Towards Build Verifiability for Java-based Systems

Jiawen Xiong[1], Yong Shi[2], Boyuan Chen[3], Filipe R. Cogo[4], Zhen Ming (Jack) Jiang[5]
Huawei China[1,2], Huawei Canada[3,4], York University[5]
Shenzhen, China[1,2], Kinston, Canada[3,4], Toronto, Canada[5]
{xiongjiawen,young.shi,boyuan.chen1,filipe.roseiro.cogo1}@huawei.com,zmjiang@eecs.yorku.ca

ABSTRACT

Build verifiability refers to the property that the build of a software system can be verified by independent third parties and it is crucial for the trustworthiness of a software system. Various efforts towards build verifiability have been made to C/C++-based systems, yet the techniques for Java-based systems are not systematic and are often specific to a particular build tool (e.g., Maven). In this study, we present a systematic approach towards build verifiability on Java-based systems. Our approach consists of three parts: a unified build process, a tool that dynamically controls non-determinism during the build process, and another tool that eliminates non-equivalences by post-processing the build artifacts. We apply our approach on 46 unverified open source projects from Reproducible Central and 13 open source projects that are widely used by Huawei commercial products. As a result, 91% of the unverified Reproducible Central projects and 100% of the commercially adopted OSS projects are successfully verified with our approach. In addition, based on our experience in analyzing thousands of builds for both commercial and open source Java-based systems, we present 14 patterns that introduce non-equivalences in generated build artifacts and their respective mitigation strategies. Among these patterns, 11 (78%) are unique for Java-based system, whereas the remaining 3 (22%) are common with C/C++-based systems. The approach and the findings of this paper are useful for both practitioners and researchers who are interested in build verifiability.

KEYWORDS

Verifiable build, Build system, Security, Software engineering

ACM Reference Format:
Jiawen Xiong[1], Yong Shi[2], Boyuan Chen[3], Filipe R. Cogo[4], Zhen Ming (Jack) Jiang[5]. 2022. Towards Build Verifiability for Java-based Systems. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513050

1 INTRODUCTION

Java is one of the most prominent programming languages in the software industry, ranked third in the TIOBE index [15]. Given the popularity of Java, both industry and open source initiatives are actively researching forms of improving the security of applications written in this programming language. Build verifiability is an important security property that ensures correspondence between the source code and the deliverable packages that are distributed to final users. Build verifiability is paramount for both commercial and Open Source Software (OSS) systems and can potentially prevent incidents such as software supply chain attacks [2, 3, 29, 34], which silently injects malicious code into a distributed package during the build process [33]. Before we can consider a build setup as a *verifiable build* [25], one of the following two properties needs to be satisfied by the generated deliverable packages: (1) generated packages by two build instances are always equivalent (i.e., have the same contents), or (2) the technical details behind occasional non-equivalences in the built packages can be explained (e.g., using Name Space Layout Randomization [1] to defend against code injection attacks). When all deliverable packages satisfy the first property, we consider the build setup as a *reproducible build* [18].

Producing a verifiable build is not trivial due to sources of non-determinism present in the build toolchain, the build environment, or the design of a software system. Prior research proposed different approaches towards producing verifiable builds of C/C++-based systems [27, 30, 31]. In particular, our prior work [32] proposed a unified process and a toolkit to produce verifiable builds for C/C++-based large-scale industrial systems. Our unified process encompasses a catalog of remediation strategies that is periodically updated whenever new sources of non-determinism are identified and mitigated. We leverage the following three different mitigation strategies: (1) *controlling*, which intercept non-deterministic build instructions at runtime and returns pre-defined values [27]; (2) *remediation*, which modifies source code and build scripts to mitigate sources of non-determinism [30]; and (3) *interpretation*, which provides a traceable explanation of eventual non-equivalences in the build artifacts that are introduced by design [25]. Our approach has been checked by an independent auditing organization for compliance [11] and is currently used by hundreds of systems within Huawei. However, the aforementioned existing approaches cannot be directly applied to Java-based systems due to the following challenges:

- **Distinct sources of non-determinism.** The sources of non-determinism that cause non-equivalences in Java packages can be different from those of C/C++ packages. There are no prior investigations on the sources of non-determinism that are exclusively related to Java systems. For example, some specific sources of non-determinism stem from how the compilation mechanism of JavaDoc [12] and JSP [16] files works. Similarly, it is unknown if there are any common sources of non-determinism between Java- and C/C++-based systems.
- **Distinct mitigation approaches.** The approaches to mitigating the sources of non-determinism are different between Java

ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513050

and C/C++-based systems. For example, the controlling mechanism used in C/C++-based systems dynamically intercepts non-deterministic build instructions at the kernel level (e.g., using `LD_PRELOAD` hooks [32]). These mechanisms cannot work directly on Java-based systems, as the non-deterministic build instructions need to be intercepted at the JVM level. As a result, new mitigation approaches are needed for Java-based systems.

- **Distinct build mechanisms.** Java-based systems have different build mechanisms compared to C/C++-based systems. Many Java-based systems are built by automated build tools (e.g., Maven and Gradle), which either use pre-compiled libraries locally or automatically retrieve them from central remote repositories (e.g., Maven Central). In addition, a Java package should, in principle, run in any platform with an installed JVM instance. This process is different from the build of C/C++ systems whose libraries are platform-dependent and typically stored in local repositories. Therefore, we also need to consider additional aspects when analyzing and improving build verifiability for Java-based systems.

