École Polytechnique Fédérale de Lausanne

Root cause localization of non-reproducible builds using llvm-diff

by Fabio Aliberti

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Anthony Vennard / Flavio Toffalini
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 10, 2022

"No dragon can resist the fascination of riddling talk and of wasting time trying to understand it."
– The Hobbit, or There and Back Again

Acknowledgments

I would like to thank Alexander De Battista Kvamme and Eskil Jarlskog for keeping me sane, and the water cooler for keeping me hydrated. Also, we would like to extend our thanks to all volunteers that maintain the continuous tests of the Archlinux packages.

Lausanne, June 10, 2022 Fabio Aliberti

Abstract

Reproducible builds ensure that two compilations of the same source code result in the same result. More often than not, this is not the case and the build is non-deterministic. Little research has been done in tracing back the root causes of such non-deterministic cases. Current solutions contain a lot of noise or are dependent on high-quality heuristics. We propose a new approach on the compiler level, that allows the back-tracing of differences in compilations. This facilitates the localization of the root cause of non-determinism, while being simple in its implementation. We test our approach against 20 real-world non-reproducible packages. Additionally, we run our tool over two large packages to see if it scales. This is successful for one of the tools. However we can not use our approach for the second one. We measure the overhead of our approach against a standard compilation. There, we observe an average overhead in memory of 5.9%. The overhead in compilation time we measured was on average 34.3%. Moreover, our approach is blind to any form of non-determinism outside the scope of the compiler.

Contents

Chapter 1

Introduction

Open-source software is built on the principle, that the more eyes look at code the more likely issues will be detected. Therefore, to ensure correctness and safety, open-source software allows for the inspection of source code. However, most software is distributed pre-compiled. There is no guarantee that a binary is generated from the source code. This can allow an attacker to publish any type of malicious software, without being detected. This detection is hard, because compilation is one way. Ideally, every compilation of the source code produces the same binary. If this is the case, an independent auditor can recompile the source code and compare the result with the distributed binary as shown in Figure 1.1. However, it is seldom the case that packages are reproducible like that. This can be due to timestamps, randomness in the build, or even compiling the code with malicious sources. The developer of a project is responsible that this does not happen. To that end, the developer can check whether or not their build process is reproducible. However, the build systems that exist are very complex, and the build process is lengthy. Thus it is often easier to check for reproducibility in the generated artifacts. A naïve approach would simply compare the final binaries bit for bit. This way, one can spot a non-deterministic build. However, with the difference alone, it is hard to find the root cause. Existing work such as [6] relies on good heuristic filtering of build logs. This limits their application to supported build environments. Others [7] rely on system call tracing. This is very complex and requires special operating system permissions. We aspire to a simple solution to trace back the causes of non-determinism. So far, there have been no attempts at this problem using the compiler directly. We introduce reprobuild, a prototype for analyzing the differences in two builds. Our tool leverages the LLVM framework to localize, and trace back the causes of non-determinism in builds. More specifically, we compare semantic differences in LLVM bitcode to determine differences. This allows us to do passes independently of build environments. Our evaluation focuses on 20 real-world Linux packages. We choose them from a list of known to be unreproducible packages. In addition to that, we also test two large packages namely Emacs and Chromium. We have found preliminary results, especially for preprocessor-based non-determinism. We can detect timestamps, build directory, and unstable order of input. However, since our approach focuses on the compilation step, we are blind to anything that is not generated

