Reproducible Builds: Increasing the Integrity of Software Supply Chains

Chris Lamb, Stefano Zacchiroli

▶ **To cite this version:**

Chris Lamb, Stefano Zacchiroli. Reproducible Builds: Increasing the Integrity of Software Supply Chains. IEEE Software, In press, 10.1109/MS.2021.3073045 . hal-03196519

HAL Id: hal-03196519
https://hal.science/hal-03196519v1

Submitted on 12 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reproducible Builds: Increasing the Integrity of Software Supply Chains

Chris Lamb
Reproducible Builds

Stefano Zacchiroli
Université de Paris and Inria, France

***Abstract*—Although it is possible to increase confidence in Free and Open Source Software (FOSS) by reviewing its source code, trusting code is not the same as trusting its executable counterparts. These are typically built and distributed by third-party vendors, with severe security consequences if their supply chains are compromised. In this paper, we present *reproducible builds*, an approach that can determine whether generated binaries correspond with their original source code. We first define the problem, and then provide insight into the challenges of making real-world software build in a "reproducible" manner—this is, when every build generates bit-for-bit identical results. Through the experience of the *Reproducible Builds project* making the Debian Linux distribution reproducible, we also describe the affinity between reproducibility and quality assurance (QA).**

> *You can't trust code that you did not totally create yourself. [. . .] No amount of source-level verification or scrutiny will protect you from using untrusted code.*
>
> — Ken Thompson (1984)

How can we be sure that our software is doing only what it is supposed to do? This was the key takeaway from Ken Thompson's 1984 Turing Lecture, "Reflections on Trusting Trust" [1]. But with people today executing far more software than they compile, the number of users who "totally create" software they run has dropped dramatically since then.

Let us narrow the issue to Free and Open Source Software (FOSS), where all source code is freely available. Hypothetically, users can examine the source of all the software they wish to use in order to confirm it does not contain spyware or backdoors—indeed, one of the original promises of FOSS was that distributed peer review [2] would result in enhanced end-user security. However, whilst users *can* inspect source code for malicious flaws, almost all software is now distributed as pre-built binaries. This permits nefarious actors to compromise end-user systems by modifying ostensibly secure code during its compilation or distribution.

For example, a Linux distribution might compile "safe" software on compromised servers and unwittingly spread malicious executables onto countless systems. Other vectors include engineers being explicitly coerced into incorporating vulnerabilities, as well as the covert compromise of developers' computers (remotely or through "evil maid" attacks [3]) so they unwittingly distribute tainted binaries via app stores and other

 Published by the IEEE Computer Society © 2021 IEEE

channels.

Software supply-chain attacks are no longer hypothetical scenarios. In December 2020, news broke that attackers subverted the SolarWinds Orion software to inject malicious code into executables at build time, resulting in a severe data breach across several US government branches [4]. 174 similar attacks have been detailed in the literature too [5]. Due to their potential impact, software supply chains have become a high-value target in recent years, and this trend appears to be accelerating.

Practical and scalable solutions to these attacks are therefore urgently needed, and an approach known as *reproducible builds* is one such countermeasure. However, it is only applicable if the software sources are widely available—although malware can be detected directly in binaries [6], [7], doing so is inefficient when source code is available for audit.

The key idea behind the reproducible builds (*R-B*) approach is that, if we can guarantee that building a given source tree always generates bit-for-bit identical results, we can establish trust in these artifacts by comparing outputs acquired from multiple, independent builders.

In this paper, we present the R-B approach from the perspective of software professionals. We show how software *users* can benefit from the increased trust in executables they run as well as how *developers* and build engineers can help make software reproducible. We also describe the *quality assurance* (QA) tools available to improve build reproducibility, highlighting how they further mutually-beneficial goals such as reducing build and test "flakiness" [8], [9]. This paper is informed in large part by the experience of the Reproducible Builds project (reproducible-builds.org), a non-profit initiative that popularized the R-B approach.

