Our Software Dependency Problem

Russ Cox
January 23, 2019

research.swtch.com/deps

For decades, discussion of software reuse was far more common than actual software reuse. Today, the situation is reversed: developers reuse software written by others every day, in the form of software dependencies, and the situation goes mostly unexamined.

My own background includes a decade of working with Google's internal source code system, which treats software dependencies as a first-class concept,[1] and also developing support for dependencies in the Go programming language.[2]

Software dependencies carry with them serious risks that are too often overlooked. The shift to easy, fine-grained software reuse has happened so quickly that we do not yet understand the best practices for choosing and using dependencies effectively, or even for deciding when they are appropriate and when not. My purpose in writing this article is to raise awareness of the risks and encourage more investigation of solutions.

What is a dependency?

In today's software development world, a *dependency* is additional code that you want to call from your program. Adding a dependency avoids repeating work already done: designing, writing, testing, debugging, and maintaining a specific unit of code. In this article we'll call that unit of code a *package*; some systems use terms like library or module instead of package.

Taking on externally-written dependencies is an old practice: most programmers have at one point in their careers had to go through the steps of manually downloading and installing a required library, like C's PCRE or zlib, or C++'s Boost or Qt, or Java's JodaTime or JUnit. These packages contain high-quality, debugged code that required significant expertise to develop. For a program that needs the functionality provided by one of these packages, the tedious work of manually downloading, installing, and updating the package is easier than the work of redeveloping that functionality from scratch. But the high fixed costs of reuse mean that manually-reused packages tend to be big: a tiny package would be easier to reimplement.

A *dependency manager* (sometimes called a package manager) automates the downloading and installation of dependency packages. As dependency managers make individual packages easier to download and install, the lower fixed costs make smaller packages economical to publish and reuse.

For example, the Node.js dependency manager NPM provides access to over 750,000 packages. One of them, `escape-string-regexp`, provides a single function that escapes regular expression operators in its input. The entire implementation is:

```
var matchOperatorsRe = /[|\\{}()[\]^$+*?.]/g;

module.exports = function (str) {
    if (typeof str !== 'string') {
        throw new TypeError('Expected a string');
    }
    return str.replace(matchOperatorsRe, '\\$&');
};
```

Before dependency managers, publishing an eight-line code library would have been unthinkable: too much overhead for too little benefit. But NPM has driven the overhead approximately to zero, with the result that nearly-trivial functionality can be packaged and reused. In late January 2019, the `escape-string-regexp` package is explicitly depended upon by almost a thousand other NPM packages, not to mention all the packages developers write for their own use and don't share.

Dependency managers now exist for essentially every programming language. Maven Central (Java), Nuget (.NET), Packagist (PHP), PyPI (Python), and RubyGems (Ruby) each host over 100,000 packages. The arrival of this kind of fine-grained, widespread software reuse is one of the most consequential shifts in software development over the past two decades. And if we're not more careful, it will lead to serious problems.

What could go wrong?

A package, for this discussion, is code you download from the internet. Adding a package as a dependency outsources the work of developing that code—designing, writing, testing, debugging, and maintaining—to someone else on the internet, someone you often don't know. By using that code, you are exposing your own program to all the failures and flaws in the dependency. Your program's execution now literally *depends* on code downloaded from this stranger on the internet. Presented this way, it sounds incredibly unsafe. Why would anyone do this?

We do this because it's easy, because it seems to work, because everyone else is doing it too, and, most importantly, because it seems like a natural continuation of age-old established practice. But there are important differences we're ignoring.

Decades ago, most developers already trusted others to write software they depended on, such as operating systems and compilers. That software was bought from known sources, often with some kind of support agreement. There was still a potential for bugs or outright mischief,[3] but at least we knew who we were dealing with and usually had commercial or legal recourses available.

