Reproducible, Verifiable, Verified Builds

Go & Versioning, Part 5

Russ Cox
February 21, 2018

research.swtch.com/vgo-repro

Once both Go developers and tools share a vocabulary around package versions, it's relatively straightforward to add support in the toolchain for reproducible, verifiable, and verified builds. In fact, the basics are already in the vgo prototype.

Since people sometimes disagree about the exact definitions of these terms, let's establish some basic terminology. For this post:

- A *reproducible build* is one that, when repeated, produces the same result.
- A *verifiable build* is one that records enough information to be precise about exactly how to repeat it.
- A *verified build* is one that checks that it is using the expected source code.

`Vgo` delivers reproducible builds by default. The resulting binaries are verifiable, in that they record versions of the exact source code that went into the build. And it is possible to configure your repository so that users rebuilding your software verify that their builds match yours, using cryptographic hashes, no matter how they obtain the dependencies.

Reproducible Builds

At the very least, we want to make sure that when you build my program, the build system decides to use the same versions of the code. Minimal version selection delivers this property by default. The `go.mod` file alone is enough to uniquely determine which module versions should be used for the build (assuming dependencies are available), and that decision is stable even as new versions of a module are introduced into the ecosystem. This differs from most other systems, which adopt new versions automatically and need to be constrained to yield reproducible builds. I covered this in the minimal version selection post, but it's an important, subtle detail, so I'll try to give a short reprise here.

To make this concrete, let's look at a few real packages from Cargo, Rust's package manager. To be clear, I am not picking on Cargo. I think Cargo is an example of the current state of the art in package managers, and there's much to learn from it. If we can make Go package management as smooth as Cargo's, I'll be happy. But I also think that it is worth exploring whether we would benefit from choosing a different default when it comes to version selection.

Cargo prefers maximum versions in the following sense. Over at crates.io, the latest `toml` is 0.4.5 as I write this post. It lists a dependency on `serde` 1.0 or later; the latest `serde` is 1.0.27. If you start a new project and add a dependency on `toml` 0.4.1 or later, Cargo has a choice to make. According to the constraints, any of 0.4.1, 0.4.2, 0.4.3, 0.4.4, or 0.4.5 would be acceptable. All other things being equal, Cargo prefers the newest acceptable version, 0.4.5. Similarly, any of `serde` 1.0.0 through 1.0.27 are acceptable, and Cargo chooses 1.0.27. These choices change as new versions are introduced. If `serde` 1.0.28 is released tonight and I add toml 0.4.5 to a project tomorrow, I'll get 1.0.28 instead of 1.0.27. As described so far, Cargo's builds are not repeatable. Cargo's (entirely reasonable) answer to this problem is to have not just a constraint file (the manifest, `Cargo.toml`) but also a list of the exact artifacts to use in the build (the

lock file, `Cargo.lock`). The lock file stops future upgrades; once it is written, your build stays on `serde` 1.0.27 even when 1.0.28 is released.

In contrast, minimal version selection prefers the minimum allowed version, which is the exact version requested by some `go.mod` in the project. That answer does not change as new versions are added. Given the same choices in the Cargo example, `vgo` would select `toml` 0.4.1 (what you requested) and then `serde` 1.0 (what `toml` requested). Those choices are stable, without a lock file. This is what I mean when I say that `vgo`'s builds are reproducible by default.

Verifiable Builds

Go binaries have long included a string indicating the version of Go they were built with. Last year I wrote a tool `rsc.io/goversion` that fetches that information from a given executable or tree of executables. For example, on my Ubuntu Linux laptop, I can look to see which system utilities are implemented in Go:

```
$ go get -u rsc.io/goversion
$ goversion /usr/bin
/usr/bin/containerd go1.8.3
/usr/bin/containerd-shim go1.8.3
/usr/bin/ctr go1.8.3
/usr/bin/go go1.8.3
/usr/bin/gofmt go1.8.3
/usr/bin/kbfsfuse go1.8.3
/usr/bin/kbnm go1.8.3
/usr/bin/keybase go1.8.3
/usr/bin/snap go1.8.3
/usr/bin/snapctl go1.8.3
$
```

