Producing Hidden Bugs Through Reproducible Builds

Author list goes here. (Most likely, the primary contributors to the document.)

ABSTRACT

Reproducible builds have gained additional deployment and scrutiny in recent years. Many major software projects, including Tor, Debian, Arch Linux, Tails, etc. have undertaken massive efforts to make many of their packages build securely. As a result, Tor and Tails build reproducibly and between 3X-9X% of packages now build reproducibly for Debian, Arch, etc... This has been touted as a major step toward improving the security of these projects.

This paper describes unexpected benefits of reproducible builds — including discovery of a wide array of previously unknown bugs. We propose a more precise goal, specifically dictating *where* reproducible build bugs should be fixed rather than just focusing on the goal of making builds reproducible. We report on our experience working on a major Linux distribution, making XXK projects reproducible (9X%) and outline the effective tools and techniques to do so. In addition to making a major Linux distribution reproduce 9X% of its packages, our philosophy improved build time by XX%, reduced the repository bandwidth needed by XX% and also uncovered XXX unrelated bugs across XX projects, including XX critical security flaws.

1. INTRODUCTION

Since Ken Thompson's Turing award lecture about 'Trusting Trust' [1], the security world has been concerned about the potential for backdoors in build infrastructure. Concerns about malicious compilers and build systems are not theoretical, with dozens of companies compromised due to attacks in the build infrastructure JC ►*tons of cites*◄. For example, in 2008? Fedora's build system was compromised by a malicious actor JC ►*cite*◄. Despite adding a hardware security module in response to this intrusion, an attacker managed to subsequently compromise the build server in 2009? and sign maliciously backdoored copies of the OpenSSH package JC ►*cite*◄.

As a response to these security concerns, there have been recent efforts to make *reproducible builds* JC ►*cite*◄. A reproducible build is one where two different parties with similar setups[1] are able to obtain bitwise identical binaries from the same source code. A build is said to be *reproducible* if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts. The idea is that if different parties are able to build the same binary from source, then either none of their build systems are compromised or they are all compromised in the same way.

Surprisingly, very few pieces of software are built reproducibly without effort on behalf of the software maintainer or changes to the build system itself. For example, no packages from Debian in 2015 would produce identical binaries, even if the build process was run twice on the same build system. The reasons for this, which are described in more detail in Section X JC ►*fix*◄, in the majority of cases deal with the use of timestamps (5X%), timezone information (3X%), locale information (2X%), or date (4X%) JC ►*please check / fix*◄. There are hundreds of more subtle issues in software that are resolved on a case-by-case basis JC ►*cite*◄.

In this paper, we describe experiences from a major Linux distribution's efforts to make software build in a reproducible manner. Interestingly, the major benefit discovered so far from reproducible builds has not been security. The effort of making builds reproducible has uncovered a large number of latent bugs in software packages. Our experiences demonstrate that reproducible builds are perhaps even more effective as a technique for software quality assurance.

Reasoning about reproducible builds as a quality assurance tool, has led to a different set of design choices and decisions than other reproducible builds efforts. Many efforts to make reproducible builds attempt to make an environment that solves common reproducibility issues (such as timestamps, locales, etc.) by running in an emulated environment or container. Others do a post-processing step to try to strip this information out of binaries, images, compressed archives, etc. Using the lens of quality assurance, we instead focus on addressing the root cause of the errors in the relevant tools (i.e., fixing issues upstream). This technique of fixing the root cause, as opposed to 'papering over' issues, has led us to find a number of latent bugs that were undiscovered for many years. JC ►*fix*◄

The use of reproducible builds also provided other ancillary benefits. Since it now becomes easy to determine when a change to part of the build toolchain causes a change, it is easier to detect when packages do not need to be rebuilt. For example, a new version of `make` could be released, which

[1] we will more precisely define this in Section X. JC ►*fix*◄

is used to build `gcc`. If `gcc` does not change, then packages that are built using `gcc`, but not `make` (or any package compiled using `make`), do need not to be recompiled. Furthermore, reproducible builds make the differences between binary versions of the same software much smaller, allowing diffs to help save repository bandwidth. Furthermore, this also helps debugging, because changes between versions are easier to locate within binaries.

