# Transparent Telemetry for Open-Source Projects

*Transparent Telemetry, Part 1*

Russ Cox

February 8, 2023

*research.swtch.com/telemetry-intro*

How do software developers understand which parts of their software are being used and whether they are performing as expected? The modern answer is *telemetry*, which means software sending data to answer those questions back to a collection server. This post is about why I believe telemetry is important for open-source projects, and what it might look like to approach telemetry in an open-source-friendly way. That leads to a new design I call *transparent telemetry*. If you are impatient, skip to the summary at the end*. Other posts in the series detail the design* and present various uses*.

## Why Telemetry?

Without telemetry, developers rely on bug reports and surveys to find out when their software isn't working or how it is being used. Both of these techniques are too limited in their effectiveness. Let's look at each in turn.

**Bug reports are not enough.** Users only file bug reports when they think something is broken. If a function is not behaving as documented, that's a clear bug to report. But if a program is misbehaving in a way that doesn't affect correctness, users are much less likely to notice. Statistics gathered by transparent telemetry make it possible for developers to notice that something is going wrong even when users do not.

For example, during the Go 1.14 release process in early 2020 we made a change to the way macOS Go distributions are built, as part of keeping them acceptable to Apple's signing tools. Unfortunately, the way we made the change also made all the pre-compiled `.a` files shipped in the distribution appear stale to builds. The effect was that the `go` command rebuilt and cached the standard library on first run, which meant that compiling any program using package `net` (which uses `cgo`) required Xcode to be installed. So Go 1.14 and later unintentionally required Xcode to compile even trivial demo Go programs like a basic HTTP server. This is not the way we want Go to work on macOS. On systems without Xcode, when `go` tried to invoke `clang`, macOS popped up a box explaining how to install it. Users simply accepted that this was necessary, perhaps even thinking `go` had displayed the popup. No one reported the bug over three years of Go releases. We didn't notice and fix the problem until late 2022 while investigating something else. With telemetry for the miss rate in the cache of pre-compiled standard library packages, the impact would have been obvious: all Macs running Go 1.14 or later would have a pre-installed package miss rate of 100%. This bug wasn't caught by our unit tests because it was caused by the distribution build machines having a modified environment different from actual user machines. The unit tests ran in the same modified environment as the build and worked fine. These kinds of unexpected differences between developer machines and user machines are inevitable at scale. Instrumenting the software on user machines is the most reliable way to understand how well it is working.

**Surveys are not enough.** Surveys help us understand what users want to do with Go, but they are only a small sample and have limited resolution. Asking about usage of infrequently-used features on a survey wastes time for a majority of respondents, and it requires large response counts to get an accurate measurement.

For example, we announced in the Go 1.13 release notes that future releases would drop support for Native Client (`GOOS=nacl`). Similarly, we announced in the Go 1.15 release notes that future releases would drop support for hardware floating point on 32-bit Intel CPUs without SSE2 instructions (`GO386=387`). Both of those removals went off okay, retroactively proving that our instincts about how few people would be affected were correct. On the other hand, we drafted an announcement for Go 1.18 removing `-buildmode=shared`, because it had essentially been broken since the introduction of modules, but when we issued Go 1.18 beta 1 we got feedback from at least a few people who were using it in some form. We still don't know how many people are using it or whether it is worth the maintenance costs, so it lingers on*. Another question is how long to keep supporting ARMv5 (`GOARM=5`), which doesn't have modern atomic instructions. More recently, we announced that Go 1.20 will be the last release to support macOS High Sierra and were promptly asked to keep it around*. Usage information would help us make more informed decisions. It's important to note the limitations of this usage information: if telemetry is disabled on all the machines that use the feature in question, or if it is only used in machines that don't stay up long enough to report anything, then we won't observe the usage. Telemetry is never perfect, but it's a useful input to the decision and much better than guessing. A survey is not any better and usually worse: there is a limit to how many questions we can reasonably ask in a survey, and asking a question where 99% of people answer "no I don't use that" is a waste of most people's time.

## Why Telemetry For Open Source?

When you hear the word telemetry, if you're like me, you may have a visceral negative reaction to a mental image of intrusive, detailed traces of your every keystroke and mouse click headed back to the developers of the software you're using. And for good reason! That mental image sounds like it must be an exaggeration but turns out to be fairly accurate. (Citations: Kindle tracking individual page turns*, VS Code telemetry logs*, and .NET telemetry events*.)