REPRODUCIBLE BUILDS

The core element of the reproducible builds model is the following property:

Definition 1. *The build process of a software product is* reproducible *if, after designating a specific version of its source code and all of its build dependencies, every build produces* bit-for-bit identical artifacts*, no matter the environment in which the build is performed.*

In other words, once we reach an agreement on the exact software version(s) we wish to build, anyone who builds that software should always generate precisely the same artifacts.

Figure 1 shows how users can leverage this property to establish trust in FOSS executables. Software development happens upstream as usual (e.g. on platforms such as GitHub and GitLab) and, from there, source code reaches downstream vendors such as Linux distributions and app stores. These vendors then build binaries from these sources, which are subsequently distributed to end-users. Note that neither the distribution nor the build process are completely trusted in this scenario, reflecting the hostile environment of the real world.

When software builds reproducibly, however, we can still establish trust in these executables. This is because users can corroborate whether their newly-downloaded binaries are identical to those that *others* have built themselves.

How this works is as follows: At the top of the supply chain, trust in a specific version of a piece of software is established through auditing the source code or, more likely, by implicitly trusting its developers (e.g. trusting version 5.11.5 of the Linux kernel as it is signed by Linus Torvalds). Later, but before executing any binaries they have downloaded, users can compare the checksums of these files with the expected values for that specific version, crucially aborting on any mismatch.

These expected checksums could originate from a limited set of trusted parties who publish statements that building some specific source code release results in a particular set of executables. However, another alternative is *distributed consensus*, where a loose-knit community of semi-trusted builders independently announce their checksums. Normally, participants in this scheme report identical checksums for a given source code release, but in case of a discrepancy (i.e. if some builders have been compromised), the checksum reported by $\geq 50\%$ of the builders may be the one to trust. Under this verification scheme, the takeover at least 50% of the builder community would be required to coerce users into running malicious binaries.

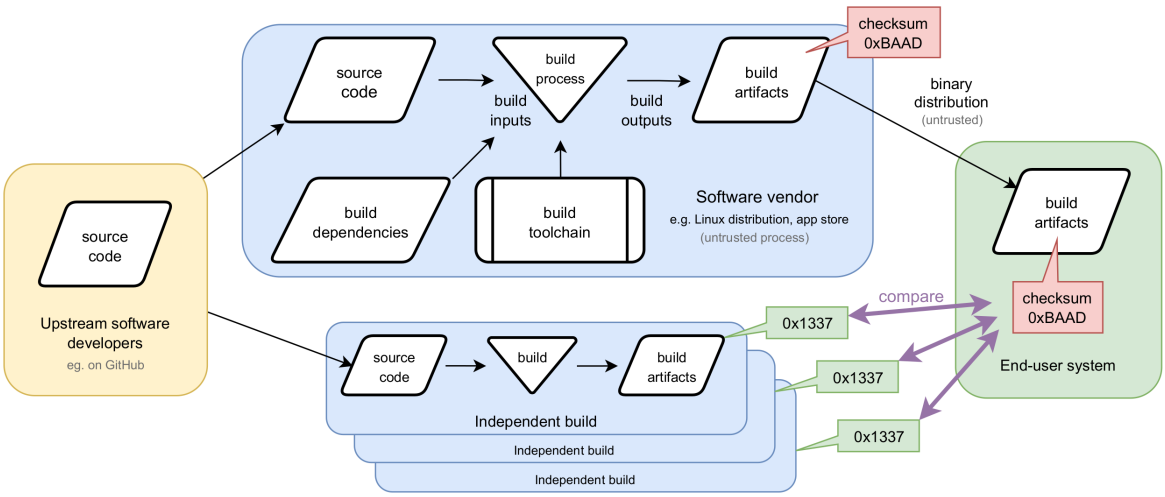

Figure 1. The *reproducible builds* approach to increasing trust in executables built by untrusted third parties. The end-user should reject the binary artifact from their software vendor, as its checksum (`0xBAAD`) does not match the one built by multiple, independent third-parties (`0x1337`).