The main contributions in this work are as follows:

- We report on the experience from a major Linux distribution's multi-year effort with reproducibly building 3XK packages. We provide details about the tools, techniques, and strategies that have proven effective at making builds reproducible. Through these efforts, 9X% of the packages now build reproducibly.

- The value of reproducible builds is clearly elucidated. While the value for security is well recognized, fixing issues with reproducible builds has also led to the discovery of XXX bugs, including XX major security vulnerabilities. This demonstrates an auxiliary benefit that goes far beyond the way in which reproducible builds are currently used.

- This paper clearly describes the choices between fixing bugs that lead to a lack of reproducibility in different places in the toolchain (fixing upstream or 'papering over' differences at the end). While, different choices have been made by different projects that perform reproducible builds, we demonstrate that this enabled us to find an array of bugs that were missed by other efforts.

- We provide evidence that other ancillary benefits of reproducible builds also provide significant reasons to utilize such techniques. Making builds reproducible locates bugs in other portions of the code, reduces the size of updates, speeds up the build process, and simplifies debugging.

The remainder of this paper is organized as follows. First, Section XX describes the concepts behind reproducibility, including the philisophical arguments about which differences should be allowed in build environments. Following this, common tools and techniques for addressing reproducibility are discussed in Section Y. Subsequently, Section Z discusses our experiences with making builds reproducible is described, including both anecdotal experiences and a quantification of which techniques worked on what set of packages. Using this context, Section XXX then does a deep dive into the more general bugs found when looking for reproducible build issues. This shows situations where our philosophy of fixing bugs in the original source paid dividends in fixing important software flaws elsewhere. We then describe the ancillary bandwidth and build time benefits of using reproducible builds (Section XYZ). Section XX discusses related work, primarily on software testing and build, before the paper concludes.

2. REPRODUCIBLE BUILDS

JC ►*1 para history / motivation. Set up the next sections*◄

2.1 Reproducible builds

According to (JC ►*cite https://reproducible-builds.org/docs/definition/*◄:
A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts.

The relevant attributes of the build environment, the build instructions and the source code as well as the expected reproducible artifacts are defined by the authors or distributors. The artifacts of a build are the parts of the build results that are the desired primary output.

Following that terminology, we also define:

Source code is usually a checkout from version control at a specific revision (a 'tag') or a source code archive.

Relevant attributes of the build environment would usually include dependencies and their versions, build configuration flags and environment variables as far as they are used by the build system (eg. the locale). It is preferable to reduce this set of attributes.

Artifacts would include executables, distribution packages or filesystem images. They would not usually include build logs or similar ancillary outputs.

The reproducibility of artifacts is **verified** by bit-by-bit comparison. This is usually performed using cryptographically secure hash functions.

Authors or distributors means parties that claim reproducibility of a set of artifacts. These may be upstream authors, distribution maintainers or any other distributor.

JC ►*text to consider integrating. Entropy is a loaded term in academia, so we would need to either be very careful with how we use it or rephrase it.*◄ A "build process" is a complicated function that has a complicated output. Ways to make that function deterministic include (1) fixing the input to the function (VM), (2) using a comparator that ignores some of the output bits (strip-nondeterminism), (3) changing the function so it is deterministic. Intuitively, since (3) dives into the guts of the build process (in contrast to (1) and (2) which are more black-boxy), it is the most difficult/invasive, but also the most <handwave>correct</handwave> and the most rewarding.

2.2 What should be assumed about the build environment?

JC ►*3-4 paragraphs, itemized list*◄

==========

Reproducibility and buildinfo files ==========

WORK IN PROGRESS

This is a plan for the moderately-far future and is not intended to block existing progress on buildinfo files that will not have the features discussed.

Building software ==========

Where the world is now: Build products contain build-specific information. Often, the product is signed "for security", and sometimes this signature is embedded inside the build product.

Where we want to get to: Build products are reproducible by *any* member of the public *entirely from source code*. Build-specific information is generated and stored separately. This includes cryptographic hashes of the build products and may be signed by the builder.

The rest of the document will describe why this approach is superior.