The phenomenon of open-source software, distributed at no cost over the internet, has displaced many of those earlier software purchases. When reuse was difficult, there were fewer projects publishing reusable code packages. Even though their licenses typically disclaimed, among other things, any "implied warranties of merchantability and fitness for a particular purpose," the projects built up well-known reputations that often factored heavily into people's decisions about which to use. The commercial and legal support for trusting our software sources was replaced by reputational support. Many common early packages still enjoy good reputations: consider BLAS (published 1979), Netlib (1987), libjpeg (1991), LAPACK (1992), HP STL (1994), and zlib (1995).

Dependency managers have scaled this open-source code reuse model down: now, developers can share code at the granularity of individual functions of tens of lines. This is a major technical accomplishment. There are myriad available packages, and writing code can involve such a large number of them, but the commercial, legal, and reputational support mechanisms for trusting the code have not carried over. We are trusting more code with less justification for doing so.

The cost of adopting a bad dependency can be viewed as the sum, over all possible bad outcomes, of the cost of each bad outcome multiplied by its probability of happening (risk).

$$\text{expected cost} = \sum_{b \in \text{bad outcomes}} \text{cost}(b) \times \text{probability}(b)$$

The context where a dependency will be used determines the cost of a bad outcome. At one end of the spectrum is a personal hobby project, where the cost of most bad outcomes is near zero: you're just having fun, bugs have no real impact other than wasting some time, and even debugging them can be fun. So the risk probability almost doesn't matter: it's being multiplied by zero. At the other end of the spectrum is production software that must be maintained for years. Here, the cost of a bug in a dependency can be very high: servers may go down, sensitive data may be divulged, customers may be harmed, companies may fail. High failure costs make it much more important to estimate and then reduce any risk of a serious failure.

No matter what the expected cost, experiences with larger dependencies suggest some approaches for estimating and reducing the risks of adding a software dependency. It is likely that better tooling is needed to help reduce the costs of these approaches, much as dependency managers have focused to date on reducing the costs of download and installation.

Inspect the dependency

You would not hire a software developer you've never heard of and know nothing about. You would learn more about them first: check references, conduct a job interview, run background checks, and so on. Before you depend on a package you found on the internet, it is similarly prudent to learn a bit about it first.

A basic inspection can give you a sense of how likely you are to run into problems trying to use this code. If the inspection reveals likely minor problems, you can take steps to prepare for or maybe avoid them. If the inspection reveals major problems, it may be best not to use the package: maybe you'll find a more suitable one, or maybe you need to develop one yourself. Remember that open-source packages are published by their authors in the hope that they will be useful but with no guarantee of usability or support. In the middle of a production outage, you'll be the one debugging it. As the original GNU General Public License warned, "The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction."[4]

The rest of this section outlines some considerations when inspecting a package and deciding whether to depend on it.

Design

Is package's documentation clear? Does the API have a clear design? If the authors can explain the package's API and its design well to you, the user, in the documentation, that increases the likelihood they have explained the implementation well to the computer, in the source code. Writing code for a clear, well-designed API is also easier, faster, and hopefully less error-prone. Have the authors documented what they expect from client code in order to make future upgrades compatible? (Examples include the C++[5] and Go[6] compatibility documents.)

Code Quality

Is the code well-written? Read some of it. Does it look like the authors have been careful, conscientious, and consistent? Does it look like code you'd want to debug? You may need to.

Develop your own systematic ways to check code quality. For example, something as simple as compiling a C or C++ program with important compiler warnings enabled (for example, `-Wall`) can give you a sense of how seriously the developers work to avoid various undefined behaviors. Recent languages like Go, Rust, and Swift use an `unsafe` keyword to mark code that violates the type system; look to see how much unsafe code there is. More advanced semantic tools like Infer[7] or SpotBugs[8] are helpful too. Linters are less helpful: you should ignore rote suggestions about topics like brace style and focus instead on semantic problems.

