An Experience Report on Producing Verifiable Builds for Large-Scale Commercial Systems

Yong Shi, Mingzhi Wen, Filipe R. Cogo, Boyuan Chen, and Zhen Ming Jiang

Abstract—Build verifiability is a safety property for a software system which can be used to check against various security-related issues during the build process. In summary, a verifiable build generates equivalent build artifacts for every build instance, allowing independent auditors to verify that the generated artifacts correspond to their source code. Producing a verifiable build is a very challenging problem, as non-equivalences in the build artifacts can be caused by non-determinsm from the build environment, the build toolchain, or the system implementation. Existing research and practices on build verifiability mainly focus on remediating sources of non-determinism. However, such a process does not work well with large-scale commercial systems (LSCSs) due to their stringent security requirements, complex third party dependencies, and large volumes of code changes. In this paper, we present an experience report on using a unified process and a toolkit to produce verifiable builds for LSCSs. A unified process contrasts with the existing practices in which recommendations to mitigate sources of non-determinism are proposed on a case-by-case basis and are not codified in a comprehensive tool. Our approach supports the following three strategies to systematically mitigate non-equivalences in the build artifacts: remediation, controlling, and interpretation. Case study on three LSCSs within Huawei shows that our approach is able to increase the proportion of verified build artifacts from less than 50 to 100 percent. To cross-validate our approach, we successfully applied our approach to build 2,218 open source packages distributed under CentOS 7.8, increasing the proportion of verified build artifacts from 85 to 99 percent with minimal human intervention. We also provide an overview of our mitigation guideline, which describes the recommended strategies to mitigate various non-equivalences. Finally, we present some discussions and open research problems in this area based on our experience and lessons learned in the past few years of applying our approach within the company. This paper will be useful for practitioners and software engineering researchers who are interested in build verifiability.

Index Terms—Verifiable build, large scale commercial system, build system, security, trustworthiness, software engineering

1 Introduction

Build verifiability is an important safety property for software releases, as independent outsiders can verify if a software release suffers from various security problems introduced during the build process (e.g., surveillance malware [1], compromised cryptographic signatures [2], supply chain attacks [3], and untrusty dependencies [4]). Independent outsiders generally refer to third party auditing agencies or government organizations. If they themselves alone can successfully validate the correspondence between source code and the generated build artifacts, the provided software release is considered as a verified build [5]. Many open source software (OSS) (e.g., BitCoin [6], Chromium [7], and Debian [8]), commercial (e.g., Facebook [9], Google [10], Huawei [5], Pinterest [11] and Telegram [12]), and governmental organizations [13], [14] are actively investigating or have already supported build verifiability, as these organizations need to demonstrate that the build artifacts they have sold to the customers or distributed openly in the wild correspond to the exact source code that they have developed. Although there are various available tools to ensure the consistency of the build environment [15], [16] and the build toolchain [9], [17], extra efforts are still required to produce verifiable builds. For example, even using a virtualized environment, there are still various non-equivalent build artifacts generated during the build processes for software systems (e.g., Debian [8] and Tor [18]).

There are two general processes proposed in the existing literature to produce verifiable builds. The first process, named *deterministic build process* [16], [19], [20], mainly focuses on the elimination of any non-deterministic build instruction. The second process, named *explainable build process* [21], mainly focuses on interpreting the non-equivalences in the build artifacts that cannot or should not be mitigated. Given the same set of source code files, the same build scripts, and the same build environment, the deterministic build process can repeatedly generate equivalent artifacts (i.e., artifacts with the exact same contents) [22]. Source code files or the build scripts are modified in order to remediate sources of non-determinism (e.g., timestamps [23], build path [24], and file ordering [25]) that cause non-equivalences in the generated build artifacts across different build processes. In turn, the explainable build process aims at providing a technical interpretation for the non-equivalences in the build artifacts. Due to security requirements or system design, certain non-equivalences in the build artifacts cannot or should not be eliminated.

- *Yong Shi and Mingzhi Wen are with the Huawei Technologies, Shenzhen 518129, China. E-mail: {young.shi, wenmingzhi}@huawei.com.*
- *Filipe R. Cogo and Boyuan Chen are with the Centre for Software Excellence, Huawei Technologies, Kingston, ON K7K 3T1, Canada. E-mail: filipe.cogo@gmail.com, boyuan.chen1@huawei.com.*
- *Zhen Ming Jiang is with the Department of Electrical Engineering & Computer Science, York University, Toronto, ON M3J 1P3, Canada. E-mail: zmjiang@cse.yorku.ca.*

Manuscript received 19 Jan. 2021; revised 13 June 2021; accepted 14 June 2021. Date of publication 25 June 2021; date of current version 19 Sept. 2022.
(Corresponding author: Filipe R. Cogo.)
Recommended for acceptance by S. Apel.
Digital Object Identifier no. 10.1109/TSE.2021.3092692

The explainable build process needs to document the root cause of the non-equivalences as well as the rationale for not eliminating them. We also highlight that, although *build verifiability* relates to *build reproducibility* [22], these are two different concepts. Build reproducibility focuses on making the generated build artifacts by two different build instances equivalent (i.e., with the exact same contents), whereas build verifiability focuses on the best effort to maximize the equivalences between the build artifacts and to interpret all non-equivalences. While build reproducibility implies build verifiability, the inverse is not necessarily true, as a verifiable build can generate an artifact containing an interpreted non-equivalence.

Build verifiability is an important property of large-scale commercial systems (LSCSs) that are deployed in critical infrastructures such as regulated communication networks. Furthermore, to instill customer confidence in the product security, it is required that these LSCSs can produce verifiable builds in a consistent and systematic manner [5]. Hence, it is vital for the companies to have appropriated processes and automated tools to detect, mitigate, and verify the non-equivalences in LSCCs builds. However, existing processes to produce deterministic builds cannot match the needs for LSCSs due to the following three challenges:

1) *Security*: Some build artifacts will not be equivalent due to the additional security mechanisms (e.g., digital signatures [26]) required for LSCSs.
2) *Third party dependencies*: LSCSs adopt a set of external commercial or open source third party packages. Ensuring equivalences in the build artifacts for LSCSs requires addressing the sources of non-determinism for LSCSs as well as their dependency packages.
3) *Scalability*: Localizing sources of non-determinism and non-deterministic build instructions is a challenging and time consuming task [19], [20], [21]. Effective techniques to address recurrent sources of non-determinism are crucial for LSCSs, which are constantly changed every day.

In this paper, we propose an approach to produce verifiable builds for LSCSs. Despite our experience report drawing on existing research results, our approach to produce verifiable builds was designed over the years of practical Research & Development and is of great value to practitioners. Our approach consists of two parts: a unified process, called VBP (Verified Build Process), and a toolset, called ToolKitA.[1] The VBP leverages prior knowledge to detect, diagnose and mitigate the non-equivalences in the build artifacts. Our VBP unifies different practices for build verifiability that are typically presented on a case-by-case basis. For example, Reproducible Builds [27] proposes an extensive set of practices for verifiable build, however these practices are not attached to a coherent and repeatable SE process. In addition to the two aforementioned strategies (the remediation strategy in the deterministic build process and the interpretation strategy in the explainable build process), our VBP also supports the controlling strategy, in which we dynamically intercept calls to non-deterministic build instructions and return a deterministic value. Regular software engineers (RSEs) and build specialists (BSPs) carry out the VBP by leveraging the ToolKitA toolset, which contains a set of tools for build profiling, extraction of build dependencies, and analysis and mitigation of non-equivalences in the build artifacts.

The VBP and ToolKitA have been applied to three mission critical LSCSs from Huawei, which are used by tens of millions of people every day. Case study results show that by leveraging our process and toolset, all of the build artifacts can be independently verified. This result represents a significant improvement compared to more than 50 percent of the unverified build artifacts previously. To further evaluate the generalizability of the VBP and ToolKitA, we have also applied our approach to verify the build process of 2,218 open source packages distributed with CentOS on version 7.8. We demonstrate the satisfatory performance of our approach by mitigating 100 percent of the non-equivalences in our LSCSs and 99.94 percent of the non-equivalences in CentOS 7.8. In summary, our paper makes the following contributions:

- We are the first to discuss the challenges and solutions to produce verifiable build for LSCSs, which have stringent security requirements, complex third party dependency relations, and many code changes. Case study results also show that our approach can work with non-Huawei systems (e.g., CentOS) with very little human intervention. We are also the first to report results on verifiably building a complete CentOS release.
- We provide a general guideline on the application of the three mitigation strategies (remediation, controlling, and interpretation) to both recurrent and new non-equivalences caused by different sources of non-determinism.
- We describe the lessons learned during several years of developing and deploying our approach within Huawei. We also present future research opportunities and open problems in this research area, which can be of interest to practitioners and software engineering researchers.

Paper Organization. The reminder of this paper is organized as follows: Section 2 presents some background information on the problem context. Section 3 describes our approach to producing a verifiable build for LSCSs using a running example. Section 4 presents an empirical assessment on applying our approach on three LSCSs from Huawei and one open source system. Section 5 discusses the lessons learned and describes some future research topics. Section 6 describes related works and Section 7 presents the threats to validity. Section 8 concludes our paper.

2 Background and Overview

In this section, we describe the problem of producing a verifiable build for LSCSs in Huawei. These LSCSs are mission critical systems used by tens of millions of people every day. Such systems have to go through rigorous auditing process by independent organizations to ensure they meet various requirements. One of the important aspect to check during the auditing process is build verifiability. First, Huawei provides auditing agencies with a system image

1. ToolKitA is anonymized due to confidentiality reasons.

(e.g., a virtual machine) with the complete set of source code (both the system implementation and the open source third party dependencies), the build toolchain (e.g., linkers and compilers), and a detailed documentation describing the configuration of the build as well as how the non-equivalences were mitigated from the build artifacts. Then these organizations perform the build process and compare the resulting build artifacts against the release-ready artifacts. This auditing aspect passes if all the build artifacts are equivalent or the non-equivalences are clearly explained and checked. It is challenging to ensure a verifiable build for the LSCSs within Huawei:

Challenge 1: Security. Existing research (e.g., [16], [19], [20], [21]) and practices (e.g., [6], [7], [27]) mainly focus on the deterministic build process, which remediates all the sources of non-determinism during the build process. However, in addition to build verifiability, there are many other security measures implemented within LSCSs to protect them against a range of different vulnerabilities. For example, the following two mechanisms are considered as common security protection mechanisms, both of which introduce non-determinism during the build process: (1) digital signatures [26], which provide a secure approach to verifying the authenticity of the LSCSs; and (2) ASLR (Address Space Layout Randomization) [28], [29], [30], [31], which is a security technique to randomize memory addresses to fend off memory safety-related vulnerabilities. This technique can become a source of non-determinism if the contents that are written into the binary artifacts are impacted by memory addresses. For example, we once found that in Berkeley DB [32], the value of a particular variable [33] is dependent on the memory address that stores the process ID of the current running process and causes non-equivalences when ASLR is enabled. Such non-equivalences should not be remediated, but explained and verified during the auditing process.

