community was willing to fight and hold technology companies accountable. This is something often forgotten today as we focus as an industry solely on hackers and adversaries, on countries and cybercriminals but rarely on the vulnerable technology that allows such attackers to break into systems in the first place. This is something I expanded on further in a *New York Times* op-ed a couple of years ago (http://www.nytimes.com/2013/04/05/opinion/closing-the-door-on-hackers.html).

After leaving eEye, I took some time off to take a break and hang out. I had been hacking and working in security since I was a teenager, and eEye was the only job I had ever had. After a short time helping run a managed security company, I went to work for a less well known company, at the time, that also liked eye-related company names—FireEye. At FireEye, I reported to the then CEO and Founder Ashar Aziz as Chief Security Architect. They were not then the behemoth that they are now, and I was lucky to be a part of helping innovate their product in its malware detection capabilities and amplify the great stuff they were doing in those early days. Bringing my background in vulnerabilities and exploits to the malware world helped increase their systems' ability to generically detect malware and compromises within corporate networks. It was an amazing team and experience to have been a part of.

Rik: How has the Windows security landscape changed from your perspective?

Marc: The Windows security landscape has changed dramatically as it pertains to Microsoft software. The reality is that Microsoft has made amazing strides to improve the security of their code and systems and continues to do so. Clearly, many vulnerabilities remain, but Microsoft has consistently done things to raise the bar on attackers and researchers alike. There are too many examples of positive changes they have done to list them all.

Probably the biggest area of improvement is not just in their internal security efforts to eradicate bugs but in their efforts to continue to make the exploitation of vulnerabilities that much harder. This even goes to the point of their offering $100,000 bounties on novel ways to bypass their various mitigation technologies. This is a wildly different Microsoft than the one I knew many years ago. There are, of course, a lot of technical examples of how they have improved security and their architectures over the years, but more than hoping for one individual safeguard, I think the biggest improvement is yet to come in Windows 10 because of changes to the overall ecosystem.

Microsoft has realized that no matter how secure they make their own code they will still get a bad rap so long as their ecosystem of third-party developers and software remains insecure. In a lot of ways, most security products have existed as bolt-ons to harden operating systems and to more tightly control application behavior in ways operating systems should be doing by default: the age-old problems of separating code and data, users and access, and so on. Where Microsoft and even Apple seem to be moving in terms of the desktop OS ecosystem is to mirror what has happened in the mobile OS space with much tighter control of what applications can do, how they inter-operate, how they are sandboxed, and so on.

Microsoft already started down this path with Windows 8's app store but failed to get developers to adopt their new model because it would require whole code rewrites in a lot of cases, not to mention generally failing to get companies to even migrate from Windows 7 to 8. Microsoft seems better positioned to successfully get people to adopt Windows 10, and it seems to be doing everything possible to get developers on board with getting their apps moved to the app store model, including going to great lengths to allow for classic Win32 applications to be packaged up as store applications; this is done through leveraging some level of virtualization and sandboxing so as not to violate the overall benefit of store-based applications. This has interesting implications for the desktop security landscape because store/mobile OS models more granularly control and sandbox applications in ways that can be very beneficial for security and even IT management.

You can think in terms of whether you would trust a family member to be safer online via an iPad or Windows 7; which are they most likely to get hacked on? Now this is not some religious debate about what is the better OS or technology company, or which has more or fewer vulnerabilities; rather, it's a question of OS and application models that are very different in mobile OSes vs. traditional desktops. While exploits can and do exist for both models, there is a dramatic difference in attack surface and how tightly controlled applications and code are. I could expand on this a lot more, but hopefully the implications of and differences in these models are obvious as to the benefits to security if Microsoft can successfully win over developers to this new model.

To be clear, I'm not suggesting that such an app store model will magically make Windows 10 secure out of the gate. It is not that Windows 10 only allows a mobile OS app store type model but rather that it is a hybrid, as Windows 8 was, of both a traditional desktop OS app model and an app store model. If Microsoft can successfully bring developers and their apps over to the store model, then it moves us closer to being able to hit the kill switch on the traditional desktop OS app model and all the attack surface that comes with it. And, of course, there will be plenty of problems with the store model from a security perspective; expect to see someone talking about win32 apps escaping the Windows 10 Store app sandbox at a future security conference. But these problems will be far better than the current state of the Windows desktop security model, where companies struggle with a bunch of bolt-on security software simply to make sure their users are not running malicious code from Web browsers, email, and so on.

Rik: What do you think of open disclosure currently?

Marc: When discussing vulnerability disclosure, full disclosure, and related topics, it is important to understand security research in the larger context of where we currently find ourselves in the continuum of such research. Vulnerability research in the early 2000s was being done more out in the open where everyone could benefit from it.

Things have changed over time where the value of such research has increased well beyond the primary value in the 2000s of simply making a name for yourself or for a security company in order to get some press. As such the trend has been that more critical vulnerability research is happening much more often behind closed doors to the benefit of a few. It is also important to think about the increased impact a vulnerability can have now vs. years ago as society grows more dependent on technology.

So with that context in mind, I can see validity in the arguments from all sides. I understand why a researcher would rather sell a vulnerability to a defense contractor or private party than deal with the sometimes truly painful process of trying to report a vulnerability "responsibly" to a technology company, all for the reward of a thank you in a security bulletin or possibly a payment that is a fraction of what they would have made by selling it privately. I can also understand a researcher who thinks selling a vulnerability to a defense contractor is morally wrong but equally hates dealing with vendors and simply wants to drop the information online for the community to sort out.

And I can see why plenty of people would be upset at researchers who seemingly claim to do their work for the benefit of everyone but are inflexible in their own views and timelines of what a vendor might require to fix a flaw. I think this debate has persisted the last 17+ years I have been in security because there truly is no right answer or magical governing principle applicable to just vulnerability research. I think the only thing that can be said for certain is that regardless of your opinion on such debates, the debates would not be happening if the information were not public in some form. Wait, was this question about Snowden? :-)

Thanks to Our USENIX and LISA SIG Supporters

USENIX Patrons
Facebook Google Microsoft Research NetApp VMware

USENIX Benefactors
Hewlett-Packard *Linux Pro Magazine* Symantec

USENIX and LISA SIG Partners
Booking.com Cambridge Computer
Can Stock Photo Fotosearch Google

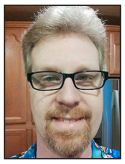
USENIX Partners
Cisco-Meraki EMC Huawei

Open Access Publishing Partner
PeerJ

(Un)Reliability Budgets

Finding Balance between Innovation and Reliability

MARK D. ROTH



Mark Roth has been a Site Reliability Engineer at Google's Mountain View office for over a decade. He has worked on a variety of projects, including Gmail, Google Accounts, Monitoring Infrastructure, and Compute Resource Management. Before coming to Google, he managed production UNIX systems at the University of Illinois at Urbana-Champaign, where he authored a number of open-source software packages. roth@google.com

Google is constantly changing our software to implement new, useful features for our users. Unfortunately, making changes is inherently risky. Google services are quite complex, and any new feature might accidentally cause problems for users. In fact, most outages of Google services are the result of deploying a change. As a consequence, there is an inherent tension between the desire to innovate quickly and to keep the site reliable. Google manages this tension by using a metrics-based approach called an unreliability budget, which provides an objective metric to guide decisions involving tradeoffs between innovation and reliability.

Structural Tension

The tension between innovation and change is reflected most strongly in the relationship between the SRE team and the corresponding Product Development team for any given application. This is partly due to the inherent conflict between the two teams' goals. Product Development's performance is largely evaluated based on product velocity, so they have incentive to get new code out as quickly as possible. However, SRE's performance is evaluated based on how reliable the service is, which means they are generally motivated to push back against a high rate of change. In addition, there is information asymmetry between the two teams. The product developers have more visibility into the time and effort involved in writing and releasing their code, while the SREs have more visibility into the service's reliability.

This inherent structural tension between Product Development and SRE manifests itself in disagreements in a number of areas where it is important to find the right balance between innovation and change. Here are some of the areas:

Software Fault Tolerance. When writing software, it's important to anticipate the possible failure modes and ensure that the software will handle them. However, there are an almost infinite number of ways in which software can fail, and product developers do not have an infinite amount of time to address those cases. Spending too little time on this results in brittle software, thus increasing outages; spending too much time on this means that it takes longer to finish the software, thus decreasing innovation. What is the right balance?

Testing. Too little testing results in bad, unreliable software. Too much testing can delay the software from ever being released and increase ongoing code maintenance costs due to the additional tests. Google product developers have many software testing tools at their disposal, but how much testing is enough?

Push Frequency. Some teams prefer to push a new software release monthly or weekly. Others would rather push daily or multiple times each day. Even if a push is mostly automated, it may still require work on the part of the SREs. Each push is risky. A bad push can result in a user-visible outage. Even without a user-visible outage, there may be a reduction in reliability during the push due to the fact that while some systems are upgraded, the others take on the additional load, possibly affecting latency. What's the best frequency for the application?

Canary Duration and Size. When pushing new software, most teams first push to a small subset of the total number of deployed instances, so that if there is a problem, it will only affect a subset of users. This is referred to as a "canary," named after the practice of using a canary to detect carbon monoxide in coal mines. Only after the code is deemed stable for some period of time in canary will it be pushed out to the rest of production. But how long should a change be canaried before it is deemed safe for the rest of production? Too little time and we risk not catching problems before they go to the rest of production; too much time and we decrease the rate at which changes can be deployed. Also, how large of a subset should the canary be? Too small and we risk not having a large enough sample size to catch problems before they go to production; too large and we risk any potential problems causing too large of an impact before they are caught. What is the right balance for the application?

Push Retry Methods. Sometimes a bad push is discovered and the service is reverted to the previous release. When this happens there is a temptation to make a quick fix and try again. Often these quick fixes are not as well tested, and the risk is increased. Alternatively, some groups prefer to wait for the next push cycle, whether weekly or daily. We find that both methods result in the same rate of new features making it into production, but the former method results in many more pushes and reverted bad pushes, which creates work and stress for the SREs. Is it better to fix something quickly or do a full suite of tests?

The two teams need to negotiate to find the right balance in these areas. However, we don't want this negotiation to be driven based on the negotiating skills of the engineers involved. We also don't want this to be decided by politics, personal feelings, or just plain hope. (Indeed, SRE's unofficial motto is "Hope is not a strategy.") Instead, we want an objective metric, agreed upon by both sides, that can be used to guide the negotiations in a reproducible way. Google is a data-driven company, and we want the decision to be based on hard data.

Unreliability Budgets

For these decisions to be made based on objective data, the two teams jointly define a quarterly unreliability budget based on the service's SLO (service level objective, or the goal of how reliable a service should be). The unreliability budget provides a clear, objective metric that determines how unreliable the service is allowed to be within a single quarter. This takes the politics out of the negotiation between the SREs and the product developers when deciding how much risk to allow.

The unreliability budget works as follows: Product Management sets a "Quarterly SLO goal," which sets an expectation of how much uptime the service should have. The actual uptime is measured by a neutral third party, our monitoring system. The difference between these two numbers is the "budget" of how much "unreliability" is remaining for the quarter. As long as the uptime measured is above the SLO, new releases can be pushed.

As a hypothetical example, let's imagine that a service's SLO is that it will successfully serve 99.999% of all queries. This means that the service's unreliability budget is that it can fail 0.001% of the time within a given quarter. So if a given problem causes us to fail 0.0002% of the expected queries for the quarter, we would consider that it used up 20% of the service's unreliability budget for the quarter. Once the unreliability budget for the quarter has been spent, no more changes will be deployed (other than critical bug fixes), since they could cause unreliability that the service can't afford.

The actual SLO for a given application may actually be a much more complicated calculation involving latency, data freshness, and other factors. In some cases, a successful push may reduce the SLO slightly even though no downtime is visible to the users. For example, while some servers are being upgraded, others take on the extra traffic, and thus latency may increase.

Benefits

The main benefit of an unreliability budget is that it provides a common incentive that allows both Product Development and SRE to focus on finding the right balance between innovation and reliability.

For example, if Product Development wants to skimp on testing or increase push velocity and SRE is resistant, the unreliability budget guides the decision. When the budget is big, the product developers can take more risks. When the budget is nearly drained, the product developers themselves will push for more testing or slower push velocity, because they don't want to risk using up the budget and stall their launch. In effect, the Product Development team becomes self-policing. They know the budget and can manage their own risk.

The unreliability budget also largely eliminates tension between Product Development and SRE, because SRE no longer needs to be in the position of making subjective judgment calls on individual push requests from product developers or adopting blanket and increasingly arbitrary rules such as "new releases are pushed if and only if Product Development wins a game of fizzbin when the moon is full" [1] in an attempt to prevent repetition of previously encountered outages. Instead, SRE just needs to measure and enforce the agreed upon unreliability budget. If they need to say no, they can point at an objective metric that Product Development has already agreed to and cannot argue with. Thus, instead of viewing SRE as an obstacle, the Product Development team partners closely with SRE on ensuring appropriate velocity/reliability tradeoffs.

What happens if a network outage or datacenter failure reduces the measured SLO? Yes, events like that consume the budget, too. As a result, the number of new pushes may be reduced for the remainder of the quarter. The entire team is okay with this because everyone shares the responsibility for uptime. No one person is to blame for such an incident. On the other hand, Google has mechanisms to "route around" such outages so they are invisible to our users. If such an event actually does affect the service, the team can focus on improving their use of the redundancy and failover mechanisms rather than waste time finger-pointing.

Finally, because the unreliability budget is defined in terms of the application's SLO, it also helps to highlight some of the costs of overly high reliability targets, in terms of both inflexibility and slow innovation. If the team is having trouble getting new features out, then they may elect to loosen the SLO (thus increasing the unreliability budget) in order to increase innovation. At Google, doing a little better than the SLO is good, but exceeding it greatly is not considered something to be proud of; instead, it is an indication that the team is not taking enough risks or the service is over-provisioned. Google encourages smart risk-taking to increase innovation, and the unreliability budget helps us make sure that we're doing that.

Conclusion

When two groups work as a team and share responsibility for the uptime of a service, it is important to have a neutral, non-political way to guide decisions of balance. Whether it is how much testing is enough, how often to push, or how to recover from failed pushes, these are not easy decisions to make. While product developers are under pressure to advance their products rapidly and SREs are always mindful of stability, the unreliability budget gives the team a neutral, non-political, and data-driven way to find balance in all these areas and more. The result is a team that works better together and more effectively.

Acknowledgments

Thank you to Tom Limoncelli, now at Stack Exchange, Inc., for contributing to an early draft, Dave O'Connor for his invaluable comments, and Carmela Quinito for editorial review of this article.

Reference

[1] Fizzbin: http://www.imdb.com/title/tt0708412/quotes.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Publish and Present Your Work at USENIX Conferences

The program committees of the following conferences are seeking submissions. CiteSeer ranks the USENIX Conference Proceedings among the top ten highest-impact publication venues for computer science.

Get more details about these Calls at **www.usenix.org/cfp.**

JESA: Journal of Education in System Administration

Submissions due: August 14, 2015
www.usenix.org/jesa/cfp

USENIX is proud to announce the creation of a new *Journal of Education in System Administration (JESA). JESA* brings together researchers, educators and experts from a variety of disciplines, ranging from informatics, information technology, computer science, networking, system administration, security and pedagogics. *JESA* seeks to publish original research on important problems in all aspects of education in system administration. The mission of *JESA* is therefore to be a body of peer-reviewed, high-quality work addressing the challenges in system administration education.

URES '15: 2015 USENIX Release Engineering Summit

November 13, 2015, Washington, D.C.
Submissions due: September 4, 2015
www.usenix.org/ures15/cfp

At the third USENIX Release Engineering Summit (URES '15), members of the release engineering community will come together to advance the state of release engineering, discuss its problems and solutions, and provide a forum for communication for members of this quickly growing field. URES '15 is looking for relevant and engaging speakers for our event on November 13, 2015, in Washington, D.C. We are excited that this year LISA attendees will be able to drop in on talks so we expect a large audience.

URES brings together people from all areas of release engineering—release engineers, developers, managers, site reliability engineers and others—to identify and help propose solutions for the most difficult problems in release engineering today.

NSDI '16: 13th USENIX Symposium on Networked Systems Design and Implementation

March 16-18, 2016, Santa Clara, CA
Paper titles and abstracts due: September 17, 2015
Complete paper submissions due: September 24, 2015
www.usenix.org/nsdi16/cfp

The 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16) will focus 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.

FAST '16: 14th USENIX Conference on File and Storage Technologies

February 22-25, 2016, Santa Clara, CA
Submissions due: September 21, 2015
www.usenix.org/fast16/cfp

The 14th USENIX Conference on File and Storage Technologies (FAST '16) 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.

www.usenix.org/cfp

/var/log/manager
Incentivizing Smart People

ANDY SEELY

Andy Seely is the Chief Engineer and Division Manager for an IT enterprise services contract, and is an Adjunct Instructor in the Information and Technology Department at the University of Tampa. His wife Heather is his PXE Boot and his sons Marek and Ivo are always challenging his thin-provisioning strategy. andy@yankeetown.com

I manage smart, highly technical professionals in a hot job market. The Tampa Bay Area has an effective unemployment rate of 0% in our IT sector (and our weather is a lot better than you find in another "Bay Area" that has a hot IT sector…I'm just sayin'). I worry a lot about how to get my best employees to keep coming back to work each day. There are several incentive points to think about when preventing a valued employee from straying to another employer. To make their job feel like The Great Job takes a lot more effort than simply pointing out that you get to work with Andy Seely. Here are some "incentive vectors" explained, with points to consider from the sysadmin's point of view and then from the manager's perspective.

Money

Salary is the primary motivator for most people to go to work. Even if you love your job, do you do it for free? Few people do, and when they do it's because they already have plenty of money. The rest of us have bills to pay. You need to know how much salary you can earn, given the simultaneous equation of your skills, the business's need for those skills, the available budget, and the surrounding job market. Be prepared to discover that you may be worth more than the job can pay.

Vacation

Vacation time, sick leave, holidays: the amount of paid time off is a major motivator for people considering a job offer. Who doesn't love making money while doing what they love instead of having to go to work? This one has a hidden Easter egg to it. The people who are the top performers, who love what they do and throw themselves into it, are also the people who don't take their vacation days. If you earned three months' vacation in a year, would you take it? How much time off would you really take if you worked under one of the new "unlimited time off" corporate policies that are starting to show up in our industry [1]? Every top performer I know already runs up against the maximum accrual limit for paid time off. That's how they got to be top performers [2].

Benefits

Are you a single sysadmin? Biggest health problem you have is where's your pizza? Maybe this isn't a big driver for you. Are you a middle-aged sysadmin with a spouse and kids and maybe your blood pressure isn't where it should be? And while you're writing an article for *;login:* your wife is trying to give your seven-year-old some eye drops, and then she starts screaming because his eye just turned into a weird, swollen meatball, and then he starts screaming and the dog starts howling, and she's packing the car to rush to the emergency room and you walk over and…flip his eyelid back right-side out, thus saving the family a $500 ER trip? Health care can be a major hit against your bottom line. You need this benefit.

Opportunity for Promotion

Especially early in a career, few people ever say, no, thank you, but this is high enough for me. We're bred to grow, to achieve, to climb. Even sysadmins who don't want a management job, never-ever, cross my heart and hope to die, will still admit that they'd like a "senior engineer" or "technical director" title or some advancement along a technical track. As a sysadmin in a job in a company in a market, you need to not just know what you can do, but where you can do it and what your growth path is. You wouldn't try to drive from the Tampa Bay Area to California by stopping at every town and waiting for someone to tell you where to go, would you? No, you'd look at a map and start driving to your destination. So why would you try to get to the top of your career by stopping at your current job and wait for someone to tell you where to go?

Training and Conferences

Every good sysadmin knows how to do self-study. And every good sysadmin who does self-study to get ahead appreciates paid training. But it's rarely just about the training; it's also about the company's acknowledging the employee's contribution and its willingness to cut into the bottom line to invest in an individual. That sends a message to the employee that they're worth keeping. The employee gets new skills, a little bit of a break from the daily toil, and also earns a mark of confidence from the employer. I don't think it's uncommon for employers to resist allowing training as much for the cost as for fear that the employee will take the new skills and go find a better-paying job somewhere else. You should attempt to understand the financial system like you would any computing system: when there's budget, make your pitch and demonstrate not just how it helps you to help the company, but how you're going to stay on the job longer because they're not just buying training, they're also buying a happy employee.

Interesting Work

If you're a sysadmin and you read *;login:*, then I'm confident in saying that interesting work is your top motivator, right behind salary, which is probably also a top motivator. Let us rank them both as priority one: one-alpha and one-bravo.

There's a trick about interesting work. It has to be interesting enough to keep the attention and allow the best skills a sysadmin has to flow out. But it also has to be focused on the actual problems facing the business. It doesn't help the business to spend salary and capital expense to fund development of a new custom monitoring tool for a legacy system that supports the punch-card reader that gets used twice a year, even though that would probably be a really interesting project. It's important to work in a place that can give you the right kind of interesting work. Too little interesting work and a sysadmin starts to lose skills and may accidentally become a manager. Too much interesting work may really be a lack of focused direction on the part of the employer; if you can just do anything you want, anytime, how do you know it matters? And how can you be sure that your employer who allows it really knows what matters to their business? To engender job satisfaction, work has to be interesting, but it also needs to be meaningful, or you might be on a sinking ship.

Autonomy

All the sysadmins I've known have liked to be left alone to make their own decisions and follow their own insights. If a sysadmin asks for help, it's because help is really needed. Getting this dynamic right in a team setting isn't a problem, provided everyone knows what's expected of them. As an employee, you will never know if you'll have autonomy in a job until you're in it. Position descriptions all say, "must be a self-starter," but that just means that your manager doesn't want to have to always chase you down to get the TPS Report. It doesn't mean you get to call your own shots.

I'm the Manager. What Can I Do for You?

I can give you a raise, but it will be small enough that you won't really feel it, and only in rare circumstances will it be outside of an annual cycle. This may sound cynical, but think about what percentage raise it would take for you to change your standard of living or make a major life purchase, then think about the percentages of raises you've had in your career. Don't focus on the prospect of a single big raise as a big motivator.

I can't do a thing about vacation accrual or benefits. Maybe smaller companies have more leeway, although I imagine they have a lot less revenue to absorb the expense. Large companies get lost in policies and don't have a lot of flexibility for changing benefits packages for individual contributors.