Keep an open mind to development practices you may not be familiar with. For example, the SQLite library ships as a single 200,000-line C source file and a single 11,000-line header, the "amalgamation." The sheer size of these files should raise an initial red flag, but closer investigation would turn up the actual development source code, a traditional file tree with over a hundred C source files, tests, and support scripts. It turns out that the single-file distribution is built automatically from the original sources and is easier for end users, especially those without dependency managers. (The compiled code also runs faster, because the compiler can see more optimization opportunities.)

Testing

Does the code have tests? Can you run them? Do they pass? Tests establish that the code's basic functionality is correct, and they signal that the developer is serious about keeping it correct. For example, the SQLite development tree has an incredibly thorough test suite with over 30,000 individual test cases as well as developer documentation explaining the testing strategy.[9] On the other hand, if there are few tests or no tests, or if the tests fail, that's a serious red flag: future changes to the package are likely to introduce regressions that could easily have been caught. If you insist on tests in code you write yourself (you do, right?), you should insist on tests in code you outsource to others.

Assuming the tests exist, run, and pass, you can gather more information by running them with run-time instrumentation like code coverage analysis, race detection,[10] memory allocation checking, and memory leak detection.

Debugging

Find the package's issue tracker. Are there many open bug reports? How long have they been open? Are there many fixed bugs? Have any bugs been fixed recently? If you see lots of open issues about what look like real bugs, especially if they have been open for a long time, that's not a good sign. On the other hand, if the closed issues show that bugs are rarely found and promptly fixed, that's great.

Maintenance

Look at the package's commit history. How long has the code been actively maintained? Is it actively maintained now? Packages that have been actively maintained for an extended amount of time are more likely to continue to be maintained. How many people work on the package? Many packages are personal projects that developers create and share for fun in their spare time. Others are the result of thousands of hours of work by a group of paid developers. In general, the latter kind of package is more likely to have prompt bug fixes, steady improvements, and general upkeep.

On the other hand, some code really is "done." For example, NPM's `escape-string-regexp`, shown earlier, may never need to be modified again.

Usage

Do many other packages depend on this code? Dependency managers can often provide statistics about usage, or you can use a web search to estimate how often others write about using the package. More users should at least mean more people for whom the code works well enough, along with faster detection of new bugs. Widespread usage is also a hedge against the question of continued maintenance: if a widely-used package loses its maintainer, an interested user is likely to step forward.

For example, libraries like PCRE or Boost or JUnit are incredibly widely used. That makes it more likely—although certainly not guaranteed—that bugs you might otherwise run into have already been fixed, because others ran into them first.

Security

Will you be processing untrusted inputs with the package? If so, does it seem to be robust against malicious inputs? Does it have a history of security problems listed in the National Vulnerability Database (NVD)?[11]

For example, when Jeff Dean and I started work on Google Code Search[12]—`grep` over public source code—in 2006, the popular PCRE regular expression library seemed like an obvious choice. In an early discussion with Google's security team, however, we learned that PCRE had a history of problems like buffer overflows, especially in its parser. We could have learned the same by searching for PCRE in the NVD. That discovery didn't immediately cause us to abandon PCRE, but it did make us think more carefully about testing and isolation.

Licensing

Is the code properly licensed? Does it have a license at all? Is the license acceptable for your project or company? A surprising fraction of projects on GitHub have no clear license. Your project or company may impose further restrictions on the allowed licenses of dependencies. For example, Google disallows the use of code licensed under AGPL-like licenses (too onerous) as well as WTFPL-like licenses (too vague).[13]

Dependencies

Does the code have dependencies of its own? Flaws in indirect dependencies are just as bad for your program as flaws in direct dependencies. Dependency managers can list all the transitive dependencies of a given package, and each of them should ideally be inspected as described in this section. A package with many dependencies incurs additional inspection work, because those same dependencies incur additional risk that needs to be evaluated.