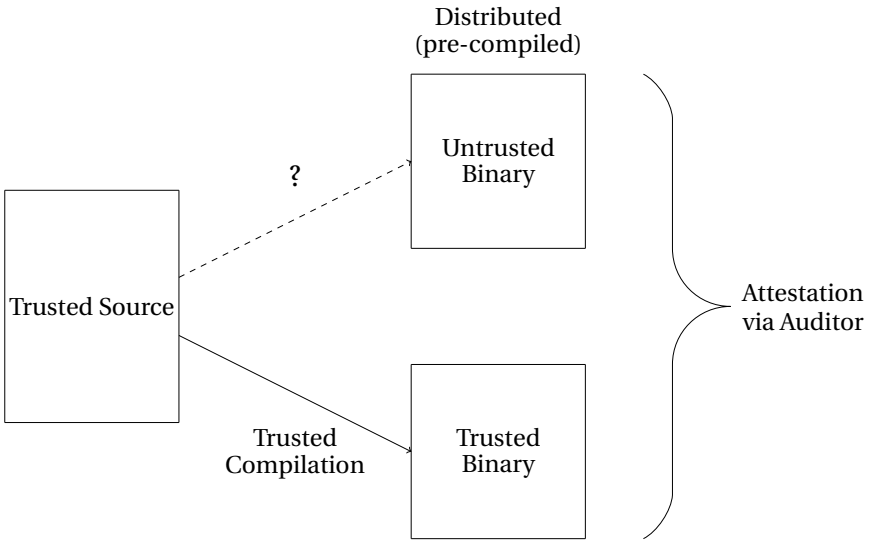

Figure 1.1: If reproducibility is achieved, attestation is possible by a trusted party.

by a compiler. Moreover, the process is not robust enough to be used for larger builds. This is because builds are not designed to support the generation of LLVM bitcode. To investigate the overhead of our tool, we run the same real-world packages against a normal compilation. Compared to those, we observe an average overhead of 34.3% runtime and 5.9% in memory. We see an increase in the overhead for larger builds. However, due to the small sample size, this is inconclusive. We aimed to investigate the viability of using LLVM as a tool to detect non-determinism. To that end, we created a tool that can be used independently of the build system. We introduce core concepts such as reproducibility in chapter 2. Our design choices are outlined in chapter 3. In chapter 4 we cover all implementation details. We discuss our findings in chapter 5. Relevant related work is covered in chapter 6. Finally, we end in chapter 7 with a conclusion.

1.1 Motivation

To build trust in distributed packages, users should have the ability to verify if it was compiled from the claimed source. Because building a package is a one-way function, this cannot be done directly. However, if a build is entirely deterministic, a user can recompile the build, and compare it against the distributed package. We want to help developers make their builds reproducible. Though there are tools, that can detect differences in non-deterministic cases, it may be challenging to find out why these differences occur. One of the main challenges is, that builds are complex and divers. There is a large number of build tools available, without a standard.

Chapter 2

Background

To understand how our tool works, some background needs to be covered. Firstly, about LLVM and its framework. This includes a compiler wrapper that helps us in our compilation. Secondly, about containers that are used to model different environments.

2.1 LLVM

LLVM [3] is a compiler infrastructure, that allows for language and platform-independent optimizations. By compiling the source to an independent intermediary representation, a large number of optimizations can be performed. Furthermore, this intermediary representation can then be compiled to the target architecture. The main idea behind this is to use a frontend per language and a backend for each architecture. This means, there is no longer the need for a compiler per language-architecture pair. In Figure 2.1, we can see the principle of this approach using LLVM IR.

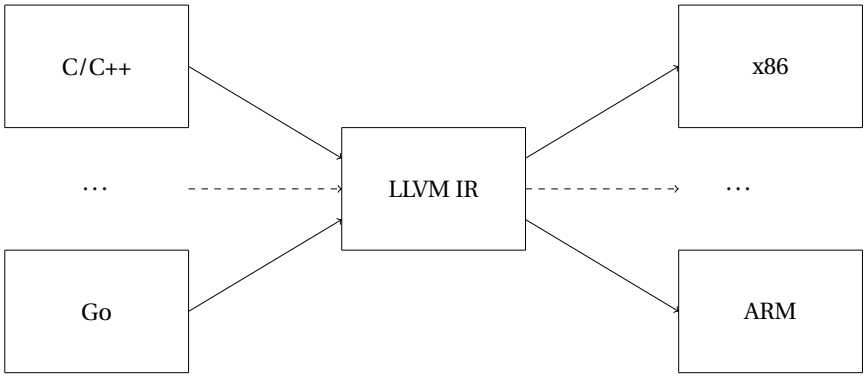

Figure 2.1: Overview of the LLVM concept.

2.2 LLVM IR and LLVM Bitcode.

LLVM IR is the intermediary representation, for the optimization passes. It is a low-level language, that can be described as a middle ground between C and Assembly. LLVM bitcode is a pseudo-binary, akin to Java bitcode. It encodes the exact behavior of the binary and can be run in the LLVM virtual machine. This bitcode can be translated back and forth to LLVM IR without information loss. Thus we may refer to the bitcode as IR in this paper, as they are interchangeable for our purposes. In a normal compilation, the bitcode is not exposed to the programmer and is used by the compiler alone. For C/C++, this compiler is Clang [1].