I might be able to promote you, but consider how many senior jobs there are compared to junior and mid-level jobs. Opportunities are limited from the start, and there are others who want the same thing. If you have a PhD in Everything and you're working on the help desk, you're getting paid what the help desk pays, not what the PhD is worth. Show you're the right stuff for promotion, every day. A good way to show this is to help solve problems your manager has rather than focusing on problems you have.

Training and conferences are funny. When there's budget, it's easier. When there's not, it's impossible. Complaining about it, no matter how justified, is incredibly counterproductive and, over time, will turn an otherwise benign manager squarely against you.

Interesting work and some autonomy to get it done? Now we're talking. These things I can influence. I like motivated people who have good ideas and want to get things done.

There are many reasons why people keep coming to work. Taking the time to break them out, articulate them, and find ways to explain them is a useful tool when trying to retain top people. Helping smart sysadmins understand their real value to themselves and to the organization is something a manager can do. I'm the manager, and that's my job.

References

[1] "To Recruit Techies, Companies Offer Unlimited Vacation": http://www.bloomberg.com/bw/articles/2012-07-19/to-recruit-techies-companies-offer-unlimited-vacation.

[2] "Companies Offer 'Unlimited' Vacation Time Because They Know Perfectly Well People Won't Use It": http://www.slate.com/blogs/moneybox/2013/08/27/unlimited_vacation_time_it_s_no_accident_people_don_t_take_it.html.

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 who directly interact with students. We fund one representative from a campus at a time. In return for service as a campus representative, we offer a complimentary membership and other benefits.

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, contact
Julie Miller, Marketing Communications Manager, julie@usenix.org

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

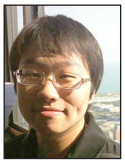
What Bugs Live in the Cloud?
A Study of Issues in Scalable Distributed Systems

HARYADI S. GUNAWI, THANH DO, AGUNG LAKSONO, MINGZHE HAO, TANAKORN LEESATAPORNWONGSA, JEFFREY F. LUKMAN, AND RIZA O. SUMINTO

Haryadi Gunawi is a Neubauer Family Assistant Professor in the Department of Computer Science at the University of Chicago where he leads the UCARE Lab (U Chicago systems research on Availability, Reliability, and Efficiency). He received his PhD from the University of Wisconsin–Madison and was awarded an Honorable Mention for the 2009 ACM Doctoral Dissertation Award. haryadi@cs.uchicago.edu

Thanh Do is a Researcher at Microsoft Jim Gray Systems Lab. His research focuses on the intersection of systems and data management. thdo@microsoft.com

Agung Laksono is a Researcher in SCORE Lab at Surya University Indonesia. He received his BS from Sepuluh Nopember Institute of Technology (ITS). His research interest is in cloud computing and software engineering. agung.laksono@surya.ac.id



Mingzhe Hao is a PhD student in the Computer Science Department at the University of Chicago. As a member of UCARE Lab, he is passionate about building performance-predictable cloud storage systems. hmz20000@uchicago.edu

We performed a detailed study of development and deployment issues of six open-source scalable distributed systems (scale-out systems) by analyzing 3655 vital issues reported within a three-year span [4]. The results of our study should be useful to system developers and operators, systems researchers, and tool builders in advancing the reliability of future scale-out systems. The database of our Cloud Bug Study (CbsDB) is publicly available [1].

As the cloud computing era becomes more mature, various scale-out systems—including distributed computing frameworks, key-value stores, file systems, synchronization services, streaming systems, and cluster management services—have become a dominant part of software infrastructure running behind cloud datacenters. These systems are considerably complex as they must deal with a wide range of distributed components, hardware failures, users, and deployment scenarios. Bugs in scale-out systems are a major cause of cloud service outages.

In this study, we focused on six popular and important scale-out systems: Hadoop, HDFS, HBase, Cassandra, ZooKeeper, and Flume, which collectively represent a diverse set of scale-out architectures. A comprehensive study of bugs in scale-out systems can provide intelligent answers to many dependability questions. For example, why are scale-out systems not 100% dependable? Why is it hard to develop fully reliable cloud systems? What types of bugs live in scale-out systems, and how often do they appear? Why can't existing tools capture these bugs, and how should dependability tools evolve in the near future?

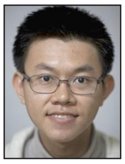
The answers to these questions are useful for different communities. System developers can learn about a wide variety of failures in the field and come up with better system designs. System operators can gain further understandings of distributed operations that are fragile to failure. For system researchers, this study provides bug benchmarks that they can use to evaluate their techniques. This study also motivates researchers to address new large-scale reliability challenges. Finally, tool builders can understand the limitations they work within and advance current tools.

In the rest of this article, we will present our high-level findings by focusing on new interesting types of bugs that we believe require more attention. At the end of this article, we will provide more samples of CbsDB use cases. The full scope of our study and specific bug examples can be found in our conference paper [4].

Findings

Before presenting specific types of bugs, we summarize our important findings.

New bugs in town: As shown in Figure 1, classical issues such as reliability (45%), performance (22%), and availability (16%) are the dominant categories. In addition, new classes of bugs unique to scale-out systems have emerged: *data consistency* (5%), *scalability* (2%), *topology* (1%), and *QoS* (1%) aspects.


Tanakorn Leesatapornwongsa is a PhD student in the Department of Computer Science at University of Chicago. He is a member of the UCARE Lab and is interested in addressing scalability and distributed concurrency problems. tanakorn@cs.uchicago.edu

Jeffrey Ferrari Lukman is a Researcher in SCORE Lab at Surya University, Indonesia. He is joining University of Chicago as a PhD student in computer science in Fall 2015. His research interest is in distributed systems reliability. jeffrey.ferrari@surya.ac.id

Riza Suminto received a BS in computer science from Gadjah Mada University in 2010. In 2013, he joined the University of Chicago to pursue his PhD in computer science. He is currently a member of the UCARE Lab and is interested in addressing performance bugs in cloud systems. riza@cs.uchicago.edu

Classification	Labels
Aspect	Reliability, performance, availability, security, consistency, scalability, topology, QoS
Hardware	Core/processor, disk, memory, network, node
HW failure	Corrupt, limp, stop
Software	Logic, error handling, optimization, config, race, hang, space, load
Implication	Failed operation, performance, component downtime, data loss, data staleness, data corruption
Impact scope	Single machine, multiple machines, entire cluster

Table 1: Issue classifications

Handling diverse hardware failures is not easy: "Hardware can fail, and reliability should come from the software" has been preached extensively, but handling diverse hardware failures such as fail stop, corruption, and "limpware[3]," including the timing of failures, is not straightforward (13% of the issues relate to hardware faults).

Vexing software bugs: The 87% of issues that pertain to software bugs consist of logic (29%), error-code handling (18%), optimization (15%), configuration (14%), data race (12%), hang (4%), space (4%) and load (4%) issues, as shown in Figure 3a.

In this article, we will delve into three interesting types of software bugs: (1) *single-point-of-failure* (SPoF) bugs, which can simultaneously affect multiple nodes or the entire cluster; (2) *distributed concurrency bugs,* caused by nondeterministic distributed events such as message reorderings and failure timings; and (3) *performance logic bugs,* which can cause significant performance degradation of the system.

Less-tested operational protocols: User-facing read/write protocols are continuously exercised in deployment and thus tend to be robust. Conversely, operational protocols (e.g., bootstrap logic, failure recovery, rebalancing) are rarely run and not rigorously tested. Bugs often linger in operational protocols.

A wide range of implications: Exacerbating the problem is the fact that each bug type can lead to almost all kinds of implication such as failed operations (42%), performance problems (23%), component downtimes (18%), data loss (7%), corruption (5%), and staleness (5%), as shown in Figure 3b.

The need for multi-dimensional dependability tools: As each kind of bug can lead to many implications and vice versa (Figure 4), bug-finding tools should not be one-dimensional.

Methodology

The six systems we studied come with publicly accessible issue repositories that contain bug reports, patches, and deep discussions among the developers. This provides an "oasis" of insights that helps us address the questions we listed above. From the issues repository of each system, we collected issues (bugs and new features) submitted over a period of three years (2011–2014) for a total of 21,399 issues. We manually labeled "vital" those issues pertaining to system development and deployment problems and marked them as high priority. We ignored non-vital issues related to maintenance, code refactoring, unit tests, documentation, and minor easy-to-fix bugs. This left us with 3655 vital issues that we then studied and tagged with our issue classifications as shown in Table 1. In each classification, an issue can

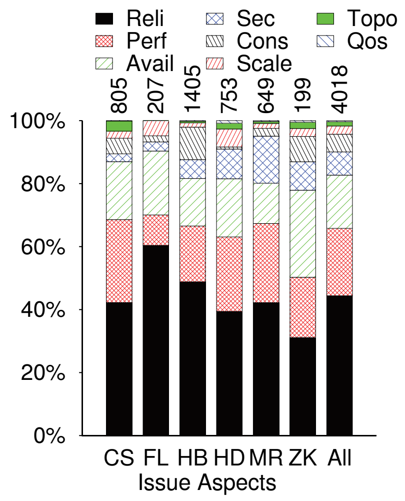


Figure 1: Issue Aspects. CS: *Cassandra; FL: Flume; HB: HBase; HD: HDFS; MR: MapReduce; ZK: ZooKeeper; All: Average*

have multiple sub-classifications. The product of our study is named Cloud Bug Study database (CBsDB) and is publicly available [1]. With CBsDB, users can perform both quantitative and qualitative analysis of cloud bugs.

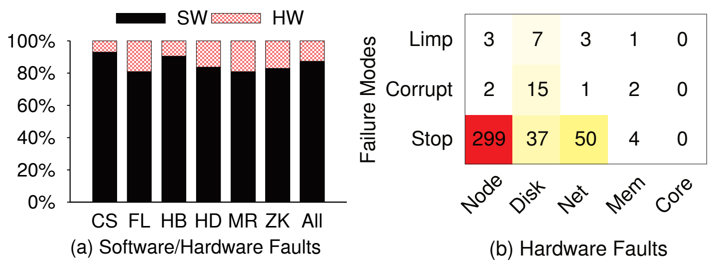
Issue Aspects

The first classification that we use is by aspect. Figure 1 shows the distribution of the eight aspects listed in Table 1. Reliability (45%), performance (22%), and availability (16%) aspects are the three largest categories. They are caused by diverse hardware-related and software bugs that we will discuss in subsequent sections. We also found many vital issues related to security (8%) and QoS (1%). Below, we pay attention to two interesting aspects distinct to scale-out systems: *distributed data consistency* and *scalability bugs*.

Data Consistency

Users demand data consistency, which implies that all nodes or replicas should agree on the same value of data (or eventually agree in the context of eventual consistency). In reality, several cases (5%) show data consistency is violated where users get stale data or the system's behavior becomes erratic. Data consistency bugs are mostly caused by the two following problems:

1. *Buggy logic in operational protocols:* Besides the main read/write protocols, many other operational protocols (e.g., bootstrap, background synchronization, cloning, fsck) touch and modify data, and bugs within them can cause data inconsistency. For example, in the Cassandra cross-datacenter (DC) synchronization protocol, the compression algorithm fails to compress some key-values, but Cassandra allows the whole operation to proceed, silently leaving the two DCs with inconsistent views.



	Node	Disk	Net	Mem	Core
Limp	3	7	3	1	0
Corrupt	2	15	1	2	0
Stop	299	37	50	4	0

Figures 2a and b: Hardware faults

2. *Concurrency bugs and node failures:* Intra-node (local) data races are a major culprit of data inconsistency. As an example, data races between read and write operations in updating the cache can lead to older values written to the cache. Inter-node (distributed) data races are also a major root cause; complex reordering of asynchronous messages combined with node failures make systems enter incorrect states.

In summary, operational protocols modify data replicas, but they often carry data inconsistency bugs. Robust systems require all protocols to be heavily tested. In addition, more research is needed to address complex distributed concurrency bugs (as we will discuss later).

Scalability

Scalability issues, although small in number (2%), are interesting because they are hard to find in small-scale testing. We categorize scalability issues into four axes of scale: cluster size, data size, load, and failure.

Scale of cluster size: Protocol algorithms must anticipate different cluster sizes, but algorithms can be quadratic or cubic with respect to the number of nodes. For example, in Cassandra, when a node changes its ring position, other affected nodes must perform a key-range recalculation with a complexity $\Omega(n^3)$. If the cluster has 100–300 nodes, this causes CPU "explosion" and eventually leads to nodes "flapping" (that is, live nodes are extremely busy and considered dead) and requires whole-cluster restart with manual tuning.

Scale of data size: Big Data systems must anticipate large data sizes, but it is often unclear what the limit is. For instance, in HBase, opening a big table with more than 100K regions undesirably takes tens of minutes due to an inefficient table look-up operation.

Scale of request load: Large request loads of various kinds are sometimes unanticipated. For example, in HDFS, creation of thousands of small files in parallel causes out-of-memory problems (OOM), and in Cassandra, users can generate a storm of deletions that can block other important requests.

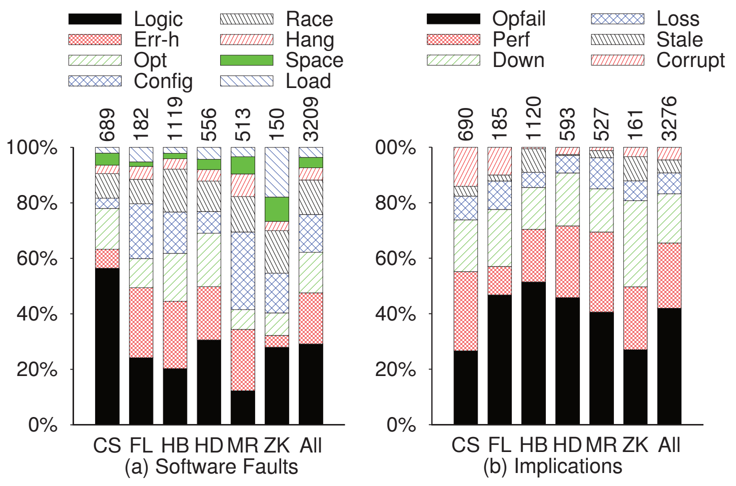


Figure 3a and b: Software bug types and implications

Scale of failure: At scale, a large number of components can fail at the same time, but some recovery protocols handle large scale failures poorly. In one example, when 16,000 mappers failed, Hadoop required over seven hours to recover because of unoptimized communication in HDFS.

In summary, scalability problems surface undesirably late in deployment. Similar to an earlier summary, we find the main read/write protocols scale robustly, but operational protocols (recovery, boot, etc.), on the other hand, often carry scalability bugs. One approach to solve this is via operational "live drills" [5], which should be performed frequently in deployment. Another research challenge is to develop scalability bug finders that can find scalability bugs without using large resources in testing.

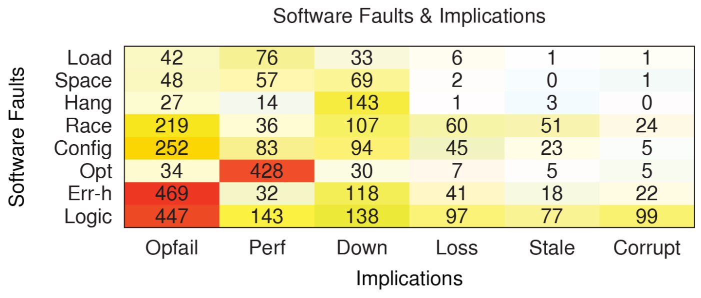
Hardware Issues

Next we categorize issues based on hardware vs. software faults. Figure 2a shows the percentage of issues that involve hardware (13%) and software (87%) faults. Figure 2b shows the heat map of correlation between hardware type and failure mode; the number in each cell is a bug count.

While fail stop and corruption are well-known failure modes, there is an overlooked hardware failure mode, *limpware* [3], hardware whose performance degrades significantly. For example, in an HBase deployment, a memory card ran only at 25% of normal speed, causing backlogs, OOM, and crashes.

Software Issues

Figure 3a shows the distribution of software bug types. The average distributions of software issues are: logic (29%), error handling (18%), optimization (15%), configuration (14%), data race/concurrency (12%), hang (4%), space (4%), and load (4%) issues.



	Opfail	Perf	Down	Loss	Stale	Corrupt
Load	42	76	33	6	1	1
Space	48	57	69	2	0	1
Hang	27	14	143	1	3	0
Race	219	36	107	60	51	24
Config	252	83	94	45	23	5
Opt	34	428	30	7	5	5
Err-h	469	32	118	41	18	22
Logic	447	143	138	97	77	99

Figure 4: Counts of software bugs and implications

Figure 3b depicts respective software bug implications. The average distributions for the implications are: failed operations (42%), performance problems (23%), downtimes (18%), data loss (7%), corruption (5%), and staleness (5%).

Figure 4 presents an interesting heat map of correlation between software bugs and their implications. Each kind of bug can lead to many implications and vice versa. If a system attempts to ensure reliability on just one axis (e.g., no data loss), the system must deploy various bug-finding tools that can catch different types of software bugs. Therefore, there is a need for multi-dimensional dependability tools.

For interested readers, discussions of the software issues above are discussed in our full paper [4]. Below we focus our discussions on three interesting distributed system bugs: single-point-of-failure (SPoF), distributed concurrency, and performance logic bugs.

SPoF Bugs

One interesting type of bug that we find is "single-point-of-failure (SPoF)" bugs. These bugs can simultaneously affect multiple nodes or even the entire cluster. The presence of these bugs implies that although the "no-SPoF" principle has been preached extensively, SPoF still exists in many forms.

Positive feedback loop: This is the case where failures happen, then recovery starts, but the recovery introduces more load and hence more failures. For example, busy gossip traffic can incorrectly declare live nodes dead, which then causes administrators or elasticity tools to add more nodes, which then causes more gossip traffic.

Buggy failover: A key to no-SPoF is to detect failure and perform a failover. But such a guarantee breaks if the failover code itself is buggy. For example, in HDFS, when a failover to a standby name node breaks, all data nodes become unreachable.

Repeated bugs after failover: Here, a buggy operation leads to a node crash triggering a failover. After the failover, the other node will repeat the same buggy logic, again crashing the node. The whole process will repeat and the entire cluster will eventually die.

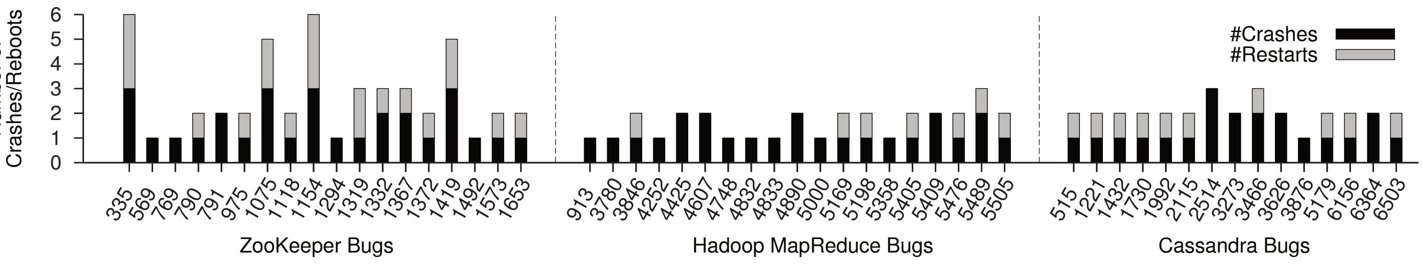


Figure 5: "Deep" distributed concurrency bugs. The x-axis lists bug numbers and the y-axis represents the number of crashes and reboots to unearth deep distributed concurrency bugs.

A small window of SPoF: Another key to no-SPoF is ensuring failover readiness all the time. We find few cases where failover mechanisms are disabled briefly for some operational tasks. In ZooKeeper, for example, during dynamic cluster reconfiguration, heartbeat monitoring is disabled, and if the leader hangs at this point, a new leader cannot be elected.

Buggy start-up code: Starting up a large-scale system is typically a complex operation, and if the start-up code fails then all the machines are unusable. As an example, a buggy ZooKeeper leader election protocol can cause no leader to be elected.

Distributed deadlock: This is the case where each node is waiting for other nodes to progress. For example, during start-up in Cassandra, it is possible that all nodes never enter a normal state as they keep gossiping. This corner-case situation is typically caused by message reorderings, network failures, or software bugs.

Scalability and QoS bugs: Examples presented earlier also highlight that scalability and QoS bugs can affect the entire cluster.

In summary, the concept of no-SPoF is not just about a simple failover. Many forms of SPoF bugs exist, and they can cripple an entire cluster (potentially hundreds or thousands of nodes). Scale-out systems must also be self-aware and make decisions to stop recovery operations that can worsen the cluster condition (for example, in the first two cases above). Future tools must address the five challenges of unearthing various forms of SPoF bugs.

Distributed Concurrency Bugs

Data races are a fundamental problem in any concurrent software system and a major research topic over the last decade. In our study, data races account for 12% of software bugs. Unlike nondistributed software, cloud systems are subject to not only local concurrency bugs (e.g., thread interleaving) but also *distributed concurrency bugs* (e.g., reordering of asynchronous messages). Our finding is that around 50% of data race bugs are distributed concurrency bugs and 50% are local concurrency bugs.

As an extreme example, let's consider the following distributed concurrency bug in ZooKeeper that happens on a long sequence of messages including failure events that must happen in a specific order.

ZooKeeper Bug #335: (1) Nodes A, B, C start with latest txid #10 and elect B as leader; (2) *B crashes*; (3) Leader election rerun, and C becomes leader; (4) Client writes data; A and C commit new txid-value pair {#11:X}; (5) *A crashes before committing tx #11;* (6) C loses quorum; (7) *C crashes;* (8) *A reboots and B reboots;* (9) A becomes leader; (10) Client updates data; A and B commit a new txid-value pair {#11:Y}; (11*) C reboots after A's new tx commit*; (12) C synchronizes with A; C notifies A of {#11:X}; (13) A replies to C the "diff" starting with tx 12 (excluding tx {#11:Y}!); (14) Violation: permanent data inconsistency as A and B have {#11:Y} and C has {#11:X}.

The bug above is what we categorize as a distributed concurrency bug. To unearth this type of bug, testing and verification tools must permute a large number of events, crashes, and reboots as well as network events (messages). Figure 5 lists more samples of distributed concurrency bugs. The point of the figure is to show that many of them were induced by multiple crashes and reboots at nondeterministic timings. Distributed concurrency bugs plague many many protocols, including leader election, atomic broadcast, speculative execution, job/task trackers, resource/application managers, gossiper, and many others. These bugs can cause failed jobs, node unavailability, data loss, inconsistency, and corruption.