Now that the `vgo` prototype understands module versions, it includes that information in the final binary too, and the new `goversion -m` flag prints it back out. Using our "hello, world" program from the tour:

```
$ go get -u rsc.io/goversion
$ goversion ./hello
./hello go1.10
$ goversion -m hello
./hello go1.10
    path  github.com/you/hello
    mod   github.com/you/hello  (devel)
    dep   golang.org/x/text     v0.0.0-20170915032832-14c0d48ead0c
    dep   rsc.io/quote          v1.5.2
    dep   rsc.io/sampler        v1.3.0
$
```

The main module, supposedly `github.com/you/hello`, has no version information, because it's the local development copy, not a specific version we downloaded. But if instead we build a command directly from a versioned module, then the listing does report versions for all modules:

```
$ vgo build -o hello2 rsc.io/hello
vgo: resolving import "rsc.io/hello"
vgo: finding rsc.io/hello (latest)
vgo: adding rsc.io/hello v1.0.0
vgo: finding rsc.io/hello v1.0.0
vgo: finding rsc.io/quote v1.5.1
vgo: downloading rsc.io/hello v1.0.0
```

```
$ goversion -m ./hello2
./hello2 go1.10
    path  rsc.io/hello
    mod   rsc.io/hello       v1.0.0
    dep   golang.org/x/text  v0.0.0-20170915032832-14c0d48ead0c
    dep   rsc.io/quote       v1.5.2
    dep   rsc.io/sampler     v1.3.0
$
```

When we do integrate versions into the main Go toolchain, we will add APIs to access this information from inside a running binary, just like `runtime.Version` provides access to the more limited Go version information.

For the purpose of attempting to reconstruct the binary, the information listed by `goversion -m` suffices: put the versions into a `go.mod` file and build the target named on the `path` line. But if the result is not the same binary, you might wonder about ways to narrow down what's different. What changed?

When vgo downloads each module, it computes a hash of the file tree corresponding to that module. That hash is also included in the binary, alongside the version information, and `goversion -mh` prints it:

```
$ goversion -mh ./hello
hello go1.10
    path  github.com/you/hello
    mod   github.com/you/hello  (devel)
    dep   golang.org/x/text     v0.0.0-20170915032832-14c0d48ead0c  h1:qgOY6WgZOaTkIIMiVjBQcw93ER...    dep   r
hello go1.10
    path  rsc.io/hello
    mod   rsc.io/hello       v1.0.0                                 h1:CDmhdOARcor1WuRUvmE46PK91ahrS...    dep   g
```

The `h1:` prefix indicates which hash is being reported. Today, there is only "hash 1," a SHA-256 hash of a list of files and the SHA-256 hashes of their contents. If we need to update to a new hash later, the prefix will help us tell old from new hashes.

I must stress that these hashes are self-reported by the build system. If someone gives you a binary with certain hashes in its build information, there's no guarantee they are accurate. They are very useful information supporting a later verification, not a signature that can be trusted by themselves.

Verified Builds

An author distributing a program in source form might want to let users verify that they are building it with exactly the expected dependencies. We know vgo will make the same decisions about which versions of dependencies to use, but there is still the problem of mapping a version like v1.5.2 to an actual source tree. What if the author of v1.5.2 changes the tag to point at a different file tree? What if a malicious middlebox intercepts the download request and delivers a different zip file? What if the user has accidentally edited the source files in the local copy of v1.5.2? The vgo prototype supports this kind of verification too.

The final form may be somewhat different, but if you create a file named `go.modverify` next to `go.mod`, then builds will keep that file up-to-date with known hashes for specific versions of modules:

```
$ echo >go.modverify
$ vgo build
$ tcat go.modverify  # go get rsc.io/tcat, or use cat
golang.org/x/text  v0.0.0-20170915032832-14c0d48ead0c  h1:qgOY6WgZOaTkIIMiVjBQcw93ERBE4m30iBm00nk...rsc.io/quot
```

The `go.modverify` file is a log of the hash of all versions ever encountered: lines

are only added, never removed. If we update rsc.io/sampler to v1.3.1, then the log will now contain hashes for both versions:

```
$ vgo get rsc.io/sampler@v1.3.1
$ tcat go.modverify
golang.org/x/text  v0.0.0-20170915032832-14c0d48ead0c  h1:qgOY6WgZOaTkIIMiVjBQcw93ERBE4m30iBm00nk...rsc.io/quot
```