What is reproducibility ==========

Build products, for us to classify them as "reproducible", *must* be exactly bit-for-bit identical. We cannot accept anything less than this, because it would greatly increase the cost of verifying identical *behaviour*. Suppose we wanted to write a "compare only behaviourial differences" program that e.g. ignores timestamps, hostnames, etc. Then,

- New data formats with their own ideas of "trivial" will need to have this logic incorporated into this program in the future. This is not scalable.

- If we want to compose build products in different ways in the future (e.g. in newer container formats, such as archives or installation media images), we would have to extend this "comparison" tool to look inside those containers, *even if* the containers themselves are bit-for-bit identical. This is not scalable either.

- For Turing-complete data formats, it is not possible to write a program even in theory that says "behaves the same" or "behaves differently". For example, a program could read its own timestamp and do different things according to this value. So the result of our diff program would not actually mean "behaves the same/different" but instead mean "behaves the same/different if the source code doesn't do certain things". Granted, reproducibility is meant to eventually help us verify source code behaviour, but tying this to the output of our diff program tangles up separate concerns and greatly increases the complexity and cost-of-reasoning of our entire system.

- For data formats containing natural language such as documentation, a similar argument to the above applies. For example, text could refer to the timestamp embedded in the page footer and mean different things depending on its value.

Therefore, no such automated "behavioural differences" tool can exist; there will always have to be some level of human review over its results - and each verifier must perform this themselves, otherwise it defeats the point. This is not scalable across the hundreds of thousands of released packages (including versions) in our FOSS ecosystem today that should all be reproducible. So, we firmly commit to bit-for-bit reproducibility, which is the only test that can be automated at scale.

Degrees of reproducibility ==========

Simply being able to reproduce a binary, even bit-for-bit identically, does not give us very much useful information. Let's introduce some thought experiments:

Thought experiment 1. If we fork the universe at the start of a build, then the build output is reproducible in both cases.

Therefore, everything is reproducible in some sense. This is not merely a pedantic example; setting parameters of a scenario to extreme values helps us identify the important parts of it. Let's reduce the extremity of the parameter a bit:

Thought experiment 2. Assuming the build does not depend on information outside of the machine (network, entropy, IO), then if we clone the state of the machine (either via VM snapshot, or the atoms of the physical machine), then the build output is reproducible in both cases.

Yes, we can reproduce a build by snapshotting a VM that was specifically set up to do the build. But what does this tell us? Not very much - a reviewer would have to not only look at the source code of the build inputs and tools, but also the snapshot of the VM, to make sure that it's not doing anything funny.

So now we see that *how* we are able to reproduce a build, matters tremendously in how useful this information is. More generally, when we verify a build, we:

1. Reproduce *some* of the universe U from the original build, call this U'. 2. Run the build on U'. 3. Verify bit-for-bit reproducibility against original product.

When we run this process across many verifiers, they will all reproduce U', and may have different values for their own U - U' . The more processes we run, the more confidence we gain, that U' is a superset of the minimal information T that we actually need to reproduce the build. But even after running this process, a human reviewer still has to review U' to check that it contains no backdoors: since it was the same across all builds, there is the possibility that U' = T and all of it was needed to affect the final build result.

So, it is in our interest (to make verification easier) to reduce U'. If we reduce U' such that it is no longer a superset of T, then we will fail to reproduce the original build. By running many reproductions with successively smaller U' (across many verifiers), we can gain confidence in what T is. Beyond that, developers can try to tweak their source code, or the source code of their build tools, to reduce T itself down.

As an ideal standard for *all* packages to aim for, T should exactly be the source code of the build input and the build tools - i.e. the **preferred form for verification** (against backdoors etc) - call this S. To verify a build for S-reproducibility, we recursively build the source code of the build tools, *not even care about their exact binary result*, use these to build the build input, then finally compare our result against the original build product[2].

In practise, we do not expect most existing packages to meet this standard, and our current (2015-12-11) reproducibility tests instead reproduce the entire *binary* build tools (i.e. an approximation of the state of the filesystem from the original build) when verifying. One has to start somewhere, and proceeding one concrete step at a time is better than trying to do too much at once.