Build verifiability of Java-based systems is supported by associated plugins with each build tool. However, these plugins require individual installation and configuration for each system and are only able to mitigate a limited set of sources of non-determinism. To tackle these challenges, in this paper, we propose a new approach to systematically diagnosing and automatically mitigating the sources of non-determinism during the build of a Java-based system. The automatic mitigation leverages dynamic bytecode instrumentation [13] to control the non-deterministic build instructions (a.k.a., the controlling mechanism). We further improve the interpretation mechanisms such that we not only explain the non-equivalences but also demonstrates their effect through a post-processing step. We have applied our approach on 46 projects from Reproducible Central [5] and 13 OSS projects that are often used by commercial applications from Huawei. The build from 55 (93.2%) of the projects can now be fully verified, compared to 0 previously. The contributions of our paper are the following:

(1) This is the first study that systematically investigates build verifiability for Java-based systems. Through our experience in verifying the deliverable packages of thousands of Java-based projects, we have derived a set of root causes and their associated mitigation strategies.
(2) Compared with existing approaches to producing verifiable builds for Java-based applications, our approach is shown: a) to mitigate sources of non-determinism that are not mitigated by existing approaches, b) to prevent the modification of existing build setups or the integration and configuration of plugins, c) to affect only specific fields and methods of specific classes, d) to integrate seamlessly with the most popularly adopted Java build tools, and e) to extend effortlessly to mitigate new sources of non-determinism.
(3) We report 14 patterns that yield non-deterministic build instructions in Java-based systems and their associated mitigation strategies. While comparing against previously reported patterns in C/C++-based systems, 11 patterns are new and unique for Java-based systems.

Paper organization: Section 2 presents the motivation and background material of our paper. Section 3 presents our approach to produce verifiable builds of Java-based systems. Section 4 presents the case study results. Section 5 discusses the results of our case study. Section 6 presents the threats to the validity of our paper. Finally, Section 7 presents our conclusions.

2 BACKGROUND AND RELATED WORKS

In this section, we present how Java-based systems are typically built (Section 2.1), the existing approaches to producing verifiable builds (Section 2.2), and how verifiable builds are currently produced for Java-based systems (Section 2.3).

2.1 The build of Java-based systems

The build of a Java-based system encompasses the four general phases as shown in Figure 1. We explain each of the general build phases below by comparing against the build process for C/C++-based systems.

Source retrieval. During the *source retrieval* phase, the *build configuration file* and the *source code* are fetched from a Version Control System (VCS). This process is typically supported by automated build tools and is similar in both Java- and C/C++-based systems, although the automated build tools are different depending on the language. The build configuration file is responsible for setting up access to *dependency repositories*, determining which *dependencies* are retrieved and linked during a build process, customizing the behaviour of the *build tool* and its associated *plugins*, and configuring options for the *compiler*.

Dependency retrieval. In Java, it is common to retrieve dependencies (e.g., `jar` files) from either local or remote (e.g., Maven Central) *dependency repositories*. Dependencies are typically retrieved by the associated package manager with the build tool. The functionalities provided by library packages are directly reused by the built application. In contrast, in C/C++-based systems, dependencies are typically retrieved from local repositories of shared libraries.

Compiling and linking. The next phase is automatically supported by build tools and plugins and is broken down into two steps: (1) The compiling step compiles the source files that are retrieved in the *source retrieval* phase. (2) The linking step binds the compiled artifacts with the obtained dependencies in the *dependency retrieval* phase to produce a set of executable files or instructions living inside runtime environments (e.g., JVM). The output of these two steps is one or more *built artifacts* (called `class files` in Java systems and `object files` in C/C++ systems) that are used as input to the next phase. In Java, the linking process is performed by the JVM. Most of the major Java build tools (e.g., Ant, Maven, and Gradle) run over the JVM, as they are Java applications themselves. Therefore, any sources of non-determinism that stem from the JVM also affect the Java build tools.

Packaging. In the last phase, the built artifacts from the previous phase (e.g., class files) and additional *package metadata* (e.g., `MANIFEST.MF` files) are archived in a *deliverable package* for distribution. In Java-based systems, deliverable packages are distributed as a deployable Java application (e.g., a `war` file) or a Java library (e.g., a `jar` file). Similarly, the deliverable packages of C/C++-based

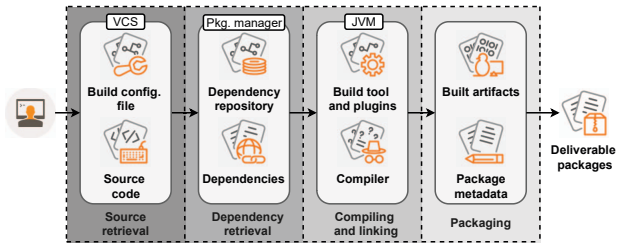

Figure 1: The build process of Java-based systems.

systems are distributed as platform-dependent executable files (e.g., in ELF format) or shared libraries (e.g., so files).

2.2 Existing approaches towards producing verifiable builds

Three main approaches can represent prior efforts towards verifiable builds. We refer to the first approach as "controlling", which comprises a mechanism that intercepts non-deterministic build instructions at runtime and replaces the returning value of these instructions with pre-defined deterministic values [27, 32]. The second approach is called "remediation" and comprises directly modifying non-deterministic instructions in the source code or build scripts. We refer to the third approach as "interpretation", which provides legitimate explanations about non-equivalence in generated build artifacts. Optionally, additional post-processing step(s) can be introduced to demonstrate the correctness of the explanations.

Prior research studies have been conducted to ensure build verifiability of C/C++-based systems. Carnavalet et al. [25] observe several related challenges to build verifiability of OSS systems. The authors manually identify and explain a set of sources of non-determinism in security-critical OSS systems. Ren et al. [30, 31] adopt an automated build profiling technique to identify accountable instructions for introducing non-equivalences in the built packages. Our prior work [32] proposes a unified process and a toolkit to produce verifiable builds of C/C++ applications. Results show that the controlling mechanism implemented by our toolkit can mitigate most of the sources of non-determinism in both large-scale commercial systems and OSS systems. Leija et al. [28] proposes a reproducible container, which can execute system calls in a deterministic way to eliminate sources of non-determinism from the build environment. Reproducible-Build [18] is a community-based effort to document the best practices and relevant tools for checking and verifying build reproducibility. It mainly focuses on C/C++ systems (e.g., packages of the Debian distribution of Linux) and highlights that producing reproducible builds of Java systems is challenging.