Challenge 2: Third Party Packages. LSCSs usually have complex third party dependencies, which can be from open source communities or other companies. Such complex dependency relations introduce the following two sub-challenges:

- *Addressing non-determinism in third party packages*: To ensure build verifiability, the whole system needs to be built from scratch. This means the complete set of source code, which includes the code of the LSCSs and the code of the third party dependencies. The third party dependencies include the third party packages that LSCSs directly depend on as well as all the additional packages that these third party packages depend on. Ensuring build verifiability requires addressing the source of non-determinism in LSCSs as well as the non-determinsm introduced in the third party dependencies.
- *Change management*: The communities or organizations behind these third party dependencies may not accept changes related to verifiable builds. For example, the GhostScript community decides not to support a deterministic build [34]. Other organizations like Apple suggest users to download from their official channels [35] and provide security checking mechanisms to ensure the downloaded artifacts match the officially released versions [36]. Therefore, local repository forks need to be introduced to track the code changes to produce a verifiable build for these third party dependencies. Such code changes need to be documented for maintenance and auditing purposes. In addition, extra effort is also needed to ensure that the local forks are constantly synchronized with the changes from the upstream repositories of these dependencies [37].

Challenge 3: Scalability. Typically, non-equivalences are mitigated in a case-by-case fashion, as they can be caused by many different sources of non-determinism (e.g., the system implementation, the build toolchain, or the build environment) [19], [20], [21]. LSCSs are maintained by thousands of developers and with hundreds of code changes every day. Due to different deployment environment and customization requirements, multiple builds can be produced for the same set of code changes. Manual analysis is extremely time consuming due to the problem complexity and the scale of the non-equivalences that need to be analyzed. Automated approaches are required to integrate the prior knowledge on diagnosing the non-equivalences and to mitigate them in large scale.

To address the above challenges, during the past few years we have developed a novel approach to produce verifiable builds for LSCSs. Our approach consists of two parts: (1) the VBP, which leverages prior knowledge on diagnosing non-equivalences in the build artifacts, addresses sources of non-determinism through the follow three strategies: remediation, controlling, and interpretation; and (2) the ToolKitA toolkit, which provides the automation tool support. Before our approach was proposed and deployed, the approach to verify build artifacts in the product team was completely manual. RSEs and BSPs manually diagnosed the non-deterministic behavior of the build and directly modified the source code files or the build files to remediate the non-equivalences. They could only address the non-equialences introduced by timestamps by using this approach. Other types of non-equivalences cannot be solved by using remediation only, e.g., the non-equivalences introduced by archiving and packaging cannot be mitigated by modifying source code files. In the next section, we will describe our approach in details.

3 Our Approach

Prior works on build verifiability are done on open-source projects and address each non-equivalence one-by-one. As explained in Section 2, LSCSs from Huawei are more complex due to security requirements, code complexity, and scale. Hence, a new approach, which is of minimal impact on regular software development activities, is needed. Our approach to achieve a verifiable build combines characteristics of the deterministic and explainable build processes. Still, our approach presents significant differences from the existing practices. First, our approach describes and evaluates the usage of a controlling technique (namely intercept & ignore lists) to effectively mitigate a variety of sources of non-determinism and support build verifiability. We show that the majority of the common sources of non-determinism can be mitigated with the usage of intercept & ignore

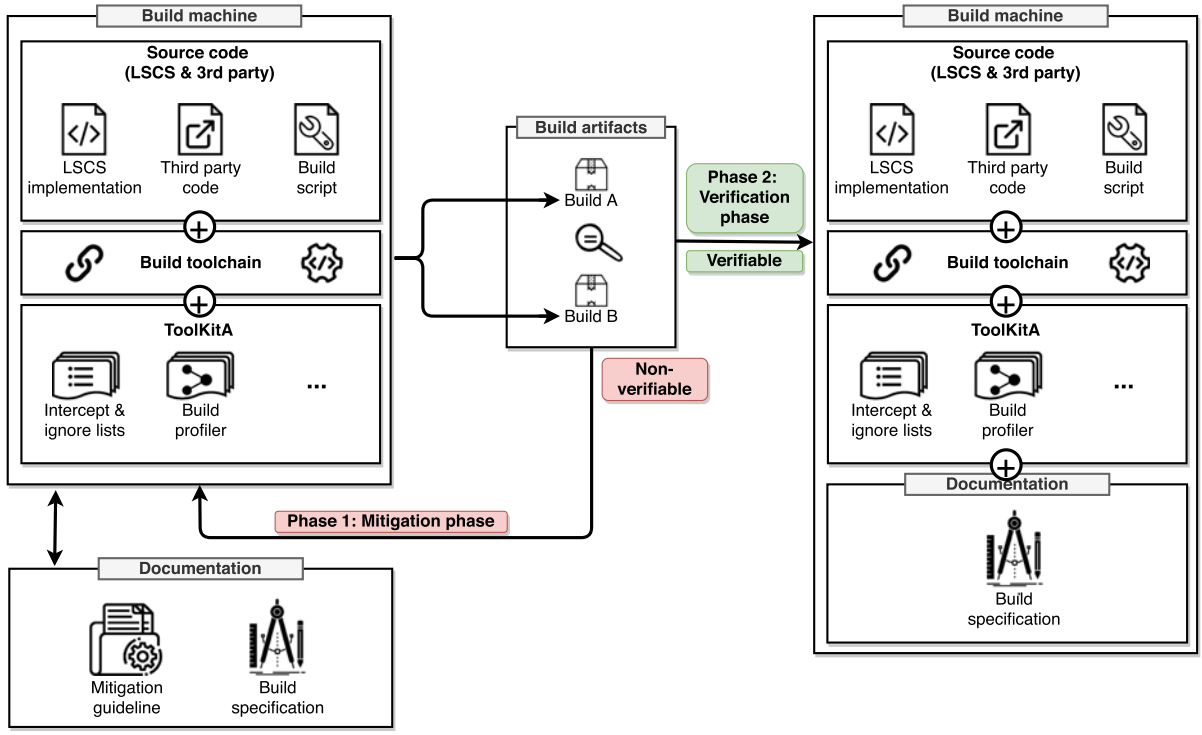


Fig. 1. The flow of our VBP for LSCSs. Boxes represent input and output information, whereas arrows represent information flow.

lists. Second, our approach integrates both process and tooling to achieve verifiable builds in LSCSs. This characteristic constrasts with existing practices, which are recommended in isolation (i.e., on a case-by-case basis) without attention to the undelying SE process that integrates these practices. Lastly, our approach conveys important recommendations for documenting and explaining sources of non-determinism that are not remediated nor eliminated from the build process. Practitioners can produce verifiable builds by applying the lessons learned from our experience report in their own context. Based on our investigation of non-verifiable builds in the past few years, we observed that many non-equivalences can be automatically mitigated by controlling the sources of non-determinism (e.g., setting initial seed values for random number generations) during the build process. Such observation is the basis for developing our approach to produce verifiable builds for LSCSs.

Fig. 1 illustrates the flow of our VBP, which consists of two phases: (1) *the mitigation phase*, which is carried out during the product development and testing stages (Section 3.1), and (2) *the verification phase*, which is carried out during the auditing stage (Section 3.2). The mitigation phase is handled by RSEs and BSPs. RSEs iteratively invoke the build process using the ToolKitA until all non-equivalences are verified. During each iteration, non-equivalences are reported to BSPs who investigate them carefully. The resulting root causes are documented in the mitigation guidelines. If there are new controllable sources of non-determinism, BSPs will also modify the configurations of ToolKitA accordingly to accommodate the new additional changes. The learning-curve for RSE is small, as they just need to learn the documentation to run ToolKitA. The effort for BSP is also small and decreases over time, as most of the non-equivalences are already diagnosed, mitigated, and documented, and very few new ones are generated. Finally, the verification phase is performed by a collaboration between the software vendor and the auditing agency.

Running Example. The build for this running example consists of three steps as shown in the build script of Fig. 2a: (1) Recording the current date and time in a text file (`time.now`); (2) Compiling `time.c` into an executable file (`time.bin`) that outputs the compiling time by invoking the `__TIME__` macro,[2] and (3) Packaging both files (`time.now` and `time.bin`) into an archive file (`time.tgz`) and writing a timestamp in the header of this file (this latter step being implicitly performed by the tar tool). The source code contents written in C for `time.c` are shown in Fig. 2b. In the following, we will explain how to use our VBP and leverage our toolset ToolKitA to produce a verifiable build for this running example.

3.1 The Mitigation Phase

The mitigation phase is broken down into six steps (see Fig. 3), which iterates until the build is deemed as verifiable. Steps 1, 2, and 6 of the mitigation phase are executed by RSEs with fully automation support from ToolKitA. Steps 3, 4, and 5 of the mitigation phase are executed by BSPs, which require manual efforts. The first step checks whether the build is verifiable. The second step employs the *build profiler* to trace the entire build process. The third step leverages the build profile, as well as the previous knowledge that is documented in the *mitigation guideline*, to diagnose the non-equivalence (i.e., to identify an isolated source of non-determinism that is accountable for the non-equivalence). The

2. https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html

```
date > time.now
gcc time.c -o time.bin
tar -zcf time.tgz time.bin time.now
```

(a) `build.sh`

```
#include <stdio.h>
int main(){
    printf("Compiling time: %s", __TIME__);
}
```

(b) `time.c`

Fig. 2. a) Build script and b) source code of our running example.

fourth step updates the mitigation guideline such that the knowledge generated during the mitigation of a non-equivalence can be reused in later iterations. The documentation of the mitigation also serves the purpose of explaining how the non-equivalences were mitigated in the verification phase. The fifth step adds specific build instructions to the *intercept & ignore lists* so that an isolated non-equivalence can be mitigated. The mitigation guideline is updated whenever any new knowledge regarding the mitigation of a non-equivalence is generated in the fifth step. The sixth step executes the build process again to validate if the non-equivalence was mitigated. At the end of this iterative process, any non-equivalences that are not eliminated (a.k.a., being controlled or interpreted) are documented for maintenance and auditing purposes.

Step 1 – Checking Verifiability. The build environment for our LSCSs is provided as a virtual machine image with the proper version of the operating system and build toolchain. The build environment is loaded with the up-to-date complete set of source code files, which includes the implementation of the LSCS, the code for their third party dependencies, and the build scripts (see Fig. 1). The build process is executed twice and the generated build artifacts are compared against each other. To unpack and compare the contents of the generated build artifacts, we use standard tools (such as diffoscope [38] in Linux). Since our LSCSs can be built in different platforms, we also integrate such standard tools for different platforms in our ToolKitA. The build is verifiable when the two sets of build artifacts have the same contents or when the differences (i.e., non-equivalences) can be interpreted. A non-equivalent artifact is successfully interpreted if it can be produced by an independent auditor, who also agrees with the technical explanation behind the differences. Otherwise, the build is considered as non-verifiable and will be profiled and analyzed in subsequent steps. In our running example, since `time.bin`, and `time.now` both include the up-to-date timestamp information, they will have different contents every time the build process is executed in the same build environment. Therefore, this build is considered as non-verifiable.

