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 a unexpected benefit of reproducible builds — discovery of a wide array of previously unknown bugs. We propose a slightly unorthodox philosophy JC ▶*Am I overstating this?*◀ 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 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 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

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

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.

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. 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

JC ▶*clear definition, concepts, 2-4 paragraphs*◀

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).*◀

...

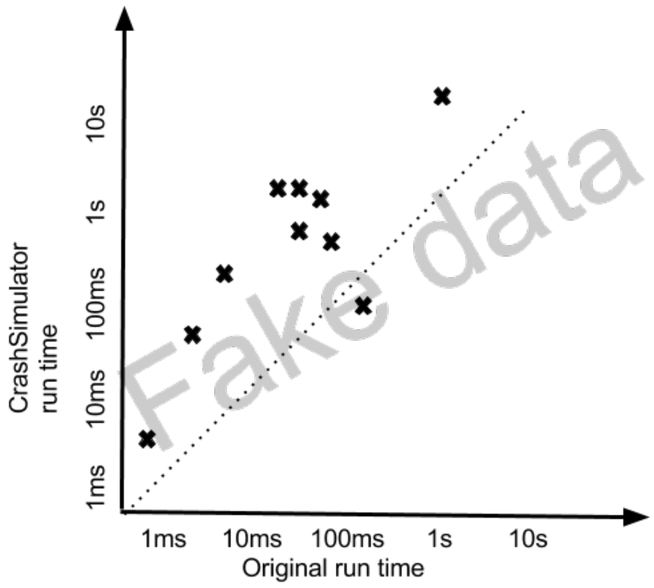

Figure 1: ***I like to sometimes create figures with fake data so that I think through what I want to show before running an experiment that takes days. DEFINITELY LABEL THIS CLEARLY SO IT IS NEVER SUBMITTED. Look how clearly I labeled this as fake data!!!***

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.

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.*◀

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.*◀

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