2.3 The current state of build verifiability for Java-based systems

There are different tools for mitigating some sources of non-determinism and verifying deliverable packages for Java-based systems [6, 8, 9, 17]. Each of these tools addresses one or more of the following three sources non-determinism: a) *timestamps*: `jar` and configuration files (e.g., `pom.xml`) contain timestamps that are either replaced by pre-defined values or stripped off [14], b) *file order*: depending on the build process, packaged files in a `jar` file can have different order. Such packages files are then sorted after the build process is finished [7], c) *metadata on* `manifest.mf` *files*: user names and tooling version that are recorded in manifest files are stripped off [10]. The aforementioned solutions are natively supported by the major automated build tools (namely Maven and Gradle).

However, two main limitations render these solutions unsuitable for verifying the build of Java-based systems in an industrial setting: (1) *Limited tool capability*: our experience on building industrial Java-based systems shows that there are several sources of non-determinism not covered by the provided solutions, such as sorting of symbol tables in the generated `jar` files and other non-equivalences introduced by specific tools (e.g., the Jasper compiler); and (2) *Complex installation and configuration processes*: since none of the existing solutions supports all the usage scenarios and build tools, one has to install and configure all of them to provide a general solution used in the industrial context. This characteristic requires huge manual effort and cannot scale to various build environments and settings, typically needed by industrial systems. These limitations motivated us to develop a new approach to produce verifiable builds for industrial Java-based systems that, compared to the native solutions offered by automated build tools, is more flexible, extensible, and generalizable. We will cover the details of our approach in the next section.

3 OUR APPROACH

As shown in Figure 2, our approach consists of five phases. (1) During the *Checking build verifiability* phase, we prepare the build environment and invoke the build process. Then we check if the deliverable package is verifiable. (2) During the *Diagnosing sources of non-determinism* phase, we study from existing literature and tool documentation to diagnose the sources of non-determinism in the deliverable package. (3) During the *Mitigating sources of non-determinism* phase, we configure our developed tools to control and interpret various sources of non-determinism. (4) During the *Documenting root causes and mitigation strategies* phase, we move back to Phase 1 to recheck the deliverable package. The process is repeated until the deliverable package is successfully verified. Then we document the root causes of non-determinism and update the corresponding mitigation strategies. Then in the (5) *Outputting the deliverable package and build specifications* phase, we output the verified deliverable package along with the build specifications, which clearly describe the build environment and setup.

To ease explanation, in the rest of the section, we will describe our approach using a running example, which is a Java-based system Foo consisting of two source code files, `Bar.java` and `Baz.java`, and a Maven configuration file `pom.xml`. After build, it will generate a deliverable package `Foo.jar`, which contains the following build artifacts: two `class` files (`Bar.class` and `Baz.class`) and three configuration files (`MANIFEST.MF`, `pom.properties`, and `pom.xml`).

Phase 1 - Checking build verifiability. The objective of this phase is to check the verifiability of the deliverable package generated from our build process. We follow the same setup previously

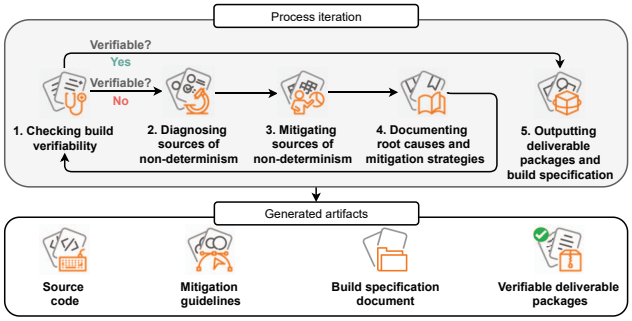

Figure 2: An overview of our approach.

adopted in C/C++-based systems [32], where we use the same build environment, build specifications, and build commands to start two build processes to produce deliverable packages. This phase is further broken down into three steps:

Step 1 - Collecting build specifications: In this step, we collect build information (e.g., JDK version, build tools, and dependencies) and record it in build specifications documents.

Step 2 - Setting up the build environment: In this step, we prepare the build environment according to the build specifications. Typically, we encapsulate the build environment in a container (e.g., docker) or a virtual machine (VM) to ensure that the build environment is consistent. In our running example, we use the same build environment throughout (docker with Ubuntu 18.04 LTS), JDK 1.8.0_111, and Maven 3.6.0).

Step 3 - Invoking the build process and checking build verifiability: In this step, we carry out the build processes following the specified procedure. Two repeated build processes are invoked in the same environment with the same setup. To ease explanation, we refer to the resulting deliverable package of the first build process as DP_1 and the resulting deliverable package of the second build process as DP_2. We compare the SHA1-checksums of DP_1 and DP_2. If they are identical, we consider that the build is verifiable and move on to Phase 5. If not, the build is not verifiable. Further inspection will be carried out in the next phase to diagnose sources of non-determinism.

Phase 2 - Diagnosing sources of non-determinism. The objective of this phase is to identify the root cause of the non-equivalences in the two deliverable packages generated in Phase 1. This phase is conducted manually and consists of the following two steps:

Step 1 - Studying existing research literature and tool documentation: As there are no prior research studies focusing on Java-based systems, in this step, we first search on existing research work on build verifiability for C/C++-based systems [25, 30–32]. Then we collect various tools for producing verifiable builds for Java-based systems in the wild [4, 7, 8] and summarize the objectives and the solutions from these tools by studying their documentation. The collected knowledge will benefit us in the diagnosis process.

Step 2 - Comparing build artifacts: In this step, we first unpack two non-equivalent deliverable packages DP_1 and DP_2 to extract two lists of build artifacts. The build artifacts usually consist of a set of `class` files and a set of text-based files. We apply diffing tools (e.g., `diffoscope`) to compare the build artifacts that have the same path and name. Since the `class` files are in the bytecode format, we first apply the `javap` command to decompile them into text-based representation. Then we compare the text-based representations line by line to examine the differences. For text-based files such as `MANIFEST.MF`, we directly examine the non-equivalences. We also check the orders and file properties (e.g., created time) of build artifacts embedded in the deliverable packages as it also introduces non-equivalences in the deliverable package. We cross-check the identified non-equivalence and collected knowledge to summarize the root cause of these non-equivalences.