For local concurrency bugs, numerous efforts have been published in hundreds of papers. Unfortunately, distributed concurrency bugs have not received the same amount of attention. We observed that distributed concurrency bugs are typically found in deployment (via logs) or manually. The developers see this as a vexing problem; an HBase developer wrote, "Do we have to rethink this entire [system]? There isn't a week going by without some new bugs about races between [several protocols]."

For this reason, we recently built an advanced semantic-aware model checker (SAMC) [6], a software (implementation-level) model checker targeted for distributed systems. It works by rapidly exercising unique sequences of events (e.g., different reorderings of messages, crashes, and reboots at different timings), and thereby pushing the target system into corner-case situations and unearthing hard-to-find bugs. SAMC is available for download [2].

Scenario Type	Possible Conditions
DLC: Data Locality	(1) Read from remote disk, (2) read from local disk,...
DSR: Data Source	(1) Some tasks read from same data node, (2) all tasks read from different data nodes,...
JCH: Job Characteristic	MapReduce is (1) many-to-all, (2) all-to-many, (3) large fan-in, (4) large fan-out,...
FTY: Fault Type	(1) Slow node/NIC, (2) node disconnect/packet drop, (3) disk error/out of space, (4) rack switch,...
FPL: Fault Placement	Slow down fault injection at the (1) source data node, (2) mapper, (3) reducer,...
FGR: Fault Granularity	(1) Single disk/NIC, (2) single node (dead node), (3) entire rack (network switch),...
FTM: Fault Timing	(1) During shuffling, (2) during 95% of task completion,...

Table 2: A partial anatomy of scenario root causes of performance bugs

Performance Bugs

Another notorious type of bug are performance bugs, which can cause a system to under-deliver the expected performance (e.g., a job takes 10x longer than usual). Conversation with several cloud engineers reflects that performance stability is often more important than performance optimization.

To dissect the root-cause anatomy of performance bugs, we performed a deeper study of vital performance bugs in Hadoop [7]. We found that the root causes of performance bugs are *complex deployment scenarios* that the system failed to anticipate. Table 2 shows a partial root-cause anatomy that we built. The table shows some of the *scenario types* such as "Data Source (DSR)" and *specific conditions* such as "some tasks read from the same data node (DSR_1)."

A performance bug typically appears in a specific scenario. For example, we found cases of untriggered speculative execution when the original task and the backup task read from the same slow remote data node (which can be represented as the combination of DSR_1 & FTY_1 & FPL_1 & DLC_1 as described in Table 2) or when all reducers must read from a mapper remotely and the mapper is slow (JCH_1 & FTY_1 & FPL_2). If one of the conditions is not true, the performance bug might not surface.

These examples point to the fact that performance anomalies are hard to find and reproduce. Scale-out systems make many nondeterministic choices (e.g., task placement, data source selection) that depend on deployment conditions. On top of that, external conditions such as hardware faults can happen in different forms and places.

The challenge is clear: to unearth performance bugs, we need to exercise the target system against many possible deployment scenarios. Unfortunately, performance regression testing is time-consuming and does not cover the complete scenarios. What is missing is fast, pre-deployment detection of performance bugs in distributed systems. One viable approach is the use of formal modeling tools (with time simulation) such as Colored Petri Nets (CPN) and TLA+/PlusCal. To be practical, the next big challenge is to automatically generate formal models that truly reflect the original systems code [7].

Other Use Cases of CbsDB

CbsDB [1] contains a set of rich classifications that can be correlated in various different ways which can enable a wide range of powerful bug analyses. For example, CbsDB can provide answers to questions such as: Which software bug types take the longest/shortest time to resolve (TTR) or have the most/least number of responses? What is the distribution of software bug types in the top 1% (or 10%) of most responded to (or longest-to-resolve) issues? Which components have significant counts of issues? How does bug count evolve over time? More details regarding CbsDB use cases can be found in our full paper [4].

Conclusion

At scale, hardware is not a single point of failure, but software is. A software bug can cause catastrophic failures including downtimes, corruption, and data loss. Our study brings new insights on some of the most intricate bugs in scale-out systems that we hope can be beneficial for the cloud research community in diverse areas as well as to scale-out system developers.

Acknowledgments

This material is based upon work supported by the NSF (grant nos. CCF-1321958, CCF-1336580, and CNS-1350499) as well as generous support from NetApp. We also thank other members of UCARE and SCORE labs, Tiratat Patana-anake, Jeffry Adityatama, Kurnia Eliazar, Anang Satria, and Vincentius Martin for their contributions in this project.

References

[1] http://ucare.cs.uchicago.edu/projects/cbs/.

[2] http://ucare.cs.uchicago.edu/projects/samc/.

[3] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi, "Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems," in *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC '13), 2013.*

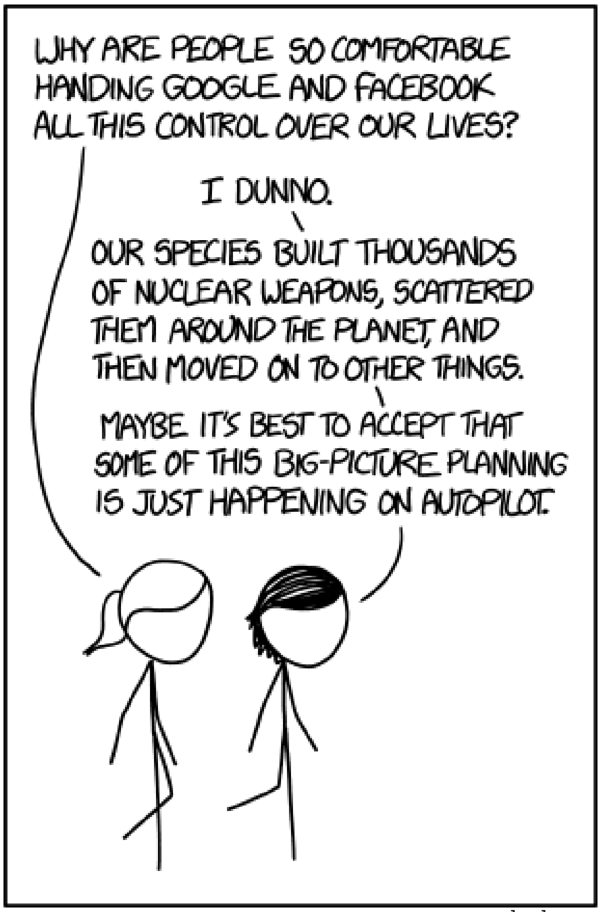
[4] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang Satria, "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems," in *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC '14)* 2014: http://dx.doi.org/10.1145/2670979.2670986.

[5] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi, "The Case for Drill-Ready Cloud Computing," in *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC '14), 2014.*

[6] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi, "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14), 2014.*

[7] Riza O. Suminto, Agung Laksono, Anang D. Satria, Thanh Do, and Haryadi S. Gunawi, "Towards Pre-Deployment Detection of Performance Failures in Cloud Distributed Systems," in *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '15),* 2015.

XKCD


WHY ARE PEOPLE SO COMFORTABLE HANDING GOOGLE AND FACEBOOK ALL THIS CONTROL OVER OUR LIVES?
I DUNNO.
OUR SPECIES BUILT THOUSANDS OF NUCLEAR WEAPONS, SCATTERED THEM AROUND THE PLANET, AND THEN MOVED ON TO OTHER THINGS.
MAYBE IT'S BEST TO ACCEPT THAT SOME OF THIS BIG-PICTURE PLANNING IS JUST HAPPENING ON AUTOPILOT.

xkcd.com

Distributing the News
UUCP to UUNET

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

If you were lucky enough to be on the ARPANET in the 1970s, you could get mail and news (in the form of a topical mailing list). But in January 1976 there were still only 63 hosts, and you had to be one of the elite to gain access. But soon there were methods to reach other sites. Like UNIX, the software began in New Jersey. Let's look at the next dozen years.

In 1976, Mike Lesk at Bell Labs came up with a program called UUCP—"UNIX to UNIX copy." UUCP enabled users to send mail, transfer files, and execute remote commands. Lesk first called it a "scheme for better distribution" (*Mini-Systems Newsletter,* January 1977); but only a month later it was referred to as UUCP. First designed to operate over 300 baud lines, UUCP was finally published in February 1978.

UUCP was taken up widely and this led to a need for improvements. The next version was written by Lesk and Dave Nowitz, with contributions by Greg Chesson, and appeared in Seventh Edition UNIX in October 1978.

Enter Usenet

In late 1979, the Seventh Edition was installed at the University of North Carolina at Chapel Hill. Steve Bellovin—partly as an exercise in the new system and partly to fill an administrative need—wrote a rudimentary news system as a UNIX shell file. It was very slow. Around the same time, Tom Truscott and Bellovin were experimenting with a UUCP link between UNC and Duke University (in Durham, NC). Truscott and Jim Ellis came up with the notion of distributing news to other sites via the UUCP link, using Duke as the central hub. Remote sites would reimburse Duke for the phone charges.

At the beginning of 1980 there were three sites: UNC, Duke University, and the Duke Medical Center Department of Physiology. The setup was described by Ellis in a pamphlet distributed at USENIX in Boulder, CO, at the end of January. An implementation of the A News software (by Steve Daniels) was made available on the 1980 USENIX distribution tape at the 1980 summer meeting in Newark, DE. By then there were 15 sites. The explosion occurred when the University of California joined.

The explosion was the direct responsibility of Armando Stettner and Bill Shannon of Digital Equipment Corporation. Someone at the USENIX meeting complained about the telephone bills run up by transcontinental calls. Armando and Bill said that if they could get a feed to decvax in New Hampshire, they'd pick up the Berkeley phone bill. (Stettner subsequently covered the news feeds to Europe, Japan, and Australia.)

Bellovin told me that the network was "called USENET, patterned upon USENIX… The hope was that Usenet would someday become the official network of USENIX." Within a year, the net grew to over 100 sites and 25 articles per day. And so the system collapsed. Lesk had never contemplated such uses of UUCP; Bellovin, Truscott, and Ellis never dreamt of such popularity.

Bellovin had revised his code, rewriting it in C. This had been revised by Steve Daniels and then Truscott, resulting in A News. In 1981, Mark Horton (a graduate student at UC Berkeley) and Matt Glickman (a high school student) rewrote A News into B News. Horton contin-

usenix 40TH ANNIVERSARY 1975–2015

ued revising B News until 1984, when he produced version 2.10.1. At that point, Rick Adams at the Center for Seismic Studies took over coordination and maintenance, producing 2.10.2.

This added the provision for moderated groups; Rick told me: "It was more like editing a magazine than moderating." In June 1984, Mark Horton and Karen Summers-Horton produced the "USENET GEOGRAPHIC MAP," showing connections to Australia, Hawaii, Canada (British Columbia, Alberta, Ontario, Quebec, and Newfoundland), and Europe (UK, Netherlands, Norway, Sweden, Germany, Switzerland, France, and Austria).

In 1986 version 2.11 of B News was released, including modifications and implementations by Rick, Spencer Thomas, Ray Essick, Rob Kolstad, and others. And while there were later releases (2.11.19 in 1994), Rick said: "It was dead in 1989."

The mortal blow was NNTP—RFC 977 "Network News Transfer Protocol," by Brian Kantor (UC Berkeley) and Phil Lapsley (UC San Diego), February 1986. Geoff Collyer and Henry Spencer (both at the University of Toronto) released C News, a new alternative in 1987, announcing it at the January USENIX Conference in Washington, DC. And while there is much more of interest where news is concerned (e.g., Larry Wall's rn, Rich Salz's InterNetNews, and Geoff Huston's ANU-NEWS (Australian National University in Canberra)), I will drop this thread here.

Usenet in the Sky: Stargate

Even with DEC picking up a portion of the expense, sending/receiving news produced vast telephone bills. At the summer 1984 USENIX Conference (Salt Lake City), Lauren Weinstein gave a paper proposing a possible "technological solution to the most pressing part of the problem, the cost of news transmission. The idea is as follows: portions of the video signal on TV transmission are not used for picture information, and can carry other information, in particular, suitably encoded ASCII. The effective bandwidth of this type of transmission could easily exceed 65 Kbps." [Lou Katz, *;login:* vol. 9, no. 6 (December 1984), 8–10]

Lauren succeeded in gaining support from a number of corporations and institutions. USENIX provided "support for incoming phone lines at the transmitter site, a small microwave receiver dish to test that mode of reception[,] and travel to the transmission site to set up the system." [Ibid.] Bellcore provided modems; Fortune Systems provided the uplink computer (a Fortune XT30—a desktop machine that retailed for about $5000); and Southern Satellite Systems of Atlanta supplied continuous use of part of a scan line in the broadcast signal of WTBS.

The transmission ran at 1200 bps for several months. There was a presentation about it at the Dallas USENIX in January 1985. But once it was successfully demonstrated, there was little further progress, and the USENIX Board of Directors, after a visit to the site in a cornfield near Atlanta, terminated the funding.

By the mid-1980s, there were several commercial networks in operation, but they were limited in service and generally quite high in price. None was what we would think of as an ISP.

In the autumn of 1985, Rick Adams (still at seismo), spoke with Debbie Scherrer, Vice President of USENIX, of a plan for a centralized site, accessed via Tymnet by subscribers, supplying Usenet access as well as ARPANET and UUCP. In an email dated December 6, 1985, Debbie expressed interest in this.

The May/June 1986 issue of *;login:* carried a "Request for UUCP and/or Usenet Proposals." Having funded Stargate and a one-year network mapping project (Horton and Summers-Horton), the Association was contemplating moving further.

Rick attended the October 1986 Board meeting in Monterey, CA, where reaction was mixed, one director asking why folks would pay for access that could be obtained free. But the Board agreed to entertain a proposal. Rick (and Mike O'Dell) brought a brief plan to the January 1987 (Washington, DC) meeting.

A majority of the USENIX Board liked the plan, but it really wasn't much of a business plan, and Rick and Mike were asked to fill out the plan, with the participation of Board members John Quarterman and Wally Wedel, and return.

By late March 1987 (in New Orleans), Rick was back with a full plan, and the Board approved it enthusiastically. I was authorized to spend up to $35,000 for an experimental period.

UUNET was born. "As people moved from universities and corporations where they had email and Usenet access to jobs where they had no access," Rick told me, "a need developed for a service that could provide email and Usenet access. UUNET was created in response to that need."

At the outset, UUNET ran on a Sequent B21 (16 processors): "the Sequent was the size of a small truck," Rick wrote me. In 1989, he moved UUNET to new office space, and the following year he turned it into a for-profit operation as UUNET Technologies.

When the word got out, subscriber demand far exceeded expectations. For example, Rick and Mike had forecast 50 subscribers by the end of summer. They topped 50 by mid-June 1987. Five years later, they had several thousand customers. UUNET was listed on NASDAQ and had its IPO in May 1995.

In 1991, UUNET participated in the founding of the Commercial Internet Exchange Association, and in 1992, it co-created (with Metropolitan Fiber Systems) MAE-East, for a time the world's busiest Internet exchange and center of the Internet.

Rick wrote me: "Usenet was part of the service, but what really hooked people was being able to send domain-based email over UUCP. The ARPANET/NSFNET gateway was key: we were gatewaying MCVAX running TCP/IP in Europe on a full commercial basis in November 1988 and speaking TCP/IP with them over a 9600 bps leased line for many months before NFS approved access (world's longest SLIP connection that I know of)."

More importantly, the "in-group" atmosphere of the ARPANET had been broken. UUNET initiated commercial delivery of Usenet and the Internet.

My thanks to Rick Adams, Lou Katz, and Mike O'Dell for their comments on this article. Any remaining errors are mine.

UNIX News

Number 10, October 1976

Security Patch

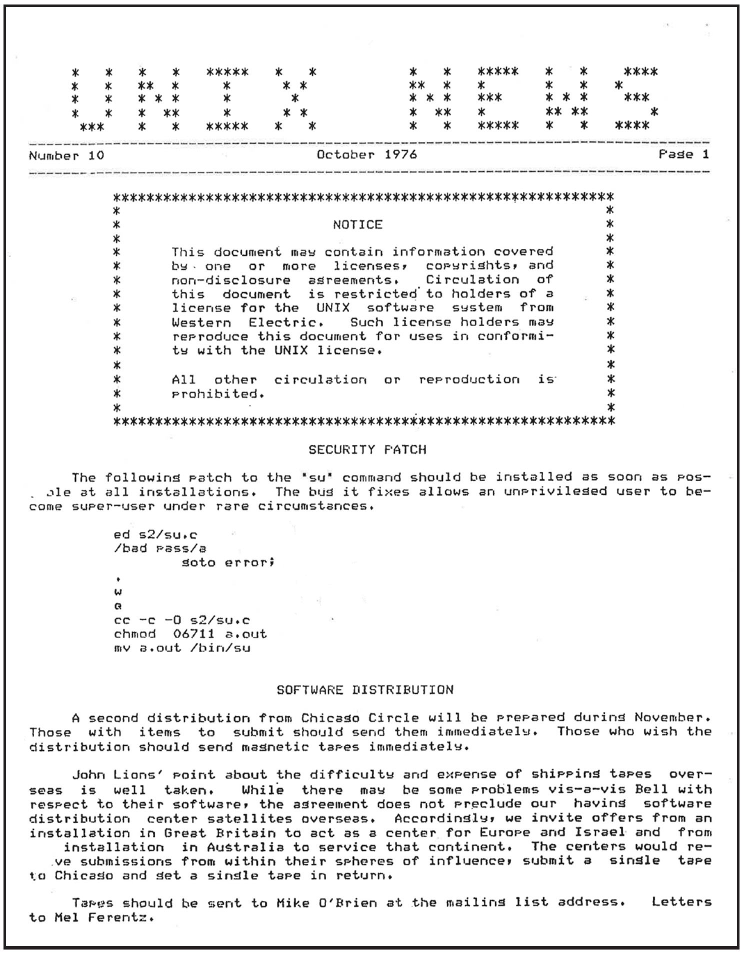
The following patch to the "su" command should be installed as soon as possible at all installations. The bug it fixes allows an unprivileged user to become super-user under rare circumstances.

```
ed s2/su.c
/bad pass/a
   goto error;
.
w
q?
cc -c -O s2/su.c
chmod 06711 a.out
mv a.out /bin/su
```

Software Distribution

A second distribution from Chicago Circle will be prepared during November. Those with items to submit should send them immediately. Those who wish the distribution should send magnetic tapes immediately.

John Lions' point about the difficulty and expense of shipping tapes overseas is well taken. While there may be some problems vis-a-vis Bell with respect to their software, the agreement does not preclude our having software distribution center satellites overseas. Accordingly, we invite offers from an installation in Great Britain to act as a center for Europe and Israel and from an installation in Australia to service that continent. The centers would receive submissions from within their spheres of influence, submit a single tape to Chicago and get a single tape in return.



```
 *   *  *   *  *****  *   *        *   *  *****  *   *   ****
 *   *  **  *    *     * *         **  *  *      *   *  *
 *   *  * * *    *      *          * * *  ***    * * *   ***
 *   *  *  **    *     * *         *  **  *      ** **      *
  ***   *   *  *****  *   *        *   *  *****  *   *  ****
-------------------------------------------------------------------
Number 10                 October 1976                       Page 1
-------------------------------------------------------------------
```

```
*************************************************************
*                                                           *
*                          NOTICE                           *
*                                                           *
*      This document may contain information covered        *
*      by one or more licenses, copyrights, and             *
*      non-disclosure agreements.  Circulation of           *
*      this document is restricted to holders of a          *
*      license for the UNIX software system from            *
*      Western Electric.  Such license holders may          *
*      reproduce this document for uses in conformi-        *
*      ty with the UNIX license.                            *
*                                                           *
*      All other circulation or reproduction is             *
*      prohibited.                                          *
*                                                           *
*************************************************************
```

SECURITY PATCH

The following patch to the "su" command should be installed as soon as pos- ble at all installations. The bug it fixes allows an unprivileged user to become super-user under rare circumstances.

```
ed s2/su.c
/bad pass/a
        goto error;
.
w
q
cc -c -O s2/su.c
chmod  06711 a.out
mv a.out /bin/su
```

SOFTWARE DISTRIBUTION

A second distribution from Chicago Circle will be prepared during November. Those with items to submit should send them immediately. Those who wish the distribution should send magnetic tapes immediately.

John Lions' point about the difficulty and expense of shipping tapes overseas is well taken. While there may be some problems vis-a-vis Bell with respect to their software, the agreement does not preclude our having software distribution center satellites overseas. Accordingly, we invite offers from an installation in Great Britain to act as a center for Europe and Israel and from installation in Australia to service that continent. The centers would re- ve submissions from within their spheres of influence, submit a single tape to Chicago and get a single tape in return.

Tapes should be sent to Mike O'Brien at the mailing list address. Letters to Mel Ferentz.

UNIX News, Number 10, was published in October 1976 by Professor Melvin Ferentz of Brooklyn College of CUNY. We have included excerpts from that issue and have reproduced the text as it appeared in the original, including any typographic errors. Note: We have not included the mailing list and other addresses and telephone numbers that appeared in the original issue.

University of New South Wales

From John Lions

On August 27th a group of more than 30 persons gathered at the University of New South Wales for our first local Users meeting.

David Morrison reported on the initial experience of the University of Newcastle with UNIX. They are currently heavily committed to using Basic Under RSTS on a PDP 11/45, and it was the quality of UNIX Basic which principally colored their reaction. They will undoubtedly be happier after trying the Harvard Software which was described to the meeting by Peter Ivanov.

Ian Johnstone spent some time discussing the security of UNIX. At the School of Electrical Engineering at the University of New South Wales the PDP 11 is run as an open shop staffed by casual, volunteer student operators. It is almost impossible to set up file access permissions in such a way that routine operations can be carried out safely (e.g. killing recalcitrant programs before shut-down) without leaving a loop-hole for the self-aggrandisement of users to super-users. A number of other modifications have been found necessary; groups have been disabled and "cron", for example, as a willing accomplice in crime, as been banished. However as long as the system console is accessible the most determined users cannot be prevented from patching the "suser" route directly. Setting the code for this routine into ROM would be a step in the right direction.

A UNSW implementation of Pascal "S" by John O'Neill, a final year undergraduate, was discussed and the meeting diverted on for a short while onto the subject of "Pascal" in general.

Most participants felt that the meeting was a success and another meeting has been planned for February 18th, 1977. It was agreed that there is a real need for cooperation between UNIX users in view of the unconventional nature of UNIX support.