Many developers have never looked at the full list of transitive dependencies of their code and don't know what they depend on. For example, in March 2016 the NPM user community discovered that many popular projects—including Babel, Ember, and React—all depended indirectly on a tiny package called `left-pad`, consisting of a single 8-line function body. They discovered this when the author of `left-pad` deleted that package from NPM, inadvertently breaking most Node.js users' builds.[14] And `left-pad` is hardly exceptional in this regard. For example, 30% of the 750,000 packages published on NPM depend—at least indirectly—on `escape-string-regexp`. Adapting Leslie Lamport's observation about distributed systems, a dependency manager can easily create a situation in which the failure of a package you didn't even know existed can render your own code unusable.

Test the dependency

The inspection process should include running a package's own tests. If the package passes the inspection and you decide to make your project depend on it, the next step should be to write new tests focused on the functionality needed by your application. These tests often start out as short standalone programs written to make sure you can understand the package's API and that it does what you think it does. (If you can't or it doesn't, turn back now!) It is worth then taking the extra effort to turn those programs into automated tests that can be run against newer versions of the package. If you find a bug and have a potential fix, you'll want to be able to rerun these project-specific tests easily, to make sure that the fix did not break anything else.

It is especially worth exercising the likely problem areas identified by the basic inspection. For Code Search, we knew from past experience that PCRE sometimes took a long time to execute certain regular expression searches. Our initial plan was to have separate thread pools for "simple" and "complicated" regular expression searches. One of the first tests we ran was a benchmark, comparing `pcregrep` with a few other `grep` implementations. When we found that, for one basic test case, `pcregrep` was 70X slower than the fastest `grep` available, we started to rethink our plan to use PCRE. Even though we eventually dropped PCRE entirely, that benchmark remains in our code base today.

Abstract the dependency

Depending on a package is a decision that you are likely to revisit later. Perhaps updates will take the package in a new direction. Perhaps serious security problems will be found. Perhaps a better option will come along. For all these reasons, it is worth the effort to make it easy to migrate your project to a new dependency.

If the package will be used from many places in your project's source code, migrating to a new dependency would require making changes to all those different source locations. Worse, if the package will be exposed in your own project's API, migrating to a new dependency would require making changes in all the code calling your API, which you might not control. To avoid these costs, it makes sense to define an interface of your own, along with a thin wrapper implementing that interface using the dependency. Note that the wrapper should include only what your project needs from the dependency, not everything the dependency offers. Ideally, that allows you to substitute a different, equally appropriate dependency later, by changing only the wrapper. Migrating your per-project tests to use the new interface tests the interface and wrapper implementation and also makes it easy to test any potential replacements for the dependency.

For Code Search, we developed an abstract `Regexp` class that defined the interface Code Search needed from any regular expression engine. Then we wrote a thin wrapper around PCRE implementing that interface. The indirection made it easy to test alternate libraries, and it kept us from accidentally introducing knowledge of PCRE internals into the rest of the source tree. That in turn ensured that it would be easy to switch to a different dependency if needed.

Isolate the dependency

It may also be appropriate to isolate a dependency at run-time, to limit the possible damage caused by bugs in it. For example, Google Chrome allows users to add dependencies—extension code—to the browser. When Chrome launched in 2008, it introduced the critical feature (now standard in all browsers) of isolating each extension in a sandbox running in a separate operating-system process.[15]

An exploitable bug in an badly-written extension therefore did not automatically have access to the entire memory of the browser itself and could be stopped from making inappropriate system calls.[16] For Code Search, until we dropped PCRE entirely, our plan was to isolate at least the PCRE parser in a similar sandbox. Today, another option would be a lightweight hypervisor-based sandbox like gVisor.[17] Isolating dependencies reduces the associated risks of running that code.