Step 2 – Profiling Build. In this step, we profile the build process to record the necessary information for further analysis. Our technique is similar to the techniques as proposed in prior studies [20], [39], [40] to intercept function calls and to construct a build call graph. Other than applying common build profiling tools such as strace for Linux, we construct the build call graph by leveraging the API hook mechanism of the operating system (e.g., `LD_PRELOAD` in Linux). A build call graph consists of build instructions as edges and files and processes as vertices. The connection between the nodes can be recovered by analyzing the instructions (e.g., open, read, write) and their arguments (e.g., files and processes) recorded from traces of the function calls. In the Linux environment, we leverage the API hook mechanism by setting `LD_PRELOAD` to our self-implemented dynamic library. Our dynamic library then intercept the `__libc_start_main` function. This function is invoked whenever a build instruction (e.g., gcc, clang, ar, and ld) is called, allowing us to record the necessary information for constructing the build call graph (i.e., the name of the build instruction, the input files, the output files, and the SHA256 values of these files) in a database. These build instruction are commonly used in the compilation process of our LSCSs and our self-implemented dynamic library allows us to specify such build instructions. After recording such information, we construct the build call graph by connecting the build instruction names in the order they were called. By analyzing the build call graph, we can pinpoint non-deterministic instructions and trace their origin to higher or lower level instructions. The build call graph is also used during the verification phase to ensure that the correct external dependencies are loaded during the build process. Fig. 4 shows an excerpt of the resulting build call graph of our running example.

Step 3 – Diagnosing Non-Equivalence. In this step, for each non-equivalent build artifact, we further analyze and resolve the non-equivalences one at a time. We analyze the build call graph to identify a call to the lower-level instructions that are not explicitly stated in the build scripts or in the source code. For example, in our Fig. 5, the build starts from the source code file `Foo.c`. After the build profiling process, we know that the build instruction `A` (e.g., `gcc`) generates the intermediate file `Foo.o`. By comparing the recorded SHA256 of the files, we know that `Foo.c` is equivalent to the same file produced by the previous build, while `Foo.o` is non-equivalent. In this case, we can identify that the process associated with gcc introuced the non-

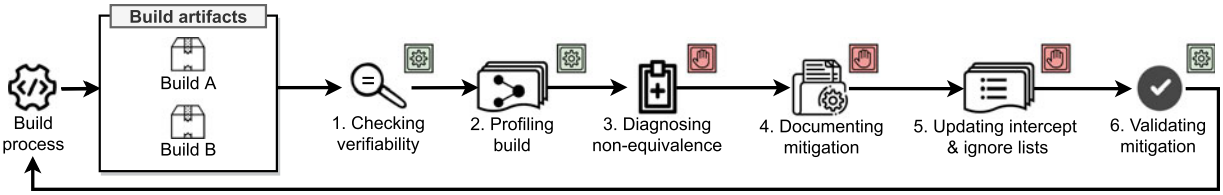


Fig. 3. The mitigation phase of our VBP. Steps 1, 2, and 6 are fully automated, whereas steps 3, 4, 5 require human intervention.

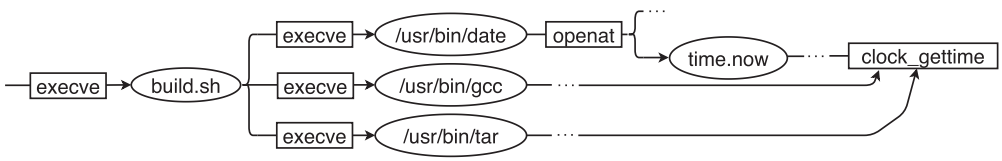


Fig. 4. The build call graph of our running example.

equivalence in the generated build artifacts. During the diagnosing process, standard tracing tools (e.g., strace in Linux) are also enabled so that we can manually identify the low-level instructions that introduce the non-equivalences. Back to our running example, we first focus on the file `time.now`. By analyzing the build call graph generated by Step 2, we know that `time.now` has non-equivalences and depends on the process `date`. By analyzing the non-equivalences (which is a date and time information), we can deduce that it is caused by the invocation of the `date` instruction as shown in Fig. 2a. We can also verify that `date` invokes the `clock_gettime` function, as shown in Fig. 4. Since the time is constantly changing, the results of invoking `clock_gettime` will be different at each invocation.

Step 4 – Documenting Mitigation. We have documented the various sources of non-determinism and their mitigation strategies in a document called the *mitigation guideline*. The information in the mitigation guideline is either from existing research and practice (e.g., [19], [20], [21], [27]) or added as a result of the investigation of prior non-equivalences in the build artifacts of our LSCSs. There are three general mitigation strategies:

- *Remediation* aims to remove the non-equivalences by modifying the LSCS implementation, the third party source code, or the build script.
- *Controlling* aims to control the non-equivalences by dynamically intercepting specific build instructions and returning pre-defined values. To control the non-equivalences, we have developed an *intercept & ignore list* mechanism (see ToolKitA in Fig. 1). We record the set of build instructions that need to be controlled in the *intercept lists*. During the build process, whenever an instruction that is in an intercept list is invoked, it will be intercepted and pre-defined values will be returned. However, returning pre-defined values in some specific runtime context may have a systemic and potentially undesirable effect (e.g., a build failure). Hence, the *ignore list* maintains a set of build instructions that are bypassed by the interception mechanism. This interception mechanism works at the kernel level and therefore is agnostic to build artifacts or build tools, as long as the build process uses the exact same instructions as annotated in the lists. This characteristic is essential to hangle the heterogeneity of the build environment of LSCSs.

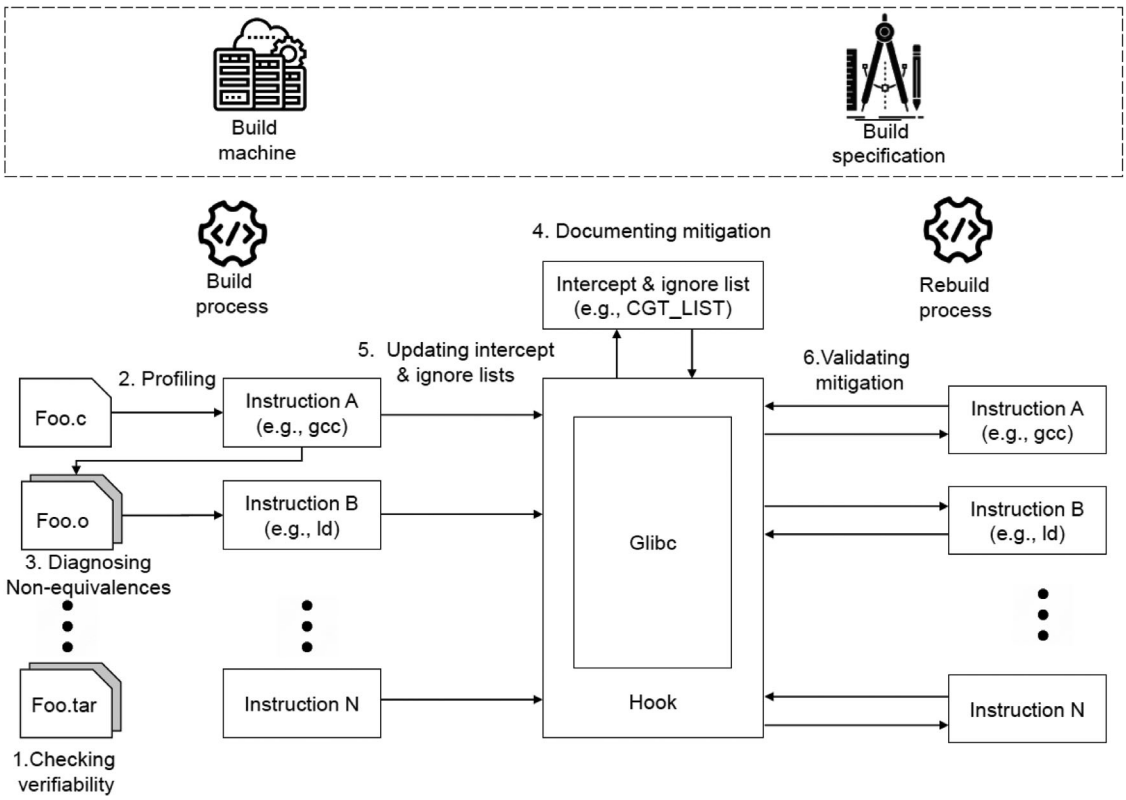


Fig. 5. Our interception mechanism and its relation with each step of the mitigation phase.

- *Interpretation* is the strategy for dealing with non-equivalences due to security requirements (e.g., digital signatures) or system design (e.g., ASLR), which cannot or should not be eliminated nor controlled. Instead, we need to explain the sources of non-determinism by documenting the location of the non-equivalences, their root causes, as well as the rationale behind not eliminating or controlling them. Such documentation needs to be detailed and accurate enough so that other practitioners and auditing agencies can verify its correctness.

In our running example, by analyzing the build call graph we can deduce that the non-equivalences in `time.now` have their origin in the invocation of the time function `clock_gettime`, which is one of the documented sources of non-determinism. The adopted mitigation strategy is to control the source of non-determinism. Assuming that the mitigation guideline is empty, we update this document with the following entry:

> *Source of non-determinism:* Date and time.
> *Description*: The date and/or time at which the build is performed is stamped in the generated build artifacts.
> *Related build instructions:* `date`
> *Recommended mitigation strategy:*
> *Controlling*: Add related build instruction to `CGT_LIST` to intercept build instructions that calls `clock_gettime`. Set pre-defined date and time to the `TIME_VALUE` variable.

It mentions that a source of non-determinism is related to the date and time instructions that are invoked during the build process and can be generally traced to the lower level `clock_gettime` function.