2.3 WLLVM

To generate the intermediary bitcode, Clang can be invoked with the `-emit-llvm` flag. For each object file that is generated, the intermediary representation is saved. However, the final linking step is not performed on this level. For our purposes, it is also necessary to use this flag independently of the build process. Wllvm (Whole program LLVM) [11] is a Clang compiler wrapper written in Python. It allows us to generate LLVM bitcode for all intermediary files and for the final binary. This is done by embedding the bitcode directly in the object files. Then it links the bitcode using the llvm-link tool in the linking step. In other words, we get the linked binary in IR, which would not be the case with the `-emit-llvm` flag. Thus, the LLVM bitcode will be semantically the same as the final binary.

2.4 Containers

To check differences in the bitcode, the compilation should run under different environments. Containers provide a convenient way of running different environments, without actually installing the packages required to build the target. Compared to virtual machines, containers do not run in their own kernel, but atop the existing host kernel. They build upon the Linux namespaces [4], to separate interactions and resources. Thus, they are more lightweight and faster to set up as opposed to VMs. Docker and Podman are the two main container providers. While the first one is more popular and has more support, the latter does not require root privileges to run. Both support an abundance of operating systems.

Chapter 3

Design

In our design, we choose to make our tool as configurable as possible while staying simple in the execution. We use a generate and compare approach, that is, we run the compilation multiple times, and compare the output of the runs. Our approach is split into three phases. The setup, the compilation and the comparison. Figure 3.1 shows this approach for two environments. In the setup phase, the different environments are set up. This includes introducing variables in the environment as necessary. The compilation step then fetches and builds from the source code. In this step, we also generate and link the necessary LLVM bitcode. Finally, we compare the resulting files and output the differences if any. This is done with llvm-diff, a tool from the LLVM project that compares LLVM bitcode.

3.1 LLVM and supported architectures.

In theory, any language that can be compiled to LLVM can be used in our tool. Currently, we have only compiled C and C++ packages. However, thanks to our design, we may use any language, that can be compiled to LLVM IR, and output it. This includes binaries and libraries that are compiled from C/C++. Even though we are currently limited in the number of languages, we are not limited in the build tools. As anyone that has tried to package anything can attest to, there are a lot of build tools out there[1]. We have considered finding a solution for a single build tool, however, we decided against it. We aim to run our comparison outside of the boundaries of these tools. To that end, LLVM is a good choice, since it operates on the compiler level, rather than the build level.

[1] See *https://xkcd.com/927/*

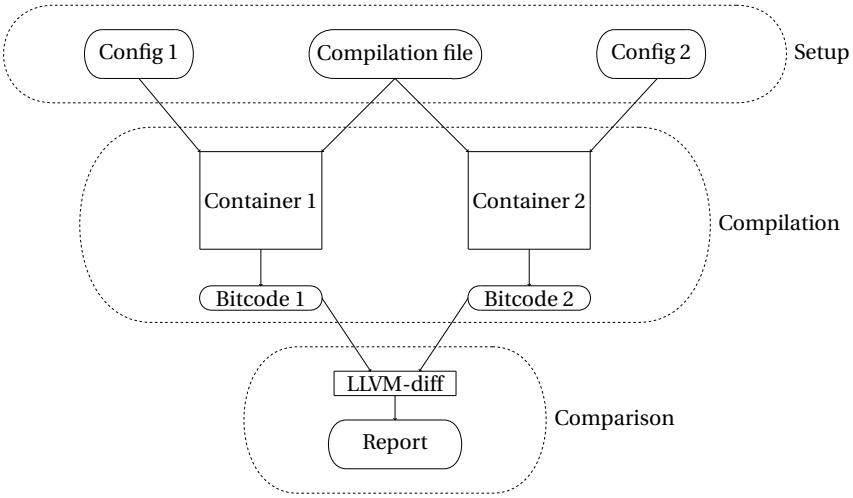

Figure 3.1: Overview of the tool

3.2 Docker Containers

Ro test various parameters, we need to run the compilation in different environments. Containers are a natural choice for this. They separate our compilation without the need for virtual machines. These would also be a suitable candidate, however, most virtual machine monitors are hard to communicate with, take a long time to start up, and need to be set up beforehand by the user. The downside of containers is that we are constrained to Unix operating systems. This is not a limitation, since we assume that a binary will unlikely be reproducible under different operating systems. Using docker requires sudo privileges, which may be undesirable. To that end, Podman is also supported. However, non-root usage is not natively guaranteed by all images, and the configuration of such cases is left to the user. To keep the tool lightweight, docker containers are spawned on the fly, and the prerequisites installed. After the compilation step is done, and our artifact is generated, the container is removed, so as not to bother the developer any longer. To ensure the developer can tweak the containers to their liking, we use configuration files.