As an interesting side note, sometimes though we can do *even better* than S-reproducibility:

~~**Tho~~ought experiment 3**. Given 'cp' as a build tool, the build output *ought to be* reproducible *no matter what the source code or the binary of the version of cp that we use is*, assuming that 'cp' is correctly implemented.

Of course, this depends entirely on the build process - for example, one does not expect different C compilers to generate the same binaries. But if any parts of build process are precisely defined like 'cp', then this reduces T even further, replacing concrete source code with this smaller definition.

Buildinfo files ==========

Before the above theory was developed, there was confusion on whether buildinfo files should be for:

- reproducing the original build product *no matter what*.
- reproducing using a specific U' that we were using on reproducible.debian.net in practise, that intentionally excluded

[2]Yes, this ignores cyclic-build-dependency and bootstrapping issues. We'll have to figure this out later, when we actually start to try it. One plausible approach is to double-diverse-compile the initial compilers (that self-build-depend) using existing binaries. One may think of it like this: DDC allows us to verify self-build-depending tools such as compilers, and S-reproduction allows us to verify other build products.

things like hostname/timestamp but for practical reasons included build path. - reproducing using T. - reproducing using S.

After developing the above theory, it becomes clear that buildinfo files should contain as much information as possible (i.e. of U), of course considering storage and distribution costs. Then, it is up to the *verifier* how much of this they want to reproduce (U'), depending on what they are aiming for.

This also gives a nice alternative for traditional reasons for including things like hostname, timestamp, build path, etc. in the build product - just put it in the buildinfo file instead, then you can have this data *and* a bit-for-bit reproducible build.

To finally state the definition:

A buildinfo file is a committment from a builder that they executed the build with certain parameters, and got a particular binary output with that input. The information should contain as much as information as possible, taking into account storage and distribution costs, and MUST *attempt to include* **all information needed to reproduce that build** (i.e. an over-estimation of T). External artefacts MUST be referenced by hash, SHA256 or stronger.

This definition is meant to allow readers of the file to:

- trace the build back to the original builder for debugging purposes

- to re-execute the build, using (subsets of) the information contained in it, and verify the build product

- to calculate the minimal set of information needed to reproduce that build product ("T" from the above section), e.g. via the following strategies:

- intersect common information from multiple buildinfo files that produce the same build product - iteratively re-execute builds, recreating succesively fewer and fewer information from the original buildinfo file

- to tweak the build input to attempt to reduce the aforementioned minimal set, which may be re-calculated for this new input by running the aforementioned strategies again.

Buildinfo files SHOULD be signed, but there may be rare applications where this is not suitable. You should have a very good reason for this, though.

The buildinfo file itself MUST NOT suggest that certain types of build-time information are "more important" for reproducibility than other types. We have already taken such a position on this matter, but holding that position should be the job of the rebuild-verification program. This reduces the complexity of the overall ecosystem of reproducibility tools.

3. REPRODUCIBLE BUILD TECHNIQUES

JC ▶*describe all the tools / techniques here. SOURCE_DATE_EPOCH, strip-nondeterminism, docker containers, reprotest, etc. 2-4 paragraphs per tool / technique*◀

- `SOURCE_DATE_EPOCH` is a distribution-agnostic standard for upstream build processes to consume a deterministic timestamp from packaging systems. Packaging systems set the environment variable `SOURCE_DATE_EPOCH` to the timestamp of the original public release of the source code being built, and build processes consume that value and use it instead of the system timestamp where appropriate, such as when embedding timestamps into user-visible portions of generated documentation. This allows a particular source tree JC ▶*DS: need to define a term for 'source package/archive'—meaning a '.tar.gz' or a tag—where 'Artifact' is defined*◀ to remain reproducible even if built on different dates. JC ▶*how does this deal with elapsed time?*◀

  The value of `SOURCE_DATE_EPOCH` is a UNIX timestamp, defined as the number of seconds (excluding leap seconds) since 01 Jan 1970 00:00:00 UTC, and is exported through the operating system's usual environment mechanism. On C-based system, it is available via the `getenv(3)` standard library function. `https://reproducible-builds.org/specs/source-date-epoch/` ?Dozens? of popular build tools, including XX, YY, and ZZ, have been modified to use `SOURCE_DATE_EPOCH`.