Step 5 – Updating Intercept & Ignore Lists. We maintain one intercept list per low level instruction that is going to be intercepted. Considering our running example, the recommended mitigation strategy by the mitigation guideline is to add the high level `date` instruction to the intercept list called `CGT_LIST` (i.e., the intercept list associated with the low level instruction `clock_gettime`). Whenever an instruction that is in the `CGT_LIST` is intercepted, it will receive a pre-defined date and time as the returning value to a call to `clock_gettime`. The snippet below shows the configuration of the interception mechanism (`libtoolkita.so`) as well as the contents of the `CGT_LIST`:

```
export LD_PRELOAD=libtoolkita.so
  export TIME_VALUE=2020-07-11,08:30:18
  export CGT_LIST=date
```

In this example, depending on the build environment, we leverage different mechanisms to intercept system calls. For Linux, we leverage the API hook mechanism (a.k.a., `LD_PRELOAD`) to pre-load our toolset ToolKitA [41] before the build process starts. During the build process, the specified system calls are intercepted and replaced by a call to a customized function (which, in turn, returns the pre-defined value). For the Windows platform, we leverage the Detours [42] tool to monitor and instrument system calls. Specified system calls in the *intercept list* are intercepted and replaced by a call to a customized function. In the example above, whenever a call to `clock_gettime` is intercepted, a customized function that returns the value stored in `TIME_VALUE` will be called instead. Currently only one instruction (`date`) is added to `CGT_LIST`, but other time- and date-related instructions that calls `clock_gettime` can be added as well. The `clock_gettime` function is also invoked by many other instructions like `gcc` and `tar`. Since we also want to control the non-determinism in the `time.bin` and `time.tgz` files, we add `gcc` and `tar` to `CGT_LIST`. At this point, the mitigation guideline is updated with two additional related build instructions to the date and time entry: `gcc` and `tar`. In some specific circumstances, by adding a build instruction to an intercept list, the build can start to fail. For example, some build instruction can use the `date` instruction to generate intermediate build files with different names (i.e., by appending the current date and time to the intermediate file names). If the `date` instruction always return the same pre-defined date and time value, the intermediate files are overwritten with each other and the build fails. In this case, the higher-level instruction that uses `date` to generate the intermediate files needs to be added to the ignore list. In Fig. 5, we demonstrate that the hook functions communicate with our intercept & ignore lists, both by intercepting the instructions in the list as well as by returning the associated predefined value.

The update of the intercept & ignore lists is the most effective step of our VBP to remediate non-equivalences in build artifacts. It configures the process for controlling the sources of non-determinism during the build process and, according to our experience (see Section 5.2), is responsible for remediating most of the non-equivalences in the build artifacts.

Step 6 – Validating Mitigation. The build process is repeated again to check if the non-equivalence investigated in the prior steps are mitigated. As shown in Fig. 5, this repeated build process (i.e., the rebuild process) will invoke the same set of build instructions. These build instructions will get the pre-defined values returned from the hook functions on top of the system kernel. The hook functions are configured by the intercept & ignore lists discussed in Step 5. If there are still some non-equivalences left at the end of this build process, we will reiterate the whole process from Step 1. This iterative process continues until all the non-equivalences are mitigated. During this process, any non-equivalences that have been mitigated will be clearly documented for maintenance and auditing purposes.

At the end of the mitigation phase, we record in the *build specification* all information that is necessary to verify the build in the verification phase.

3.2 The Verification Phase

Once all the non-equivalences are mitigated, the build is considered as verifiable and can be audited by an independent organization. The same build environment as used in the mitigation phase, along with the source code, the intercept & ignore lists, as well as the build specification are all provided to the auditing agency. The build specification contains the following information regarding the build:

Fig. 6. Interpreting the non-equivalences in the build artifacts. Non-equivalent regions of the build artifacts are highlighted in grey.

- The configuration of the build environment (e.g., an specification of the virtual machine and operating system, as well as the version of the build toolchain).
- A detailed rationale for each instruction that is contained in the *intercept & ignore list*. Please refer to Section 3.1 for an example of mitigating the non-equivalences caused by date and time.
- A detailed rationale for each non-equivalence that is interpreted (i.e., neither remediated nor controlled). In addition, for each non-equivalence in the build artifacts, the location of the non-equivalence is recorded and mapped to the rationale described in the *non-equivalence inventory*. Fig. 6 shows one example.

The auditing agency executes the build process and checks the resulting build artifacts against the provided (release ready) artifacts. For all the controlled non-equivalences, they can edit or remove the contents in the intercept & ignore lists to examine and verify their impact on the build artifacts. For the interpreted non-equivalences, they carefully examine the documentation and the build artifacts to see if they are legitimate. The build call graph can be used during this examination to ensure that the loaded external dependencies are also legitimate. To pass the auditing process, all the non-equivalences need to be mitigated and independently verified.

4 Case Study

In this section, we present an in-field evaluation of our approach to produce verifiable builds. In Section 4.1, we discuss our case study setup and explain the studied LSCSs. In Section 4.2, we compare the proportion of equivalent build artifacts that are generated before and after applying our approach for three LSCSs within Huawei and for version 7.8 of CentOS. In Section 4.3, we provide an overview of our mitigation strategies for various sources of non-determinism described in the *mitigation guideline* (see Section 3). This guideline records our experience on successfully mitigating many non-equivalences for these three LSCSs in the past few years.

4.1 Case Study Setup

We report our experience on applying our approach to three representative LSCSs within Huawei. All three studied LSCSs are mission critical systems used by tens of millions of people every day. They have been widely deployed in many of the products within Huawei. As these LSCSs are deployed in the access-point of the wireless networks, they are considered to be the main focus of security related auditing by the third party auditing agencies. For this reason, these three LSCSs have been submitted to different security audits and their ability to produce verifiable build artifacts was validated by an independent auditing agency [5]. Each of these systems consist of millions or tens of millions lines of code and have complex third party dependency relations.

To evaluate the generalizability of our approach, we have also applied our approach to the open source packages distributed under CentOS version 7.8, which is a popular Linux distribution. CentOS is widely used inside Huawei and the software packages contained in the distribution are used by software systems inside Huawei. Hence, conducting a study on CentOS will improve the product quality and also contribute to the open source community. For example, Huawei has created an open source project called OpenEuler, which is forked from CentOS, that could benefit from this study. In addition, existing works towards deterministic builds are found only in other Linux distributions such as Debian [8]. Our work complements those works and provides practical contributions to the CentOS community.

4.2 Performance Evaluation

In this section, we report the methodological details (Section 4.2.1) and report results of our performance evaluation on three LSCSs within Huawei (Section 4.2.2) and a large-scale open source system, CentOS (Section 4.2.3).

4.2.1 Evaluation Method and Evaluation Metrics

For the studied systems (including the three LSCSs in Huawei and CentOS), we compute two metrics between two identical build processes: the number (percentage) of equivalent build artifacts and the number (percentage) of non-equivalent build artifacts that are interpreted. The number (percentage) of equivalent build artifacts represents how many build artifacts are equivalent before and after we remediate or control the non-equivalences. The number (percentage) of non-equivalent build artifacts that are interpreted denotes the number of build artifacts that cannot be remediated or controlled, but can only be interpreted after investigation.

We first conduct the same build process twice before applying our approach and collecting the build artifacts. Then we compare the percentage of build artifacts that can be repeatedly produced (a.k.a., equivalent build artifacts). For example, one system generates 100 build artifacts during the build process. If the same build process is repeated again and only 20 of the build artifacts are equivalent to the ones generated in the previous build, then the percentage of equivalent build artifacts is $20/100 * 100\% = 20\%$.

After applying our unified process to mitigate sources of non-determinism, we conduct the same build process twice and collect the generated build artifacts. By applying our

TABLE 1
The Percentage of Equivalent Build Artifacts Before and After Applying Our Approach for LSCSs

System	%(#) of equivalent artifacts		# interpreted artifacts
	Before	After	
$LSCS_1$ (A)	32%(9)	100.00%(28)	0
$LSCS_1$ (B)	0%(0)	100.00%(37)	0
$LSCS_2$	83%(1,877)	99.70%(2,253)	6
$LSCS_3$	88%(64,029)	99.99%(72,784)	1
CentOS	85.87%(81,483)	94.56%(89,724)	5,021

$LSCS_1$ is built in platforms A and B.

unified process, we build the system using our ToolKitA with the build specification. We then compare the percentage of build artifacts which are equivalent. For example, after applying our approach, there are 99 build artifacts that are equivalent compared to 20 before applying our approach. Then we consider the percentage of equivalent build artifacts is 99/100 * 100% = 99%. The remaining one build artifact is mitigated by interpretation. In other words, we achieve the verifiable build by generating 99 percent equivalent build artifacts and 1 percent interpreted build artifacts.

4.2.2 Evaluation Results for the LSCSs

Table 1 shows our results on the three LSCSs. The first LSCS, $LSCS_1$, is supported on two different platforms and hence produces two different sets of build artifacts. All three LSCSs under study are required to be built from the complete set of source code. Before adopting our approach, one of three LSCSs have less than 50 percent of equivalent build artifacts. After adopting our approach, all the build artifacts in $LSCS_1$ (`PlatformA`) and $LSCS_1$ (`PlatformB`) are equivalent. Also, 99.70 percent of the build artifacts in $LSCS_2$ are equivalent and 99.99 percent of the build artifacts in $LSCS_3$ are equivalent. The remaining 0.3 percent of the non-equivalent build artifacts (6 build artifacts) in $LSCS_3$ and the remaining 0.001 percent of the non-equivalent build artifacts (1 build artifact) in $LSCS_3$ are mitigated by interpretation (see Section 3.1). In other words, the builds of all three LSCSs are verifiable.

The released versions of these three systems (one system supported in two platforms) along with another four systems have been audited by the independent auditing agencies between 2019 and 2020. Following the verification phase of our VBP, the auditors are able to verify the build of the LSCSs by leveraging our toolset ToolKitA and the provided documentation (i.e., build specification). They have compared and inspected their produced build artifacts against the released versions and confirmed the builds are verifiable: "*HCSEC* (the independent auditing agency) *has now verified binary equivalence across eight product builds*" (page 20 of [5]).

4.2.3 Evaluation Results for CentOS

To evaluate the generalizability of our approach, we have also applied our approach to the open source packages distributed under CentOS version 7.8, which is a popular Linux distribution. We have followed the VBP with the same *intercept & ignore list* configured for our LSCSs. We successfully compiled 2,218 of these open source packages, which contains 889,430 source code files and 245 million lines of code. Similarly to Section 4.2.2, we evaluate our approach by comparing the percentage of build artifacts that can be repeatedly generated before and after applying our approach. The results are shown at the last row of Table 1.

In total, we generate 94,888 build artifacts from the CentOS 7.8 distribution. Before adopting our approach, 85.87 percent (81,483) of the build artifacts are equivalents. After directly adopting our approach using the existing intercept & ignore list, 94.56 percent (89,724) of the build artifacts are equivalent. Among the remaining 5.44 percent of the non-equivalent build artifacts (5,164 build artifacts), 5.29 percent (5,021) can be mitigated by interpretation. Among the remaining 143 $(5164 - 5021 = 143)$ build artifacts, 0.096 percent (91 build artifacts) of the build artifacts can be further mitigated by controlling. When investigating those non-equivalences, we observed that they are associated with files generated by source code written in Python (i.e., files with the `.pyc` and `.pyo` extensions) in specific versions. Therefore, we updated the *intercept & ignore list* configured from our LSCSs with the `python2` and `python3.6` instructions, as our LSCSs are mainly implemented in C and C++. After adopting our approach, we observed that 99.94 percent (94,836) of the build artifacts are deemed as verified. We are still investigating the root causes of the remaining 0.06 percent (52) non-equivalent build artifacts. The results clearly show that our approach on the LCSCs can be applicable to open source software packages with satisfactory performance.