REPRODUCIBILITY IN THE SMALL

How hard is it to ensure that independent builds always result in bit-for-bit identical executables? First, let us consider the software vendor in Figure 1. Each build takes as its input the source to be built, all of its build-time dependencies, and the entire build toolchain including the compiler, linker and build system. The build produces a set of artifacts (executables, data, documentation, etc.) as its output. Any change in these inputs may legitimately affect its output.

However, even once all inputs have been controlled for, the build may still be unreproducible—that is, producing different artifacts when the build is repeated. This results from two main classes of problem: uncontrolled build inputs and build non-determinism.

Uncontrolled build inputs occur when toolchains allow the build process to be affected by the surrounding environment. Common examples include system time, environment variables and the arbitrary build location on the filesystem. Uncontrolled inputs can be seen as analogous to breaking encapsulation in software design; a tight coupling between a high-level process and the low-level implementation details.

Build non-determinism occurs when aspects of the build behave non-deterministically *and* these "random" behaviours are encoded in the final artifacts. For example, if the output is derived in any way from the state of a pseudorandom number generator or the arbitrary order of process scheduling.

To address uncontrolled build inputs, it is tempting to "jail" builds into sanitized environments that always present a canonical interface to the underlying build system. Indeed, this was the approach taken by early projects such as Bitcoin and Tor (rbm.torproject.org). However, jails result in slower build times and impose technical and social restrictions on developers who may be accustomed to choosing their tooling. Most jails cannot address non-determinism issues either.

The ultimate and preferred solution is to ensure that any code run during the build *only* depends on the legitimate build inputs (the source being built, the build dependencies and the toolchain), and that any non-deterministic behavior does not affect the resulting artifacts.

We will now review some individual causes of unreproducible builds and show how to address them.

Build timestamps

Timestamps are, by far, the biggest source of unreproducibility. It is a common practice to explicitly embed dates into binaries via C's `__DATE__` macro (see Listing 1), but many tools record dates into build artifacts as well. For example, `help2man` generates UNIX manual pages directly from the output of `--help`, and

Listing 1. The `__DATE__` C preprocessor macro *"expands to a string constant that describes the date on which the preprocessor is being run."*

```
void usage() {
    fprintf (stderr,
             "foo-utils version "
             "3.141 (built %s)\n",
              __DATE__);
}
```

in its default configuration, it embeds the current date into generated files. As this value changes from day-to-day, this results in an unreproducible build. TEX's `\date` macro also embeds the current date, with similar implications for generated documentation.

The value of these timestamps is extremely limited, particularly as they don't convey which version of the software was actually built; after all, older software can always be built later. Embedded timestamps should therefore be avoided entirely, but for cases where that is not possible (e.g. in file formats that mandate their presence), the Reproducible Builds project proposed the `SOURCE_DATE_EPOCH` environment variable as a way to communicate an acceptable timestamp to build systems [10]. This typically represents the last modification time of the source tree as extracted from the software's changelog file.

Build paths

The filesystem path where the build took place is often embedded in generated binaries too, usually via the `__FILE__` preprocessor macro (see Listing 2) or by assertion statements that reference their corresponding line of code. Other sources of build paths include logging messages, locations of detached debug symbols, `RPATH` entries in ELF binaries, and many other instances that are intended, ironically, to assist the software development process.

To help address this issue, the Reproducible Builds project worked with the GNU GCC developers to introduce the `-ffile-prefix-map` and `-fdebug-prefix-map` options which support embedding relative (rather than absolute) paths.

Listing 2. The `__FILE__` C preprocessor macro *"expands to the name of the current input file"*. This results in non reproducibility when the program is built from different directories, e.g. `/home/lamby/tmp` vs. `/home/zack/tmp`.

```
fprintf (stderr,
         "DEBUG: boop (%s:%s\n",
         __FILE__, __LINE__);
```

Filesystem ordering

Contrary to the output of `ls(1)`, the POSIX Unix standard does not specify an ordering for results returned by the underlying `readdir(3)` system call. As a result, directories accessed in naive "`readdir` order" may be processed in a non-deterministic manner. If this arbitrary ordering influences any build artifacts, the build will not be reproducible.