3.3 Configuration files

Having a form of configuration is necessary, as the compilation has to run in two containers that may be set up differently. Moreover, configuration files allow us to easily maintain differences. As opposed to command-line options, they are more adaptive to changes in a workflow and more easily documented. If anything does not work we can tweak the configuration files, instead of remembering the command line options. This configuration file in JSON format contains a list of JSON objects that describe the containers. Figure 3.2 shows an example of such a configuration file. The information, that has to be included is most importantly the path to a compilation script. This is a bash/python executable script that will be run in the shell. Usually, these are the same

```
[
    {
        "container": "podman",
        "prereq": [
            "clang",
            "llvm",
            "make",
            "python",
            "python-pip",
            "wget",
            "tar"
        ],
        "compscript": "nano/compile.sh",
        "environment": "Arch",
        "mount": "env1",
        "internal": "/build"
    },
    {
        "container": "podman",
        "prereq": [
            "clang",
            "llvm",
            "make",
            "python",
            "python-pip",
            "wget",
            "tar"
        ],
        "compscript": "nano/compile.sh",
        "environment": "Arch",
        "mount": "env2",
        "internal": "/build"
    }
]
```

Figure 3.2: Example of a configuration file for the nano package.

across different objects but can be changed, if the build process is different across the containers. Moreover, if one wishes to change locales or directories one can do so in these. The compilation script contains all information that is needed to generate the LLVM bitcode result files of that container. This means, that the actual implementation of the generation of the LLVM bitcode is up to the user. We recommend, however, using a compiler wrapper such as wllvm that outputs the linked LLVM bitcode file of the binary to simplify the process. As stated previously, we do not expect a binary to be the same across operating systems. However, it may be the case, that it is independent of Linux distribution. For this reason, we allow for different docker images to be compared against one another. Which Linux distribution is used, can be changed from container to container. We recommend using the already supported distributions i.e Arch, Ubuntu, or Debian. However, a custom container can also be used. In this case, installing required packages has to be done via a separate init script instead of the configuration file. For the supported containers, the list of required packages can be put into the JSON object directly.

Chapter 4

Implementation

In our implementation, we choose a way that is easily modifiable. We do not assume any structure of the build system. Instead, the compilation step takes care of generating the output that has to be compared. The output comes from at least two runs of the compilation.

4.1 Creating different environments

As mentioned in chapter 3, we use containers to model different environments. Though the kernel used is the same, naturally, the compilation time will always be different. Moreover, the build directory can be conveniently set in the configuration file. Any other parameters such as locales can be changed in the compilation or initialization file. To make sure, that the environments are new each time, an image is pulled from the hub. As soon as the container is spawned, it executes the compilation script.

4.2 Generating output

This compilation script should completely build the package and produce the bitcode from source. Luckily many Linux tools written in C/C++ have a similar build process. The standard recipe `./configure && make` comes from the GNU build standard[1]. The `./configure` step generates all the build files needed, based on the environment. Then the compilation is performed with `make`. In this case, using wllvm is straightforward, as we need to set it as our compiler during the configure step. However, not all tools follow this recipe. Sometimes, the compiler environment variable is more of a suggestion, and it is required to use Clang/GCC always. In those cases,

[1] Even though it is considered more as a suggestion rather than an enforced rule, one will hardly find a GNU package that does not follow this recipe

we need to inject wllvm as the compiler in the generated makefile directly. It is usually done with the `sed` command to search and replace the compiler variable with wllvm. In general, one has to know how the build tool prepares for the compilation. In the end, this is up to the developer. In the compilation file, they can generate the output however they like.

4.3 Comparing the output

To compare the outputs, we use llvm-diff, a difference engine for LLVM bitcode. By running over the CFG, the bitcode can be compared, even if some optimizations such as inlining have been performed. However, this raises the problem of function ordering. Even though the control flow graph may be the same, the functions may appear out of order. This is not checked by llvm-diff natively. Thus we need to adjust the code, to report the order of the functions. The llvm-diff tool already checks if all functions in one file are contained in the other one. So it is sufficient to check for the exact order there.