4.3 An Overview of Our Mitigation Guideline

The non-equivalences in the build artifacts are caused by different sources of non-determinism. Based on our experience on diagnosing many non-equivalent build artifacts in the past few years, we categorize the sources of non-determinism into the following three categories:

- *Environment* refers to the non-equivalences caused by the interactions between the build process and the build environment, either by means of function calls or global environment variables.
- *Build toolchain* refers to the non-equivalences caused by the non-deterministic behavior of various tools used during the build process (e.g., linkers, compilers, and archivers).
- *System* refers to the non-equivalences caused by the implementation of the system under development. It includes the implementation of the system itself, the source code of the external third party dependencies, and the build scripts.

In the remaining of this section, we describe the details of our mitigation guideline based on the above three categories of sources of non-determinism.

Environment. The recommended strategy to mitigate the non-equivalences caused by the environment is controlling. There are two main mechanisms for controlling the interaction between the build process and the environment: (1) utilizing *intercept & ignore list* for intercepting build instructions that invoke certain functions; and (2) providing pre-defined values for the referenced global environment variables.

We explained a running example in Section 3 on how we use the first mechanism to address the most common source

of non-determinism: date and time. To further illustrate the details, we describe two function calls that interacts with date and time: `time` and `__xstat`. `time` returns the number of seconds since the Epoch. `__xstat` returns the last modification time of a file. Many build instructions may invoke these two functions. If a non-deterministic build instruction invokes `time`, then we add this build instruction to `INTERCEPT_LIST_TIME`. Furthermore, if a non-deterministic build instruction invokes `__xstat`, then we add this build instruction to `INTERCEPT_LIST_XSTAT`. For example, gcc compiler invokes `__xstat` to write the file modification time into the compiled artifacts, hence we add `gcc` to `INTERCEPT_LIST_XSTAT`.

All of the above function calls can be invoked from different locations such as the implementation of LSCSs themselves (shown in Section 3), the third party dependencies (e.g., openssl, mkimage), the build scripts (e.g., Make file), and some tools in the build toolchain (e.g., tar and gcc). Furthermore, the function calls can be invoked throughout the build process. Therefore, the best strategy is to centrally control the build instructions that invoke these function calls such that they return the same pre-defined date and time. Other intercept lists are also specified for other environment-induced sources of non-determinism. Here we describe two more examples: file ordering and pseudo-random number generation.

1) *File ordering*: Depending on the type of file system, reading the contents of a directory would return a list of files whose order is unspecified. The differently ordered list of files can either directly cause the non-equivalences if they are embedded in the generated artifacts; or indirectly if these files are processed in a different order during each build process. We found that a set of function calls (e.g., `readdir` and `nsfw`) whose behavior may vary depending on the types of file systems. To control the non-deterministic behavior, we add the build instructions that may invoke these function calls to the corresponding intercept list (e.g., make).
2) *Pseudo-random number generation*: Pseudo-random numbers are generated using different seeds and algorithms during each build process. Function calls that read a generated random number and write them on an intermediate artifact will cause non-equivalences (e.g., `rand`, `srand`, and `random`). For example, to generate unique serial numbers during the build process, pseudo-random number generation functions need to be invoked (e.g., oggenc [43] invokes `srand`). Hence, we add oggenc to the corresponding intercept list. Therefore, every time oggenc is executed, the same seed is used by `srand` during the pseudo-random number generation.

Other environment-induced non-equivalences are caused by the global environment variables, which may vary across different build environments. To control such non-deterministic behavior, we set the same pre-defined values to these environment variables. Below, we show two such examples:

1) *Locale*: Depending on the locale, information like the format of the time, and character encoding can be different. Such information is usually embedded in the build artifacts. To control this source of non-determinism, we can override the locale of the environment by setting a same pre-defined value for the global environment variable `LC_ALL`.
2) *Hash seeds*: Hash functions generally randomize their hash seeds while computing the hash values for numbers or strings. This can introduce non-equivalences in the build artifacts if hash functions are invoked by different build instructions. To control such source of non-determinism, we set the hash seeds as an environment variable (e.g., `PYTHONHASHSEED` for Python and `PERL_HASH_SEED` for Perl), so that different build instructions use the same hash seeds while computing the hash functions.

Build Toolchain. The recommended strategy to mitigate the non-equivalences caused by the build toolchain is remediation. We describe two sources of non-determinism that are induced by the build toolchain.

1) *Absolute file path.* The absolute path of the source files is written to the final artifacts as a debug information by many tools in the build toolchain (e.g., compilers, archiving tools, etc.). Many tools provide utilities for suppressing such outputs. For example, the outputted file paths in gzip can be easily removed by adding the command option `-n`. Similarly, we can set the command option `-fdebug-prefix-map=OLD=NEW` in gcc to prevent outputting the absolute path names during the compilation process.
2) *Randomly generated intermediate files.* The absolute path of the intermediate files are sometimes randomly generated and written to the final build artifacts, which cause non-equivalences. For example, the Windriver compiler writes the randomly generated intermediate files to the build artifacts. To address this source of non-determinism, we add the command option `-save-temps` in the build commands to specifically indicate the generated file paths.

System. Depending on the problem context, the recommended mitigation strategy for system induced non-equivalences can vary. Here we describe three common system induced sources of non-determinism:

1) *Non-initialized variables*: In programming languages like C, variables that are not explicitly initialized by the programmer are assigned with different values as they capture the random bytes in memory, causing non-equivalences. To mitigate such non-equivalences, the remediation strategy is recommended. Practitioners need to fix the problem by explicitly initializing variables.
2) *Digital signatures or encrypted information*: The built system requires that generated artifacts have stamped digital signatures or encrypted information, which causes non-equivalences. The preferred mitigation strategy is interpretation. For LSCSs, these non-equivalences cannot be eliminated or controlled due to security concerns. Instead, the location and the causes of such non-equivalences need to be clearly documented for product maintenance and auditing process.

3) *Documentation*: LSCSs include many documentation files generated during the build process. Some of them are generated by Microsoft applications and have a binary format. These binary documentation files can be non-equivalents even when the encoded textual contents are identical. For example, hhc.exe is a build-in tool provided by Microsoft for compiling chm documents. During the compiling process, random numbers are injected into the final documents, causing non-equivalences in the build artifacts. Since this tool is a commercial tool, it is impossible for us to diagnose and fix this problem. Similar to the digital signatures, such non-equivalences are interpreted and explained in the documentation. ps2pdf is another tool used for converting PostScript documents into pdf files. This tool randomly generates UUID (universally unique identifier) for each document. To address this source of non-determinism, we apply the remediation strategy by modifying the build scripts to specify pre-defined UUIDs.

5 Discussions and Future Work

In this section, we discuss the lessons learned by applying our approach to LSCSs within Huawei and present some future research directions.

5.1 Building LSCSs From Source Code

Before checking build verifiability for LSCSs, we need to ensure LSCSs can be built successfully from source code. However, this process is non-trivial due to the complex third party dependency relations in the LSCSs.

- *Software Bill of Materials (SBOM)*: LSCSs have multiple third party dependencies, each of which can also have additional third party dependencies. Hence, it is crucial to obtain the SBOM, which is a list of third party dependencies and their versions for the LSCSs [44]. We obtain this information by manually identify the exhaustive list of third party dependencies, which are involved during the build process. These third party dependencies can be from open source communities, from Huawei, or from other companies. For open source dependencies, we download their source code from the official website and re-build the projects while turning on the build profiler of ToolKitA. We manually identify additional dependent open source packages and download them. This process continues until all the dependent open source projects are downloaded. Same procedure is also repeated for packages from Huawei. For third party packages of other companies, since we cannot obtain their source code, we just record their names and versions in the build specification document. Currently, there are no well defined processes to systematically recover the SBOMs in the context of LSCSs. More research is needed in this area.
- *Unified Build Process*: The verifiable build process for LSCSs requires the entire system, including all third-party dependencies and internal libraries, to be built from source code. Therefore, each LSCS encompass a very heterogeneous set of components and this heterogeneity is amplified when many LSCSs are developed by the same company. For example, in Huawei, different LSCSs use different programming languages and open source dependencies which, in turn, use different build tools and build processes. Therefore, it is important to implement an unified build process accross all product teams within the same company. Although there are build tools (e.g., Bazel [17] and Buck [9]) that support build processes for different projects and different programming languages, unifying the build processes accross different product teams using the existing build toolchain is still an open research area. The independent auditing agency mention that "*binary equivalence remains a bespoke project, rather than a consistent output of* Huawei*'s build process*" [5]. Hence, a huge amount of effort is currently devoted in Huawei to adapt our toolset ToolKitA and to modify the build scripts or changing the build toolchain in order to unify the entire build process accross different product teams. Research is needed to support automated integration of the build processes for different LSCSs, which have complex dependency relations.

5.2 Comparison Among Different Mitigation Strategies

5.2.1 Advantages and Disadvantages

Although both controlling and remediation can be used to remove the non-equivalences in the build artifacts, both of them have their advantages and disadvantages.

Compared to remediation, controlling has three advantages: First, it is non-intrusive as practitioners only need to modify the *intercept* list instead of changing the source code or the build scripts of LSCSs or the third party packages. Second, different from remediation, whose impact is specific to the changed artifacts, controlling has a more general impact. For example, all the build instructions which reference a particular global environment variable will have a pre-defined value (e.g., see `LC_ALL` in Section 4.3). Third, as the prior resolution on controlling various environment-induced non-equivalences are encoded in the *intercept* list, recurrent problems caused by the environment are automatically mitigated. However, we need to take extra care on updating the *intercept* lists, as some build instructions(e.g., make, git) may behave in an unexpected way once intercepted. For example, if not careful, the pre-defined date and time would cause clock skew and the build process will either be blocked or behave differently [45]. In addition, our current implementation of ToolKitA intercept system calls that are performed by programs allocated at the user space and, therefore, it is not able to intercept system calls that are directly performed by the kernel space. Further research and development should propose a similar hooking mechanism for the kernel space.

Compared to controlling, remediation has one advantage, which is that we can independently verify whether the source of non-determinism is addressed. The impact of the remediation is localized and do not impact other build processes. On the other hand, the disadvantage is that we have to analyze each individual source of non-determinism of

different external dependencies, which is time-consuming. For example, to fix date and time issue, multiple patches are proposed for open source projects (e.g., openssl [46], sysstat [47], and lsof [48]).

When the non-equivalences should not or cannot be remediated or controlled due to design or security requirements, they should be interpreted. For each of the interpreted non-equivalences, we clearly document its location, the source of non-determinism, and the rationale. However, such analysis might need to be repeated each time new build requests come in, as their locations and contents may change from build to build. How to effectively track or mitigate the same interpreted non-equivalences across different builds or how to mitigate recurrent interpreted non-equivalences remain as an open research problem.