Even with these examples and other off-the-shelf options, run-time isolation of suspect code is still too difficult and rarely done. True isolation would require a completely memory-safe language, with no escape hatch into untyped code. That's challenging not just in entirely unsafe languages like C and C++ but also in languages that provide restricted unsafe operations, like Java when including JNI, or like Go, Rust, and Swift when including their "unsafe" features. Even in a memory-safe language like JavaScript, code often has access to far more than it needs. In November 2018, the latest version of the NPM package `event-stream`, which provided a functional streaming API for JavaScript events, was discovered to contain obfuscated malicious code that had been added two and a half months earlier. The code, which harvested large Bitcoin wallets from users of the Copay mobile app, was accessing system resources entirely unrelated to processing event streams.[18] One of many possible defenses to this kind of problem would be to better restrict what dependencies can access.

Avoid the dependency

If a dependency seems too risky and you can't find a way to isolate it, the best answer may be to avoid it entirely, or at least to avoid the parts you've identified as most problematic.

For example, as we better understood the risks and costs associated with PCRE, our plan for Google Code Search evolved from "use PCRE directly," to "use PCRE but sandbox the parser," to "write a new regular expression parser but keep the PCRE execution engine," to "write a new parser and connect it to a different, more efficient open-source execution engine." Later we rewrote the execution engine as well, so that no dependencies were left, and we open-sourced the result: RE2.[19]

If you only need a tiny fraction of a dependency, it may be simplest to make a copy of what you need (preserving appropriate copyright and other legal notices, of course). You are taking on responsibility for fixing bugs, maintenance, and so on, but you're also completely isolated from the larger risks. The Go developer community has a proverb about this: "A little copying is better than a little dependency."[20]

Upgrade the dependency

For a long time, the conventional wisdom about software was "if it ain't broke, don't fix it." Upgrading carries a chance of introducing new bugs; without a corresponding reward—like a new feature you need—why take the risk? This analysis ignores two costs. The first is the cost of the eventual upgrade. In software, the difficulty of making code changes does not scale linearly: making ten small changes is less work and easier to get right than making one equivalent large change. The second is the cost of discovering already-fixed bugs the hard way. Especially in a security context, where known bugs are actively exploited, every day you wait is another day that attackers can break in.

For example, consider the year 2017 at Equifax, as recounted by executives in detailed congressional testimony.[21] On March 7, a new vulnerability in Apache Struts was disclosed, and a patched version was released. On March 8, Equifax received a notice from US-CERT about the need to update any uses of Apache

Struts. Equifax ran source code and network scans on March 9 and March 15, respectively; neither scan turned up a particular group of public-facing web servers. On May 13, attackers found the servers that Equifax's security teams could not. They used the Apache Struts vulnerability to breach Equifax's network and then steal detailed personal and financial information about 148 million people over the next two months. Equifax finally noticed the breach on July 29 and publicly disclosed it on September 4. By the end of September, Equifax's CEO, CIO, and CSO had all resigned, and a congressional investigation was underway.

Equifax's experience drives home the point that although dependency managers know the versions they are using at build time, you need other arrangements to track that information through your production deployment process. For the Go language, we are experimenting with automatically including a version manifest in every binary, so that deployment processes can scan binaries for dependencies that need upgrading. Go also makes that information available at run-time, so that servers can consult databases of known bugs and self-report to monitoring software when they are in need of upgrades.

Upgrading promptly is important, but upgrading means adding new code to your project, which should mean updating your evaluation of the risks of using the dependency based on the new version. As minimum, you'd want to skim the diffs showing the changes being made from the current version to the upgraded versions, or at least read the release notes, to identify the most likely areas of concern in the upgraded code. If a lot of code is changing, so that the diffs are difficult to digest, that is also information you can incorporate into your risk assessment update.

You'll also want to re-run the tests you've written that are specific to your project, to make sure the upgraded package is at least as suitable for the project as the earlier version. It also makes sense to re-run the package's own tests. If the package has its own dependencies, it is entirely possible that your project's configuration uses different versions of those dependencies (either older or newer ones) than the package's authors use. Running the package's own tests can quickly identify problems specific to your configuration.