Open-source software projects have tended to avoid this kind of telemetry, for two reasons. The first is the significant privacy cost to users of collecting and storing detailed activity traces. The second is the fact that access to this data must be restricted, which would make the project less open than most strive to be. When the choice is between this kind of invasive tracking or doing nothing, doing nothing seems like an easy call. Still, doing nothing has real disadvantages. It means open-source developers like me tend not to understand as well how our software is used or how it performs. Then, because we lack that knowledge, we end up wasting time by maintaining features that aren't used, hurting users by removing features that are still being used, and delivering a poorer user experience by failing to notice when our software is underperforming in real-world usage.

Some open-source projects have adopted traditional telemetry, with mixed success and varying levels of user pushback. For example: Audacity*, GitLab*, and Homebrew*. Homebrew's telemetry seems to be generally accepted by users, and VS Code's detailed telemetry has not stopped it from being used by 74% of developers, as reported by the 2022 StackOverflow survey*. It could even be that the benefits from telemetry are part of how VS Code's developers have been able to build a tool that users like so much. Even so, the vast majority of projects, even large ones that would benefit, stay away from telemetry.

I believe that the choice between invasive tracking and doing nothing at all is a false dichotomy, and it's harming open source. Not having basic information about how their software is used and how well it is performing puts open-

source developers at a disadvantage compared to commercial software developers. Not having this information makes it more difficult to understand what's important and what isn't working, making prioritization that much harder. Not having clear prioritization in turn exacerbates the pre-existing problems with maintainer burnout.

Eric Raymond famously declared that "given enough eyeballs, all bugs are shallow," which he explained as meaning that "[g]iven a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone." Perhaps this was true in 1997 (perhaps not), but it's certainly not true today, as the Go macOS cache bug shows. A quarter century later, software is much larger, and open-source software is used by far more people who didn't develop it and aren't familiar with how it should and should not behave. Eyeballs don't scale.

I believe that open-source software projects need to explore new telemetry designs that help developers get the information they need to work efficiently and effectively, without collecting invasive traces of detailed user activity.

## Transparent Telemetry

This series of blog posts presents one such design, which I call *transparent telemetry*, because it collects as little as possible (kilobytes per year from each installation) and then publishes every bit that it collects, for public inspection and analysis.

I'd like to explore using this system, or one like it, in the Go toolchain, which I hope will help Go project developers and users alike. To be clear, I am only suggesting that the instrumentation be added to the Go command-line tools written and distributed by the Go team, such as the `go` command, the Go compiler, `gopls`, and `govulncheck`. I am *not* suggesting that instrumentation be added by the Go compiler to all Go programs in the world: that's clearly inappropriate. Also, throughout these posts, "developer" refers to the authors of a given piece of software, while "user" refers to the users of that software. From the point of view of the Go toolchain, "developer" means a Go toolchain developers like me, while "user" means one of the millions of Go programmers using that toolchain.

With transparent telemetry, as programs from the Go toolchain run, they would increment counters for various events of interest (for example: cache hit, use of a given feature, measured latency in a given range) in a per-week on-disk file. These files hold only counter values, not user data nor user identifiers. Some counter names include a short stack trace (function names and line offsets only, no argument data).

The Go team at Google would run a collection server. Each week, with 10% probability (averaging ~5 times per year) the user's Go installation would download a "collection configuration" to find out which counter values are of interest to the server and at what sample rate. The collection configuration would be served in a Go module validated using the Go checksum database*, for added confidence that all clients are being served the same configuration. Based on the sample rates, the Go installation might send a report containing the counter values of interest. Typical sample rates would be around 2% (averaging ~1 report per installation per year), but very rare events could be sampled at a higher rate, up to the 10% limit. As more systems take part in transparent telemetry, the overall sample rate on any given system will decrease, because only a fixed number of samples is necessary*.

The report would contain no ID of any form – no user login, no machine ID, no MAC address, no IP address, no IP address prefix, no geolocation information, no randomly-generated pseudo-ID, no other kind of identifiers. The report would contain basic information about the toolchain, such as its version

and what operating system and architecture it was built for. The report could also contain coarse-grained information about the version of the host operating system (for example, "Windows 8") and other tools the Go toolchain uses, such as the local C compiler ("gcc 2.95").