5.2.2 Soundness

Verifiable builds generate build artifacts that satisfy either one of the following conditions: (1) the generated build artifacts are equivalent; (2) if the build artifacts are not equivalent, the differences can be interpreted. The soundness of the explainable build process presented by Carnavalet *et al.* [21] can be validated, as it explains every source of non-determinism found in the build artifacts. However, this process lacks tooling support and the study is mainly done manually. In turn, the deterministic build process was proposed by Ren *et al.*[20] with an automated tool. RepLoc is a tool to support the mitigation of non-equivalences by means of remediation and, therefore, to produce reproducible builds. Since some non-equivalences are not possible to be remediated, this strategy does not necessarily satisfy the conditions for a verifiable build (unless all the non-equivalences can be interpreted). Therefore, the soundness of producing verifiable builds using only remediation cannot be validated. We can validate the soundness of our VBP in achieving verifiable builds based on the definition. First, our VBP can control a variety of sources of non-determinism, including those described by Ren *et al.* Second, for the remaining non-equivalent build artifacts, we interpret the differences similar to Carnavalet *et al.* The third party auditing agencies can independently check the verifiability of our VBP on the LSCSs.

5.3 The Cost of Applying VBP

The cost of applying our unified process to produce a verifiable build can be broken down into: (1) the cost of localizing and mitigating a new source of non-determinism; and (2) the cost of localizing and mitigating similar causes of recurrent non-equivalences across multiple releases or different systems.

As for (1), we observe that using our controlling strategy with the intercept & ignore lists to eliminate non-equivalences in the generated build artifacts is the strategy that contributes the most for the cost-effectiveness of our approach. The main reason is that controlling avoids having to recurrently localize build instructions that are associated with the same source of non-determinism (e.g., having to individually localize all build instructions that cause timestamp differences). The key technical characteristic of our controlling strategy is to intercept instructions at the system level, which allows to remediate the sources of non-determinism associated with different instructions all together (see Section 3.1 – Step 4). When sources of non-determinism can be automatically localized, the controlling strategy still outweighs the remediation mechanism with respect to cost-effectiveness. For example, RepLoc [20] is the state-of-the-art tool that automatically identifies the build instructions that are accountable for non-equivalences and generates a ranked list of source files containing such build instructions. In this approach, the BSPs need to manually verify the instructions within the ranked list of files and remediate the non-equivalences one by one. All the non-equivalences that are reported by Ren *et al.*[20] can be mitigated through the controlling strategy.

As for (2), using the controlling strategy requires manual efforts at the beginning when BSPs need to identify the non-equivalences and localize the sources of non-determinism. As this process evolves, the effort for BSPs decreases because recurrent sources of non-determinism are all mitigated at once by our controlling mechanism (due to interception of instructions at the system level). The controlling mechanism also has a positive impact in another aspect regarding the cost effectiveness of our unified process, which is the portability among different systems and releases. Our unified process has been successfully adapted and applied to verifiably build hundreds of products within Huawei. RSEs can reuse the documented mitigations and run our ToolKitA with a standard set of build instructions in the intercept & ignore lists, saving a significant amount of effort usually spent in Steps 3, 4, 5 and 6 of our Mitigation Phase. New types of non-equivalences that are occasionally identified are mitigated by BSPs running a complete cycle of our Mitigation Phase, which will update the standard intercept & ignore list as well as the mitigation guideline. Informal feedback of the product teams suggests that both RSEs and BSPs are positive regarding the usage of our approach.

There are currently 239 products inside Huawei that apply our approach, including the three LSCSs described above. The product teams are satisfied with the verifiable builds generated and acknowledge that our approach has reduced a great amount of manual effort. Most of the non-deterministic behavior can be addressed by simply executing our ToolKitA (with a standard set of build instructions added to the intercept & ignore lists). By means of informal feedback from the product teams in Huawei, we have found that most of the non-deterministic instructions come from dynamic libraries, which can be controlled using our approach. However, very few products use static libraries (less than 1 percent in our LSCSs), for which the API hook mechanism does not apply. Our current solution is to convert the static libraries into dynamic libraries. How to efficiently control static libraries remains as an open research problem.

5.4 Sources of Non-Determinism in LSCSs

Identifying the sources of the non-determinism in non-verifiable builds is challenging and time consuming [19], [20], [21]. First, build artifacts have to be analyzed such that the non-equivalences can be traced back to the problematic files or build scripts that generate them. Then, additional manual inspection is required to further locate the specific lines of source code or build instructions. As we have described in the *mitigation guideline* in Section 4.3, the sources of non-

determinism can be induced from the environment, the build toolchain, or the system. By analyzing thousands of non-equivalences from LSCSs and their third party dependencies in the past few years, we find that the majority of the problems in the non-verifiable builds are caused by various interactions with the environment (e.g., querying the current date and time or returning a list of files). This is the main reason why we can support a verifiable build process for multiple LSCSs in a short period of time and produce 100 percent or over 99 percent of the equivalent build artifacts.

To cross-validate our findings, we have also applied our approach to all the open source packages distributed under CentOS7.8, which is a popular Linux distribution. The detailed evaluation results are discussed in Section 4.2.3. As shown in Section 4.2.3, before adopting our approach, there are 13,405 non-equivalent binary artifacts. Among these build artifacts, the majority ($61.48\% = \frac{8241}{13405}$) of the non-equivalent build artifacts are mitigated by controlling. In addition, 5,021 ($37.46\% = \frac{5021}{13405}$) non-equivalent build artifacts are mitigated through interpretation. These 5,021 build artifacts are the driver files of the operating system kernel and the non-equivalences are caused by digital signatures, which should be reserved due to security requirements. We first use the *intercept & ignore* list directly configured from the LCSCs without any modification. After careful examination, we found another 91 ($0.68\% = \frac{91}{13405}$) non-equivalent build artifacts can be mitigated through controlling by adding the related processes into the *intercept & ignore* list. In the end, through our approach, only 52 ($0.39\% = \frac{52}{13405}$) non-equivalent build artifacts remain. On one hand, this study clearly demonstrates the generalizability of our approach. On the other hand, it also shows that the main sources of non-determinism in CentOS are also environment-related, followed by a non-trivial portion of the non-equivalences caused by security requirements. This also demonstrates the advantage of our verified build process over the existing deterministic build processes, which mitigates the non-equivalences by remediation in a case-by-case basis.

5.5 Integration With the Development Workflow

Currently, our verifiable build process is recommended to run frequently when a project just starts or if the build toolchain has been recently updated. Once the non-equivalences have been successfully mitigated, the verifiable build process can be executed less frequently due to the performance overhead imposed to the build process (e.g., build profiling). Once the LSCSs are close to release deadlines, this process needs to be executed on the builds of each release candidate. However, such scheduling recommendation may not be optimal due to the following two reasons: (1) occasionally the verifiable build process can be invoked even when no changes are made to the existing build processes. This results in a waste of computing resources and analysis effort (especially for the interpreted non-equivalences). (2) If some non-equivalences are introduced in a series of code changes during which the verifiable build process is not invoked, practitioners will spend more time to pinpoint the sources of non-determinism. Future research may look into automated recommendation on when to invoke the verifiable build process in a cost-effective fashion.

6 Related Works

In this section, we discuss four areas of related work to this paper: (1) security-related issues in the build process, (2) fault localization, (3) analysis of runtime behavior of the build process, and (4) a comparison with related practices with the steps of the mitigation phase.

6.1 Security-Related Issues in the Build Process

The type of toolchain attack that a verifiable build is able to detect was first explained by Thompson [49]. In this attack, a compromised compiler is used to inject malicious code during the compilation of a source code. As a result, the generated build artifacts are compromised. Therefore, to ensure that a built software system is not compromised, in addition to inspecting the source code for malicious code, the whole build toolchain needs to be turstworthy. Wheeler [50] describes a countermeasure to this attack that is based on the idea of comparing the generated artifacts by a trusted compiler against those generated by a non-trusted compiler. Carnavalet *et al.* [21] discuss a set of challenges for verifiable builds in security-critical OSSs. The authors also describe the sources of non-determinism identified while checking build verifiability for the TrueCrypt system. Nikitin *et al.* [51] describe a decentralized software update infrastructure that builds the release and compares the generated artifacts against the deployed binaries. Leija *et al.* [16] describe the design of a container technology – called reproducible container – which can deterministically execute x86 – 64 instructions and Linux system calls. Compared to the papers above, our paper is the first research work focusing on the challenges and solutions for producing verifiable builds in the context of LSCSs.

libfaketime [52] is an open source utility that returns pre-specified date and time values and can be used during a verifiable build process. In comparison, our *intercept & ignore list* mechanism implemented on ToolKitA is a more generic mechanism which supports returning pre-defined values for many different build instructions and, therefore, control different sources of non-determinism. Moreover, our toolset ToolKitA implements the ignore list to avoid injecting pre-defined values in specific build instructions.

6.2 Fault Localization

Prior studies in fault localization [53] focus on reproducing the production environment [54], recovering call graphs [55], recording and replaying program execution [56], and generating artificial faults [57]. There are only two works on the localization of sources of non-determinism in verifiable builds [19], [20]. Ren *et al.* [19] present a tool, which automatically localizes the sources of non-determinism by leveraging information retrieval techniques. Their proposed tool can achieve an accuracy of 79 percent for the top-10 ranked files. In their subsequent work, Ren *et al.* [20] propose another tool that analyzes the build call graph to localize the sources of non-determinism. The accuracy of their new tool is 90 percent for the top-10 ranked instructions. Both tools aim to support the deterministic build process whose main objective is to remediate the sources of non-determinism. This is because they mainly focus on the build verifiability of OSSs, whose build environment varies. However, checking build verifiability in LSCSs has a different sets of challenges. LSCSs have the same build

TABLE 2
The Differences and Similarities of Steps of Our Mitigation Phase and Existing Practices Towards Build Verifiability