Again, upgrades should not be completely automatic. You need to verify that the upgraded versions are appropriate for your environment before deploying them.[22]

If your upgrade process includes re-running the integration and qualification tests you've already written for the dependency, so that you are likely to identify new problems before they reach production, then, in most cases, delaying an upgrade is riskier than upgrading quickly.

The window for security-critical upgrades is especially short. In the aftermath of the Equifax breach, forensic security teams found evidence that attackers (perhaps different ones) had successfully exploited the Apache Struts vulnerability on the affected servers on March 10, only three days after it was publicly disclosed, but they'd only run a single `whoami` command.

Watch your dependencies

Even after all that work, you're not done tending your dependencies. It's important to continue to monitor them and perhaps even re-evaluate your decision to use them.

First, make sure that you keep using the specific package versions you think you are. Most dependency managers now make it easy or even automatic to record the cryptographic hash of the expected source code for a given package version and then to check that hash when re-downloading the package on another computer or in a test environment. This ensures that your build use the

same dependency source code you inspected and tested. These kinds of checks prevented the `event-stream` attacker, described earlier, from silently inserting malicious code in the already-released version 3.3.5. Instead, the attacker had to create a new version, 3.3.6, and wait for people to upgrade (without looking closely at the changes).

It is also important to watch for new indirect dependencies creeping in: upgrades can easily introduce new packages upon which the success of your project now depends. They deserve your attention as well. In the case of `event-stream`, the malicious code was hidden in a different package, `flatmap-stream`, which the new `event-stream` release added as a new dependency.

Creeping dependencies can also affect the size of your project. During the development of Google's Sawzall[23]—a JIT'ed logs processing language—the authors discovered at various times that the main interpreter binary contained not just Sawzall's JIT but also (unused) PostScript, Python, and JavaScript interpreters. Each time, the culprit turned out to be unused dependencies declared by some library Sawzall did depend on, combined with the fact that Google's build system eliminated any manual effort needed to start using a new dependency.. This kind of error is the reason that the Go language makes importing an unused package a compile-time error.

Upgrading is a natural time to revisit the decision to use a dependency that's changing. It's also important to periodically revisit any dependency that *isn't* changing. Does it seem plausible that there are no security problems or other bugs to fix? Has the project been abandoned? Maybe it's time to start planning to replace that dependency.

It's also important to recheck the security history of each dependency. For example, Apache Struts disclosed different major remote code execution vulnerabilities in 2016, 2017, and 2018. Even if you have a list of all the servers that run it and update them promptly, that track record might make you rethink using it at all.

Conclusion

Software reuse is finally here, and I don't mean to understate its benefits: it has brought an enormously positive transformation for software developers. Even so, we've accepted this transformation without completely thinking through the potential consequences. The old reasons for trusting dependencies are becoming less valid at exactly the same time we have more dependencies than ever.

The kind of critical examination of specific dependencies that I outlined in this article is a significant amount of work and remains the exception rather than the rule. But I doubt there are any developers who actually make the effort to do this for every possible new dependency. I have only done a subset of them for a subset of my own dependencies. Most of the time the entirety of the decision is "let's see what happens." Too often, anything more than that seems like too much effort.

But the Copay and Equifax attacks are clear warnings of real problems in the way we consume software dependencies today. We should not ignore the warnings. I offer three broad recommendations.

1. *Recognize the problem.* If nothing else, I hope this article has convinced you that there is a problem here worth addressing. We need many people to focus significant effort on solving it.
2. *Establish best practices for today.* We need to establish best practices for managing dependencies using what's available today. This means working out processes that evaluate, reduce, and track risk, from the original adoption decision through to production use. In fact, just as

some engineers specialize in testing, it may be that we need engineers who specialize in managing dependencies.

3. *Develop better dependency technology for tomorrow.* Dependency managers have essentially eliminated the cost of downloading and installing a dependency. Future development effort should focus on reducing the cost of the kind of evaluation and maintenance necessary to use a dependency. For example, package discovery sites might work to find more ways to allow developers to share their findings. Build tools should, at the least, make it easy to run a package's own tests. More aggressively, build tools and package management systems could also work together to allow package authors to test new changes against all public clients of their APIs. Languages should also provide easy ways to isolate a suspect package.