  The use of `SOURCE_DATE_EPOCH` is a compromise solution that is not universally loved by reproducible builds projects for a few reasons (cite https://wiki.debian.org/ReproducibleBuilds): First, it requires changes in the build system for different projects, which makes it difficult to gain adoption in some cases. Instead it may be easier to simply have the build system omit information that leads to non-determinism. Second, this technique breaks build systems, because files will have the same timestamp (e.g., setting the system clock to some constant time T, breaks various build tools such as GNU make). Third, it also does not actually remove the irreproducibility issues. Setting the system clock to some constant time T at the start of the build, then letting it run normally, does not achieve reproducibility in the cases where the build takes a non-deterministic amount of time and something embeds "the current time" near the end of the build process. Finally, mandating a specific start time or build path prevents parallel builds from being done without extra system-level support (e.g. virtualisation or separate kernel namespaces), and prevents users without permissions to use that path from verifying reproducibility. Combined, these limitations make `SOURCE_DATE_EPOCH` a solution with significant drawbacks.

  JC ▶*DS: something about 'WW packages had build processes that would fail within their support lifetime, and have been fixed'—this makes developing/backporting vulnerability patches easier*◀

- *strip-nondeterminism* is a tool that is run over the finished archives and image files that removes common sources of non-determinism. This includes timestamps and similar metadata from the files. JC ▶*need much more detail.*◀ The drawback of this is...

In addition, there are several tools that are used to aid efforts to make builds reproducible.

- *diffoscope* is a recursive and content-aware diff utility used to locate and diagnose reproducibility issues. Unlike the traditional `diff(1)` utility which mostly focuses on determining if files differ, *diffoscope* is optimized to help dig through differences that relate to reproducible builds. One major improvement is in the way that binary differences are handled. Rather than just observing a difference, diffoscope parses the files in the appropriate format and understands the context of the change. For example, differences in PDFs fields

may be decoded to indicate that the difference is in the timestamp field. JC ►fix?◄

Another major improvement is that diffoscope will recursively unpack archives of many kinds. Package formats are usually just standard archive formats, with additional fields that indicate metadata for the package. In fact, many archive formats contain several archives, which can be nested inside each other. For example, the Hello World Debian package `hello_2.10-1+b1_amd64.deb` has an outer AR archive, which contains a format file, a gzip-compressed tarball containing metadata, and an xzip-compressed tarball containing the files to be installed—some of which are themselves compressed. In larger packages, the files to be installed may include archive files, such as JAR or ISO files. By recursively processing files, diffoscope can pinpoint that a difference is due to the XX inside of YY inside of ZZ in this package.

- *a .buildinfo file* captures information about the system environment in which a build is performed. One obvious use of this information is to examine the environment a build happened in, perhaps for forensic purposes. However, a `.buildinfo` file may also be used to recreate that build environment later. This enables different users to build software in a reproducible manner, with the `.buildinfo` as a guide. JC ►How do we clearly distinguish this from requiring something like `SOURCE_DATE_EPOCH`?◄

  It is also envisioned that the `.buildinfo` file may have a variety of other uses in the future. First, a debian user could elect to only install binary packages that have been successfully built by multiple builders. JC ►How does this work? Are these files signed?◄ Second, a `.buildinfo` file could be used by a debian derivative to attest to the packages it has deterministically rebuilt. Third, a triage process could use `.buildinfo` files to identify toolchain changes that have an effect on large numbers of binary packages (e.g., a backdoor in a commonly used library cite Xcode Ghost). Finally, `.buildinfo` files used as part of a cross-building process could demonstrate the correctness of the cross-compiling toolchain by reproducing the exact binary artifacts.

4. EXPERIENCES MAKING BUILDS REPRODUCIBLE

JC ►This part of the eval focuses on efforts to make builds reproducible. What sorts of flaws were found where? How were they fixed? We need to make most of this quantifiable. Lots of data about number of packages fixed with each type of fix.◄

JC ►some anecdotes about fixing bugs / interesting bugs are most welcome. This should be appropriately organized. (I can fix the org later).◄

...

5. BUGS FOUND

This section decribes the flaws found via making software reproducible. These bugs are *not* merely situations where reproducibility does not work, but instead covers (seemingly) unrelated bugs uncovered via this process.