Step	Our approach	Existing practices	Comparison
Step 1 – Checking verifiability	Uses an integrated tool to ToolKitA to support checking build verifiability in multiple platforms.	Use reprotest [65] to vary build environment and decide whether two build instances produce equivalent build artifacts. Use diffoscope [38] to visualize the non-equivalences in build artifacts.	*Similarities*: The step in our approach for checking build verifiability uses the same techniques and concepts as the existing tools. *Differences*: Existing practices are typically supported on a single platform. Our ToolKitA for LSCSs works on multiple platforms.
Step 2 – Profiling build	Uses a build profiler tool integrated into ToolKitA and leverages an API hook mechanism to construct a build call graph and identify non-deterministic build instructions. Stores the build call graph in a database.	Van der Burg *et al.* [39] use build profiling to find inconsistent licences in OSS projects. Ren *et al.* [20] use build profiling to find root causes of non-equivalences for deterministic builds. Adams *et al.* [64] use for design recovery. Bezemer *et al.* [40] use for the identification of unspecified dependencies. Zhou *et al.* [54] use for fault localization.	*Similarities*: Our build profiler is similar to existing practices in the sense that it also intercepts specific system calls during the build process and constructs a build call graph. *Differences*: While ToolKitA records specific build instructions that are intercepted by using the API hook mechanism, existing practices typically parse the output of a profiling tool such as strace. As a result, our ToolKitA recovers the build call graph during the build, while existing practices recover the build call graph after the build process is finished.
Step 3 – Diagnosing non-equivalence	Mostly done by manual analysis of the sources of non-determinism that are still not mitigated. The build call graph constructed through the API hook mechanism helps us to identify high-level instructions that introduce non-determinism. The low-level instructions are identified by BSPs manually.	Carnavalet *et al.* [21] report the results of a manual analysis to identify sources of non-determinism and diagnose non-equivalences. Ren *et al.* [19], [20] propose automated tools to identify build instructions that are accountable for non-determininsm.	*Similarities*: Similar to Carnavalet *et al.* [21], we rely on manual effort to diagnose sources of non-determinism. However, our manual diagnosis is supported by a build call graph constructed during the build process. *Differences*: RepTrace [19] is the state of the art tool to automatically support the diagnosis of non-equivalences and its effectiveness was demonstrated to support the remediation strategy. However, after adding sufficient instructions in the intercept & ignore lists, we are able to easily produce a verifiable build of different systems, avoiding the need of sophisticated support for diagnosing non-equivalences. The usage of intercept & ignore lists offers flexibility to control a multitude of sources of non-determinism including but not limited those identified by RepTrace: Random numbers generated by different seeds, Randomness in file system ordering, Name of the build system, File attributes for owner (st_uid), group (st_gid), disk block location (ino_t) and device ID (st_dev), Debugging information on executable files, Intermediate files (logs, lock-files, etc.) generated in a non-deterministic form, Documentation in binary format (e.g., MS Office)
Step 4 – Documenting mitigation	Uses the mitigation guideline to document prior knowledge about the sources of non-determinism. For each category of source of non-determinism (environment, toolchain, or system), the mitigation guideline recommends the most suitable mitigation strategy (remediation, controlling, or interpretation).	Reproducible Builds [27] records a comprehensive knowledge base related to verifiable builds, including reusable practices and tools.	*Similarities*: Both Reproducible Builds [27] and our approach explicitly document the knowledge generated during the mitigation phase. We also reuse specific knowledge about the mitigation of sources of non-determinism throughout our process. *Differences*: In our documentation, we categorize the sources of non-determinism into the environment, the build toolchain, and the system. For each category, our approach recommends the most suitable mitigation strategy.
Step 5 – Updating intercept & ignore lists	Uses the intercept & ignore lists to manage the controlled build instructions.	Uses libfaketime to intercept time system calls and return predefined values.	*Similarities*: Both ToolKitA and libfaketime use a hook mechanism to intercept system calls and return predefined values instead of the value returned by the originally called function. *Differences*: libfaketime is over specific and does not support the heterogeneity of LSCSs. In particular, libfaketime supports controlling time-related build instructions, whereas our intercept & ignore lists support any build instruction.
Step 6 – Validating mitigation	Records a complete build specification that is used during the verification phase, including the complete build environment in a virtual machine and interpretation of sources of non-determinism.	Use build specifications as recommended by Reproducible Builds [66] and Debian [67].	*Similarities*: Build specifications are common documents for systems that can be verifiably built. *Differences*: Our LSCSs need to be provided to an independent auditing agency and thus requires a complete build specification that includes the build environment (i.e., virtual machine, build toolchain, set of source code, and pre-compiled proprietary libraries), as well as an explanation of the interpreted sources of non-determinism, the performed remediations, and the controlled build instructions.

environment, but new challenges like stringent security requirements, complex third party dependencies, and scalability. Therefore, the main focus of this paper is to describe our verified build process, which is more suitable for LSCSs.

6.3 Analyzing the Runtime Behavior of the Build Process

Various studies have been performed to analyze the behavior of the build process. Many studies focus on reducing the duration of the build by analyzing the runtime behavior of the build process [58], [59], [60]. Kerzazi *et al.* [61] and Zolfagharinia *et al.* [62] studied the build failures in commercial systems and OSSs. Tu *et al.* [63] and Adams *et al.* [64] profile the build process to collect build logs and system tracing data. Such profiling data is used to construct build call graphs, which can be used in a variety of purposes, such as detecting licensing inconsistency in OSSs [39], detecting undefined build dependencies in Make files [40], or localizing sources of non-determinism during the verifiable build process [20]. In this paper, we also generate and analyze the build call graphs to diagnose and mitigate the non-equivalences in the build artifacts.

6.4 Related Practices With the Steps of the Mitigation Phase

In this section, we describe the main differences and similarities between our mitigation phase and the existing practices to support build verifiability. In Table 2, for each step of the mitigation phase, we describe related research results and tools, and compare them with our approach.

7 Threats to Validity

In this section, we discuss the threats to validity.

7.1 External Validity

7.1.1 Programming Languages

Currently our approach focus on mitigating non-equivalences in C/C++ based systems. We are currently working on extending our support to systems written in other programming languages. However, the main idea behind them are similar. For example, we are working on supporting a verifiable build for Java-based systems. In particular, we are extending our build profiler to Java-based build tools (e.g., Maven and Gradle). In addition, we are also implementing a similar *intercept & ignore list* mechanism at the JVM level in order to control the environment-induced non-equivalences.

7.1.2 Build Environment

Our approach assumes a homogeneous build environment, as we use the same system image for the build both in the development and in the auditing environments. Such a homogeneous environment assumption is mainly because the LSCSs of Huawei are used in embedded devices, in which the deployment environment is fixed. Our approach will not work for systems which need to be deployed in a heterogeneous development, since those systems need to be deployed in various environmental settings. For example, Debian needs to support different locales (e.g., United States, United Kingdom, and German). For such systems, a deterministic build process is more appropriate [19], [20]) .

7.1.3 Mitigation Guideline

In Section 4.3, we summarize our recommended strategies to mitigate the non-equivalences in LSCSs and the third party dependencies. Although we also tried our process on many other open source systems to ensure generalizability, the reported sources of non-determinism and their associated recommended strategies are not exhaustive and are only based on our experience.

7.2 Internal Validity

Controlling is an important mitigation strategy introduced in our approach. It is a very effective approach to eliminating the environment-induced non-equivalences. However, it might cause side effects and introduce problems during the build process, as all the functions specified in the *interrupt list* will return pre-defined values. Therefore, exceptions need to be specified in the *ignore list*, such that whenever a function specified in the *ignore list* is invoked, real computed values would be returned.

7.3 Construct Validity

When we evaluate the performance of our approach, we measure the number of build artifacts that can be repeatedly produced (a.k.a., equivalent build artifacts). The improvement shown in Section 4.2 is a result of both remediating and controlling the sources of non-determinism in LSCSs. For the non-equivalences that cannot be mitigated through the aforementioned two strategies, we interpret them. If all of the non-equivalences in the build artifacts can be mitigated by any of the above three strategies, the build process for this system is considered as verifiable. This process has also been communicated and accepted by various audit organization.

8 Conclusion

Build verifiability is an important safety property to ensure that build artifacts correspond to the source code of that system. Previous research and practices mainly focus on developing and enhancing deterministic build. However, LSCSs within Huawei have challenges that deterministic builds are not able to handle, ranging from security requirements, third party dependencies, to large-scale code changes. To cope with these challenges, we have developed an integrated approach to produce verifiable builds for LSCSs. Our approach includes a unified build process and a toolset. Our approach supports three strategies to mitigate the non-equivalences in the build artifacts: remediation, controlling, and interpretation. We apply our approach to three LSCSs, showing effective results. We improved the proportion of build artifacts that are successfully verified from less than 50 to 100 percent. We cross validated our results by using our approach to build 2,218 open source packages distributed under CentOS 7.8, increasing the proportion of verified build artifacts from 85 to 99.9 percent. We also give an overview of our mitigation guidelines and share the lessons learned based on our experience in the past few years.

Finally, we describe some open research problems related to verifiable builds, which can be of interest to practitioners and software engineering researchers.

Acknowledgments

The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Huawei and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of Huawei's products.

References

[1] H. Security, *Malware Trends*, 2020. Accessed: Aug. 2020. [Online]. Available: https://us-cert.cisa.gov/sites/default/files/documents/NCCIC_ICS-CERT_AAL_Malware_Trends_Paper_S508C.pdf

[2] A. C. L. U. (ACLU), *How Malicious Software Updates Endanger Everyone*, 2020. Accessed: Augu. 2020. [Online]. Available: https://www.aclu.org/issues/privacy-technology/consumer-privacy/how-malicious-software-updates-endanger-everyone

[3] D. A. Wheeler, *Attack on SolarWinds Could Have Been Countered by Reproducible Builds*, 2020. Acessed: Jan. 2021. [Online]. Available: https://lists.reproducible-builds.org/pipermail/rb-general/2020-December/002109.html

[4] M. Perry, *Deterministic Builds Part One: Cyberwar and Global Compromise*, 2020. Accessed: Aug. 2020. [Online]. Available: https://blog.torproject.org/deterministic-builds-part-one-cyberwar-and-global-compromise

[5] HCSEC, "HCSEC annual report, "Huawei Cyber Security Evaluation Centre Oversight Board, 2020. [Online]. Available: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/923309/Huawei_Cyber_Security_Evaluation_Centre__HCSEC__Oversight_Board-_annual_report_2020.pdf

[6] B. Project, *Bitcoin Reproducible Build*, 2020. Accessed: Aug. 2020. [Online]. Available: https://bitcoinops.org/en/topics/reproducible-builds

[7] C. Project, *Chromium Reproducible Build*, 2020. Accessed: Aug. 2020. [Online]. Available: https://www.chromium.org/developers/testing/isolated-testing/deterministic-builds

[8] Debian, *Debian Reproducible Build How-to Wiki*, 2020. Accessed: Jun. 17, 2020. [Online]. Available: https://wiki.debian.org/ReproducibleBuilds/Howto

[9] Buck, *Buck: A High Performance Build Tool*, 2020. Accessed: Aug. 2020. [Online]. Available: https://buck.build

[10] D. McNutt, *The 10 Commandments of Release Engineering*, 2020. Accessed: Aug. 2020. [Online]. Available: https://www.usenix.org/legacy/events/lisa10/tech/slides/mcnutt.pdf

[11] R. Malik, *Developing Fast & Reliable iOS Builds at Pinterest (Part One)*, 2020. Accessed: Aug. 2020. [Online]. Available: https://medium.com/pinterest-engineering/developing-fast-reliable-ios-builds-at-pinterest-part-one-cb1810407b92

[12] Telegram, *Telegram Reproducible Build*, 2020. Accessed: Aug. 2020. [Online]. Available: https://core.telegram.org/reproducible-builds

[13] SwissCovid, *SwissCovid: DP3T Android App for Switzerland – Reproducible Build*, 2020. Accessed: Aug. 2020. [Online]. Available: https://github.com/DP-3T/dp3t-app-android-ch/blob/c64feecf68ee62391013ca3cd668a34443c63322/REPRODUCIBLE_BUILDS.md

[14] C.-W.-A. Backlog, *Allow for Reproducible Builds*, 2020. Accessed: Aug. 2020. [Online]. Available: https://github.com/corona-warn-app/cwa-backlog/issues/21

[15] L. Courtès and R. Wurmus, "Reproducible and user-controlled software environments in HPC with Guix," in *Proc. Eur. Conf. Parallel Process.: Parallel Process. Workshops*, 2015.