4.4 Reporting the cause

For all other cases, that are not function ordering, llvm-diff automatically reports the violating line of the IR file. This output is in LLVM IR, and as such human-readable, but perhaps difficult to understand. Therefore, the tool must output the location in the source code. By embedding debug information in the LLVM bitcode, we can trace back where the differences are in the source code.

Chapter 5

Evaluation

In this project, we intend to analyze our tool by asking the following project questions.

- **PQ1:** Does our tool correctly report the cause of non-determinism?
- **PQ2:** Does our tool scale to real-world examples.
- **PQ3:** What kind of overhead does using our tool produce?

To answer PQ1, we create small examples, that we know how and where they are non-reproducible. Then we checked with real-world programs, that are known to be non-reproducible. To answer PQ2, we test it on a large project, in particular Emacs, as well as a huge project namely Chromium. We investigate PQ3, by analyzing the time, and memory it takes for the compilation whilst generating LLVM IR. This is then compared to a standard computation.

Evaluation environment. Our evaluation was performed using an Intel®Core™i7-8700 CPU @ 3.20GHz with 16GB of memory, running Ubuntu 20.04. All containers checking the differences use Archlinux as the distribution. We run the containers using Docker, but it should be noted, that all run tests can be run with Podman.

Sample selection. To test against real-world packages, we need to know whether a given build is reproducible or not. Several Linux distributions take part in the reproducible builds project. One of which is Archlinux, which aims to attest all packages available from their package manager. They provide a list [8] of continuously tested packages. This will serve as a rough baseline for our evaluation. Note, that not all packages listed in the continuous tests are useable. Since our tool currently works with C/C++ packages only. We choose 20 non-reproducible and two reproducible packages from the list of tested packages. The two reproducible packages are

chosen by preference, as the Author uses them frequently. The 20 non-reproducible packages are picked at random[1].

5.1 PQ1: Asserting correctness

To see if our tool can detect basic non-determinism, we create artificial examples, each of which contains a single form of non-determinism. In our small examples, we can detect various non-determinism. This includes non-deterministic macros, linking faulty libraries, and unstable order of inputs. As expected, the tool can find and report back the underlying issue. For the order of inputs, we can detect that it occurs. However, we cannot detect where it arises. This is because the non-determinism does not lie in the source code, but rather in the order of the compilation. These examples served as a baseline, and it was expected that we can detect such examples. However, the real world is more complicated. For this reason, we choose real-world packages to check the correctness of our tool. We successfully build and compare 20 such projects. During our testing, we detect the underlying cause for 11 of those. However, we notice for the rest of the packages, that nothing is detected. Upon further inspection, we see the underlying cause being the log or configuration files. For those, that we can detect, all of them come from timestamps in the form of ctime usually. Within the log/configuration files the main cause for the non-determinism seems to be the hostname and build path.

Answer to PQ1: Or tool correctly detects and traces back the root cause for our examples. However, we observe that there are many files, over which we do not have control over.

5.2 PQ2: Investigate scalability

In our tests, we also use four, what we consider, large projects. Of those, we will focus on one in particular.

Large projects Emacs [2] is a highly extensible display editor. It is one of the larger projects from the GNU environment. It is interesting to us because it is also not reproducible according to the continuous tests. Our tool successfully compiled emacs and detected the underlying cause of the non-determinism. The rest of the large packages follow this example. Compared to the smaller examples, they may contain multiple executables or library files. We detect non-determinism in all of them, the compilation time being the prevalent cause. However, in these projects, we see automatically generated documentation for the first time. Interestingly, this documentation also contains the compilation time.

[1] This randomness is modulo checking whether or not we can use the package.