In our running example, after unpacking the deliverable package, five build artifacts are extracted. Four build artifacts: `Bar.class`, `Baz.class`, `MANIFEST.MF`, and `pom.xml` are equivalent, while `pom.properties` is not equivalent. Listing 1 shows the non-equivalences that exist in the auto-generated messages, which contain timestamps. The timestamp differences are caused by the build environment, as two build processes are invoked at different times. This pattern of non-equivalence is further explained in Section 4.2 as [P1]. In addition, the order of the build artifacts in the deliverable package is not deterministic. In DP_1, the file `Bar.class` is listed before `Baz.class`, and it is the other way around in DP_2. This is caused by the multi-threaded compilation of the Java compiler. This pattern of non-equivalence is further explained in Section 4.2 as [P11].

Listing 1: Example differences of timestamp in `pom.properties`.

```
# Generated by Maven
- # Sun Sep 18 22:43:23 EDT 2021
+ # Sun Sep 18 22:45:35 EDT 2021
```

Phase 3 - Mitigating sources of non-determinism. The objective of this phase is to leverage automated techniques to mitigate sources of non-determinism. This phase consists of two steps, which are automatically invoked during the compiling and linking, and the packaging phases of the build process, respectively.

Step 1 - Applying bytecode instrumentation to control sources of non-determinism: In this step, sources of non-determinism are controlled by dynamically altering the behavior of build tools via bytecode instrumentation. Modern build tools (e.g., Ant, Maven, and Gradle) for Java-based systems are also implemented in Java. Hence, these build tools must first be loaded by JVM to start the build process. By default, the build tools write metadata such as timestamps into various build artifacts. Some metadata is non-deterministic and cannot be easily mitigated. Hence, we develop a technique called JavaBEPEnv, which includes a custom Java Agent program. It leverages the Java Instrument API to modify the bytecode of build tools dynamically. Java Instrument API is a set of APIs supported by JVM to instrument bytecode when it is being loaded in the JVM. To control the sources of non-determinism, JavaBEPEnv replaces the non-determinism introducing methods (e.g., `currentTimeMillis()`) with customized methods. These customized methods have the same method signatures as the non-determinism introducing methods, but they only return deterministic information (e.g., a fixed timestamp) to keep the outputted information consistent. JavaBEPEnv can be configured to be attached to JVM when the build process starts and control the non-deterministic behavior automatically.

Step 2 - Applying post-processing to interpret sources of non-determinism: In the build process of Java-based systems, many non-equivalences are caused by different phases of the process (e.g., compiling and linking, and packaging), which cannot be easily controlled. Post-processing the build artifacts by rules could explain the non-equivalences and verify they are not malicious.

Interpreting sources of non-determinism means that an open and transparent technique will be automatically applied to build artifacts to eliminate non-equivalences. The technique should be transparent to third-party stakeholders who want to verify the build independently. We implement such a technique called JavaBEPFix, including rules like: (1) it leverages the Byte Code Engineering Library (BCEL) to modify the non-equivalent `class` files. BCEL is an open-source library to analyze, transform, and manipulate class files. With BCEL, JavaBEPFix is able to interpret many sources of non-determinism that cause non-equivalent `class` files, such as [P6] Constant pool and [P7] Temporary variables (details shown in Section 4.2). (2) It automatically unpacks the deliverable package, sorts the build artifacts in a deterministic order based on predefined configuration (e.g., sort the files by name), and repacks the build artifacts into a post-processed deliverable package. (3) It also leverages the standard java.nio API to keep the creation time of build artifacts consistent. Other sources of non-determinism that can be interpreted by JavaBEPFix are discussed in Section 4.2.

Similar to C/C++-based systems, to mitigate some sources of non-determinism in the deliverable packages of Java-based systems, we also need to leverage the remediation strategy (e.g., editing source code or configuration files, or upgrading dependencies). We will not mention the details here due to page limitations. Further details about how remediation is applied to produce verifiable builds can be found in our prior work [32].

In our running example, both JavaBEPEnv and JavaBEPFix are enabled during the build process. As a result, the timestamp generated in `pom.properties` files will always be the same, as the methods that generate timestamps are intercepted and transformed. All the build artifacts are automatically sorted and repacked into a deliverable package by JavaBEPFix.

Phase 4 - Documenting root causes and mitigation strategies. The objective of this phase is to document root causes of non-determinism and the corresponding mitigation strategies. For each newly discovered source of non-determinism, we document its root cause and summarize it into a specific category. We also describe the recommended mitigation strategies. Such documentation is beneficial for producing verifiable builds in Java-based systems, as it can be reused whenever a documented root cause is identified. Please refer to Section 4.2 for a detailed documentation of various patterns of non-determinism in Java-based systems. After this phase, we move back to Phase 1 and check if the new deliverable package after mitigation is verifiable.

In our running example, we document the patterns of timestamp and entries in deliverable packages. The root cause of non-equivalent timestamps is the build environment. The mitigation strategy is to control the timestamp using `JavaBEPEnv`. The root cause of randomly ordered entries in deliverable packages is due to multi-threaded compiling. The mitigation strategy is to interpret the non-equivalences by sorting the built artifacts.

Phase 5 - Outputting deliverable packages and build specifications. This phase begins once we deem the deliverable package as verifiable, after Phase 1. The objective of this phase is to output the verifiable deliverable packages along with build specifications.

The build specifications consist of three parts: (1) the build environment; (2) the build commands; and (3) the additional operations applied on the build artifacts for build verifiability. The build environment could either be a docker file (if the build is started within a container), a VM image, or a detailed description of the host operating system (OS), and dependent libraries (e.g., the detailed versions of JDK), and so on. The build commands include the exact instructions to start the build process. The applied mitigation strategies are also documented. Independent builders can leverage the build specifications to verify the deliverable packages. Similar to our prior work [32], the outputs of this phase are provided to independent agencies for security auditing if needed.