For example, the build system of the *PikePDF* library located its own source files using Python's `glob` routine. But as `glob`'s result value inherits the non-determinism of `readdir(3)`, *PikePDF*'s source files were linked in an arbitrary order.

This is a particularly pernicious problem as some filesystem implementations return different orderings "more often" than others. To avoid these issues, build systems should impose a deterministic order on any directory iteration encoded in its artifacts, e.g. via an explicit `sort()`.

Archive metadata

`.zip` and `.tar` archives store timestamps and user ownership information in addition to the files themselves. However, if this metadata is inherited from the surrounding build environment, it will not be replicated when building elsewhere. For example, if a `.tar` archive stores files as belonging to the build user (e.g. `lamby`), another user (e.g. `zack`) building the same software will obtain a different result.

This can be avoided by instructing tools to ignore on-disk values in favour of metadata chosen by the build system (e.g. using `tar(1)` with `--owner=0` and `--clamp-mtime=T`), or by normalizing metadata before archiving begins (e.g. by using `touch(1)` with `SOURCE_DATE_EPOCH` as a reference timestamp).

Listing 3. Perl's hash type does not define an ordering of its keys, so a call to `sort` should be inserted before `keys %h` to make it deterministic.

```
my %h = ( a => 1, b => 2, c => 3);
foreach my $k (keys %h) {
    print "$k\n";
}
```

Randomness

Even when the entire environment is controlled for, many builds remain inherently nondeterminstic. For example, builds that iterate over hash tables (such as Perl's "hash" or the `dict` type in Python < 3.7) exhibit arbitrary behaviour as their respective elements are returned in an undefined order—the code in Listing 3, for example, may print any combination of `abc`, `bac`, `bca`, etc. This affects reproducibility if these results form any part of the build's artifacts.

Parallelism (such as via processes or threads) can also prevent reproducibility if the arbitrary completion order is encoded into build results too. Similar to filesystem ordering, these issues can be resolved by imposing determinism in key locations, seeding any sources of randomness to fixed values or sorting the results of hash iterations and parallelized tasks before generating output.

Uninitialized memory

Many data structures have undefined areas that do not affect their operation. The FAT filesystem, for example, contains unused regions that may be filled with arbitrary data. In addition, modern CPU architectures perform more efficiently when data is naturally aligned, and the padding added to ensure alignment can result in similarly undefined areas. These regions containing "random" data affect reproducibility when stored in build results.

One solution is to explicitly zero-out memory regions that may persist in artifacts. For example, Listing 4 shows a patch for *GNU mtools* that ensures generated FAT directory entries do not embed uninitialized memory.

Listing 4. A patch for *GNU mtools* ensuring that a `direntry_t` struct does not contain uninitialized memory.

```
--- a/direntry.c
+++ b/direntry.c
@@ -24,6 +24,7 @@

 void initializeDirentry(
   direntry_t *entry, Stream_t *Dir) {
+  memset(entry, 0, sizeof(direntry_t));
   entry->entry = -1;
   entry->Dir = Dir;
```

REPRODUCIBILITY IN THE LARGE

Now that we know how to address some individual reproducibility issues, we turn to the problems that arise when making large software *collections* reproducible.

The Reproducible Builds project started in 2014 with the aim of making the Debian operating system (www.debian.org) completely reproducible. This is a formidable goal, as not only is Debian a extremely mature Linux distribution, it is one of the largest curated collections of FOSS software in general.

Seven years later, over 95% of the 30 000+ packages in Debian's development branch can now be built reproducibly, and as the Linux distribution with the largest total number of reproducible packages, it serves as an extremely relevant case study. The evolution of this effort can be found at wiki.debian.org/ReproducibleBuilds.

Adversarial rebuilding

Given its scale, Debian developers realized they would need a programmatic way to test for reproducibility. To this end, they developed a continuous integration (CI) [11] system which builds each package in the Debian archive twice in a row, using two independent build environments that are deliberately configured to differ as much as possible. For instance, the clock on the second build is set 18 months in the future, and the hostname, language, system kernel, etc., are all varied so that if any environmental differences are used as a build input, the two builds will differ as a result. The large number of variations applied (30+) can validate build reproducibility to a high degree of accuracy.