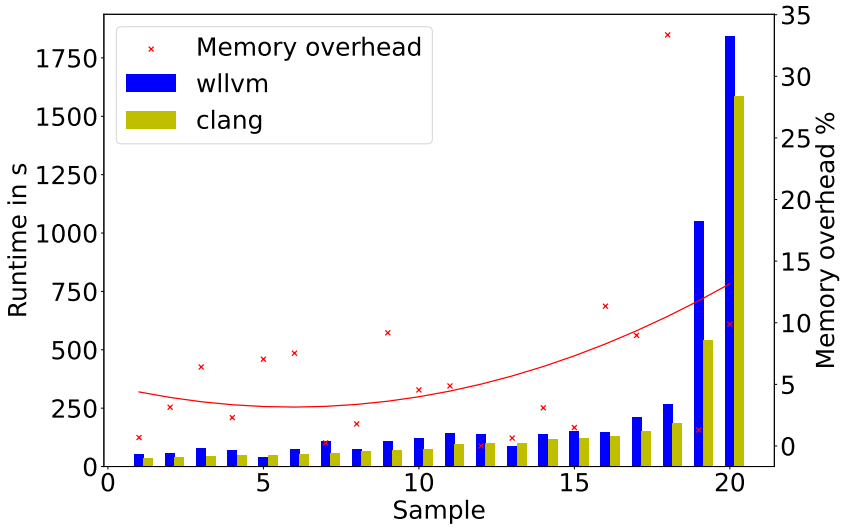

Figure 5.1: Comparison of compilation with and without generating LLVM IR. The bar diagram shows the comparison of the runtime. The scatter plot is the memory overhead, in percent of the baseline. The regression of the memory overhead is displayed as a line in the plot.

Case Study Massive project Seeing our result in Emacs, we decide to stress test our approach against a massive project. We investigate the possibility of compiling Chromium with the wllvm compiler wrapper. It should be noted, that Chromium is not one of the packages listed in the continuous tests. From a correctness standpoint, it may not be interesting. But from a scaling perspective, it is interesting, as it is one of the largest code bases in open-source. However, this was not successful. The main issue is unsupported compiler flags. This means we cannot use the compiler wrapper, to generate bitcode. The alternate approach would be to compile with the `-emit-llvm` flag. However, due to the complexity of the project, it is not feasible.

Answer to PQ2: Our tool scales to larger projects. However, we are limited by the ability to generate the LLVM bitcode.

5.3 PQ3: What is the overhead of our approach?

To assess the performance of our tool, we run the same projects as before, with and without generating the bitcode. This allows us to see the overhead of our tool. Figure 5.1 shows these results. We time the run time from within our tool from the spawning of the docker to the deletion. The memory usage is monitored with `docker -status`. We take the maximum observed time as our metric. The compilation with the compiler wrapper results in a time overhead of 34.3%. Unfortunately, we see a trend of increasing overhead toward larger projects. In particular, the second to last sample project "bluez-qt" even exhibits 93% overhead. This can be explained through

the large number of binaries contained in this package. On average, the memory overhead is 5.9%. Again we see the overhead increasing towards larger packages. Here the outlier with 33% overhead is a package called stochas [10]. A library used for statistics. Though it is not confirmed, we suspect that it is because it is not a binary, but a library. We have not observed this same trend in the other libraries we tested, but they were rather small compared to this.

Answer to PQ3: We observe a 34.3% overhead in compilation time to regular compilation. The memory overhead is 5.9%. The trend towards larger builds is an increase in memory and time. However, due to the small sample size, this may be inconclusive.

5.4 General observations

During our testing, we have observed several times, that the output of the differences can be quite unreadable, due to their large size. This is due to the output of llvm-diff, but also the case for any command-line difference tool. The main cause of non-determinism we observed lay in the timestamps of compilations. However, there was a concerning amount of differences, we did not catch. These are mainly in configuration files, log files, documentation, and even the odd python script. Overall, these findings suggest that an LLVM-based approach may be insufficient to detect causes of non-determinism in builds.

Chapter 6

Related Work

Since reproducible builds are quite new, there is little research in its field. Thus we do not limit the related work to works with the same goal, but more general in the field of reproducible builds.

6.1 Reproducible containers

For instance, there is an exploration [5] of reproducing builds using perfectly deterministic containers. This is done by forcing all the system calls to be deterministic. Since the source will be compiled the same if all system calls are the same. Because all possible interactions that might lead to non-determinism can only come from the system[1]. By spoofing the system calls to be deterministic, they reproduce builds across containers. They are however limited, as the underlying system may not be in precisely the same state as the first time. Note, that we had a similar idea initially, however, we dropped it, since the spoofing of system calls requires the cooperation of the OS, and would have been out of scope for this project. Instead, we chose to root cause localization as our focus.