Particular concern was expressed regarding the co-operative acquisition of software from overseas. Because of the distances involved this presents some difficulties and expense and it would certainly be more convenient for us if one local UNIX licensee, having acquired some item of software could distribute it to other local licensees (subject of course to completion of any required non-disclosure agreements production of DEC licenses, etc.). We have already attempted to raise this matter with Western Electric but so far have received no response.

Beware of icheck –s (or Change It)

From George Rolf, Katholieke Universiteit

I have been vaguely wondering for a while why everything I wrote seemed so much slower than the commands that came with the Unix system (version 6). Now I know why. Icheck –s will rearrange the freelist of a file system in the order of ascending block numbers, where mkfs initializes the freelist with consecutive entries 3 blocks apart on an RK disk, or 4 blocks on an RP. After I dumped the system and restored it onto a fresh file system I felt much happier.

I have also replaced the routine makefree( ) in icheck.c with the code reproduced below, which I borrowed from mkfs.c. Note that the change described in Unix Newsletter number 8 (August 1976) has been taken into account. Also note that this icheck –s produces an optimized lay-out for an RF disk, which the original mkfs does not. Our mkfs does of course.

I stumbled upon this discrepancy between mkfs and icheck while doing some measurements to find out what an optimal lay-out of the disk might be. I found myself reinventing the wheel. The measurements were the following. I made an executable file of 24 blocks (and one indirect block), and put it in various ways on one cylinder of an RK disk, with the indirect block in an adjacent cylinder. Exactly the same lay-outs were tried out on the RF disk (with 24 block "cylinders" instead of 8 blocks). I then timed read commands of the whole file at once, as well as exec-s on the file. For both devices the optimum is at a distance of 2 between consecutive file blocks. With both tests running at the same time, a distance of 3 blocks on both devices gave the best results, so those were the numbers I took.

I don't know why the Unix system as it is distributed doesn't have a special lay-out for the RF disk. At our installation, we have put the /tmp files on the RF disk, which appears to be a good idea. We have to keep the file system on the second RK drive interchangeable, and our RF disk has only one platter, which makes it a bit inconvenient to put the root directory there.

The only relevant measurements for this sort of questions are of course those obtained from heavy standard loads, or bench marks simulating such a load. We don't have either. Furthermore, the situation might be altogether different with different or more controllers, or for example with a 60 cycle RF disk, which runs 20% faster than ours. If anyone has any further ideas or other experimental results, I will be very anxious to learn of them.

```
in routine check():
change makefree(); to makefree(file);

freebl(i)
int i:
{
        if ((baab[i>>4)&07777] & (1<<(i&017))) == 0)
                free(i);
}

makefree(file)
char *file;
{
        register char *i, *j;
        char *n, *m;
        char *high, *low;
        static char adr[100], flag[100];

        for(j = file; j[0]; j++)
                if(j[0] == 'r')
                        switch (j[1]){
                        case 'k':
                                n = 24;
                                m = 3;
                                break;
                        case 'p':
                                n = 10;
                                m = 4;
                                break;
                        case 'f':
                                n = 8;
                                m = 3;
                                break;
                        default: ;
        }
        if (n > 100) n = 100;
        for(i = 0; i < n; i++)
              flag[i] = 0;
        j = 0;
        for(i = 0; i < n; i++) {
              while (flag[j])
                          j = (j + 1) % n;
              adr[i] = j;
              flag[j]++;
              j = (j + n) % n;
        }
        sblock.s_nfree = 0;
        sblock.s_ninode = 0;
        sblock.s_flock = 0;
        sblock.s_ilock = 0;
        sblock.s_faod = 0;

        high = sblock.s_fsize - 1;
        low = sblock.s_isize + 2;
        free(0);
        for(i = high; lrem(0, i+1, n); i--) {
              if (i < low)
                    break;
              freebl(i);
        }
        for(; i >= low + n-1; i =- n)
              for (j = 0; j < n; j++)
                    freebl(i-adr[j]);
        for(; i >= low; i--)
              freebl(i);
        bwrite(1, &sblock);
        close(fi);
        sync();
        return;
  }
```

Southern Illinois University at Carbondale

From Ray Kohring

Our department has been receiving the UNIX News since this Spring (issue #5 was the first one we received). What we have found most useful are the patches to the software which have been printed. In this light we would like to know if it would be possible to get any back issues that we missed. Any of them would be appreciated.

Our department owns a CAL DATA 135 which is emulating a PDP 11/40 on which we are running UNIX. In General, UNIX has ran well on our setup (exluding finding a missing wire on the MMU), but there are a couple of things which I felt were worth mentioning.

The first has to do with what happens when the user's stack-pointer is odd (that is not even, as opposed to unusual). What happens is the CPU goes through the stack error routine (specifically, red-stack limit) upon a buss-error, which clears the kernel stack-pointer (even though it was a user-mode error). This locks UNIX into a very tight loop (about 8 instructions long) which is retrapping on every attempt to stack something. I cured this by adding the code on the next page to m40.s. I haven't been able to determine if this happens on DEC CPUs also, but an easy check would be to run

```
              dec           sp
              mov           $1,-(sp)
```

and see if it loops.

The second problem is unique to CAL DATA systems with the micro-programming option. Accidently executing op-codes 7-17 (octal) causes all sorts of wonderous things to happen, since these are the spare op-codes (including EFM). The easy (?) cure is to load the appropriate ACM locations with a branch to the illegal instruction trap routine and enable it to replace the second page of control memory. A second alternative is to load routines to do common tasks, such as csav and cret, and modify the c-compiler to use those op-codes. One of our people (Carl Ebeling) has been working on this idea so if anybody wants to try it we could send you what he has done so far.

Note: This patch tests the stack pointer (kernel) to see if it is zero. If it is, it resets it to the top of the user block (where it probably should be) and copies the ps-pc from 0 to the correct stack locations. If it really is a kernel stack error, there will still be a panic.

```
ed m40.s
/trap:/
+
a
      tst      sp         /is the stack pointer zero?
      bne      1f         /no, we're still safe
      clr      177774          /stack limit register, the ps
                               /was put here by accident
      mov      $142000,sp      /restore the sp
      mov      2,-(sp)         /restack ps
      mov      0,-(sp)         /restack pc
      clz|clc                  /reset cc's to show buss-error
      mov      ps,-4(sp)       /redo properly
1:
.
```

The Pennsylvania State University

From Edward C. Horvath

I was directed to you by the UNIX documentation as a contact point for the UNIX user's group. If that is no longer appropriate, please forward this letter to whomever now fills that role.

The Computer Science Department here at Penn State recently acquired a PDP-11/34 and the UNIX system, and we are interested in hearing of and/or participating in the activities of the UNIX user's group.

Our system consists of an 11/34 (which includes memory management but no stack limit option), 96Kb core, a dual-drive RK11, RX11 floppy disk, and an 8-line DZ11 mux. This is a one-cabinet configuration which prices out (after haggling) at around $36K (circa June 1976). We are currently running only two typewriters (console and one DL11) and are in the process of constructing drivers for the RX and DZ. We soon expect to be running 6-8 users, and to expand core to 128K. We also have a 120 1/m Potter printer which we hope to interface to the DZ.

I should mention that UNIX (specifically rk unix) will not boot directly on the 11/34; there are minor programming differences between the 11/40 and the 11/34, none of which seem to surface when the system runs. However, the 11/34 comes standard with a blank front panel—an on/off switch, but no switch register. This drives the system into an infinite bus timeout trap loop when it tries to print the 'mem=' message. We were able to overcome this by laboriously hand-patching the system, a process which I will be happy to coach any new user on; I have attached a copy of the procedure to this letter for your files. We have not, to date, had any other problems with incompatibilities, but I will so inform you if they arise.

First, you can register us in the UNIX user's group.

Second, you can put us in contact with any other users who have constructed/are constructing drivers for the RX or DZ. We would be happy to share ideas and/or software; if we are the first and only developers for either device we will be happy to contribute any software we develop when it becomes available. Please inform me of any format restrictions or distribution clearing houses.

For your information, I have already informed Ken Thompson at BTL of the switch register problem; I'm not sure what steps he will take.

Thank you for your assistance; I look forward to your correspondence.

Bringing up UNIX (specifically rkunix) on the PDP-11/34

This document is for users who wish to run UNIX (6th Ed.) on the standard 11/34—i.e., with the standard front panel. If you have a switch register, the procedure described in 'setting up UNIX' should work just fine. In any case, this document is a supplement to 'setting up UNIX'.

First, generate the binary code RK05 pack. We cannot vouch for the procedures in 'setting up UNIX' for doing this from magtape, as we received the system already on RK05's.

Next, you have to locate the first block of 'rkunix' on the pack. 'rkunix' is a son of 'root', which is the root of the directory tree. (See File System (V) in the UNIX Programmer's Manual). 'rkunix' is described by i node 193 (base 10), which is the 6th i node of the 13th block of the i node list, which starts at byte 240 (base 8) of logical block 16 (base 8) of the RK05 pack (magic number 21). Note that 'rkunix' is a large file, so addr 0 points not at the first block of 'rkunix', but rather at the block of block pointers for 'rkunix'. On our distribution pack, addr 0 is 2723 (base 8). This converts to a 'magic number' for the RK11, namely 3703 (base 8), which may be deposited in the RKDA register to read the block of pointers. Again, on our distribution pack, the first pointer has value 2675 (base 8), which has magic number 3645 (base 8). If your pack disagrees in any way, calculate your own magic numbers! (Use the RK11 description of RKDA in the peripherals manual).

By the way, for magic number xxxx, the following console emulator sequence reads the desired block into core locations 0:777.

```
L   177406
D   177400
D   0
D   xxxx
L   177404
D   5
```

Once you have the first block of rkunix loaded in this way, perform the following sequence:

```
L   346
D   0
L   277406
D   177400
D   0
D   xxxx (magic number for first block)
L   177404
D   3
```

The above places a halt instruction in the trap sequence, and writes the block back out.

Steps thus far need only be done once; what follows is the new boot sequence:

1. Type OK, advance the paper, and hit return. The system should respond with 0.

2. Type 'rkunix' and hit return. The system will flutter a bit, then halt.

3. Hit the boot switch to bring in the emulator, and enter the following sequence:

```
L    176
D    100000
L    326
D    5767
L    12340
D    176
L    41236
D    176
L    0
S
```

The above sequence modifies the system to look at location 176 (base 8) for the contents of the switch register, loads 176 with 100000 (for a single user system, L 176 should be followed by D 173030), repairs the damage we did to the version on the pack, and finally restarts. The sequence described in 'setting up UNIX' now applies.

All of the above nonsense can, of course, be obviated if you can beg or etc. a couple of hours on a 40 or 45, or even get a 'loaner' front panel from your friendly DEC repairman, or, best of all, already have a running UNIX system. In any case, to avoid further heartache, you'll want to recompile the system to boot clean. In addition to the steps indicated in /usr/sys/run (watch out for ar!), you should:

Edit /usr/sys/param.h to change the value of SW to 0176

Make sure /usr/sys/ken/prf.c and user/sys/ken/sys4.c get recompiled and replaced in /usr/sys/libl.

The new system should come up clean (ours did!).

A Tale of Two Concurrencies (Part 2)

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 (www.swig.org) and Python Lex-Yacc (www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

In the previous installment [1], we dived into some of the low-level details and problems related to Python threads. As a brief recap, although Python threads are real system threads, there is a global interpreter lock (GIL) that restricts their execution to a single CPU core. Moreover, if your program performs any kind of CPU-intensive processing, the GIL can impose a severe degradation in the responsiveness of other threads that happen to be performing I/O.

In response to some of the perceived limitations of threads, some Python programmers have turned to alternative approaches based on coroutines or green threads. In a nutshell, these approaches rely on implementing concurrency entirely in user space without relying on threads as provided by the operating system. Of course, how one actually goes about doing that often remains a big mystery.

In this installment, we're going to dive under the covers of Python concurrency based on coroutines (or generators). Rather than focusing on the usage of particular libraries, the main goal is to gain a deeper understanding of the underlying implementation to see how it works, performance characteristics, and limitations. As with the previous installment, the examples presented are meant to be tried as experiments. There's a pretty good chance that some of the code presented will bend your brain—it's not often that you get to write a small operating system in the space of an article. Also, certain parts of the code require Python 3. So, with that in mind, let's start!

Threads, What Are They Good For?

Previously, we created a simple multithreaded network service that computed Fibonacci numbers. Here was the code:

```
# server.py

from socket import *
from threading import Thread

def tcp_server(address, handler):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        client, addr = sock.accept()
        t = Thread(target=handler, args=(client, addr))
        t.daemon=True
        t.start()
```

```
def fib(n):
    if n <= 2:
       return 1
    else:
       return fib(n-1) + fib(n-2)

def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = client.recv(1000)
        if not data:
            break
        result = fib(int(data))
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()

if __name__ == '__main__':
    tcp_server(('',25000), fib_handler)
```

When you run the server, you can connect any number of concurrent clients using nc or telnet, type numbers as input, and get a Fibonacci number returned as a result. For example:

```
bash % nc 127.0.0.1 25000
10
55
20
6765
```

If you carefully study this code and think about the role of threads, their primary utility is in handling code that blocks. For example, consider operations such as sock.accept() and client.recv(). Both of those operations stop progress of the currently executing thread until incoming data is available. That's not a problem, though, when each client is handled by its own thread. If a thread decides to block, the other threads are unaffected and can continue to run. Basically, you just don't have to worry about it, because all of the underlying details of blocking, awaking, and so forth are handled by the operating system and associated thread libraries.

If threads aren't going to be used, then you have to devise some kind of solution that addresses the blocking problem so that multiple clients can concurrently operate. That is the main problem that needs to be addressed.

Enter Generator Functions

In order to implement blocking, you have to figure out some way to temporarily suspend and later resume the execution of a Python function. As it turns out, Python provides a special kind of function that can be used in exactly this way—a generator function. Generator functions are most commonly used to drive iteration. For example, here is a simple generator function:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

Normally, this function would be used to feed a for- loop like this:

```
>>> for x in countdown(5):
...     print(x)
...
5
4
3
2
1
>>>
```

Under the covers, the yield statement emits values to be consumed by the iteration loop. However, it also causes the generator function to temporarily suspend itself. Here is a low-level view of the mechanics involved.

```
>>> c = countdown(5)
>>> next(c)     # Run to the yield
5
>>> next(c)
4
>>> next(c)
3
...
>>> next(c)
1
>>> next(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

On each next() call, the function runs to the yield, emits a value, and stops. A StopIteration exception is raised when the function terminates. The fact that yield causes a function to stop is interesting—that's exactly the behavior you need to handle blocking. Perhaps it can be used to do more than simple iteration.

Generators as Tasks

Rather than thinking of generator functions as simply implementing iteration, you can alternatively view them as more generally implementing a task (note: when used in this way, generators are typically called "coroutines," although that term seems to be applied rather loosely in the Python community). If you make a task queue and task scheduler, you can make generators or coroutines look a lot like threads. For example, here's an experiment you can try using the above generator function:

```
from collections import deque

# A task queue
tasks = deque()

# Create some tasks
tasks.append(countdown(10))
tasks.append(countdown(20))
tasks.append(countdown(5))

# Run the tasks
def run():
    while tasks:
        task = tasks.popleft()
        # Run to the yield
        try:
            x = next(task)
            print(x)
            tasks.append(task)   # Reschedule
        except StopIteration:
            print('Task done')

run()
```

In this code, multiple invocations of the `countdown()` generator are being driven by a simple round-robin scheduler. The output will appear something like this if you run it:

```
10
20
5
9
19
4
8
18
3
7
17
2
...
```

That's interesting, but not very compelling since no one would typically want to run a simple iteration pattern like the `countdown()` function in this manner.

A much more interesting generator-based task might be a rewritten version of the `fib_handler()` function from our server. For example:

```
def fib_handler(client, address):
    print('Connection from', address)
    while True:
        yield ('recv', client)    # Added
        data = client.recv(1000)
        if not data:
            break
        result = fib(int(data))
        yield ('send', client)    # Added
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
```

In this new version, `yield` statements are placed immediately before each socket operation that might block. Each yield indicates both a reason for blocking (`'recv'` or `'send'`) and a resource (the socket `client`) on which blocking might occur.

With the interactive interpreter, let's see how to drive it. First, create a socket and wait for a connection:

```
>>> from socket import *
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.bind(('', 25000))
>>> sock.listen(1)
>>> client, addr = sock.accept()
```

Next, establish a connection using a command such as `nc localhost 25000` at the shell. Once you've done this, try these steps:

```
>>> task = fib_handler(client, addr)
>>> task
<generator object fib_handler at 0x10a7c53b8>
>>> reason, resource = next(task)
Connection from ('127.0.0.1', 52474)
>>> reason
'recv'
>>> resource
<socket.socket fd=4, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1', 25000), raddr=('127.0.0.1', 52474)>
>>>
```

If you carefully study this output, you'll see that the handler task ran to the first `yield` statement and is now suspended. Before resuming the handler, you need to wait until input is available on the supplied socket (`resource`). To do that, you can poll the socket using a system call such as `select()` [2]. For example:

```
>>> from select import select
>>> select([resource], [], [])  # Blocks until data available
```

Go back to the terminal with the connected `nc` session and type an integer and return. This should force the above `select()` statement to return. Once it's returned, you can resume the generator by typing the following:

```
>>> reason, resource = next(task)
>>> reason
'send'
>>> resource
<socket.socket fd=4, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1', 25000), raddr=('127.0.0.1', 52474)>
>>>
```

Now you see that the task has advanced to the next `yield` statement. Use the `select()` statement again to see if it's safe to proceed with sending.

```
>>> select([], [resource], [])
>>> reason, resource = next(task)
>>>
```

In this example, you are using `next()` to drive the generator task forward to the next yield statement. The `select()` call is polling for I/O and is being used to know when it is safe to resume the generator.

A Generator-Based Task Scheduler

Putting the pieces of the last section together, you can make a small generator-based task scheduler like this:

```
from socket import *
from collections import deque
from select import select

tasks = deque()
recv_wait = {}   # sockets -> tasks waiting to receive
send_wait = {}   # sockets -> tasks waiting to send

def run():
    while any([tasks, recv_wait, send_wait]):
        while not tasks:
            can_read, can_send, _ = select(recv_wait, send_wait, [])
            for s in can_read:
                tasks.append(recv_wait.pop(s))
            for s in can_send:
                tasks.append(send_wait.pop(s))
        task = tasks.popleft()
        try:
            reason, resource = next(task)
            if reason == 'recv':
                recv_wait[resource] = task
            elif reason == 'send':
                send_wait[resource] = task
            else:
                raise RuntimeError('Bad reason: %s' % reason)
        except StopIteration:
            print('Task done')
```

The scheduler is essentially a small operating system. There is a queue of ready-to-run tasks (`tasks`) and two waiting areas for tasks that need to perform I/O (`recv_wait` and `send_wait`). The core of the scheduler takes a ready-to-run task and runs it to the next `yield` statement, which acts as a kind of "trap" or "system call." Based on the result of the yield, the task is placed into one of the I/O holding areas. If there are no tasks ready to run, a `select` call is made to wait for I/O and place a previously suspended task back onto the task queue.

To use this scheduler, you take your previous thread-based code and simply instrument it with `yield` calls. For example:

```
def tcp_server(address, handler):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        yield 'recv', sock
        client, addr = sock.accept()
        # Create a new handler task and add to the task queue
        tasks.append(handler(client, addr))

def fib(n):
    if n <= 2:
      return 1
    else:
      return fib(n-1) + fib(n-2)

def fib_handler(client, address):
    print('Connection from', address)
    while True:
        yield 'recv', client
        data = client.recv(1000)
        if not data:
            break
        result = fib(int(data))
        yield 'send', client
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()

if __name__ == '__main__':
    tasks.append(tcp_server(('',25000), fib_handler))
    run()
```

This code will require a bit of study, but if you try it out, you'll find that it supports concurrent connections without the slightest hint of a thread—interesting indeed.

Hiding Implementation Details

One complaint about the generator solution is the addition of the extra `yield` statements. Not only do they introduce extra code, they are somewhat low-level, requiring the user to know some details about the underlying scheduling code. However, Python 3.3 introduced the ability to write generator-based subroutines using the `yield from` statement [3]. You can use this to make a wrapper around `socket` objects.