To identify any reliance on non-deterministic filesystem ordering, the R-B project also developed a FUSE-based [12] virtual filesystem called *disorderfs* (salsa.debian.org/reproducible-builds/disorderfs) which can provide a view of

Listing 5. An example `.buildinfo` file, recording both the environment and results of building Debian's `black` package. (Excerpt: see buildinfo.debian.net/sources/black/20.8b1-1 for the full version.)

```
Source: black
Version: 20.8b1-1
Checksums-Sha1:
  9915459ae7a1a5c3efb984d7e5472f7976e996b1 2584 black_20.8b1-1.dsc
  14bfd3011b795f85edbc8cc4dc034a91cfaa9bcd 111096 black_20.8b1-1_all.deb
  69c3d4ae7115c51e7b00befe8b4afd5963601d66 285684 python-black-doc_20.8b1-1_all.deb
Checksums-Sha256: [...]
Build-Architecture: amd64
Installed-Build-Depends: autoconf (= 2.69-11.1), automake (= 1:1.16.2-4), [...],
  gcc (= 4:10.2.0-1), [...], python3 (= 3.8.2-3), [...]
  xz-utils (= 5.2.4-1+b1), zlib1g (= 1:1.2.11.dfsg-2)
```

a filesystem with configurable orderings. The R-B CI system *reverses* the filesystem ordering between the builds, revealing any dependency on non-deterministic filesystem ordering.

Recording build information

As per Definition 1, a reproducible build must always use the same original source, toolchain and build dependencies, and to ensure these inputs can be replicated correctly, Debian devised the `.buildinfo` file format.

Once a Debian package is built, the precise source version and the versions of all its build dependencies are recorded in a `.buildinfo` file. This file also contain checksums of any generated `.deb` artifacts, the Debian binary package format. (An example file may be found in Listing 5.)

`.buildinfo` files are a crucial building block for any process wishing to validate reproducibility. A `.buildinfo` is produced during an initial build and is then used to reconstruct a second build environment. The build is repeated within this second environment and the checksums from this latter build are compared with the ones in the *original* `.buildinfo`—if they do not match, the build is unreproducible or a build host has been tampered with.

Users can employ `.buildinfo` files to implement the consensus-driven approach outlined above, verifying downloaded packages by comparing them against the checksums in `.buildinfo` files distributed by Debian and other builders. In this scenario, `.buildinfo` files are cryptographically signed to represent a *build attestation*, e.g. "I, Alice, given source X and environment Y, have built a package with checksum K." Bob would verify that Alice really made this claim, and then compare Alice's *K* against his downloaded file, potentially trusting Alice's *K* over any divergent (and likely malicious) claim from Eve.

Debian currently hosts over 20 million `.buildinfo` files in a number of experimental services. However, centralized distribution schemes inherit many of the issues of the SSL certificate authority ecosystem, particularly in representing an obvious target to attack [13]. Decentralized alternatives remain a future challenge at this point, as does a practical consensus mechanism to determine the "valid" checksum for any given package.

Root cause analysis

As we have outlined, it is trivial to detect mismatches between builds simply by comparing the checksums of their artifacts. However, it can be extremely difficult to understand the root cause of this difference.

Therefore, the R-B project developed *diffoscope* (diffoscope.org), a visual "diff" tool that recursively unpacks a large number of archive formats and translate tens of binary formats into human-readable forms. As a result, it can display the meaningful, code-level differences between, for example, two compiled Java `.class` files, even if they were contained in `.tar` in a `.xz` (in a `.deb` in a `.iso`, etc.).