We have included all the above information in our running example and delivered them to third-party auditing agencies. They acknowledged the build is verifiable.

4 CASE STUDY

In this section, we present the evaluation of applying our approach on Java-based systems. In the past few years, we have applied our approach on various systems ranging from OSS to commercial systems within Huawei. These systems are from different application domains such as server-based systems, middleware libraries, and mobile apps. Due to confidentiality, we cannot directly discuss the evaluation details on our commercial systems. Hence, to demonstrate the effectiveness of our approach, we have applied our approach on various representative Java-based OSS systems. The case study setup and the evaluation results are described in Section 4.1. Section 4.2 shares the sources of non-determinism and our proposed mitigation strategies based on our experience on applying our approach on thousands of Java-based commercial and open source systems.

4.1 Performance Evaluation

In this section, we describe our case study setup (Section 4.1.1) and evaluation results (Section 4.1.2).

4.1.1 Case Study Setup. Here we describe how to set up our case study on two different datasets of OSS projects. We first present the setup of OSS projects from Reproducible Central [5]. Next, we present the setup of OSS projects that are commonly adopted as dependencies within Huawei.

OSSs from Reproducible Central. The Reproducible Builds [18] presents a collection of efforts on producing reproducible builds for C/C++-based systems. Recently, efforts towards reproducible builds for Java-based systems are also included [7]. Reproducible Central [5] is part of Reproducible Build efforts, which rebuilds open source Java-based systems and compare the deliverable packages with the stored ones in Maven Central. As of September 4, 2021, it contains 391 releases of 118 projects. Among them, 112 releases in 46 projects (the builds of 39% projects) cannot be reproduced or verified. For demonstration purposes, we only focus on the build verifiability of the most recent releases of these 46 projects which are not verified. Table 1 shows the basic information

Table 1: Evaluation results after applying our approach. BVP represents build verifiable projects.

Dataset	Projects	BVP		SLOC		Files	
		Before	After	Min	Max	Min	Max
RC	46	0 (0%)	42 (91%)	108	578,998	1	2,310
CA	13	0 (0%)	13 (100%)	4,710	700,668	51	7,540

of these projects. The sizes of these 46 projects range from 108 lines to 578,998 lines, and they contain from 1 to 2,310 files. Examples of these 46 projects include dubbo and kubernetes-client. For brevity, we call these projects as Reproducible Central (RC) projects.

To examine if our approach can produce verifiable builds for RC projects, we follow the projects' build specification and build commands in a fresh docker environment as described in Section 3. The main focus is to check if we can produce verifiable deliverable packages. Note that the deliverable packages stored in Maven Central were not built with our approach, hence many of the non-determinism have not been mitigated (e.g., not controlling the timestamp differences). To demonstrate the effectiveness of our approach, we have to compare the two deliverable packages built in our local environment, instead of comparing the deliverable packages against the ones in the central repository.

Commonly adopted OSSs within Huawei. Since the selected projects from Reproducible Central are generally of a smaller scale, to ensure generalizability, we have also selected the most recent releases of 13 open source projects which are widely adopted within Huawei. As shown in Table 1, the sizes of these commercially adopted projects range from 4,710 to 700,668 lines of code and have 51 to 7,540 source files. Examples of these 13 projects include Spring Framework and SLF4J. For brevity, we call these projects Commercially Adopted (CA) projects. In a similar setup as RC projects above, we also build the CA projects locally twice using the same build specification and build environments. Then, we verify the resulting deliverable packages.

4.1.2 Case Study Results. In this section, we report the evaluation results of 46 RC projects and 13 CA projects. Table 1 shows our evaluation results. For the build of 46 RC projects, before applying our approach, none of them are verifiable. After applying our approach, 42 (91%) of them are successfully verified. The build of four RC projects failed to be fully verified due to additional non-deterministic APIs from third party libraries.

These four projects leverage third-party libraries to generate Java source code files, XML files, and index files. For example, `org.apache.royale.compiler` from Apache Royale uses JFlex, a lexical analyzer generator which can generate Java programs based on specifications. A set of Java source code files is generated for later use in the build process. The auto-generated source code files have differences in the comments (e.g., randomly sorted documentation for parameters used in a method), which lead to the non-equivalences in the deliverable packages. For such types of non-equivalences, we plan first to identify and locate the non-determinism introducing methods. Then we will extend the current implementation of `JavaBEPEnv` to dynamically instrument and alter the existing behavior of these methods. In particular, for the non-determinism introducing methods, we intercept them by defining custom methods with the same method signatures through Java Instrumentation API, and implement custom program logic to avoid non-determinism.

For the 13 CA projects, before applying our approach, none of them achieve build verifiability. After applying our approach, the build of all 13 CA projects can be successfully verified. The evaluation results on both RC and CA projects demonstrate the effectiveness of our approach towards build verifiability for Java-based systems.

4.2 Our Mitigation Guidelines

This section describes various patterns of sources of non-determinism and the corresponding mitigation strategies. Table 2 presents the list of patterns. For each pattern, we include the root cause, the description, the mitigation strategy, the specificity, and one code example. There are a total of 14 patterns from five categories of root causes. Three types of strategies are applied to mitigate the sources of non-determinism: Control, Interpretation, and Remediation. Control includes using JavaBEPEnv to dynamically alter non-deterministic behaviors or ensuring the build environment is consistent. Interpretation involves using JavaBEPFix to post-process non-equivalent build artifacts. Remediation includes modifying source code, configurations, or upgrading JDK versions. In the remainder of this section, we describe each pattern in detail.

4.2.1 [RC1] Environment. The environmental factors refer to the build environment that should be documented in build specifications, including the types of host OS (e.g., Windows or Linux), and the dependent JDK versions. Without such documentation, many non-equivalences might be introduced to the build artifacts. Such non-determinism could usually be avoided if a docker or VM is provided for the build, except for timestamp-related non-determinism. Below we list the common patterns we find during our evaluation.