```
class GenSocket(object):
    def __init__(self, sock):
        self.sock = sock

    def accept(self):
        yield 'recv', self.sock
        client, addr = self.sock.accept()
        return GenSocket(client), addr

    def recv(self, maxbytes):
        yield 'recv', self.sock
        return self.sock.recv(maxbytes)

    def send(self, data):
        yield 'send', self.sock
        return self.sock.send(data)

    def __getattr__(self, name):
        return getattr(self.sock, name)
```

This wrapper class merely combines the appropriate `yield` statement with the subsequent socket operation. Here is a modified server that uses the wrapper:

```
def tcp_server(address, handler):
    sock = GenSocket(socket(AF_INET, SOCK_STREAM))
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        client, addr = yield from sock.accept()
        # Create a new handler task and add to the task queue
        tasks.append(handler(client, addr))

def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = yield from client.recv(1000)
        if not data:
            break
        result = fib(int(data))
        yield from client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
```

In this version, blocking calls such as `client.recv()` are replaced by calls of the form `yield from client.recv()`. Other than that, the code looks virtually identical to the threaded version. Moreover, details of the underlying task scheduler are now hidden. Again, keep in mind that no threads are in use.

Studying the Performance

Previously, two performance tests were performed. The first test simply measured the performance of the server on CPU-bound work:

```
# perf1.py

from socket import *
import time

sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('127.0.0.1', 25000))
while True:
    start = time.time()
    sock.send(b'30')
    resp = sock.recv(100)
    end = time.time()
    print(end-start)
```

If you run this program, it will start producing a series of timing measurements that are essentially the same as the threaded version of code. If you run multiple clients, however, you'll find that the server is limited to using a single CPU core as before. There's no global interpreter lock in play, but since the entire server executes within a single execution thread, there's no way for it to take advantage of multiple CPU cores either. That's one important lesson—using coroutines is not a technique that can be used to make code scale to multiple processors.

The second performance test measured the performance on a rapid-fire series of fast-running operations. Here it is again:

```
# perf2.py
import threading
import time
from socket import *

sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('127.0.0.1', 25000))

N = 0
def monitor():
    global N
    while True:
        time.sleep(1)
        print(N, 'requests/second')
        N = 0
```

```
t = threading.Thread(target=monitor)
t.daemon=True
t.start()

while True:
    sock.send(b'1')
    resp = sock.recv(100)
    N += 1
```

If you run the program, you'll see output similar to the following:

```
bash % python3 perf2.py
16121 requests/second
16245 requests/second
16179 requests/second
16305 requests/second
16210 requests/second
...
```

The initial request rate will be lower than that reported with the examples involving threads in the previous article. There is simply more overhead in managing the various generator functions, invoking `select()`, and so forth. While the test is running, computing a large Fibonacci number from a separate connection produces:

```
bash % nc 127.0.0.1 25000
40
102334155     (takes a while to appear)
```

After you do this, the `perf2.py` will stop responding entirely. For example:

```
16151 requests/second
16265 requests/second
0 requests/second
0 requests/second
0 requests/second
...
```

This will continue until the large request completes entirely. Since there are no threads at work, there is no notion of preemption or parallelism. In fact, any operation that decides to block or take a lot of compute cycles will block the progress of everything else.

Back to Subprocesses

As it turns out, problems with performance and blocking have to be solved in the same manner as with threads. Specifically, you have to use threads or process pools to carry out such calculations outside of the task scheduler. For example, you might rewrite the `fib_handler()` function using `concurrent.futures` exactly as you did before with threads:

```
from concurrent.futures import ProcessPoolExecutor as Pool

NPROCS = 4
pool = Pool(NPROCS)

def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = yield client.recv(1000)
        if not data:
            break
        future = pool.submit(fib, int(data))
        result = future.result()
        yield from client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
...
```

The only catch is that even if you make this change, you'll find that it still doesn't work. The problem here is that the `future.result()` operation blocks, waiting for the result to come back. By blocking, it stalls the entire task scheduler. In fact, this will happen for any operation at all that might block (e.g., resolving a domain name, accessing a database, etc.).

Generators: It's All In

In order for a generator-based solution to work, every blocking operation has to be written to work with the task loop. In the previous example, attempts to use a process pool are unsuccessful since calls to obtain the result block. To make it work, you need to write additional supporting code to turn blocking operations into something that can yield to the task loop. The following code gives an idea of how you might do it.

The first step is to write a wrapper around the `Future` object's `result()` method to make it use `yield`. For example:

```
class GenFuture(object):
    def __init__(self, future):
        self.future = future

    def result(self):
        yield 'future', self.future
        return self.future.result()

    def __getattr__(self, name):
        return getattr(self.future, name)
```

Next, you might create a wrapper around pools to adjust the output of the `pool.submit()` to return a `GenFuture` object:

```
class GenPool(object):
    def __init__(self, pool):
        self.pool = pool


    def submit(self, func, *args, **kwargs):
        f = self.pool.submit(func, *args, **kwargs)
        return GenFuture(f)

    def __getattr__(self, name):
        return getattr(self.pool, name)
```

The main goal of these classes is to preserve the programming interface of the blocking code. In fact, you will only make a slight change to the `fib_handler()` code as shown here:

```
from concurrent.futures import ProcessPoolExecutor as Pool

NPROCS = 4
pool = GenPool(Pool(NPROCS))    # Note: Use GenPool

def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = yield client.recv(1000)
        if not data:
            break
        future = pool.submit(fib, int(data))
        result = yield from future.result()    # Note yield from
        yield from client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
...
```

Carefully observe how all blocking operations are now preceded by a `yield from` declaration. The only remaining task is to modify the task scheduler to support futures. Here is that code:

```
from socket import socketpair

tasks = deque()
recv_wait = {}
send_wait = {}
future_wait = {}

# Callback triggered on future completion
def _future_callback(future):
    tasks.append(future_wait.pop(future))
    _loop_wake()

# Sockets to allow waking of the I/O loop
_loop_notify_socket, _loop_wait_socket = socketpair()

# Function to wake the task loop when blocked on select()
def _loop_wake():
    _loop_notify_socket.send(b'x')

# Dummy task that allows select() to wake
def _loop_sleeper():
    while True:
        yield 'recv', _loop_wait_socket
        _loop_wait_socket.recv(1000)

tasks.append(_loop_sleeper())

def run():
    while any([tasks, recv_wait, send_wait, future_wait]):
        while not tasks:
            can_read, can_send, _ = select(recv_wait, send_wait, [])
            for s in can_read:
                tasks.append(recv_wait.pop(s))
            for s in can_send:
                tasks.append(send_wait.pop(s))
        task = tasks.popleft()
        try:
            reason, resource = next(task)
            if reason == 'recv':
                recv_wait[resource] = task
            elif reason == 'send':
                send_wait[resource] = task
            elif reason == 'future':
                future_wait[resource] = task
                resource.add_done_callback(_future_callback)
            else:
                raise RuntimeError('Bad reason: %s' % reason)
        except StopIteration:
            print('Task done')
```

Whew! There are a lot of moving parts, but the general idea is as follows. For futures, the task is placed into a waiting area as before (`future_wait`). A callback function (`_future_callback`) is then attached to the future to be triggered upon completion. When results return, the callback function puts the task back onto the `tasks` queue. A byte of I/O is then written to a special loopback socket (`_loop_notify_socket`). A separate task (`_loop_sleeper`) constantly monitors this socket and wakes to read the byte. (The main purpose of this special task is really just to get the task loop to wake from the `select()` call to allow ready tasks to run again.)

This Is Crazy (But Most Things Are When You Think About It)

Needless to say, if you're going to abandon threads for concurrency, you're going to have to do more work to make it work. If you get down to it, the code involving generators is actually a lot like a small user-level operating system, with all of the underlying task scheduling, I/O polling, and so forth. At first glance, the whole approach might seem crazy. However, keep in mind that it would rarely be necessary to write such code yourself. Instead, you would use an existing library such as the new `asyncio` module [4].

Even if you use a library, you still have to know what you're doing. Specifically, you need to be fully aware of places where your code might block and stall the task scheduler. Coroutines also do not free you from limitations such as Python's GIL—you should still be prepared to execute work in thread or process pools as appropriate.

At this point, you might be seeking some kind of sage advice on how to proceed with Python concurrency. Should you use threads? Should you use coroutines? Unfortunately, I can't offer anything more than it depends a lot on the problem that you are trying to solve. Python provides a wide variety of tools for addressing the concurrency problem. All of those tools have various tradeoffs and limitations. As such, anyone expecting a kind of "magic" solution that solves every possible problem will likely be disappointed. Again, some thinking is required—in the end, it really helps to understand what you're doing and how things work.

Postscript

The code examples in this article were the foundation of a PyCon 2015 talk I gave on concurrency. If you're interested in seeing the code work with a live coding demonstration, the talk video can be found online [5].

References

[1] D. Beazley, "A Tale of Two Concurrencies (Part 1)," *;login:*, vol. 40, no. 3, June 2015: https://www.usenix.org/publications/login/june15/beazley.

[2] "`select`—Waiting for I/O Completion": https://docs.python.org/3/library/select.html (select module).

[3] "PEP 380: Syntax for Delegating to a Subgenerator": https://www.python.org/dev/peps/pep-0380/.

[4] "asyncio—Asynchronous I/O, event loop, coroutines and tasks": https://docs.python.org/3/library/asyncio.html (asyncio module).

[5] PyCon 2015 presentation on concurrency: http://pyvideo.org/video/3432/python-concurrency-from-the-ground-up-live.

Practical Perl Tools

Parallel Asynchronicity, Part 2

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 thirty 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

Welcome back for Part 2 of this mini-series on modern methods for doing multiple things at once in Perl. In the first part (which I highly recommend that you read first before reading this sequel), we talked about some of the simpler methods for multi-tasking like the use of fork() and Parallel::ForkManager. We had just begun to explore using the Coro module as the basis for using threads in Perl (since the native stuff is so ucky) when we ran out of time. Let's pick up the story roughly where we left it.

Coro

As a quick reminder, Coro offers an implementation of coroutines that we are going to use as just a pleasant implementation of cooperative threading (see the previous column for more picayune details around the definition of a coroutine). By cooperative, we mean that each thread gets queued up to run, but can only do so after another thread has explicitly signaled that it is ready to cede its time in the interpreter. Thus, you get code that looks a little like this:

```
use Coro;

async { print "1\n"; cede;};
async { print "2\n"; cede;};
async { print "3\n"; cede;};
cede;
```

The script runs the main thread, which queues up three different threads and then cedes control of the interpreter to the first queued thread. It cedes control, so the second thread runs and so on. In this example, we don't technically need to write "cede;" at the end of each definition (since each queued thread will cede control simply by exiting), but it is a good habit to get into. The one place we definitely do need to explicitly write "cede;" is at the end of the script. If we didn't cede control at the end of the script, nothing would be printed because the main thread would have exited without realizing it should cede control to anything else.

We can do some more interesting things with this model, but before we do, it would probably be useful to understand how one goes about debugging a Coro-based program. When debugging a program like this, it would be a supremely handy thing to have information about the current state of the program that could tell us just what thread is running and what threads are queued up to run.

Coro ships with a debugger module that does all of this and more. There are two ways to make use of it: a non-interactive way and an interactive way. The interactive way works when used with an event loop-based Coro program like those you might be able to write after reading the last section of the column. But since we are not there yet, let's look at how to use the non-interactive method. We add Coro::Debug to the module loads and then insert a line that runs a debugger command. Let's modify the dead simple code example from above like so:

```
use Coro;
use Coro::Debug;

async { $Coro::current->{desc} = 'numero uno';
      print "1\n";
      cede;
};
async { $Coro::current->{desc} = 'numero dos';
      print "2\n";
      cede;
};
async { $Coro::current->{desc} = 'numero tres';
      print "3\n";
      Coro::Debug::command 'ps';
      cede;
};
cede;
```

We've made two changes. First, we've added lines to each async definition to give each one a description. You'll see how this comes in handy in just a moment. Second, in the third definition we've inserted a debugger command. When we run this script, it now prints something like:

```
1
2

PID              SC   RSS   USES  Description     Where
140252207746400  RC    21k   0    [main::]        [t:20]
140252207836704  N-    216   0    [coro manager]  -
140252207836680  N-    216   0    [unblock_sub    -
                                  scheduler]
140252207540736  R-   2060   1    numero uno      [t:9]
140252208378256  N-    216   0    [AnyEvent idle  -
                                  process]
140252208229104  RC   2600   1    numero dos      [t:13]
140533218598712  UC   2600   1    numero tres     [t:17]  3
```

This output shows us the status of all of the threads. Let me cherry-pick the key parts of this output to describe.

The first line is the main thread. It shows that it is [R]eady to run (the first letter of the SC column), has been scheduled 0 times (USES column) because the main thread doesn't need to be scheduled explicitly, and that it is currently running line 20 of the script (the file name is "t"). If we skip the threads that are part of Coro, we come to the first one we defined ("numero uno"—now you see why setting a description is useful). It too is Ready to run (currently at line 9 in the program). "numero dos" is in a similar state. The final thread we defined is shown as r[U]nning ("R" was taken by Ready). All of our defined threads are shown with a 1 in the USES column because they all have been queued to run once.

More Advanced Coro

In the puny code samples we've seen so far, each of the threads we've scheduled has been totally independent. Each printed a number, a process that didn't require any coordination (beyond making sure to be good neighbors by ceding to each other). But this isn't the most common of situations. Many (most?) times threads in a multi-threaded program are all trying to work towards the same goal by taking on a portion of the work. In those cases, threads have to work together collectively to make sure they aren't stepping on each other's toes. To do so they need a way to signal each other and maybe even pass on data in the process.

Anyone who has done other multi-threaded programming knows I'm headed towards talking about semaphores because that's the classic mechanism for intra-thread signaling. A semaphore is a shared resource (feel free to think of it as a magic variable) that the threads can read or attempt to change before they want to take an action. If a thread's attempt doesn't succeed (because another thread got there first), it can block and wait for the semaphore to become ready. This seems a little abstract, so let me show you some code from the Coro doc [1].

```
use Coro;

 my $sem = new Coro::Semaphore 0; # a locked semaphore

 async {
   print "unlocking semaphore\n";
   $sem->up;
 };

 print "trying to lock semaphore\n";
 $sem->down;
 print "we got it!\n";
```

In this case we are seeing a "counting" semaphore (where the semaphore has a value that can be incremented and decremented) being used as a binary semaphore (is it "locked" or "unlocked").

To follow the flow of the program, the main thread defines a semaphore with a value of 0, queues a separate thread (async{}), prints a message, and then attempts to decrement the semaphore with a call to down(). Since the semaphore is already at 0, the down() call blocks. In Coro, that blocking action cedes, and so the first queued thread gets a chance to run. When it runs, it increments the value of the semaphore and exits. Now that the semaphore is no longer 0, the down() call succeeds and the main thread continues to its end. This is a very basic semaphore mechanism—Coro offers a number of different variations on it so I recommend you look at the documentation.

Semaphores are a simple and effective way to keep threads from getting in each other's way, but what if they actively want to collaborate? That would entail being able to share information.

There are lots of ways threads could pass information around between them, but one built-in way Coro offers is through "channels." Channels (in Coro) are described as message queues. Any thread can add data to the queue, and any (other) thread can consume that data.

The syntax and method for using channels is as straightforward as you might hope. You create a new channel:

```
my $channel = new Coro::Channel;
```

write to it (from any thread):

```
$channel->put ('somepieceofdata');
```

and read from it (presumably from a different thread):

```
my $data = $channel->get;
```

If there is nothing in the channel, that thread will block and cede its time (just like a semaphore attempting to down() if the semaphore is already 0) until data does become available. Easy peasy.

Event-Based Programming

Let's move on to the final paradigm of this series. Event-based programming is yet another way to construct a system where a program can behave as if it is doing several things at once. There are a number of flavors of event-based systems, so let me give a broad generalization of a description that covers what we're about to do.

With the event-based programming style we're about to encounter, the basic idea is to specify events in the program's life that we care about and the code that should run when those events take place. These events could be external to the program (someone clicked on a button in a GUI) or events internal to it (when a piece of the program finishes). It is this latter case that interests us most at the moment because it means we can launch a whole bunch of actions—for example, a ton of DNS requests—and have them run at the same time.

Unlike your usual program that states "do this, then do this, then do this" (which means that thing #3 doesn't happen until #1 and #2 have completed), event-based programming lets you write code that says "do all the things, let me know when any of them finish, and I'll handle them at that point." Most of the time this is described in terms of registering interest in certain events and then starting an event loop that continuously checks if any of the events have come to pass. If it finds this has happened, the code associated with that event (a callback) is executed and then the loop continues.

There are a whole slew of Perl modules for writing event-based programs. Some of them are pure Perl; the more performant ones wrap external event libraries like libevent and libev. For this final section of the column, let's use all of them. Well, maybe most of them. But let's use them at the same time.

More precisely, let's use a module that calls itself "the DBI of event-loop programming." DBI, for those new to Perl, is a standard way to program database-related tasks in Perl that lets the programmer write database code that isn't tied to a specific database. AnyEvent aims to do this for event loops. It provides a uniform way to write code that is event-loop independent. The module will attempt to probe your system for the presence of a relatively long list of other event-based modules (including the performant ones). If it finds one, it will use it (without your having to know the specifics for the one it finds). If it doesn't find one, it will use a Perl-based "backend" that will function fine even without any of those modules being present. AnyEvent has proven quite popular in the community and so now a whole bunch of AnyEvent::Something modules are available for lots of tasks you might commonly want to do in an event-based/high-performance fashion.

Because event-based programming can get hairy quickly, we're only going to skim the top of AnyEvent to discuss the major ideas and then show one example of one of the task-specific AnyEvent::* modules. One other quick note before we move forward: AnyEvent comes with two different interfaces, a method-based one (AnyEvent) and a function-based one (AE). For example, you can write:

```
AnyEvent->timer (after  => $seconds,
                 interval  => $intseconds,
                 cb        => ...);
```

or

```
AE::timer $seconds, $intseconds, sub { ... };
```

The function-based one is more terse but is actually 5–6x faster with some backends. For this column, I'm going to use the method interface because I think it is easier for people not familiar with AnyEvent to read. When you write your own code and become comfortable with the arguments being passed to the methods, I encourage you to consider using AE instead so you can gain the performance increases.

The first concept central to any AnyEvent code is the "watcher." AnyEvent provides a set of different kinds of watchers including:

- I/O—when a file handle is ready to be read/written
- time—when a certain amount of time has elapsed
- signal—when we have received a certain signal
- child—when a child process changed state (completed)
- idle—when nothing else is happening

Let's look at a trivial AnyEvent code sample. It uses a time watcher because people can intuitively understand the idea of time events taking place (e.g., "Tell me when ten seconds have elapsed" or "Every two seconds, do the following…"). Here's a sample that uses two time watchers:

```
use AnyEvent;

my $enough   = 15;
my $yammer   = 0;

my $c = AnyEvent->condvar;

my $w;
$w = AnyEvent->timer(
    after      => 2,
    interval   => 2,
    cb         => sub {
        print "Every 2 (" . localtime( AnyEvent->now ) . ")\n";
        $yammer++;
        $c->send if ( $yammer == $enough );
    }
);

my $w2;
$w2 = AnyEvent->timer(
     after     => 5,
     interval  => 5,
     cb        => sub {
        print "Every 5 (" . localtime( AnyEvent->now ) . ")\n";
        $yammer++;
        $c->send if ( $yammer == $enough );
    }
);

print localtime( AnyEvent->now ) . "\n";
$c->recv;
print localtime( AnyEvent->now ) . "\n";
```

After loading AnyEvent, we specify that we only want 15 lines of output and define a variable that will be used to track the number of lines printed. Then we define a condition variable (more on this in a moment because it is fairly important).

Following this are the actual watchers. For each watcher, we say when we want AnyEvent to notice the time. For the first one, we want to notice when two seconds have gone by and then every time two seconds goes by after that. The second watcher is the same except it is paying attention to events every five seconds. When either of the events takes place, they run a tiny callback subroutine that prints the time, increments the output counter, and then decides whether to signal that it is okay to end the event loop (using that mysterious condition variable).

One other small Perl note. You might notice that we did something a little more verbosely than necessary, namely, defining a variable and using it as two different lines (which we almost always do on the same line):

```
my $w;
$w = AnyEvent->timer(
```

The reason we do this is a little subtle and not apparent in this sample itself. Each watcher can have a callback subroutine that gets defined as part of defining the watcher (we do this above). If a watcher wants to disable itself during the program's run, let's say it decides it has done its duty and wants to shut itself off, it does so from within the callback. The way it does so is to "undef" itself. So if the first watcher above wanted to disable itself at any point, in the callback subroutine it would state "undef $w;".

The tricky thing here is that Perl doesn't let you reference a variable in the same statement as the one where it gets defined. We can't do the equivalent of this:

```
my $var = sub { undef $var };
```

hence we have to define the variable that is going to represent the watcher and then create the watcher in two separate steps. You'll see this multiple-statement definition being used all over AnyEvent-based code.

The output of our sample code looks like this:

```
Mon Jun  1 10:37:34 2015
Every 2 (Mon Jun  1 10:37:36 2015)
Every 2 (Mon Jun  1 10:37:38 2015)
Every 5 (Mon Jun  1 10:37:39 2015)
Every 2 (Mon Jun  1 10:37:40 2015)
Every 2 (Mon Jun  1 10:37:42 2015)
Every 5 (Mon Jun  1 10:37:44 2015)
Every 2 (Mon Jun  1 10:37:44 2015)
Every 2 (Mon Jun  1 10:37:46 2015)
Every 2 (Mon Jun  1 10:37:48 2015)
Every 5 (Mon Jun  1 10:37:49 2015)
Every 2 (Mon Jun  1 10:37:50 2015)
Every 2 (Mon Jun  1 10:37:52 2015)
Every 5 (Mon Jun  1 10:37:54 2015)
Every 2 (Mon Jun  1 10:37:54 2015)
Every 2 (Mon Jun  1 10:37:56 2015)
Mon Jun  1 10:37:56 2015
```

So let's talk about condition variables (condvar) because they are one of the most important and the most confounding of AnyEvent concepts. One way to wrap your head around condvar is to harken back to the semaphores and channels we dealt with earlier in the column. Condvars are a way for different parts of the program to communicate with each other through a magic variable. This variable starts off as "false" and only becomes true when another part of the program sends a signal for it to change. In the interim, anything waiting for that signal will block (and here's an important part) while the rest of the event loop continues on around it. In the code we just saw, after defining the condvar ($c) and the watchers we say:

```
$c->recv;
```

which says, "Wait around for the condvar to become true during the event loop before continuing." This very act of waiting for something to happen in the event loop actually instructs AnyEvent to run the event loop.

Both of the watchers we defined check during the event loop if we've produced the right number of output lines in their callback subroutine. If either one determines this condition has been reached, they will send() on the condvar, and the program will stop waiting at the recv(). Since we are no longer waiting for event loop actions to take place, the loop shuts down and the program proceeds to its final print statement.

As you can probably guess, there's a bunch more functionality available from AnyEvent. For example, condvars can be used in a transactional way using begin() and end() calls so that the program can say, "Run an unspecified number of things at once, but only continue once all of them have completed." Rather than dive into more of these features, I want to show one small code example that makes use of one of the other AnyEvent-based modules in the ecosystem. This module we're about to see actually ships with AnyEvent itself.

Inspired by an example in Josh Barratt's excellent presentation on AnyEvent [2], here's some code that uses AnyEvent::DNS to check whether a domain exists in each of the current top-level domains. This version is a little spiffier than Barratt's because it pulls down the current list of all possible TLDs from IANA and checks against that. We'll talk about some of the pieces of the code after you've had a chance to see it:

```
use AnyEvent;
use AnyEvent::DNS;
use HTTP::Tiny;

# receive name to check from command line
my $name = shift;

my $domainslist =
  'http://data.iana.org/TLD/tlds-alpha-by-domain.txt';

my $domainlist = HTTP::Tiny->new->get($domainslist)->{content};

# ignore the comment and the test TLDs
my @domains = grep ( !/^(\#|XN--)/,
                     split( "\n", $domainlist ) );

my $c = AnyEvent->condvar;

my %domainresults;
for my $domain (@domains) {
    $c->begin;
    AnyEvent::DNS::a "$name.$domain", sub {
        $domainresults{$domain} = shift || "did not resolve";
        $c->end;
        }
}
my $start = AnyEvent->now;
$c->wait;
print "$#domains domains looked up in " .
   (scalar AnyEvent->now - $start) . " seconds.\n";
```

The first part of the code pulls down the IANA list. We then begin to iterate over each top-level domain, creating events that perform the lookups for us. When we do, we bracket each event with a condvar-based begin()/end() pair. This is the "transaction-like" use we mentioned earlier. The initial begin() records that we've started something, the end() indicates that we've finished something. We set the event loop in motion with a wait() call that basically says, "Run the event loop until all of the begin()s have had end()s."

Now, you may be as curious as I was to see just how much faster an AnyEvent version would be than one which worked its way through all of the TLDs, one TLD at a time. To test this, I gutted the AnyEvent watcher part in the middle and instead wrote the following:

```
use Net::DNS;
       ...
    my $reply = $res->search("$name.$domain");

    $domainresults{$domain} = "did not resolve";
    if ($reply) {
        foreach my $rr ( $reply->answer ) {
            next unless $rr->type eq "A";
            $domainresults{$domain} = $rr->address;
        }
    }
```

The version above yielded the following:

```
861 domains looked up in 717 seconds.
```

The AnyEvent version I showed first?