In most cases, *diffoscope* indicates which category of fix is required to make a build reproducible. For instance, when programs embed build dates into their binaries, *diffoscope* clearly

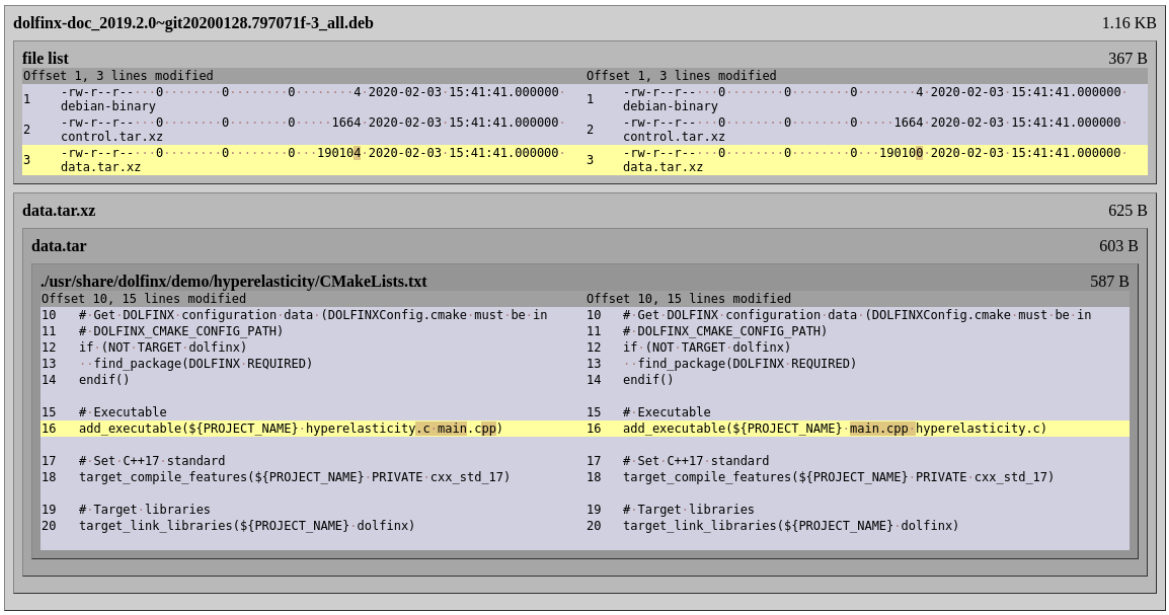

Figure 2. *diffoscope* recursively unpacks archives of many kinds and transforms various binary formats into more human-readable forms in order to compare them.

highlights these date-based variations, and the surrounding context tends to assist in identifying which part of the original source code to fix.

An example *diffoscope* output is shown in Figure 2, where two versions of the `dolfinx-doc` package differ. Here, *diffoscope* indicates that the difference is in `CMakeLists.txt`, a generated file which contains the same entries with a different ordering between the two builds. This would appear to be a filesystem ordering issue, solved by the addition of an explicit sort. However, this may be a problem affecting all software that uses CMake, so the issue may be better addressed there; alas, *diffoscope* cannot entirely replace a software engineer's judgement.

Quality Assurance (QA)

Adopting a methodical approach to verify the reproducibility of builds can be highly complementary to QA efforts. This is because problems that affect build reproducibility are often symptoms of larger, systemic issues.

To begin with, systematic *reproducibility* testing implies systematic *build* testing, so will easily identify software that fail to build under any circumstances. Other software will fail to build only in the extreme environments designed to test for reproducibility, but will become more robust as a result of behaving well there. For example, some software will fail to build in the future due to hardcoded SSL certificates with expiry dates—these are detected due to the build environment's artificial future clock. Others fail to build in rarely-used timezones due to incorrect assumptions about time offsets—the test suite for the Ruby *Timecop* library failed in this way (bugs.debian.org/795663).

Due to these serendipitous quality improvements, addressing reproducibility issues improves the correctness and robustness of the Debian distribution as a whole. However, even less-critical problems can be identified through reproducibility testing as well. For example, an issue in the Doxygen documentation generator (bugs.debian.org/970431) led to broken hyperlinks that linked to their build-time location (e.g. `/tmp/build/foo/usage.html`) instead of their run-time one (`/usr/share/doc/foo/usage.html`). This was trivial to identify with *diffoscope*, and fixing it corrected broken documentation for hundreds of end-users.