The server would collect each day's uploaded reports, update telemetry graphs served publicly on go.dev, and post the full set of uploaded reports for public download, inspection, and analysis.

Although the report would not include any identifiers, the TCP connection uploading the report would expose the system's public IP address to the server if a proxy is not being used. This IP address would not be associated with the uploaded reports in any way. Standard system maintenance, including DoS prevention, might require logs that include the IP address, but uploaded reports will be kept separate from those logs. The privacy policy would be similar to the one used by the Go module mirror and checksum database*.

The Go home page* and download page* already include a notice about the default use of the Go module mirror and a link to more information. That notice and link would be updated to disclose on-by-default telemetry. To opt out, users would set `GOTELEMETRY=off` in their environment or run a simple command like `go env -w GOTELEMETRY=off`; The first telemetry report is not sent until at least one week after installation, giving ample time to opt out. Opting out stops all collection and reporting: no "opt out" event is sent. It is simply impossible to see systems that install Go and then opt out in the next seven days.

## Summary

Transparent telemetry has the following key properties:

- The decisions about what metrics to collect are made in an open, public process.
- The collection configuration is automatically generated from the actively tracked metrics: no data is collected that isn't needed for the metrics.
- The collection configuration is served using a tamper-evident transparent log, making it very difficult to serve different collection configurations to different systems.
- The collection configuration is a cacheable, proxied Go module, so any privacy-enhancing local Go proxy already in use for ordinary modules will automatically be used for collection configuration. To further ameliorate concerns about tracking systems by the downloading of the collection configuration, each installation only bothers downloading the configuration each week with probability 10%, so that each installation only asks for the configuration about five times per year. [*Update*, 2023-02-24: The design has changed to be opt-in*, which requires raising these probabilities.]
- Uploaded reports only include total event counts over a full week, not any kind of time-ordered event trace.
- Uploaded reports do not include user IDs, machine IDs, or any other kind of ID.
- Uploaded reports only contain strings that are already known to the collection server: counter names, program names, and version strings repeated from the collection configuration, along with the names of functions in specific, unmodified Go toolchain programs for stack traces. The only types of non-string data in the reports are event

counts, dates, and line numbers.

- IP addresses exposed by the HTTP session that uploads the report are not recorded with the reports.
- Thanks to sampling*, only a constant number of uploaded reports are needed to achieve a specific accuracy target, no matter how many installations exist. Specifically, only about 16,000 reports are needed for 1% accuracy at a 99% confidence level. This means that *as new systems are added to the system, each system reports less often*. With a conservative estimate of two million Go installations, about 16,000 reporting each week corresponds to an overall reporting rate of well under 2% per week, meaning each installation would upload a report on average less than once per year. [*Update*, 2023-02-24: The design has changed to be opt-in*, which requires raising these probabilities.]
- The aggregate computed metrics are made public in graphical and tabular form.
- The full raw data as collected is made public, so that project maintainers have no proprietary advantage or insights in their role as the direct data collector.
- The system is on by default, but opting out is easy, effective, and persistent. [*Update*, 2023-02-24: The design has been changed to be opt-in*.]

## Next Steps

For more detail about the design, see the next post*. For more use cases, see the post after that*.

Although these posts use Go as the example system using transparent telemetry, I hope that the ideas apply and can be adopted by other open-source projects too, in their own, separate collection systems. For example, even though VS Code collects high-resolution event traces (sometimes tens or hundreds of events per minute), a close reading of those traces shows hardly anything is new in each event. That is, VS Code suffers the reputational hit of collecting lots of data but appears to gather relatively little actual information. Perhaps using transparent telemetry in VS Code or a similar editor could offer the editor's developers roughly equivalent insights and development velocity at a much lower privacy cost to users.

I am posting these to start a discussion about how the Go toolchain can adopt telemetry* in some form to help the Go toolchain developers make better decisions about the development and maintenance of Go. I have written an implementation of local counter collection to convince myself it could be made cheap enough, but no other part of the system exists today in any form. I hope that the system can be built over the course of 2023.

* Asterisks mark hyperlinked text.