[P1] Timestamp. Build tools call JVM-level functions to retrieve the current timestamp. The timestamp is then written to build artifacts, causing non-equivalences across different build instances. In addition, generated files might contain time-related information, such as the creation time. Table 2 shows an example. During the build process of `wcm-caconfig-editor-1.8.0`, the timestamps are recorded in the `properties.xml` file, which causes the build artifacts to be non-equivalent.

Solution: Use JavaBEPEnv to replace the timestamp introducing function calls at JVM-level with customized functions. The customized functions return the pre-defined timestamp instead of real timestamp values, preventing non-deterministic information from being written into build artifacts. For the timestamps recorded in the file attributes, use JavaBEPFix to process all the files and assigning pre-defined timestamps to them.

[P2] JDK version. The JDK version used in the build process is written into `MANIFEST.MF`. As shown in Table 2, during the build process of dropwizard [22], the two build instances invokes two different JDK versions: 1.8.0_292 and 1.8.0_275, as the build specification does not record the exact JDK version while it simply notes `JDK1.8`. When independent builders try to verify the build,

Table 2: Our mitigation guideline

Root cause	Name	Description	Strategy	Java-specific	Example

[RC1] Environment	[P1] Timestamp	Time related information was written into files by build tools or embedded in the file properties.	Control or Interpretation	No	`META-INF/vault/properties.xml` (wcm-caconfig-editor-1.8.0)
					`- <entry key="created">2021-01-17T13:47:15.000Z</entry>`
					`+ <entry key="created">2021-01-17T13:46:49.000Z</entry>`

	[P2] JDK version	JDK versions written into `MANIFEST.MF`	Control	Yes	`META-INF/MANIFEST.MF` (io.dropwizard.metrics@metrics-servlets-4.1.22)
					`- Build-Jdk: 1.8.0_292`
					`+ Build-Jdk: 1.8.0_275`

	[P3] Git information	Git related information written into `git.json` and packaged in final artifact.	Control	No	`classes/git.json` (ladapchai-0.8.1)
					`- "git.local.branch.ahead": "0"`
					`+ "git.local.branch.ahead": "NO_REMOTE"`

	[P4] User information	Users who invoked the build process written into `MANIFEST.MF`.	Remediation or Control	Yes	`META-INF/MANIFEST.MF` (io.dropwizard.metrics@metrics-servlets-4.2.1)
					`- Build-By: runner`
					`+ Build-By: ?`

[RC2] JDK	[P5] LineNumberTable	Non-deterministic LineNumber Table generated by javac by default.	Remediation	Yes	`io/fabric8/maven/docker/HelpMojo.class` (docker-maven-plugin-0.36.1)
					`LineNumberTable:`
					`- Line 29:0`
					`+ Line 28:0`

	[P6] Constant Pool	Redundant/randomly ordered elements in Constant pool.	Interpretation	Yes	`io/fabric8/maven/docker/HelpMojo.class` (docker-maven-plugin-0.36.1)
					`- #12 = Methodref #160.#279 // java/io/InputStream.close:()V`
					`+ #91 = Methodref #88.#90 // java/io/InputStream.close:()V`

	[P7] Temporary variables	The temporary variables have different assigned IDs.	Interpretation	Yes	`ClassA.class` `ClassA.class` (Internal project)
					`- astore 15` `+ astore 13`
					`- aload 14` `+ aload 12`

	[P8] Javadoc	Javadoc entries randomly sorted due to JDK bug.	Control	Yes	(JDK-8013887/Internal project)
					`- com.sun.source.tree` `+ com.sun.source.doctree`
					`- com.sun.source.doctree` `+ com.sun.source.util`
					`- com.sun.source.util` `+ com.sun.source.tree`

	[P9] Inner class order	The order of inner classes is non-deterministic.	Interpretation	Yes	`InnerClasses:` (Internal project)
					`- public static #160= #518 of #517; //Foo=class A/B/C/...`
					`- public static #189= #520 of #519; //Bar=class A/B/C/...`
					`public static #162= #650 of #657; //Baz=class A/B/C/...`
					`+ public static #160= #518 of #517; //Foo=class A/B/C/...`
					`+ public static #189= #520 of #519; //Bar=class A/B/C/...`

	[P10] Method order	Methods in class files are randomly ordered.	Interpretation	Yes	(Kubernetes-client-project-5.4.1)
					`Io/fabric8/kubernetes/api/model/WatchEventFluent.class`
					`A withAuthInfoObject(final AuthInfo p0);`
					`- A withAPIServiceObject(final APIService p0);`
					`A withResourceRequirementObject(...);`
					`+ A withAPIServiceObject(final APIService p0);`

[RC3] Multi-thread	[P11] Entries in deliverable packages	Files packaged in archive files randomly sorted due to multithreading.	Interpretation	No	`- ... META-INF/MANIFEST.MF` (liquibase-percona-4.3.1.jar)
					`- ... META-INF/services/`
					`+ ... META-INF/MANIFEST.MF`
					`+ ... liquibase/`

[RC4] Other tools	[P12] Properties in files	Properties in `MANIFEST.MF` are randomly ordered.	Interpretation	Yes	(io.dropwizard.metrics:metrics-4.2.1)
					`- Export-Package:com.codahale.metrics.health;uses:="com.codahale.metrics";version="4.2.1" (...) com.codahale.metrics.health.annotation;version="4.2.1"`
					`+ Export-Package:com.codahale.metrics.health.annotation;version="4.2.1",com.codahale.metrics.health;uses:="com.codahale.metrics";version="4.2.1" (...)`

	[P13] JSP compilation	Different source code generated by Jasper due to cache option.	Control	Yes	`JasperGeneratedFile.java` (Internal project)
					`+ static { _jspx_dependants = new`
					`+ java.util.HashMap<java.lang.String,java.lang.Long>(2);`
					`+ _jspx_dependants.put("dep1.jar",Long.valueOf(1685L));`
					`...}`