Reproducibility testing can even flag spurious content *within* documentation. For instance, manual packages generated by executing an underlying program can fail in several ways, often printing error messages that are mistak-

Listing 6. An example `ConfigData.pm`. As it was created at build time, all users shared the same `OpenIDConsumerSecret`.

```
{
 'cgibin' => '/usr/lib/cgi-bin/gbrowse',
 'conf' => '/etc/gbrowse',
 'databases' => '/var/lib/gbrowse/databases',
 'htdocs' => '/usr/share/gbrowse/htdocs',
 'OpenIDConsumerSecret' => '639098210478536',
 'tmp' => '/var/cache/gbrowse'
},
```

enly shipped as the package's "documentation" (e.g. bugs.debian.org/972635). These bugs are difficult to detect if the failure does not occur on a developer's own machine, but they can be easily spotted whilst testing for reproducibility as the error messages are deliberately designed to appear in different languages.

Even security issues can be discovered whilst testing for reproducibility. In one example, the *GBrowse* biological genome annotation viewer failed to build reproducibly (bugs.debian.org/833885), and *diffoscope* identified a configuration file that contained a different `OpenIDConsumerSecret` value between builds (see Listing 6). Although this secret was being securely generated, it was being created at *build time*, so the same value was distributed to all users of the package—the fix was to generate the secret at installation time so that each deployment possessed its own unique key. The mechanics of reproducibility testing suggest that this issue would not have been readily discovered another way.

Community engagement

Although the causes of build unreproducibility often reside within the source code of individual projects, it is far more effective to detect issues via centralized testing in distributions such as Debian due to the uniform build interfaces these large collections provide. Nevertheless, the social norms of the FOSS community dictate that fixes should be integrated upstream, instead of remaining in distribution-specific patch sets.

To this end, the Reproducible Builds project has contributed to hundreds of individual FOSS projects, in addition to working with key toolchains such as GCC, Rust, OCaml, etc. This community-oriented approach ensures that as many users as possible can benefit from the specific advantages of reproducible builds, as well as from the software quality improvements achieved while pursuing that goal.

THE REPRODUCIBLE BUILDS ECOSYSTEM

Taking Debian to its current state required over seven years of cross-community work that was spearheaded by the Reproducible Builds project (reproducible-builds.org), a non-profit organisation that aims to increase the integrity of software supply chains by advocating for and implementing the approach outlined in this paper.

Although originating in Debian around 2014, many other FOSS projects have joined the initiative such as Arch Linux, coreboot, F-Droid, Fedora, FreeBSD, Guix, NixOS, openSUSE and Qubes. One milestone of this joint effort is Tails (https://tails.boum.org/), the operating system used by Edward Snowden to securely communicate the NSA's global surveillance activities in 2013 [14]—Tails began releasing reproducible ISO images in 2017 to improve end-user verifiability and security.

The Reproducible Builds project has also developed several tools (reproducible-builds.org/tools) that facilitate various QA processes related to reproducibility. Some of these, such as *diffoscope* and *disorderfs*, have been highlighted in this paper.

Increasing the security of open source software is clearly a worthwhile goal, and software professionals and organisations can always provide assistance. This is not only by addressing any uncontrolled build inputs and sources of non-determinism in the software they maintain, but by working with the Reproducible Builds project itself in terms of code, donations and other traditional forms of community contribution.

CONCLUSION

In this article, we have outlined what it means for software to build reproducibly and how that property can be leveraged by end-users to establish trust in open source executables, even when they are built by untrusted third parties. We also surveyed several causes of unreproducibility and located their causes in build systems and

similar logic. We also described some of the quality assurance (QA) processes and tools that can be used to make large open source software collections reproducible—using this model, the Debian operating system has achieved 95% reproducibility in over 30 000+ packages.

Additional work is still needed to address the software that is not yet reproducible. In the case of Debian, there are no insurmountable obstacles preventing the project from reaching 100%—the remaining 5% "only" need fixes similar in kind to those already discussed. However, this has not yet been achieved, partly because time and effort are not inexhaustible or fungible resources in volunteer communities, but also due to regressions in previously-reproducible packages. Improved awareness and prioritisation of reproducibility amongst software developers would reduce the incidence of such events.