6.2 Root cause localization

In that area, [6] leverages build logs and rule-based filtering to detect non-determinism in builds. Their source code is not public and thus this comparison is based on their paper alone. In contrast to our tool, they can detect non-determinism in any type of file. Based on an existing difference in two builds, they rank the source files from likely to unlikely to contain the issue. However, it is very complex and requires good heuristic filtering. Notice also, that the granularity of the localization is on the file level. Compared to that, we can localize the violating position,

[1] This includes but is not limited to time, urandom, tid, info about the filesystem, locales.

thanks to the debug information. The same authors published a paper [7] that leverages system calls for their analysis. As such they are independent of build tools. However, the process is very complex and is prone to a lot of noise. The authors mitigate this by filtering irrelevant system calls, based on pre-defined rules. However, it is unclear how well this solution scales to larger projects. As far as we can infer from their paper, they have not tested their tool on projects that take more than five minutes to build. Our tool leverages the existing LLVM framework to determine differences in builds. As such it is much simpler in its implementation. However, with this simplicity comes also false negatives. As opposed to [7], we cannot detect all forms of non-determinism. However, similarly to [7], we are required to run in two different environments. In other words, if the non-deterministic behavior is no triggered across the environments, we cannot find it.

Chapter 7

Conclusion

No one has attempted to localize root causes of non-reproducible builds using the LLVM infrastructure. As such the status quo has been advanced. We provide a simple way to detect non-determinism in Linux builds. We can detect non-determinism in binaries. Our solution can be used across different build systems. Though it may not scale for large builds our tool incurs negligible overhead in time and memory for smaller builds. We see the need for more future work in the field of reproducible builds and more general package attestation. We provide a few ideas of our own, based on the findings of this project. For instance, as stated in the evaluation section, we can only work with C/C++ packages. However, many packages use other languages, such as Python or Haskell. In fact, 494 out of 801[1] non-reproducible community packages of the Archlinux continuous tests are Haskell based. Another 70[2] packages are Python-based. For reproducible builds in general, it would be interesting to investigate the reason, why so many Haskell packages are unreproducible. On the other hand, one can continue in the field of LLVM-based attestation. There, it might be possible to do a sort of meta attestation. By comparing the compiled LLVM bitcode with the bitcode obtained by lifting the distributed binary. This omits the necessity, to compile the source multiple times. The robustness of such an approach would have to be investigated. Our research focused on the reproducibility of binaries and libraries. We do not detect any other form of non-determinism such as documentation or configuration files. Though we believe that this is a sufficient form of attestation, one might argue the opposite. To these people we present this excerpt from the definition of reproducible builds: "[...] can recreate bit-by-bit identical copies of all **specified artifacts**" [9]. This leaves some room for interpretation in our favor, as we could detect the non-determinism in all specified files. Namely binaries.

[1] Quick survey of packages with "haskell" in their name. The actual number may be larger. 03.06.2022 *https://tests.reproducible-builds.org/archLinux/state_community_FTBR.html*

[2] Again from the same dataset, with the same heuristic as the Haskell packages.

Bibliography

[1] *Clang: a C language family frontend for LLVM.* URL: https://clang.llvm.org/. (accessed: 08.06.2022).

[2] *GNU Emacs - GNU Project.* URL: https://www.gnu.org/software/emacs/. (accessed: 05.06.2022).

[3] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". *See* https://llvm.org. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[4] *namespaces(7) — Linux manual page.* URL: https://man7.org/linux/man-pages/man7/namespaces.7.html. (accessed: 05.06.2022).

[5] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. "Reproducible Containers". In: New York, NY, USA: Association for Computing Machinery, 2020, pp. 167–182. ISBN: 9781450371025. URL: https://doi.org/10.1145/3373376.3378519.

[6] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. "Automated Localization for Unreproducible Builds". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 71–81. DOI: 10.1145/3180155.3180224.

[7] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. "Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 527–538. DOI: 10.1109/ASE.2019.00056.

[8] *Reproducible Archlinux ?!* URL: https://tests.reproducible-builds.org/archlinux/archlinux.html. (accessed: 28.05.2022).

[9] *Reproducible builds definition.* URL: https://reproducible-builds.org/docs/definition/. (accessed: 08.06.2022).

[10] *Stochas Open-source advanced probabilistic polyrhythmic sequencer plugin.* URL: https://stochas.org/. (accessed: 05.06.2022).

[11] *WLLVM.* URL: https://github.com/travitch/whole-program-llvm. (accessed: 28.05.2022).