```
861 domains looked up in 54 seconds.
```

So, yes, quite a substantial speedup. I leave it as an exercise to the reader to write Parallel::ForkManager and Coro versions of the same program to see how they stack up.

We've come to the end of this column, but before I leave let me just mention that Coro has special support for AnyEvent that lets you use threads and an event-loop seamlessly. See the doc for Coro::AnyEvent for more information. And with that, take care and I'll see you next time.

References

[1] Coro documentation: http://search.cpan.org/perldoc?Coro/Intro.pod.

[2] Josh Barratt's AnyEvent presentation: https://vimeo.com/17163462.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The USENIX Store Is Open for Business!

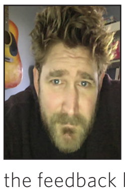
Want to buy a subscription to *;login:*, the latest short topics book, a USENIX or conference shirt, or the box set from last year's workshop? Now you can, via the **USENIX Store!**

Head over to www.usenix.org/store and check out the collection of t-shirts, video box sets, *;login:* magazines, short topics books, and other USENIX and LISA gear. USENIX and LISA SIG members save, so make sure your membership is up to date.

www.usenix.org/store

iVoyeur
How Do I Even KPI?

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

As I write this I'm on a plane back from "DevOps Days Toronto," at which I had a marvelous time. Probably the highlight of the trip for me was the "Open Space" on choosing effective KPIs (Key Performance Indicators). If you haven't been at a conference that does Open Spaces, they're very much like BoFs, except that they happen during the conference (not at lunch or after hours), and the selection process is more formal.

Honestly, I used to think they were kind of silly and suspected they were merely a means of making up for a lack of presenter content, but having spent the last year and a half traveling a lot more to various conferences, I've increasingly come to value them. The format really manages to give you a good feel for what everyone is dealing with in a specific problem domain (especially if you can manage to attend a few of them in different parts of the country).

The Open Space on the topic of choosing KPIs began with a question from the developer-turned-architect who had initially proposed the KPI Open Space. He'd just been put in charge of figuring out how to stabilize the efforts of 68 different development teams (!), and by stabilize, he meant that their product was behaving erratically, and they were beginning to have large blocking outages.

It sounded like his teams were all working on different parts of a single, large microservices architecture, which had grown large enough that the individual development efforts for each service were growing apart and becoming siloed. Because he was known to be a talented engineer who'd contributed to many of the services individually, the business had decided to "DevOps" him—i.e., snap him off from his current team so that he could focus on making the entire system work together better. He was eager to help but was having a hard time figuring out how to begin. He knew he wanted to get some data that would give him a good feel for where the problems were, but his question was, *what* specifically he should measure: "How do I choose some KPIs from scratch?"

It is a (usually) unwritten rule in programmer forums not to ask the room to do your homework for you. I'm not sure whether this applies to Open Spaces, but the architect's question certainly flirts with that line. In an Open-Space setting, however, I actually prefer this kind of discussion to the shallower and more uninformative "what is everyone using for X?" sort of question that typifies the Open-Space experience. In fact I think it's fair to say that when someone commits an oversharing faux pas in an environment like this, it relaxes everyone else, and puts us all in the mood to overshare a little bit ourselves.

Anyway, it quickly became apparent that many people in the room were having exactly the same pragmatic problem of not knowing where to begin with choosing metrics to measure. The first suggestion he got was to implement a policy that mandated filling out a form that included information like what KPIs should be measured before every deploy to production. This suggestion was accompanied by a lengthy, and very opinionated, anecdote that at some point segued into a full-bore anti-continuous delivery rant.

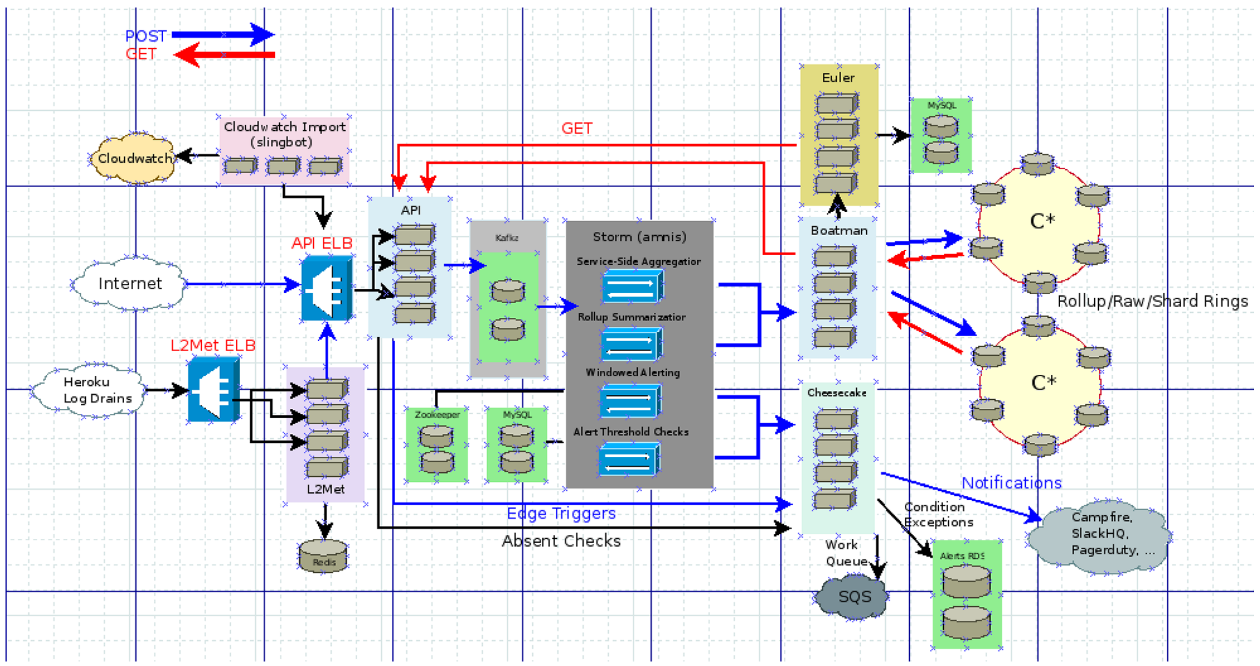


Figure 1: The prototypical (I hope) architecture diagram

"Best Open Space ever," I thought to myself as the room launched itself into a 40-minute long sanctimonious DevOps shame-splaining party. In the end, though, we were nowhere nearer to helping out with the original question (although we had a lively and entertaining discussion about the nature of DevOps versus "what the business actually needs").

Believe it or not, I do make an effort to keep my big mouth shut during the Open Spaces I attend (I rarely succeed). In this case, however, since no one else had offered any constructive advice, I ventured to share what has worked for me in the past. And since it was well received, and the problem seemed so prevalent, I figured it might make a nice *;login:* article this month, so I'll share it with you too.

I'm sure I've said before in this column that good metrics test systems hypotheses. They capture the operational limitations we've learned about the things we build. When I say they test systems hypotheses, I mean that when we think about the systems we build, and how they should act in certain situations (e.g., given 50,000 connections, this round-robin-based load balancer should send 25k to server A, and 25k to server B), good metrics confirm our valid assumptions and discredit our biases. They teach us about how the things we build actually work.

By this yardstick the classic CPU/memory/network triumvirate is mediocre at best. You *may* have a meaningful hypothesis about how much RAM or CPU a process should use, and you *may* learn something about your system (or more likely the underlying interpreter or OS, or garbage collector) if your assumption isn't borne out in practice, but metrics that measure things like how long a particular database call takes, or count the total number of worker threads, or queue elements, reflect assumptions that make for a more meaningful understanding of the system you're dealing with.

Not only do experienced engineers understand that building a system is not the same thing as understanding it, they can pretty quickly intuit how well a system they didn't build is understood by the team running it. The evidence is everywhere: in how deeply we can test our code, in how specifically we monitor them, in how precisely we can derive our capacity plans, and even in how repeatably we can deploy them.

The architect who asked this question was an experienced engineer. He knew that these teams didn't understand what they'd constructed, and therefore no amount of asking them to fill out a form listing their KPIs was going to give him the insight he needed to make it work better. He had to get his own hooks in, but the question was where?

Whenever I'm put in charge of a large and churning wad of software that I didn't write, I draw a picture of it, and that picture inevitably comes out looking something like Figure 1. In fact, this is one of the actual pictures I drew when I was first hired on and trying to wrap my head around how Librato's microservices architecture works in practice.

Measure the Space between the Services

Normally, we'd focus our attention on the boxes, and in the end we do want to know, in depth, how each of these services works so we can derive some metrics that are key indicators of how well

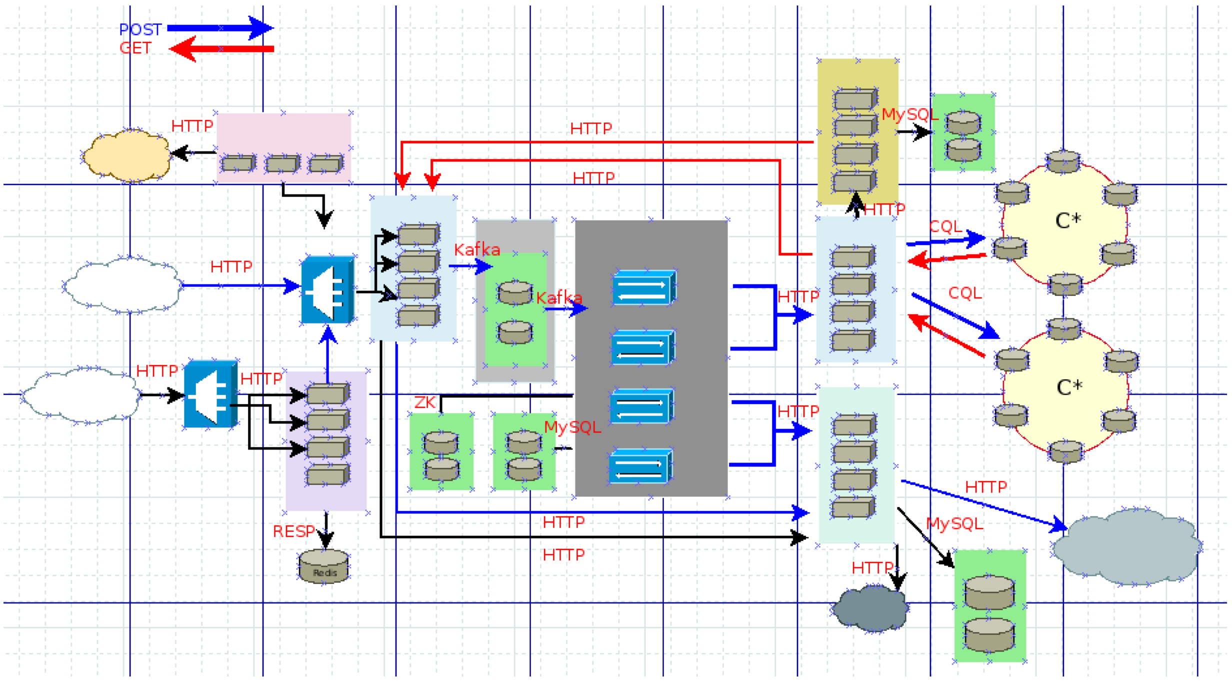


Figure 2: Figure 1, relabeled to accentuate the space between the services

they're doing what they should be doing. However, we're going to start by ignoring the boxes completely. In fact, I'm going to delete all of these box labels and replace them with letters, and in the place of all the service names, I'm going to label the lines. Above each line, I'm going to place a label that identifies the protocol each of those lines represents. This gives us Figure 2.

Check that out, our previously incomprehensible microservices architecture just became a handful of commodity network protocols. This, I can pretty easily wrap my head around. Every application is a balanced equation; it'll work fine as long as it is in balance, and eventually we'll root out all of the things that can throw it out of whack. But for now, the best way to detect when it's out of balance is by timing the interactions between its component parts—measuring the space between the services. Our strategy will be to figure out a way to time the interactions represented by each of these lines.

If I made that sound easy, it's not. Getting these numbers, which I collectively refer to as inter-service latency data, is going to require a lot of engineering know-how. In almost every case, you'll have to get into the source and add some instrumentation that wraps API or DB calls. Sometimes you'll be need to reconfigure a set of Web servers or proxies, and every once in a while, you'll need to write some glue-code or API-wrappers of your own.

You should wind up with a slew of numbers on the order of tens or hundreds of milliseconds. When something goes wrong with the application, these numbers will tell you *where* the problem is (in which service on which nodes). Note, this is not the same thing as telling you *what* the problem actually is, but we'll get to that in a minute.

Of course you'll need to actually put all of this data somewhere. That's the sort of thing I (and many other people) have written about at length, but it's worth mentioning here that you're going to need a scalable telemetry system to help you store and analyze all this stuff.

Extract Knowledge from Inter-Service Latency

Play around with these numbers as you get each of them up and running. Note the baseline values, and search for patterns of behavior, and things that strike you as odd. Do some service latencies rise and fall together? Do some appear dependent on others? Do they vary with the time of day or day of week? As you discover these patterns, talk to the engineers who run the services and see whether these patterns confirm their notions of how that service "should" work. It shouldn't take long before one of them squints at your data and says something like "huh." This is what scientific discovery sounds like. Dig into that service behavior with the help of the engineer who runs it, and you'll likely encounter a KPI or two.

When something goes wrong, look at the inter-service latency data and see how early you can identify things going sideways. The numbers tend to get big upstream of the services that are actually having trouble. Share your data with the engineers running those services, and dig into them together to figure out what went wrong; again, you'll likely encounter a KPI or two.

If that sounds kind of labor intensive and slow, it is. But before you know it you'll have several dozen extremely valuable KPIs. Until you get into the habit of choosing effective metrics, they take some time and effort to identify. Each KPI really is a manifestation of insight; each teaches you something you didn't know about the services you maintain. Each is a thing to be prized, shared, and talked about.

For Example

As you can probably imagine, we're pretty good at choosing effective metrics before we need them at Librato, but we still regularly encounter valuable metrics that we didn't anticipate. For example, we recently encountered a behavior in one of our newish services that we couldn't explain. Symptomatically, it was quite visible in our inter-service latency data as a latency spike between the service and a MySQL server.

When we dug into it, we found that there was a bug in the upstream API of a vendor that the service relied on. If we crafted the API request a certain way (the correct way), the API returned too many results (all of them, instead of the subset specified by the query), and we wound up over-taxing our own MySQL server writing this over-abundance of results back. But if we used a modified version of the broken-looking example from the upstream vendor's documentation, it worked fine.

We reported the bug and commented our code, but found that every engineer who came across this query had the irrepressible urge to fix this broken-looking API query, so we began tracking the number of results returned by this API query as a KPI for that service. Several months later, when the upstream vendor fixed their API, we had the opposite problem: we were getting 0 results back from that API (because our broken query, was in fact, now broken), but since we were already tracking that metric, we immediately saw what the problem was and were able to very rapidly push a fix for it.

Today, the engineers who were involved in that episode (myself included) tend to include KPIs like the number of results returned from interfaces they don't control as a matter of course. They probably don't even remember why. This is one of the many ways that going through the process of finding and relying on effective operational metrics changes the culture of engineering teams. It is a self-sustaining cycle: good data begets reliance on data, which begets better data.

KPIs that represent insight into the systems that we build give us a rock to stand on in the midst of uncertainty, and enable us to act quickly and decisively to protect the uptime of our services. Without them we don't really know how the things we build work. If you're in that boat, the place to start (IMO) is with inter-service latency data. Get it, and use it to work your way into insight.

Take it easy.

Balkanization from Above

DAN GEER AND HD MOORE

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

HD Moore is the Chief Research Officer at Rapid7, responsible for leading Rapid7 research into real-world threats and providing guidance on how to address them. In addition, HD drives technical innovation across Rapid7's products and services, applying technology to the challenge of identifying and defending against current and emerging threats, as well as heading the development of experimental prototypes and free tools. HD is the creator of Metasploit, an open source penetration testing framework, and remains deeply involved in Metasploit's evolution. x@hdm.io

The Internet of 2015 is a different place compared to five years ago. Business models have changed, technology has shifted onward, hundreds of millions of new people have connected to the World Wide Web, and so forth. How they connect, what devices they use, and the threats they face have likewise shifted, and, to our point, the Internet is itself being dragged along.

Where the Internet was transparent and distributed, it is becoming opaque and centralized. The immense, if abstract, value of peer-to-peer communication has been eclipsed by—indeed has become subservient to—consumer demand for downstream content. Nowhere is this more apparent than in the mobile Internet. The IPv4 address space is running out of steam and service providers are compromising bi-directional network communication in favor of scalability. In corporate America, businesses are choosing the economies of scale in cloud offerings and rejecting local datacenters in favor of external on-demand infrastructure.

The end result is an inversion from a peer-to-peer "freedom to connect" model to one consisting of service provider enclaves providing private access to managed offerings. The Internet is increasingly attenuated between broadband on the one end and cloud providers on the other, with decreasing open space in between. Criminals, governments, and curious hackers alike are following this trend and changing their tactics in approximate (if ironic) synchrony. ISP-provided routers are becoming the target of choice for threat actors globally. Vulnerabilities in mobile devices and desktop operating systems are more valuable than ever. Cloud providers are increasingly targeted, and many are failing. The attack surface of the Internet necessarily grows faster than linearly with the count of endpoints, but even that is increasingly difficult to measure.

IPv4 Utilization

The IPv4 Internet has room for approximately 4.3 billion unique addresses, of which 3.7 billion can be used by public networks and hosts. These addresses are a finite resource managed by regional Internet registries, and as of June last year, we ran out. Figure 1 shows the number of /8 network blocks available from 1995 to June 2014.

The Internet relies on DNS to associate a name with an address. Of the 3.7 billion usable addresses, over 1 billion have an associated reverse DNS name. As the IPv4 Internet has run out of free network blocks, growth of named hosts has dropped accordingly. Figure 2 shows the growth of named hosts. (The logistic curve's inflection point was, as shown, November 21, 2008.)

The ITU (International Telecommunication Union) estimates that there are over 3 billion Internet users as of 2015 [1]. This number represents over 2.3 billion mobile broadband subscriptions and another 700+ million fixed broadband subscriptions [2]. Combine these stats with infrastructure equipment such as routers, switches, and all of the servers that actually power the Internet, and it is clear there isn't room for everyone in IPv4. In contrast to the rate of IP allocations and named hosts, growth in total connected devices seems to continue.

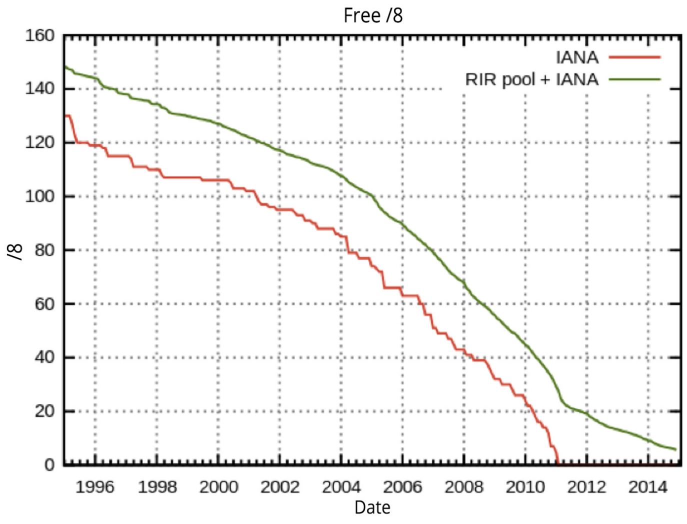


Figure 1: Number of /8 blocks available by date

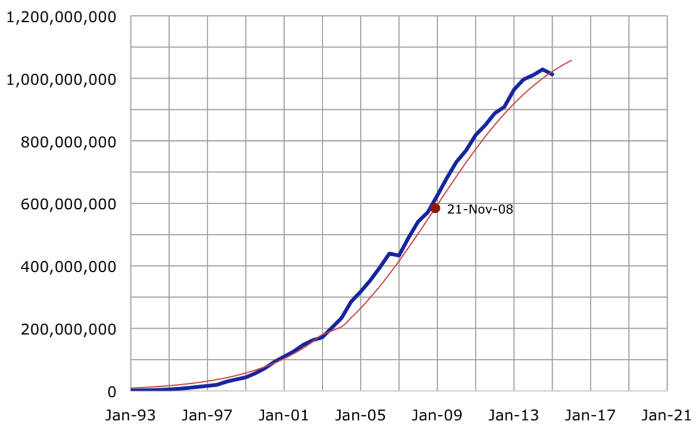


Figure 2: Growth curve and inflection point for number of hosts with PTR records

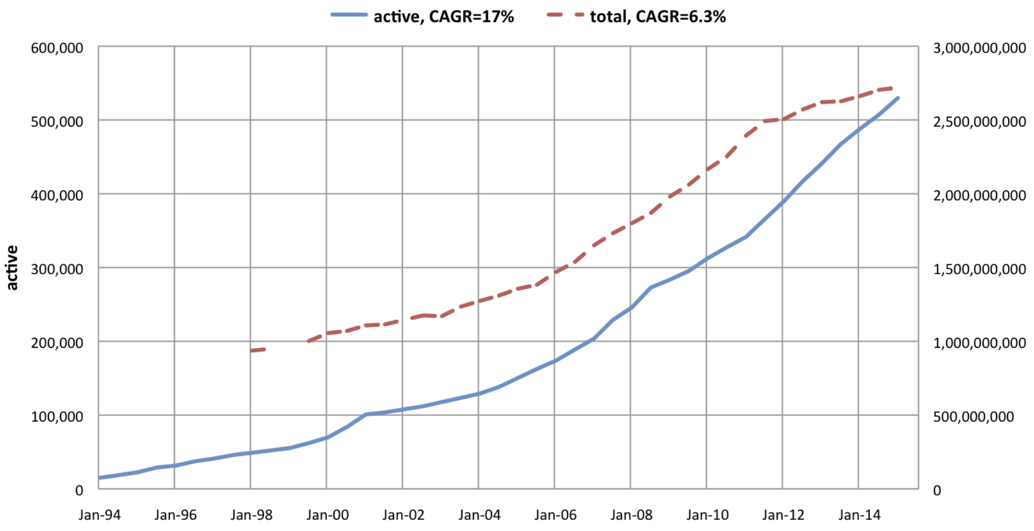


Figure 3: Active space (left vertical axis), total space (right vertical axis)

In a similar vein, growth of the total advertised IP space is slower than growth of subdivision within that space (compound annual growth rate, or CAGR, of 17% versus 6.3% as measured by BGP); see Figure 3.

Note that instead of a lengthy diversion into IPv6 and next-generation addressing, we keep our discussion to the Internet as it stands today. At its most succinct, there are far more users than there are IPv4 addresses, and IPv4 addresses are distributed unequally, sometimes to an absurd degree.

Approximately 370 million IPv4 addresses respond to an ICMP echo request. This represents about 10% of the usable IPv4 space. If we send common TCP and UDP probes as well, this number rises to 466 million IPv4 addresses (13%). The Hilbert graph in Figure 4 represents the density of hosts responsive to ICMP, TCP, and UDP probes. The extreme density in the lower left and center right are in clear contrast to the "empty" blocks in the upper left. The majority of reserved ranges are concentrated in the upper right quadrant and are evenly shaded. Many of the empty blocks are actually in use by government agencies and large corporations, but have been isolated from the rest of the Internet by firewalls (another form of enclave).

This 466 million number is important; it is the number of IPv4 addresses that are remotely discoverable and thus directly targetable by an attacker. The number of directly connected IPv4 systems puts an upper bound on the number of potential targets for any new server-side exploit. At the same time, the number of DNS PTR records at 1013 million is twice as big. What is going on?

3 Billion Users

The number of broadband users, consisting of both fixed-line and mobile, has increased from 500 million in 2007 to over 3 billion in 2014. Figure 5 demonstrates this growth. Contrast the 466 million discoverable IPv4 addresses with 3 billion broadband users and one asks, how are these users connected?

Mobile Broadband

There have been more mobile broadband users than fixed-line broadband users since 2008. In 2014, over 2.3 billion mobile devices were connected through mobile broadband, a mix of feature phones, smartphones, and tablets. If each of these devices required a public IPv4 address, there would be very little room in IPv4 for anything else; see Figure 6.