Other challenges remain for the reproducible builds ecosystem too. Cryptographically signed artifacts are becoming more common, which cannot be made reproducible without distributing signing keys to builders. One solution is to adopt detached signatures, but the addition of parallel distribution channels for these (unreproducible) files would require extensive changes to existing software distribution channels.

The verification of open source software for mobile devices also remains problematic. With the notable exception of F-Droid, not only are the build processes of the major app stores unreproducible (or not even FOSS), the checksums of artifacts are hidden from end-users, rendering any distributed validation scheme impossible. Significant usability and transparency improvements are needed to make meaningful progress in this area.

Finally, we are left with the recursive question of whether we can trust even *reproducible* binaries without trusting where our compilers and other toolchain components come from. To address this, the parallel Bootstrappable Builds (bootstrappable.org) project seeks to minimize the amount of binary code required to bootstrap a minimal C compiler—at time of publication, a binary as small as 6 KB is enough to activate a chain of steps from *TCC* [15] to GCC from which almost all toolchains can then be obtained. Ken Thompson would likely approve, whilst still pointing out that 6 KB is too much untrusted code.

ACKNOWLEDGMENTS

The authors would like to thank the Reproducible Builds and the wider Debian community for their feedback on this paper, as well as for their invaluable work on increasing the trustworthiness of free and open source software. The authors also thank Giovanni Mascellani for their insightful discussions on bootstrappable builds.

REFERENCES

1. Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.
2. Peter C. Rigby, Brendan Cleary, Frédéric Painchaud, Margaret-Anne D. Storey, and Daniel M. Germán. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, 2012.
3. Joanna Rutkowska and Alexander Tereshkin. Evil maid goes after TrueCrypt. *The Invisible Things Lab*, 2009. https://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html.
4. Catalin Cimpanu. Third malware strain discovered in SolarWinds supply chain attack. *ZDNet*, 2021. https://www.zdnet.com/article/third-malware-strain-discovered-in-solarwinds-supply-chain-attack/, retrieved 2021-03-01.
5. Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *DIMVA 2020: The 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 12223 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2020.
6. Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3):1–40, 2017.
7. Edward Amoroso. Recent progress in software security. *IEEE Software*, 35(2):11–13, 2018.
8. Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. Empirical study of restarted and flaky builds on Travis CI. In *MSR 2020: The 17th International Conference on Mining Software Repositories*. IEEE, 2020.
9. Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International*

Symposium on Foundations of Software Engineering, pages 643–653, 2014.

10. Chris Lamb and Ximin Luo. SOURCE_DATE_EPOCH specification. Technical report, Reproducible Builds project, 2017. https://reproducible-builds.org/specs/source-date-epoch/.
11. Mathias Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, 2014.
12. Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies, FAST 2017*, pages 59–72. USENIX Association, 2017.
13. Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.
14. Susan Landau. Making sense from Snowden: What's significant in the NSA surveillance revelations. *IEEE Secur. Priv.*, 11(4):54–63, 2013.
15. Fabrice Bellard. TCC: Tiny C compiler. https://bellard.org/tcc/, 2003. Retrieved 2020-10-05.

Chris Lamb is a freelance programmer with over fifteen of experience of developing open source software. He has contributed to the Debian operating system since 2006 and was elected to serve as the Project Leader in 2017 and 2018. He is also a director of the Open Source Initiative (OSI) as well as Software in the Public Interest (SPI), Inc. Today, he is now highly active in the Reproducible Builds project, through which he has received a grant from the Linux Foundation. Contact him at chris@chris-lamb.co.uk.

Stefano Zacchiroli is Associate Professor of Computer Science at Université de Paris on leave at Inria, France. He is co-founder and current CTO of the Software Heritage project. He is a member of the steering committee of the Reproducible Builds project. He has served as Debian Project Leader over the period 2010-2013. Contact him at zack@irif.fr.