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, ... have undertaken massive efforts to make many of their packages build securely. As a result, Tor and XXX 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' JC ►*cite*◄, 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 JC ►*perhaps find an official definition instead*◄. 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, only XX% of packages from Debian in 2014? would even produce identical binaries 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 is used to build `gcc`. If `gcc` does not change, then pack-

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

ages that are built using `gcc`, but not `make` (or any package compiled using `make`), need not be compiled. JC ►*could use help with this example.*◄ Furthermore, reproducible builds make the differences between binary versions of the same software much smaller, allowing diffs to help save repository bandwidth.

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.

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 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*◄

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*◄

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.

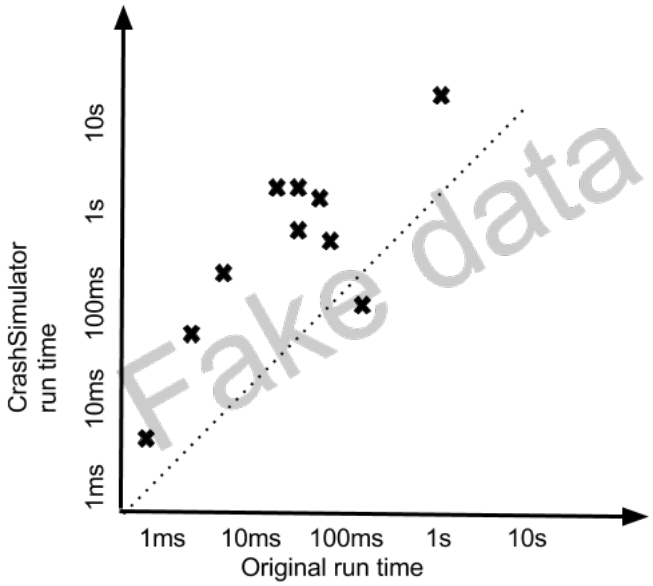

Figure 1: ***I like to sometimes create figures with fake data so that I think through what I want to show before running an experiment that takes days. DEFINITELY LABEL THIS CLEARLY SO IT IS NEVER SUBMITTED. Look how clearly I labeled this as fake data!!!***

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 binaries. JC ►*What issues does this cause in the installed software? Is there some sort of security / stability flaw in the eventual code?*◄

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 concurrently. However, if the Python2 scripts complete in the same second they are started, the Python3 build scripts fail to run. JC ►*why?*◄ 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