There's a lot of good software out there. Let's work together to find out how to reuse it safely.

References

1. Rachel Potvin and Josh Levenberg, "Why Google Stores Billions of Lines of Code in a Single Repository," *Communications of the ACM* 59(7) (July 2016), pp. 78-87. https://doi.org/10.1145/2854146
2. Russ Cox, "Go & Versioning," February 2018. https://research.swtch.com/vgo
3. Ken Thompson, "Reflections on Trusting Trust," *Communications of the ACM* 27(8) (August 1984), pp. 761–763. https://doi.org/10.1145/358198.358210
4. GNU Project, "GNU General Public License, version 1," February 1989. https://www.gnu.org/licenses/old-licenses/gpl-1.0.html
5. Titus Winters, "SD-8: Standard Library Compatibility," C++ Standing Document, August 2018. https://isocpp.org/std/standing-documents/sd-8-standard-library-compatibility
6. Go Project, "Go 1 and the Future of Go Programs," September 2013. https://golang.org/doc/go1compat
7. Facebook, "Infer: A tool to detect bugs in Java and C/C++/Objective-C code before it ships." https://fbinfer.com/
8. "SpotBugs: Find bugs in Java Programs." https://spotbugs.github.io/
9. D. Richard Hipp, "How SQLite is Tested." https://www.sqlite.org/testing.html
10. Alexander Potapenko, "Testing Chromium: ThreadSanitizer v2, a next-gen data race detector," April 2014. https://blog.chromium.org/2014/04/testing-chromium-threadsanitizer-v2.html
11. NIST, "National Vulnerability Database – Search and Statistics." https://nvd.nist.gov/vuln/search
12. Russ Cox, "Regular Expression Matching with a Trigram Index, or How Google Code Search Worked," January 2012. https://swtch.com/~rsc/regexp/regexp4.html
13. Google, "Google Open Source: Using Third-Party Licenses." https://opensource.google.com/docs/thirdparty/licenses/#banned
14. Nathan Willis, "A single Node of failure," LWN, March 2016. https://lwn.net/Articles/681410/
15. Charlie Reis, "Multi-process Architecture," September 2008. https://blog.chromium.org/2008/09/multi-process-architecture.html
16. Adam Langley, "Chromium's seccomp Sandbox," August 2009. https://www.imperialviolet.org/2009/08/26/seccomp.html
17. Nicolas Lacasse, "Open-sourcing gVisor, a sandboxed container runtime," May 2018. https://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime
18. Adam Baldwin, "Details about the event-stream incident," November 2018. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident
19. Russ Cox, "RE2: a principled approach to regular expression matching," March 2010. https://opensource.googleblog.com/2010/03/re2-principled-approach-to-regular.html
20. Rob Pike, "Go Proverbs," November 2015. https://go-proverbs.github.io/
21. U.S. House of Representatives Committee on Oversight and Government Reform, "The Equifax Data Breach," Majority Staff Report, 115th Congress, December 2018. https://oversight.house.gov/report/committee-releases-report-revealing-new-information-on-equifax-data-breach/

22. Russ Cox, "The Principles of Versioning in Go," GopherCon Singapore, May 2018. https://www.youtube.com/watch?v=F8nrpe0XWRg

23. Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," *Scientific Programming Journal*, vol. 13 (2005). https://doi.org/10.1155/2005/962135

Coda

This post is a draft of my current thinking on this topic. I hope that sharing it will provoke productive discussion, attract more attention to the general problem, and help me refine my own thoughts. I also intend to publish a revised copy of this as an article elsewhere. For both these reasons, unlike most of my blog posts, *this post is not Creative Commons-licensed*. Please link people to this post instead of making a copy. When a more final version is published, I will link to it here.