[RC5] Compound effect	[P14] Lambda expression	Auto-generated methods for lambda expression during compilation are not consistent.	Control	Yes	`- #25 = Methodref #4.#30 // L1.lambda$new$0:()V` (Internal project)
					`+ #25 = Methodref #4.#30 // L1.lambda$new$1:()V`
					`- #25 = Methodref #4.#30 // L2.lambda$new$1:()V`
					`+ #25 = Methodref #4.#30 // L2.lambda$new$0:()V`

a different JDK version is recorded in the `MANIFEST.MF`, causing non-equivalences in the build artifact.
Solution: Make sure the same JDK version is used during the build processes. The adopted JDK version should be documented in the build specification for future references as well.
[P3] Git information. Many software projects use Git as the VCS to manage the evolution and the maintenance of the projects. Some build processes record Git related information (e.g., the `commit ID` or the user who started the process) in the build artifacts. Such information might be non-deterministic if the build processes are started in different environments. As shown in Table 2, one of the build environment for `ladapchai-0.8.1.jar` does not configure the remote repository [20], causing the field `git.local.branch.ahead` to be non-equivalent.
Solution: Make sure the same Git setup is used during the build processes. The Git setup information should be documented in the build specification for future references as well. Recommend to use the same build environment (e.g., VMs or containers) to ensure the consistency of the build environment.
[P4] User information. The `user ID` of the user who invokes the build process can be written into `MANIFEST.MF` file. As shown in Table 2, the user information is recorded in the `Build-By` field in `MANIFEST.MF` during the build process for `metrics-servlets-4.2.1`. As two build processes can be conducted in different environments, the user information can also be different.
Solution: Modifying the build configuration can mitigate this issue. A configuration field `Built-by` under `manifestEntries` can be set with a consistent name to avoid non-deterministic `user IDs`. Alternatively, build within consistent environments (similar to [P2] and [P3]) can also mitigate this source of non-determinism.

4.2.2 [RC2] JDK. Some non-deterministic behavior is caused by JDK during the build process. Below we list the common patterns that we find during our evaluation.
[P5] LineNumberTable. `LineNumberTable` is an optional attribute that represents the relation between source code and bytecode. It can vary during the compiling phase. Table 2 shows an example. During the build process for `docker-maven-plugin` [24], the values `LineNumberTable` are different in the `HelpMojo.class` file.
Solution: Modify the build configuration to mitigate this issue.For example, a configuration parameter `-g:none` can be added with `javac` to prevent `LineNumberTable` from being written into the bytecode. If the build process is started by Maven, we can also disable the generation of `LineNumberTable` by adding such configuration in the `pom.xml`.
[P6] Constant pool. The constant pool is a data structure inside `class` files. It records the symbolic references that JVM uses to link with the actual contents of variables, methods, interfaces, etc. We find that the constant pool might contain duplicated elements. The indices of the duplicated elements are used in a non-deterministic way when these elements are referenced. Furthermore, the constant pool might be randomly ordered accross two build instances, causing the indices to be different. Table 2 shows an example of this pattern. During two build instances of `docker-maven-plugin` [19], the indices of the reference to the `close` method are recorded as 12 and 91, respectively.
Solution: Use JavaBEPFix to mitigate this issue. In particular, it will post-process the `class` files by deduplicating the constant pool and then sort it in a deterministic order.
[P7] Temporary variables. Temporary variables are variables with a short lifetime. For example, a return statement `return (a+b);` would create a temporary variable when compiled to bytecode. Such variables will be assigned with a temporary `ID` by the compiler, which is used for instructions such as `astore` and `aload`. We find the same build process could yield different `IDs` for the temporary variables. The example shown in Table 2 is adapted from our internal project. At the same locations of a class file, the `IDs` of the temporary variables are different in two build instances.
Solution: Use JavaBEPFix to mitigate this issue. In particular, it will automatically post-process the `class` files to reassign temporary variables with deterministic `IDs`.
[P8] JavaDoc. Lower versions of JDK can cause entries in the JavaDoc being randomly sorted [21]. The example shown in Table 2 is adapted from the issue report `JDK-8013887` [21], where the three JavaDoc entries have different orders during two identical build processes. We found such non-determinism exists in Huawei's internal projects.
Solution: Upgrade the JDK version to be higher than or equal to `1.8_b105` to mitigate this issue.
[P9] Inner class order. Inner classes are classes that are defined inside another class. When the source code of the class that contains inner classes is compiled to bytecode, the list of inner classes is listed in the bytecode. As shown in Table 2, three inner classes, `Foo`, `Bar`, and `Baz` are listed. However, the order of the inner classes in the list is non-deterministic.
Solution: Use JavaBEPFix to mitigate this issue. In particular, it will automatically post-process the `class` files by sorting the inner classes in a deterministic way.
[P10] Method order. The order of the compiled methods in the `class` files might be non-deterministic. Table 2 shows an example. Across two build instances of `kubernetes-client-project-5.4.1`, file `WatchEventFluent.class` is not equivalent. We find that the only difference is the order of the methods inside the class (e.g., the method `withAPIServiceObject` appears before or after the method `withResourceRequiremetnObject`).
Solution: Use JavaBEPFix to mitigate this issue. In particular, it will automatically post-process the `class` files by sorting the methods in a deterministic way.

4.2.3 [RC3] Multi-thread. Multi-threaded compilation is widely adopted by modern build tools, as it can accelerate the build process. However, build artifacts might be generated in a non-deterministic order, causing non-equivalences in deliverable packages.
[P11] Entries in deliverable packages. Each deliverable package contains a list of build artifacts, which are compiled and packaged in a multi-threaded manner. The sequence of these build artifacts in the deliverable package is not deterministic, as it depends on which thread execution finishes first. As a consequence, although the contents of each build artifact are equivalent during two build instances, but the deliverable package as a whole is not. As shown in Table 2, during the build processes of `liquibase-percona-4.3.1.jar`, the entries are not ordered the same, causing the resulting deliverable package to be different.

Solution: Use JavaBEPFix to mitigate this issue. In particular, after the original deliverable package is generated, it first unpacks the deliverable package and then re-packages the build artifacts in a deterministic way (e.g., by name).