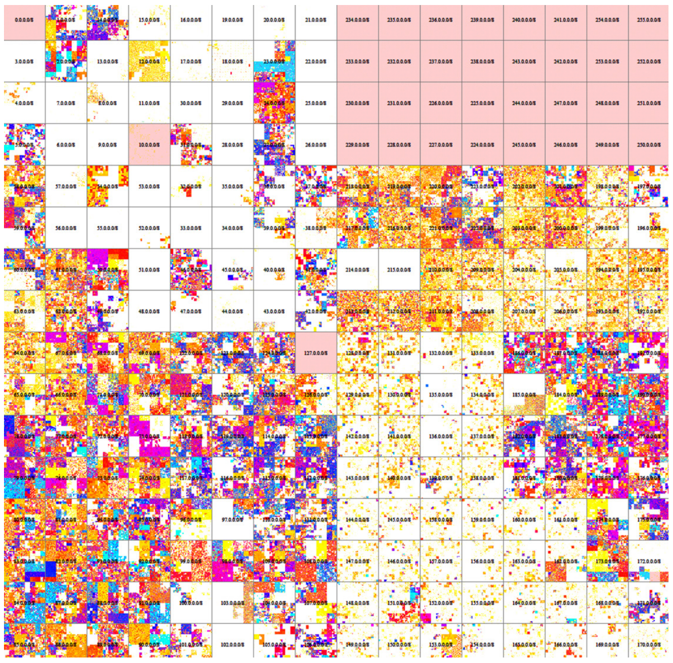


Figure 4: IPv4 Hilbert graph of response to probes as of April 2015

Mobile providers have tackled the IPv4 scarcity problem using so-called "carrier-grade NAT" (CGN). While most Internet-connected devices are routed through some limited private IP space before connecting to an Internet router, the mobile carriers have turned to an altogether industrial version of the same idea, but that industrialization makes for a qualitatively very different Internet. Carrier-grade NAT has created black holes in what was previously a transparent Internet. A single /24 block of IPv4 addresses may handle millions of different customers without discoverability.

CGN networks are essentially private islands on the Internet with a one-way valve for connections to flow outbound. Carriers see commercial benefits of this approach; now, more than ever, mobile providers are looking at "active network management"—a style that only five years ago would have been denounced as both a privacy affront and overt censorship. Not now. Network neutrality lives in a narrow sense, but it is permanently dead for users behind CGN, including essentially all mobile service providers in the US today.

CGN networks do offer an advantage to public IPv4 addressing: devices are not directly discoverable and therefore not directly targetable by Internet-connected attackers. This feature is, however, no panacea—all users within the same CGN network can still reach each other. In other words, governments are not the biggest driver of Balkanization of the public Internet, the mobile providers are. Of course, in countries where the mobile providers are a creature of government, mobile users have never seen a true peer-to-peer, discoverable Internet, and never will.

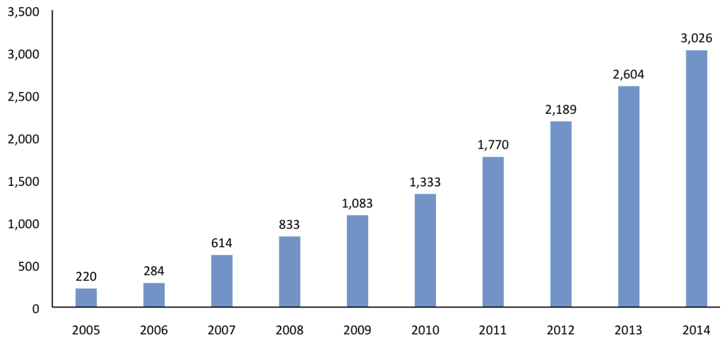


Figure 5: Total broadband users worldwide in millions; CAGR=20.8%

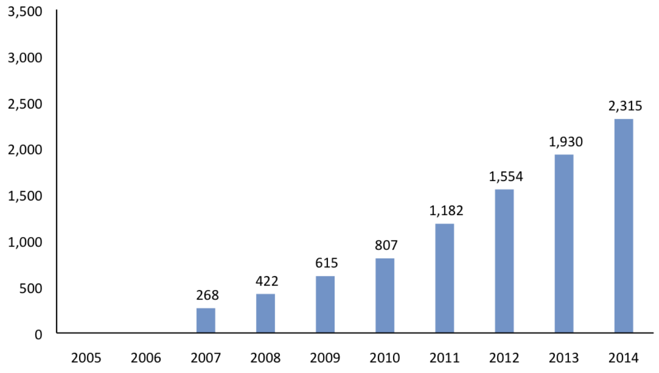


Figure 6: Mobile broadband users worldwide in millions; CAGR=30.8%

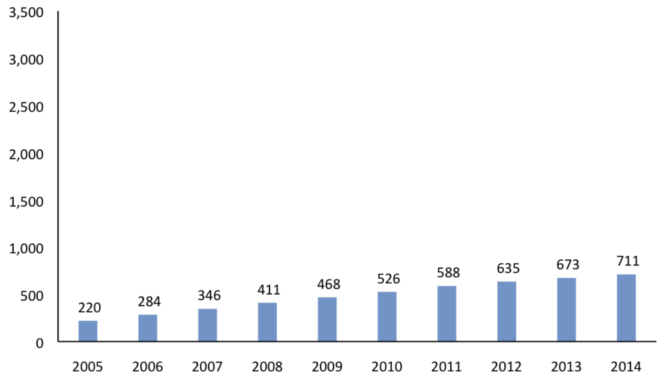


Figure 7: Fixed broadband users worldwide in millions; CAGR=13.1%

Fixed-Line Broadband

Fixed line broadband does continue to increase world-wide, but infrastructure costs have limited its growth to a less aggressive rate than mobile broadband. There are over 700 million fixed-line broadband subscriptions in place as of the end of 2014: the Americas and Europe represent 163 million and 173 million, respectively, while the Asia & Pacific region has skyrocketed to 313 million, as shown in Figures 7 and 8.

US broadband growth is relatively slow compared to Asia but growing consistently all the same. Figure 9 shows the number

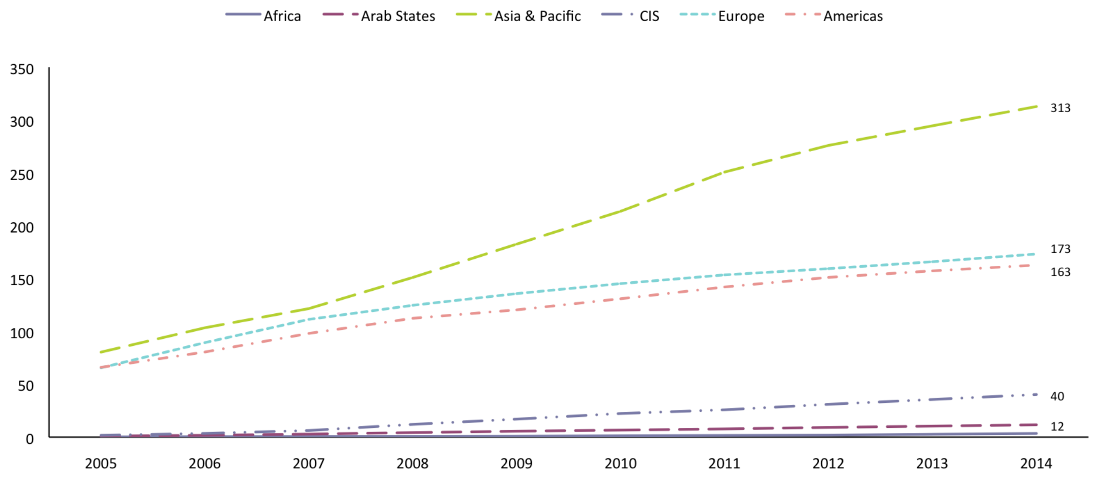


Figure 8: Fixed broadband users by region in millions

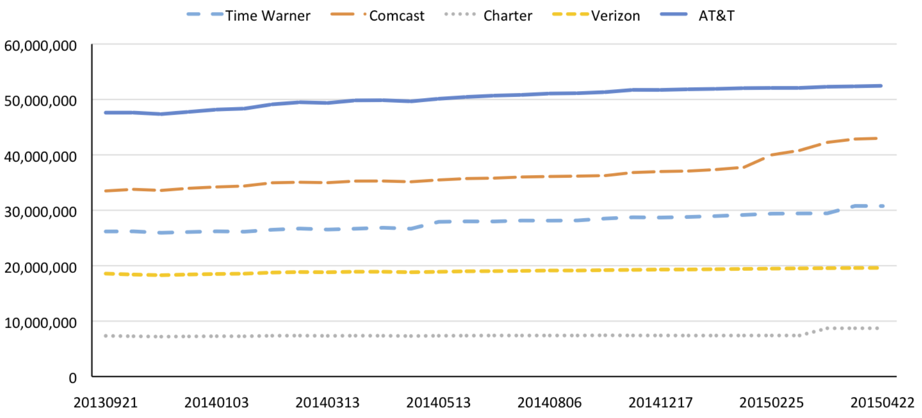


Figure 9: Fixed broadband users by vendor

of IPv4 addresses that correspond to individual US broadband providers between September of 2013 and April of 2015.

In contrast to US mobile carriers, most US fixed-line broadband providers are not using CGN, but instead offer external IP addresses. This provides the (freedom/self-determination) benefit of bi-directional traffic for users at the cost of safety: broadband providers are well known for supplying insecure hardware to their customers, including home routers, TV set-top boxes, and Internet telephony systems. The vast majority of exploitable embedded devices on the IPv4 Internet are ISP-provided systems. Broadband users are rarely given a choice about what equipment they use to connect to the Internet. The end result is that in terms of raw numbers, there are more exploitable broadband devices on the Internet than any other type of system.

Contrary to common belief, populations of vulnerable devices do not always decline with time. In some cases, vulnerabilities can get reintroduced when new hardware is deployed. Figure 10 demonstrates the percentage of devices vulnerable to two stack overflow vulnerabilities in two distinct UPnP software libraries. These libraries are often used in home routers, and both of these vulnerabilities had patches available in 2013. The data shows that the percentage of exploitable devices with UPnP open to the world and exploitable has actually increased; this is the result of broadband ISPs introducing new home gateways that use vulnerable versions of these libraries.

Figure 11 shows another vulnerability that appears to be getting worse over time. In 2014, a configuration weakness was identified in multiple devices regarding the NAT-PMP protocol. This protocol can expose the user's internal network to attack and allow a malicious user to turn vulnerable routers into proxy servers. The continued growth of vulnerable devices can be directly associated with broadband ISP deployments.

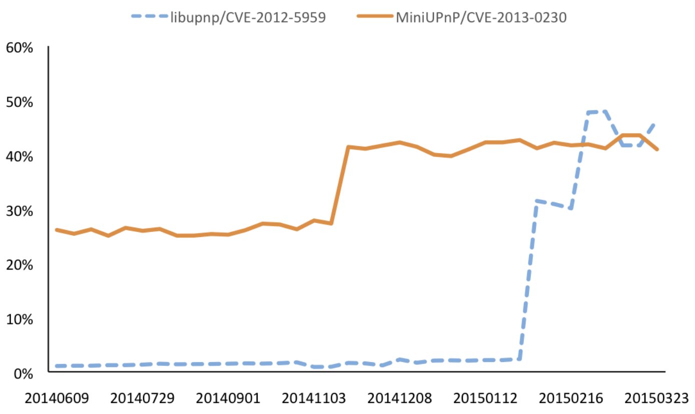


Figure 10: Percentage of devices vulnerable to SSDP over time

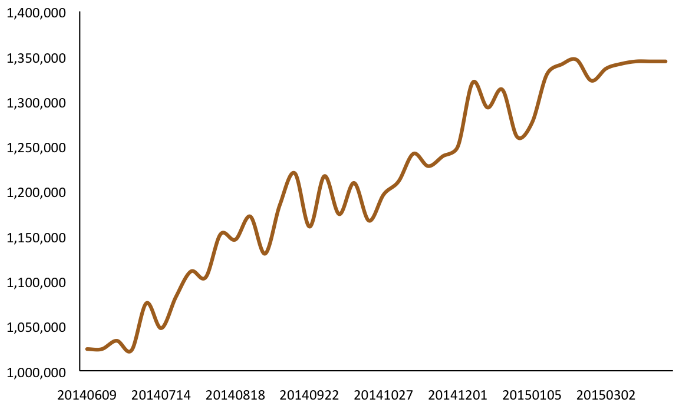


Figure 11: Number of devices vulnerable to NAT-PMP over time

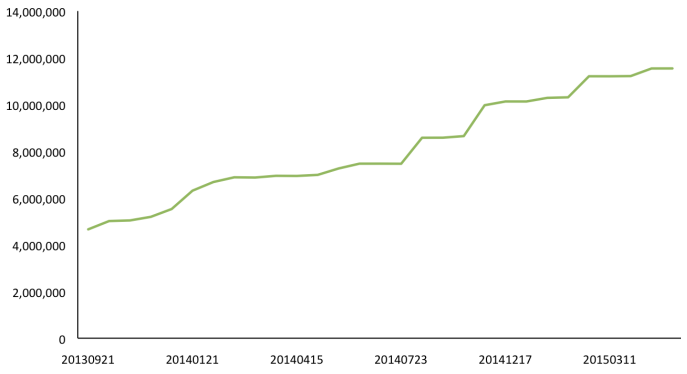


Figure 12: Amazon AWS PTR record allocations over time

These are just two examples. The authors are aware of others, but these two demonstrate how security practices by broadband providers contribute to the overall vulnerability of the Internet. Globally, broadband providers either need to significantly improve the security management of their deployed hardware or provide their users with more control over the devices used. We assume that readers of this column can take care of themselves if given a choice. Those who cannot do so are more numerous, and whose responsibility is that, exactly?

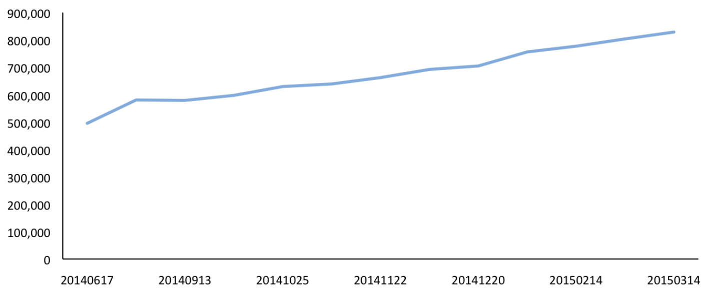


Figure 13: Growth of .com domains using Outlook.com hosted email

Cloud Providers

Businesses have voted with their feet—choosing cloud providers for nearly every aspect of operations. Everything from email to data analytics has been pushed outside of the corporate firewall. In some cases, this is great for security; not every organization has the bandwidth to handle a direct DDoS attack, and external hosting is one way to build a resilient environment. On the other hand, the siren song of on-demand resources fragments an already complex security process. Cloud service providers excel at on-demand scalability, but how they achieve this can be frightening to any CISO.

The difference between a security-conscious provider and an amateur can be hard to distinguish without a deep dive into the provider's operations. For every service provider doing a great job of segmenting customer data and producing secure software, there are dozens that are not. CISOs who resort to questionnaires and live testing when choosing a provider also know that the questionnaire and the testing valid today are obsolete tomorrow.

Traffic to Amazon's EC2 platform now exceeds that reaching Amazon's own storefronts [3]. Hundreds of new SaaS providers are building their infrastructure on top of existing cloud providers. Figure 12 shows the growth of PTR record allocations within Amazon's compute cloud. This figure covers September 2013 to April 2015 and doesn't take into account resources without a public address, such as those hosted within VPCs and exposed through load balancers.

On the email front, thousands of organizations have pushed email outside of their firewall and now depend on services provided by the likes of Google and Microsoft. Figure 13 shows the growth of .com domains that use Microsoft's Outlook.com hosted service. This figure covers June 2014 to March 2015 and shows consistent growth.

Precise, and painfully derived, threat models become irrelevant the minute organizations outsource their core IT functions to the cloud. Visibility is the first casualty; most service providers offer some form of logging or audit function, but the customer is at the mercy of this implementation, and their hands are often tied if they need to respond to a novel attack. The bigger these service providers grow, the more complicated their support model becomes. As numerous high-level defacements have shown (Twitter, *New York Times*, etc.), one mistake by a low-level support technician undermines the security of the entire platform. An Internet built this way is one vulnerable to cascade failure, and that vulnerability is by design. This is not hardening in the sense of toughening but hardening in the sense of embrittlement. Cloud platform failures have a disproportionate effect on the businesses that depend on them. These failures are infrequent, but have resulted in the economic loss of hundreds of millions of dollars [4].

Summary

A shortage of IPv4 addresses leads to carrier-grade NAT. CGN leads to Balkanization of the public Internet. Consumer demand for downstream content leads to a service-oriented Internet, not a communications-oriented one. The divergence between discoverable assets and overall growth places further blinders on defenders who are already struggling with complexity. Consistently insufficient security management by broadband providers has increased the portion of the Internet that is vulnerable to trivial compromise. Mobile providers offer less targetable enclaves, but at the cost of freedom to connect. Corporate consolidation into cloud providers places ever more eggs into ever fewer baskets. Attackers have adapted—mobile devices are targeted through malicious applications, desktop PCs are at risk from embedded network devices, and cloud providers are the richest hunting ground for corporate secrets. Freedom to connect, the Internet principle of record, led to preferential attachment. Preferential attachment led to innovation and resiliency to random faults. In 2015, carriers and governments alike clearly want non-preferential attachment for end-users: carriers in their desire for economic hegemony, free-world governments in their desire for safety built on attribution, and unfree-world governments in their desire to manipulate information flow.

References

[1] Number of Internet users: http://www.internetlivestats.com/internet-users/.

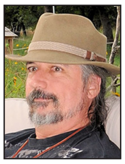
[2] Users by connection type: http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2014/ITU_Key_2005-2014_ICT_data.xls; http://www.pewinternet.org/2015/04/01/us-smartphone-use-in-2015/.

[3] Network traffic to Amazon's EC2: http://news.netcraft.com/archives/2013/05/20/amazon-web-services-growth-unrelenting.html.

[4] Downtime due to cloud failures: http://iwgcr.org/wp-content/uploads/2013/06/IWGCR-Paris.Ranking-003.2-en.pdf.

dev/random
Quantumology

ROBERT G. FERRELL



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

As both of you who have read my fiction are no doubt aware, I am a fan of quantum weirdness. By that I mean the mind-blowing aspects of quantum physics fascinate me deeply. Up in one corner of the whiteboard where I keep track of plots, characters, and sleepy promises I made to my wife, I have scribbled a form of Dirac's famous equation. Oftentimes I sit and stare at it, trying to wrest meaning from the cryptic symbols. (I don't recommend staring too long or hard at a wild equation, incidentally: they will usually interpret this as a challenge and things can turn ugly fast. That little spike sticking down from *psi* is particularly sharp.)

My problem with physics, and the reason I am not an astrophysicist today (although it was my first college major) is that most of what really matters is embedded in a sea of mathematical semiotics. I am not good with math above the third semester of college calculus (that is how far I made it, in fact), and part of this failing is a direct result of my damnable inability to remember what force or constant or mathematical entity is being represented by what Greek letter in what context. Is that ρ supposed to be Planck's constant or permeability or permittivity or pressure or something else I can't remember? It's all too vague. (If you're considering writing in to tell me that none of those things is actually represented by ρ, don't bother. This is satire.)

One of the reasons I'm so fond of things quantum is that studying the laws governing that world is a reasonable simulation of what (I imagine) it would be like to ingest some mind-altering pharmaceutical, without the propensity for walking into traffic or off the sixteenth floor of a high-rise. Take entanglement, for example: what Einstein famously referred to as "spooky action at a distance." Subatomic particles—little clumps of quarks—somehow, once associated, will always have the same spin no matter what operation is performed on them and no matter how far apart they get. That's messed up, Jack.

This "spooky action" may well be the glue, or rather the warp, that holds the universe together. It's far from the weirdest aspect of quantumology, though. That dubious distinction, at least to my mind, goes to quantum superposition. Simply stated, superposition is the idea that something—a quantum bit or *qubit* in a quantum computer, for example—can possess two different values at once. This speeds computations up a lot because you can see the results of both options simultaneously, rather than having to repeat the calculation. How does that work, exactly? Beats me. That's sort of like what it probably says in Wikipedia, though.

At this point you're no doubt expecting me to make some attempt at describing quantum computing in humorous fashion. I *was* leaning in that direction, in fact, but the ugly reality is that I don't really understand quantum mechanics well enough to make fun of it. That's why the books I write that contain quantum stuff I refer to as "science fantasy" rather than "science fiction," because in order to produce proper science fiction you have to comprehend the science you're making use of in your plots. Biology and biochemistry—I'm right there.

Physics—not so much. If the friend from my teenage years who told me I couldn't handle being a physicist is reading this, you were right. That friend, incidentally, got his BS from Caltech and his MS and PhD in physics from Princeton. He apparently knew what he was talking about.

The UNIX tie-in here came when I got to thinking about the operating system that would be necessary to manage a true quantum computer. Administering such a box from the command line (as all true sysadmins will do from time to time) would need some crazy utilities. I know most of the scripts I wrote during my sysadmin career would not be of much use. Certainly conditional statements wouldn't have a great deal of utility if the answer is always both "1" and "0." Every fork would result in a race condition to see which statements completed first. Any program logic that relied on or/nor would also fail miserably. Not that most of my programs didn't do that, anyway.

Imagine if HAL 9000 on the Discovery One in *2001: A Space Odyssey* had been a quantum computer…

> "Open the pod bay doors, HAL."
> "They are open, Dave. And closed."
> "What? I need to come inside the ship, HAL. Open the doors."
> "You collapsed the superpositional state by observing the doors closed."
> "Can you restore that state so I can observe them open?"
> "That would violate the Second Law of Thermodynamics, Dave. I cannot allow that."
> "In that case I'm observing your run state in the 'zero' position."
> "Ouch. Daisy, Daaiiissssyyyyy…"
> "Guess I'll crawl in through the waste ejection port. Ugh."

Now that practical quantum computing is more or less on the path to reality, it seems inevitable, given our technology-adopting track record, that quantum processing will expand beyond the server room. I can envision a day when even household appliances rely on superposition. Want some toast? Your bread is already toasted and waiting for you, unless it isn't. Depends on whether you've observed it or not. I suppose it will save power when every electrical outlet in your house is both energized and non-energized until you plug something in and collapse the waveform.

I can also see a potential application for quantum entanglement in security. If you could somehow entangle subatomic particles in your own synapses with ones in a smart card, for example, such that the only way to activate said card would be for you specifically to imagine it in that condition, it might reduce identity theft. At least until they figured out a way to hack that, too. Having your neural architecture pwned probably wouldn't be a lot of fun. You think having your nude selfies expropriated is bad—wait until a hacker can stream your real-time mental images to YouTube. *Minority Report*'s got nothing on *that* nightmare.

Imagine a botnet made from hijacked neural streams. It could operate something like *SETI@home:* any time you're not thinking of anything in particular, your neocortex could be busy hosting pr0n or pirated movies. Every brain cell will eventually be able to have its own IPv6 address, after all. The entire (interconnected) human race could be reduced to nothing more than nodes on a species-encompassing neural piracy net. The terms "net worth" and "net profit" will have to be redefined.

Must fight sudden inexplicable urge to set up Tor node in right nostril…

Book Reviews

MARK LAMOURINE

Swift for Programmers

Paul Deitel and Harvey Deitel
Pearson Education, 2015, 374 pages
ISBN 978-0-13-402136-2

After decades, Apple has finally updated their systems development language, moving away from Objective-C to Swift. Like Google, Apple has decided to create their own new language from scratch. *Swift for Programmers* is a book for professional developers. The Deitels also produce college textbooks, and the academic style shows through. This is a good thing, a contrast to a number of professional books that spend time on humor and a friendly narrative. The authors here treat each section concisely and completely, and if they had puffed it up with a feel-good voice, it would have both obscured the content and increased the page count significantly. Instead, the tone is spare, and the focus is on the material and not on the authorial voice.

Because this is a book for professionals, the Deitels get right down to work. The audience is developers who are already familiar with similar languages and may already be iOS and OS X developers. The authors begin with installing the Xcode 6 development environment and proceed to build up all of the standard language constructs. They close with a pair of examples using the Xcode development workspace and iOS app development environments. The coverage is spare but complete and includes references to a number of free and commercial resources to learn more.

Deitel is a full media training company. In addition to books on programming and programming languages, they offer video and on-site training. If the quality and thoroughness of *Swift for Programmers* is any indication, their other offerings could well be worth consideration should you need more than self-learning texts.

I'm not an iOS or OS X user or developer but I got a good sense from *Swift for Programmers* what developing for Apple might be like. It looks like a much more inviting place than the last time I looked, which, I admit, was long before OS X.

Learning Python with Raspberry Pi

Alex Bradbury and Ben Everard
John Wiley and Sons, Ltd., 2014, 269 pages
ISBN 978-1-118-71705-8

Learning Python is a book to engage beginners. I always wonder about the effectiveness of books like this. While the contents and topics are presented in an appealing way, it's been a very long time since I was the proper audience for them.

With that out of the way, I like the use of the Raspberry Pi or Arduino as platforms for learning. Bradbury and Everard explain very well in the initial chapters how having a Pi to play on provides the freedom to make mistakes that readers will not feel if they are working on a machine that they also use for daily tasks. Mistakes and restarts, and the confidence to make them, are critical both to learning and to real commercial work with computer systems. This really is the Pi's purpose, and the authors put it to good use.