When go.modverify exists, vgo checks that all downloaded modules used in a given build are consistent with entries already in the file. For example, if we change the first digit of the rsc.io/quote hash from w to v:

```
$ vgo build
vgo: verifying rsc.io/quote v1.5.2: module hash mismatch
    downloaded:   h1:w5fcysjrx7yqtD/aO+QwRjYZOKnaM9Uh2b40tElTs3Y=
    go.modverify: h1:v5fcysjrx7yqtD/aO+QwRjYZOKnaM9Uh2b40tElTs3Y=
$
```

Or suppose we fix that one but then modify the v1.3.0 hash. Now our build succeeds, because v1.3.0 is not being used by the build, so its line is (correctly) ignored. But if we try to downgrade to v1.3.0, then the build verification will correctly begin failing:

```
$ vgo build
$ vgo get rsc.io/sampler@v1.3.0
vgo: verifying rsc.io/sampler v1.3.0: module hash mismatch
    downloaded:   h1:7uVkIFmeBqHfdjD+gZwtXXI+RODJ2Wc4O7MPEh/QiW4=
    go.modverify: h1:8uVkIFmeBqHfdjD+gZwtXXI+RODJ2Wc4O7MPEh/QiW4=
$
```

Developers who want to ensure that others rebuild their program with exactly the same sources they did can store a go.modverify in their repository. Then others building using the same repo will automatically get verified builds. For now, only the go.modverify in the top-level module of the build applies. But note that go.modverify lists all dependencies, including indirect dependencies, so the whole build is verified.

The go.modverify feature helps detect unexpected mismatches between downloaded dependencies on different machines. It compares the hashes in go.modverify against hashes computed and saved at module download time. It is also useful to check that downloaded modules have not changed on the local machine since it was downloaded. This is less about security from attacks and more about avoiding mistakes. For example, because source file paths appear in stack traces, it's common to open those files when debugging. If you accidentally (or, I suppose, intentionally) modify the file during the debugging session, it would be nice to be able to detect that later. The vgo verify command does this:

```
$ go get -u golang.org/x/vgo  # fixed a bug, sorry! :-)
$ vgo verify
all modules verified
$
```

If a source file changes, vgo verify notices:

```
$ echo >>$GOPATH/src/v/rsc.io/quote@v1.5.2/quote.go
$ vgo verify
rsc.io/quote v1.5.2: dir has been modified (/Users/rsc/src/v/rsc.io/quote@v1.5.2)
$
```

If we restore the file, all is well:

```
$ gofmt -w $GOPATH/src/v/rsc.io/quote@v1.5.2/quote.go
$ vgo verify
all modules verified
$
```

If cached zip files are modified after download, vgo `verify` notices that too, although I can't plausibly explain how that might happen:

```
$ zip $GOPATH/src/v/cache/rsc.io/quote/@v/v1.5.2.zip /etc/resolv.conf
  adding: etc/resolv.conf (deflated 36%)
$ vgo verify
rsc.io/quote v1.5.2: zip has been modified (/Users/rsc/src/v/cache/rsc.io/quote/@v/v1.5.2.zip)
$
```

Because vgo keeps the original zip file after unpacking it, if vgo `verify` decides that only one of the zip file and the directory tree have been modified, it could even print a diff of the two.

What's Next?

This is implemented already in vgo. You can try it out and use it. As with the rest of vgo, feedback about what doesn't work right (or works great) is appreciated.

The functionality presented here is more the start of something than a finished feature. A cryptographic hash of the file tree is a building block. The `go.modverify` built on top of it checks that developers all build a particular module with precisely the same dependencies, but there's no verification when downloading a new version of a module (unless someone else already added it to `go.modverify`), nor is there any sharing of expected hashes between modules.

The exact details of how to fix those two shortcomings are not obvious. It may make sense to allow some kind of cryptographic signatures of the file tree, and to verify that an upgrade finds a version signed with the same key as the previous version. Or it may make sense to adopt an approach along the lines of The Update Framework (TUF), although using their network protocols directly is not practical. Or, instead of using per-repo `go.modverify` logs, it might make sense to establish some kind of shared global log, a bit like Certificate Transparency, or to use a public identity server like Upspin. There are many avenues we might explore, but all this is getting a little ahead of ourselves. For now we are focused on successfully integrating versioning into the go command.