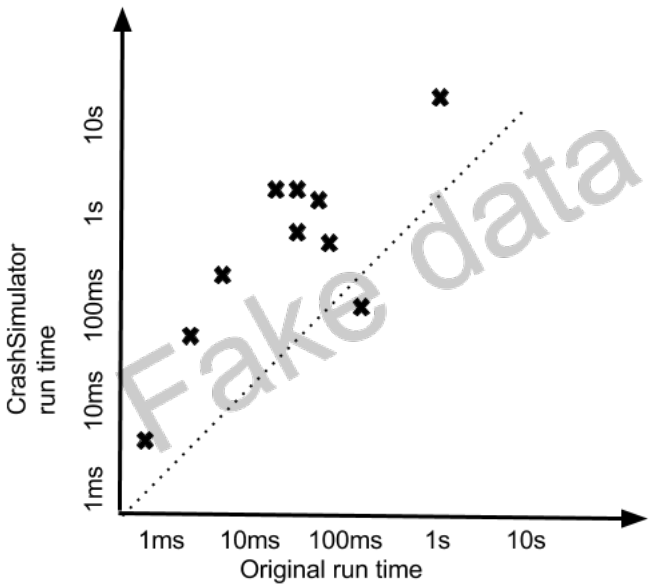

Figure 1: ***I like to sometimes create figures with fake data so that I think through what I want to show before running an experiment that takes days. DEFINITELY LABEL THIS CLEARLY SO IT IS NEVER SUBMITTED. Look how clearly I labeled this as fake data!!!***

JC ►likely have a table showing some quantitative data about the bugs. Hopefully we have at least a few dozen examples. It would be fine to categorize and group them it there are too many to list.◄

JC ►General ideas about what was found. Discussing interesting details of most types of bugs.◄

race conditions. One type of bug that was frequently found were race conditions in the build process. This was often the case where a build script would start multiple copies of the same tool. One cause for this was when those programs would share common temp files, caches, etc. JC ►cite `https://bugzilla.opensuse.org/show_bug.cgi?id=1021353` `https://bugs.launchpad.net/intltool/+bug/1687644` ◄. The issue is that the programs will either overwrite or corrupt each other's state, leading to errors in the output files. JC ►What issues does this cause in the installed software? Is there some sort of security / stability flaw in the eventual code?◄ In these two cases, this only affected manual pages, but it is possible that similar flaws exist in code that produces executable output.

Another way that race conditions cause issues deals with checking timestamps on files. For example, a build script may start two different programs that check some property of timestamps and decide how to behave. For example, in the build script for `python-bottle`, the Python2 and Python3 build scripts are run one after another. However, if both the Python2 and Python3 build scripts complete within the same second the 2nd install script will think it is already done, because the output file size and timestamp already have the expected value.

JC ►are there time-of-check-to-time-of-use bug or something else?◄

Another variant on this involves copying build files over each other at different times. For example, in `mtr` the `Makefile.dist` file was being copied over the `Makefile`, leading

to a race condition. To quote the bug report: JC ►*cite* *http://pkgs.fedoraproject.org/cgit/rpms/mtr.git/commit/?id=9dd4325251e1c28064f4bf5b84ae2f95e3118200*◄ Note that make doesn't wait for this background task to finish. During rpm build we are building mtr twice. After first build we call distclean. If second invocation of configure script runs in less than 3 seconds then the Makefile generated by configure will be overwritten by background copy. We don't want that and since we are calling configure explicitly we don't really need this "feature" at all.

6. RELATED WORK

JC ►*Explain the rough grouping of the related work at a high level*◄

JC ►*break it down into categories either by paragraph or so, or have subsections.*◄

JC ►*be sure to talk about how recent efforts to add binary diversity are not counter to this. Any efforts to do binary diversity, could be done on the machine, after the package was unpacked, but before it is installed.*◄

7. CONCLUSION

JC ►*Explain the few key takeaways. Benefits, eval results, usage, what is new. If code / data is available, reiterate. optionally, explain future work.*◄

8. REFERENCES

[1] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, Aug. 1984. Turing award lecture.