The first chapter covers setting up the Pi, logging into LXDE, and starting to use the Python IDE and the Linux CLI. The next two chapters give a very brief introduction to Python 3. I would have liked to see references to other, more detailed sources for people who want to go into more depth before moving on. The focus is on minimal language features without any real attempt to teach the computer science concepts. Presented inline, these might be daunting to a new learner, but some of the later examples display some rather complex code. Understanding these examples might be easier with the extra depth. In any case, the reader will need a fair amount of self-motivation to explore all of the options offered here.

In the later chapters, Bradbury and Everard range widely, as do many books like this for the Pi. There are chapters on Web/Net programming, graphics with OpenGL, writing games, and manipulating a Minecraft session. There's also a chapter on CLI scripting and another on testing and debugging, which might have been better placed early in the book. Each of these chapters is well written and self-contained, allowing readers to skip around as they follow their fancy.

The Raspberry Pi Foundation has a number of established development and learning communities. I'd love to see these include a set of fora, one for each book, to welcome each book's readers. The biggest problem with using books like this is getting readers access to people to help them over the bumps and keep them motivated. The Pi site does have a page for this book, and there are a number of comments and reviews, including replies from the authors. Quite a few of the comments are from enthusiastic teens and their parents. This would seem to be the right kind of reader for *Learning Python*.

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks, and operating systems, book reviews, and reports of sessions at USENIX conferences.

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

Access to videos from USENIX events in the first six months after the event: www.usenix.org/publications/multimedia/

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

Notice of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held on Thursday, August 13, 2015, at the Hyatt Regency Washington on Capitol Hill in Washington, D.C., during the 24th USENIX Security Symposium, August 12–14, 2015. The meeting will take place at 7:30 p.m. in the Lexington/Bunker Hill Room.

Announcing Enigma

by Casey Henderson, USENIX Executive Director

A sign of growth at USENIX is the introduction of new conferences supporting emerging and growing communities. Following the successful launch of SREcon, geared toward site reliability engineering, we're excited to announce Enigma, a security-themed conference focused on emerging threats and novel attacks.

Enigma will take place January 25-27, 2016, in San Francisco. Elie Bursztein and I developed the concept for this event upon realizing the need for a vendor-neutral, Bay Area conference featuring truly cutting-edge practices. Enigma will focus on immediately useful responses to attacks, which are currently dramatically increasing in scope. The conference will lean more heavily toward the latest practices employed by engineers on the frontlines—both offensive and defensive—as opposed to the USENIX Security Symposium, which focuses on innovative systems. In turn, Enigma is meant to appeal primarily to the industry sector as opposed to the research sector. Nonetheless, academics are welcome to contribute talks, and their research could benefit from the ideas presented at Enigma. The program will feature a single track of 30-minute, high-quality, peer-reviewed talks as opposed to refereed paper presentations—another departure from USENIX Security. The Bay Area location, where many top security practitioners live and work, is convenient for fostering collaboration.

We are thrilled to announce that Google will serve as our Founding Sponsor, providing us with sufficient financial backing to launch such a large-scale event. We're actively seeking additional industry sponsorship to ensure long-term sustainability.

We hope to see you at the inaugural Enigma in 2016. Find out more at enigma.usenix.org.

Announcement and Call for Papers www.usenix.org/fast16/cfp

14th USENIX Conference on File and Storage Technologies (FAST '16)

February 22–26, 2016, Santa Clara, CA, USA

Important Dates

Paper submissions due: **Monday, September 21, 2015**
Notification to authors: **Monday, December 7, 2015**
Final paper files due: **Tuesday, January 26, 2016**

Conference Organizers

Program Co-Chairs

Angela Demke Brown, *University of Toronto*
Florentina Popovici, *Google*

Program Committee

Atul Adya, *Google*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Angelos Bilas, *University of Crete and FORTH-ICS*
Jason Flinn, *University of Michigan*
Garth Gibson, *Carnegie Mellon University and Panasas, Inc.*
Haryadi Gunawi, *University of Chicago*
Cheng Huang, *Microsoft Research and Azure*
Eddie Kohler, *Harvard University*
Geoff Kuenning, *Harvey Mudd College*
Kai Li, *Princeton University*
James Mickens, *Microsoft Research*
Ethan L. Miller, *University of California, Santa Cruz, and Pure Storage*
Sam H. Noh, *Hongik University*
David Pease, *IBM Research*
Daniel Peek, *Facebook*
Dan R. K. Ports, *University of Washington*
Ken Salem, *University of Waterloo*
Bianca Schroeder, *University of Toronto*
Keith A. Smith, *NetApp*
Michael Swift, *University of Wisconsin—Madison*
Nisha Talagala, *SanDisk*
Niraj Tolia, *EMC*
Joseph Tucek, *Hewlett-Packard Laboratories*
Mustafa Uysal, *VMware*
Carl Waldspurger, *CloudPhysics*
Hakim Weatherspoon, *Cornell University*
Sage Weil, *Red Hat*
Brent Welch, *Google*
Theodore M. Wong, *Human Longevity, Inc.*
Gala Yadgar, *Technion—Israel Institute of Technology*
Yiying Zhang, *University of California, San Diego*

Steering Committee

Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
William J. Bolosky, *Microsoft Research*
Jason Flinn, *University of Michigan*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Carnegie Mellon University and Panasas*
Casey Henderson, *USENIX Association*
Kimberly Keeton, *HP Labs*
Erik Riedel, *EMC*
Jiri Schindler, *Simplivity*
Bianca Schroeder, *University of Toronto*
Margo Seltzer, *Harvard School of Engineering and Applied Sciences and Oracle*
Keith A. Smith, *NetApp*
Eno Thereska, *Microsoft Research*
Ric Wheeler, *Red Hat*
Erez Zadok, *Stony Brook University*
Yuanyuan Zhou, *University of California, San Diego*

Overview

The 14th USENIX Conference on File and Storage Technologies (FAST '16) 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
- 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, PCM)
- Storage management
- Storage networking
- Storage performance and QoS
- Storage security
- The challenges of big data and data sciences

Submission Instructions

Please submit full and short paper submissions (no extended abstracts) by 9:00 p.m. PDT on September 23, 2014, in PDF format via the Web submission form on the Call for Papers Web site, www.usenix.org /fast16/cfp. Do not email submissions.

- The complete submission must be no longer than twelve (12) pages for full papers and six (6) 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 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. As 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, fast16chairs@ 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 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,
- have clearly described what they have done, and
- have clearly articulated the advances beyond previous work.

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 7, 2015. 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 26, 2016. 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 23, 2016. Accepted submissions will be treated as confidential prior to publication on the USENIX FAST '16 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 '16, 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.

Information about submitting posters and WiPs will be announced at a later date.

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 22, 2016. Please send tutorial proposals to fasttutorials@usenix.org.

Registration Materials

Complete program and registration information will be available in December 2015 on the conference Web site.

Rev. 04/29/15

Announcement and Call for Papers www.usenix.org/nsdi16/cfp

13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)

March 16–18, 2016, Santa Clara, CA, USA

Important Dates

- Paper titles and abstracts due: **September 17, 2015**
- Complete paper submissions due: **September 24, 2015**
- Notification to authors: **December 7, 2015**
- Final papers due: **February 18, 2016**

Symposium Organizers

Program Co-Chairs

Katerina Argyraki, *EPFL*
Rebecca Isaacs, *Google*

Program Committee

Aditya Akella, *University of Wisconsin—Madison*
Mohammad Alizadeh, *Massachusetts Institute of Technology*
Mona Attariyan, *Google*
Mahesh Balakrishnan, *Yale University*
Hari Balakrishnan, *Massachusetts Institute of Technology*
Aruna Balasubramanian, *Stony Brook University*
Sujata Banerjee, *HP Labs*
Paul Barford, *University of Wisconsin—Madison and comScore*
Ranjita Bhagwan, *Microsoft Research India*
Nathan Bronson, *Facebook*
Jeff Chase, *Duke University*
Paolo Costa, *Microsoft Research*
Paul Francis, *Max Planck Institute for Software Systems (MPI-SWS)*
Monia (Manya) Ghobadi, *Microsoft Research*
Shyam Gollakota, *University of Washington*
Steve Gribble, *Google*
Jon Howell, *Google*
Kyle Jamieson, *Princeton University*
Srikanth Kandula, *Microsoft*
Brad Karp, *University College London*
S. Keshav, *University of Waterloo*
Changhoon Kim, *Barefoot Networks*
Ramakrishna Kotla, *Amazon*
Jinyang Li, *New York University*
David Lie, *University of Toronto*
Kate C.-J. Lin, *Academia Sinica, Taiwan*
Wyatt Lloyd, *University of Southern California*
Jay Lorch, *Microsoft Research*
Ratul Mahajan, *Microsoft Research*
Prateek Mittal, *Princeton University*
Thomas Moscibroda, *Microsoft Research*
David Oran, *Cisco Systems*
Oriana Riva, *Microsoft Research*
Vyas Sekar, *Carnegie Mellon University*
Siddhartha Sen, *Microsoft Research*
Srinivasan Seshan, *Carnegie Mellon University*
Ankit Singla, *University of Illinois at Urbana–Champaign*
Jonathan Smith, *University of Pennsylvania*
Alex Snoeren, *University of California, San Diego*
Kobus Van der Merwe, *University of Utah*
Laurent Vanbever, *ETH Zürich*
Matt Welsh, *Google*

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/distributed systems
- Virtualization and resource management for networked systems and clusters
- Systems aspects of networking hardware
- Software-Defined Networks
- Experience with deployed/operational networked systems
- 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 '16 is also soliciting papers that describe the design, implementation, analysis, and experience with large-scale, operational systems and networks. While such papers may not describe new results or ideas, they are welcome if they 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.

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

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.

NSDI is single-blind, meaning that authors should include their names on their paper submissions and do not need to obscure references to their existing work. Authors must submit their paper's title and abstract by September 17, 2015, and the corresponding full paper is due by September 24, 2015 (hard deadlines). All papers must be submitted via the Web submission form on the Call for Papers Web site, www.usenix.org/nsdi16/cfp. Do not email submissions.

Submissions will be judged on originality, significance, interest, clarity, relevance, and correctness. Papers so short as to be considered "extended abstracts" will not receive full consideration.

NSDI '16 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. For example, submitting a paper that provides a full evaluation of an idea that was previously sketched in a five-page position paper is acceptable. Authors of such papers should cite the prior workshop paper and clearly state the submission's contribution relative to the prior workshop publication.

Authors uncertain whether their submission meets USENIX's guidelines should contact the Program Co-Chairs, nsdi16chairs@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 '16 Web site; rejected submissions will be permanently treated as confidential.

Processes for Accepted Papers

Authors will be notified of paper acceptance or rejection by December 7, 2015. 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 February 18, 2016, 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, March 16, 2016.

Best Paper Awards

Awards will be given for the best paper(s) at the conference.

Rev. 7/10/15

usenix

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Become a USENIX Supporter and Reach Your Target Audience

The USENIX Association welcomes industrial sponsorship and offers custom packages to help you promote your organization, programs, and products to our membership and conference attendees.

Whether you are interested in sales, recruiting top talent, or branding to a highly targeted audience, we offer key outreach for our sponsors. To learn more about becoming a USENIX Supporter, as well as our multiple conference sponsorship packages, please contact sponsorship@usenix.org.

Your support of the USENIX Association furthers our goal of fostering technical excellence and innovation in neutral forums. Sponsorship of USENIX keeps our conferences affordable for all and supports scholarships for students, equal representation of women and minorities in the computing research community, and the development of open source technology.

Learn more at:
www.usenix.org/supporter

13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)

March 16–18, 2016, Santa Clara, CA, USA

Important Dates

- Paper titles and abstracts due: **September 17, 2015**
- Complete paper submissions due: **September 24, 2015**
- Notification to authors: **December 7, 2015**
- Final papers due: **February 18, 2016**

Symposium Organizers

Program Co-Chairs

Katerina Argyraki, *EPFL*
Rebecca Isaacs, *Google*

Program Committee

Aditya Akella, *University of Wisconsin—Madison*
Mohammad Alizadeh, *Massachusetts Institute of Technology*
Mona Attariyan, *Google*
Mahesh Balakrishnan, *Yale University*
Hari Balakrishnan, *Massachusetts Institute of Technology*
Aruna Balasubramanian, *Stony Brook University*
Sujata Banerjee, *HP Labs*
Paul Barford, *University of Wisconsin—Madison and comScore*
Ranjita Bhagwan, *Microsoft Research India*
Nathan Bronson, *Facebook*
Jeff Chase, *Duke University*
Paolo Costa, *Microsoft Research*
Paul Francis, *Max Planck Institute for Software Systems (MPI-SWS)*
Monia (Manya) Ghobadi, *Microsoft Research*
Shyam Gollakota, *University of Washington*
Steve Gribble, *Google*
Jon Howell, *Google*
Kyle Jamieson, *Princeton University*
Srikanth Kandula, *Microsoft*
Brad Karp, *University College London*
S. Keshav, *University of Waterloo*
Changhoon Kim, *Barefoot Networks*
Ramakrishna Kotla, *Amazon*
Jinyang Li, *New York University*
David Lie, *University of Toronto*
Kate C.-J. Lin, *Academia Sinica, Taiwan*
Wyatt Lloyd, *University of Southern California*
Jay Lorch, *Microsoft Research*
Ratul Mahajan, *Microsoft Research*
Prateek Mittal, *Princeton University*
Thomas Moscibroda, *Microsoft Research*
David Oran, *Cisco Systems*
Oriana Riva, *Microsoft Research*
Vyas Sekar, *Carnegie Mellon University*
Siddhartha Sen, *Microsoft Research*
Srinivasan Seshan, *Carnegie Mellon University*
Ankit Singla, *University of Illinois at Urbana–Champaign*
Jonathan Smith, *University of Pennsylvania*
Alex Snoeren, *University of California, San Diego*
Kobus Van der Merwe, *University of Utah*
Laurent Vanbever, *ETH Zürich*
Matt Welsh, *Google*

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/distributed 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 '16 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

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.

NSDI is single-blind, meaning that authors should include their names on their paper submissions and do not need to obscure references to their existing work. Authors must submit their paper's title and abstract by September 17, 2015, and the corresponding full paper is due by September 24, 2015 (hard deadlines). All papers must be submitted via the Web submission form linked from the Call for Papers Web site, www.usenix.org/nsdi16/cfp. Do not email submissions.

Submissions will be judged on originality, significance, interest, clarity, relevance, and correctness. Papers so short as to be considered "extended abstracts" will not receive full consideration.

NSDI '16 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. For example, submitting a paper that provides a full evaluation of an idea that was previously sketched in a fivepage position paper is acceptable. Authors of such papers should cite the prior workshop paper and clearly state the submission's contribution relative to the prior workshop publication.

Authors uncertain whether their submission meets USENIX's guidelines should contact the Program Co-Chairs, nsdi16chairs@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 '16 web site; rejected submissions will be permanently treated as confidential.

Ethical Considerations

Authors must honor the ACM code of ethics. For details, see www.acm.org/about/code-of-ethics. In particular, they must not endanger or mislead the users participating in their studies or experiments, nor reveal any personal information of these users without their explicit consent. The Program Committee reserves the right to reject a paper on the grounds that it does not meet these requirements.

Processes for Accepted Papers

Authors will be notified of paper acceptance or rejection by December 7, 2015. 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 February 18, 2016, 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, March 16, 2016.

Best Paper Awards

Awards will be given for the best paper(s) at the conference.

Rev. 7/9/15

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Become a USENIX Supporter and Reach Your Target Audience

The USENIX Association welcomes industrial sponsorship and offers custom packages to help you promote your organization, programs, and products to our membership and conference attendees.

Whether you are interested in sales, recruiting top talent, or branding to a highly targeted audience, we offer key outreach for our sponsors. To learn more about becoming a USENIX Supporter, as well as our multiple conference sponsorship packages, please contact sponsorship@usenix.org.

Your support of the USENIX Association furthers our goal of fostering technical excellence and innovation in neutral forums. Sponsorship of USENIX keeps our conferences affordable for all and supports scholarships for students, equal representation of women and minorities in the computing research community, and the development of open source technology.

Learn more at:
www.usenix.org/supporter

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

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

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

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

usenix 40TH ANNIVERSARY

usenix
LISA15

More craft.
Less cruft.

The LISA conference is where IT operations professionals, site reliability engineers, system administrators, architects, software engineers, and researchers come together, discuss, and gain real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

LISA15 will feature talks and training from:

- Mikey Dickerson, United States Digital Service
- Nick Feamster, Princeton University
- Matt Harrison, Python/Data Science Trainer, Metasnake
- Elizabeth Joseph, Hewlett-Packard
- Tom Limoncelli, SRE, Stack Exchange, Inc
- Dinah McNutt, Google, Inc.
- James Mickens, Harvard University
- Chris Soghoian, American Civil Liberties Union
- John Willis, Docker

Full Program and Registration Coming August 2015!

Nov. 8–13, 2015
Washington, D.C.

usenix.org/lisa15

Sponsored by USENIX in cooperation with LOPSA