[16] O. S. N. Leija *et al.*, "Reproducible containers," in *Proc. 25th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2020, pp. 167–182.

[17] Bazel, *Bazel – FAQ*, 2020. Accessed: Aug. 2020. [Online]. Available: https://bazel.build/faq.html

[18] M. Perry, *Deterministic Builds Part Two: Technical Details*, 2020. Accessed: Aug. 2020. [Online]. Available: https://blog.torproject.org/deterministic-builds-part-two-technical-details

[19] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, "Automated localization for unreproducible builds," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 71–81.

[20] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie, "Root cause localization for unreproducible builds via causality analysis over system call tracing," in *Proc. 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 527–538.

[21] X. de Carne de Canavalet and M. Mannan, "Challenges and implications of verifiable builds for security-critical open-source software," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 16–25.

[22] R. Builds, *Reproducible Build Definition*, 2020. Accessed: Aug. 2020. [Online]. Available: https://reproducible-builds.org/docs/definition

[23] Lsof org, *Allow Reproducible Builds*, 2020. Accessed: Aug. 2020. [Online]. Available: https://github.com/lsof-org/lsof/pull/94

[24] Debian, *dpkg-buildpackage: Set BUILD_PATH_PREFIX_MAP for Build Tools to Generate Reproducible Output*, 2020. Accessed: Aug. 2020. [Online]. Available: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=862116

[25] Debian, *dh-buildinfo: Please Produce Stable Output*, 2020. Accessed: Aug. 2020. [Online]. Available: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=722186

[26] N. I. of Standards and T. (NIST), *Digital Signature Standard (DSS)*, 2020. Accessed: Aug. 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186–4.pdf

[27] R. Builds, *Reproducible Build Buy-in*, 2020. Accessed: Aug. 2020. [Online]. Available: https://reproducible-builds.org/docs/buy-in

[28] M. S. R. C. (MSRC), *Software Defense: Mitigating Common Exploitation Techniques*, 2020. Accessed: Aug. 2020. [Online]. Available: https://msrc-blog.microsoft.com/2013/12/11/software-defense-mitigating-common-exploitation-techniques/

[29] M. S. R. C. (MSRC), *Clarifying the Behavior of Mandatory ASLR*, 2020. Accessed: Aug. 2020. [Online]. Available: https://msrc-blog.microsoft.com/2017/11/21/clarifying-the-behavior-of-mandatory-aslr/

[30] *The Unreproducible Package*, 2021. Accessed: Apr. 2021. [Online]. Available: https://github.com/bmwiedemann/theunreproduciblepackage

[31] *Value Initialization*, 2021. Accessed: Apr. 2021. [Online]. Available: https://reproducible-builds.org/docs/value-initialization/

[32] Oracle, *Oracle Berkeley DB*, 2021. Accessed: Jul. 2021. [Online]. Available: https://www.oracle.com/database/technologies/related/berkeleydb.html

[33] Oracle, *Oracle Berkeley DB repository*, 2021. Accessed: Jul. 2021. [Online]. Available: https://github.com/berkeleydb/libdb/blob/master/src/os/os_uid.c

[34] Ghostscript, *Support SOURCE_DATE_EPOCH for Reproducible Builds*, 2020. Accessed: Aug. 2020. [Online]. Available: https://bugs.ghostscript.com/show_bug.cgi?id=696765

[35] A. Developer, *Validating Your Version of Xcode*, 2020. Accessed: Aug. 2020. [Online]. Available: https://developer.apple.com/news/?id=09222015a

[36] A. Support, *Safely Open Apps on Your Mac*, 2020. Accessed: Aug. 2020. [Online]. Available: https://support.apple.com/en-ca/HT202491

[37] C. Sung, S. Lahiri, P. Choudhury, M. Kaufman, and C. Wang, "Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study," in *Proc. 42nd Int. Conf. Softw. Eng.*, 2020, pp. 172–181.

[38] reprotest, 2021. Accessed: Apr. 2021. [Online]. Available: https://diffoscope.org

[39] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel, "Tracing software build processes to uncover license compliance inconsistencies," in *Proc. Int. Conf. Automated Softw. Eng.*, 2014, pp. 731–742.

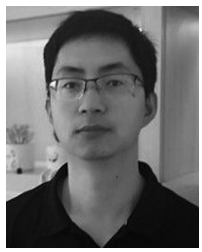
[40] C.-P. Bezemer, S. McIntosh, B. Adams, D. M. German, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 3117–3148, 2017.

[41] LD_PRELOAD, *ld.so(8) — Linux Manual Page*, 2020. Accessed: Aug. 2020. [Online]. Available: https://man7.org/linux/man-pages/man8/ld.so.8.html

[42] Microsoft, *Microsoft Research Detours Package*, 2021. Accessed: Jan. 2021. [Online]. Available: https://github.com/microsoft/Detours

[43] *OggSerialNumbers*, 2020. Accessed: Aug. 2020. [Online]. Available: https://wiki.debian.org/ReproducibleBuilds/OggSerialNumbers
[44] C. Yanko, *Using a Software Bill of Materials (SBOM) is Going Mainstream*, 2020. Accessed: Aug. 2020. [Online]. Available: https://blog.sonatype.com/software-bill-of-materials-going-mainstream
[45] Clockskew, *Make Mac Bundles Built With LXC Match Their KVM Counterparts*, 2020. Accessed: Aug. 2020. [Online]. Available: https://gitlab.torproject.org/legacy/trac/-/issues/12240
[46] openssl, *Fix SOURCE_DATE_EPOCH bug; use UTC*, 2020. Accessed: Aug. 2020. [Online]. Available: https://github.com/openssl/openssl/commit/8a8d9e190533ee41e8b231b18c7837f98f1ae231
[47] sysstat, *How can we Eliminate the Build Differences Caused by the Time Stamps for Reproducible Build*, 2020. Accessed: Aug. 2020. [Online]. Available: https://github.com/sysstat/sysstat/issues/164
[48] lsof, *Allow Reproducible Builds*, 2020. Accessed: Aug. 2020. [Online]. Available: https://github.com/lsof-org/lsof/pull/94
[49] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984.
[50] D. A. Wheeler, "Countering trusting trust through diverse double-compiling," in *Proc. 21st Annu. Comput. Secur. Appl. Conf.*, 2005, pp. 13–48.
[51] K. Nikitin *et al.*, "CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1271–1287.
[52] Wolfcw, *libfaketime*, 2020. Accessed: Aug. 2020. [Online]. Available: https://github.com/wolfcw/libfaketime
[53] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.
[54] X. Zhou *et al.*, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 683–694.
[55] Y. Yuan, L. Xu, X. Xiao, A. Podgurski, and H. Zhu, "RunDroid: Recovering execution call graphs for android applications," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 949–953.
[56] E. T. Barr, M. Marron, E. Maurer, D. Moseley, and G. Seth, "Time-travel debugging for JavaScript/Node.js," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 1003–1007.
[57] S. Pearson *et al.*, "Evaluating and improving fault localization," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 609–620.
[58] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 123–133.
[59] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Identifying and understanding header file hotspots in C/C++ build processes," *Automated Softw. Eng.*, vol. 23, no. 4, pp. 619–647, 2016.
[60] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos, "Reducing build time through precompilations for evolving large software," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance*, 2005, pp. 59–68.
[61] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? An empirical study," in *Proc. 30th IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 41–50.
[62] M. Zolfagharinia, B. Adams, and Y.-G. Guéhéneuc, "A study of build inflation in 30 million CPAN builds on 13 Perl versions and 10 operating systems," *Empir. Softw. Eng.*, vol. 24, pp. 3933–3971, 2019.
[63] Q. Tu and M. W. Godfrey, "The build-time software architecture view," in *Proc. 18th IEEE Int. Conf. Softw. Maintenance*, 2001, pp. 398–407.
[64] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter, "Design recovery and maintenance of build systems," in *Proc. 23rd IEEE Int. Conf. Softw. Maintenance*, 2007, pp. 114–123.
[65] *reprotest*, 2021. Accessed: Apr. 2021. [Online]. Available: https://pypi.org/project/reprotest/
[66] *Formal Definition*, 2021. Accessed: Apr. 2021. [Online]. Available: https://reproducible-builds.org/docs/formal-definition
[67] *ReproducibleBuilds BuildinfoFiles*, 2021. Accessed: Apr. 2021. [Online]. Available: https://wiki.debian.org/ReproducibleBuilds/BuildinfoFiles

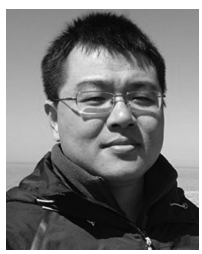
Yong Shi received the master's degree in computer science from the Dalian University of Technology, Dalian, China. He is currently an expert on software engineering and cyber security with the Trustworthiness Theory, Technology & Engineering Lab, Huawei Technologies Co., Ltd. His research interests include build engineering, security defense, and static analysis.



Mingzhi Wen received the BSc degree from the Mathematics Department, Xidian University, Xi'an, China. He is currently a principle engineer with the Beijing Research Center, Huawei Technologies Co., Ltd. His research interests include trustworthy build, including reproducible build processes and accurate tracking of the build processes. He has 14 years of experience in the cyber security field.

Filipe R. Cogo received the BSc and MSc degrees in computer science from the Universidade Estadual de Maringá (UEM), Maringá, Brazil, and the PhD degree from Queen's University, Kingston, Canada. He is currently a software engineering researcher with the Centre for Software Excellence, Huawei Canada. His research interest include empirical software engineering, mining software repository, and AI4SE.

Boyuan Chen received the BEng degree from the School of Computer Science, University of Science and Technology of China, Hefei, China, and the MASc and PhD degrees from the Department of Electrical Engineering and Computer Science, York University, Toronto, Canada. He is currently a senior researcher with the Centre for Software Excellence, Huawei Canada. His research interests include software engineering and artificial intelligence, in particular, software logging, SE for AI, empirical software engineering, DevOps, and software testing. He has papers published in top venues like ICSE, ASE, EMSE, and CSUR.



Zhen Ming Jiang received the BMath and MMath degrees in computer science from the University of Waterloo, Waterloo, Canada, and the PhD degree from the School of Computing, Queen's University, Kingston, Canada. He is currently an associate professor with the Department of Electrical Engineering and Computer Science, York University, Toronto, Canada. His research interests include software engineering and computer systems. Some of his research results are already adopted and used in practice on a daily basis. For more information, please visit http://www.cse.yorku.ca/∼zmjiang.