4.2.4 [RC4] Other tools. Build processes of some Java-based systems depend on third-party tools or plugins. These tools and plugins might introduce non-determinism into build artifacts, causing non-equivalences in the deliverable packages.

[P12] Properties in files. Some properties in `MANIFEST.MF` files might be randomly ordered. For example, the `Export-Package` property records the packages that are visible outside the deliverable package. In the build of `dropwizrd-metrics-4.2.1.jar`, the list of `Export-Package` does not have the consistent sequence. Similar issue is identified with the `Private-Package` attribute. This is caused by a third-party build plugin tool.

Solution: Use JavaBEPFix to mitigate this issue. In particular, it will automatically locate the non-deterministic file properties (e.g., `Private-Package` and `Export-Package`) in `MANIFEST.MF` file and sort the relevant properties.

[P13] JSP compilation. `JSP` is a Java-based technique to create dynamically generated webpages. The `JSP` files can be parsed to Java source code files so that they share all the APIs and functionalities provided by `JVM`. To do that, techniques such as Tomcat Jasper engine are applied. A source of non-determinism is identified during this process, as the caching option in the Jasper engine cause the generated source code to have non-equivalent static variable definitions as shown in Table 2.

Solution: Use JavaBEPEnv can mitigate this issue. It sets the caching option to false and prevents the inconsistent information being generated through dynamic instrumentation.

4.2.5 [RC5] Compound effects. Some non-equivalences are due to a combination of root causes such as multi-thread and JDK behavior.

[P14] Lambda expressions. This pattern occurs when there are multiple lambda expressions in the source code. When compiling with lower versions of JDK in a multi-threaded setting, the `ID` of the same lambda expressions may be assigned in a different way [23]. As shown in Table 2, the `ID`s of the lambda expressions of the two files, `L1.java` and `L2.java`, are different when the two files are compiled in a different order.

Solution: Upgrade JDK to a version that is newer than `jdk8-b44`.

5 DISCUSSION

In this section, we discuss how our approach can help on tackling related industrial challenges to build verifiability of Java-based systems and present some future research directions.

Towards trustworthy software supply chains. Software supply chain attacks explore the dependency relationships between different software components and target software systems. One form of supply chain attack is the injection of malicious code during the build process [29], particularly by hijacking third-party libraries distributed through central repositories and linked to the built package. Verifiable build plays an important role in preventing this type of software supply chain attack. In particular, systems with a verifiable build can have their integrity jointly verified by independent builders that share checksums of the generated build artifacts, such that others can compare against the build artifacts that they produce themselves. For example, Lamb and Zacchiroli [26] discuss how reproducible builds can help OSS users in establishing trust in distributed packages through package managers. Compared to deliverable packages built locally, those in the remote central repositories usually suffer from more types of non-equivalences due to inconsistent build environment, lack of automated techniques, and other related factors. Our approach proposes an important step to support a trustworthy software supply chain, as it helps developers to deploy verifiable build artifacts in central repositories.

Comparison across different OS platforms. Deliverable packages should be verifiable even when built across different OS platforms, as one major advantage of `Java` is the compatibility across different OSs. We have conducted additional experiments on 13 CA projects on both `Linux` and `Windows` platforms. Our results show that the deliverable packages are verifiable when built in `Linux` and `Windows` separately by applying our approach. However, when comparing the deliverable packages between `Linux` and `Windows`, none of the build packages are equivalent. This is due to the environmental differences between the two OS platforms. Take the build processes for `Logback` as an example. The two OS platforms can have different users, which triggers the `[P4]` pattern. In addition, even using the same JDK versions, there are differences in the class files while building in `Linux` and `Windows`. Although one of the main advantages of Java is that software systems can be "built once and run anywhere", the verifiability of the build across different platforms is still not satisfied. This issue can be currently resolved by specifying the OS platforms as part of the build specifications or using pre-setup VMs. Such requirement is in accordance with the definition of build verifiability [18]. However, mitigating the sources of non-determinism from different platforms remains to be an interesting piece of future work.

Comparison among systems implemented in different programming languages. Due to the different setups of interpreted vs. compiled programming languages, the mitigation strategies also differ in the following two aspects: (a)*Different approaches for same mitigation strategies:* Various prior work has been done towards build verifiability for C/C++-based systems [30–32]. The strategy of control in our prior work [32] is similar to JavaBEPEnv proposed in this paper. This mechanism intercepts non-determinism introducing functions (e.g., functions that return timestamp) and returns pre-defined values. However, the approach in [32] cannot be directly used in Java-based systems. The control strategy of our prior work focuses on process level, where the system level functions are intercepted. However, in the build process of Java-based software systems, JVM is the only process that is instantiated. Furthermore, simply intercepting the JVM process will likely cause the congestion in the build process, as many multi-threaded operations in JVM is time-sensitive. (b) *Java-specific patterns*: As shown in Table 2, there are three common patterns associated with non-equivalent build artifacts between Java- and C/C++-based systems. However, there are also eleven patterns that are unique for Java-based systems. For example, six patterns that are caused by JDK behavior are specific to Java-based systems. Other software systems that are built on top of JVM (e.g., Kotlin and Scala) may benefit from our approach and future work should investigate the build verifiability for systems written in JVM-supported programming languages.

6 THREATS TO VALIDITY

In this section, we present the threats to validity.

External Validity. We try to be as comprehensive as we can in our case study by selecting 46 projects from Reproducible Central, and 13 projects from commercially adopted OSS systems. However, our approach and our mitigation strategies towards build verifiability may not cover all the build verifiability scenarios for Java-based systems. In addition, our approach and findings are only limited to Java-based systems and may not be applicable to software systems written in other programming languages.

Internal Validity. To avoid confounding factors, we ensure the build environment is consistent before we perform our experiments. Although Java is a programming language that is OS independent, we still only perform our experiments on the same OS platforms (either Linux or Windows) depending on the usage scenarios. This setup is in accordance with the definition of build verifiability [18].

Construct Validity. To check if the build is verifiable, we track all the deliverable packages generated by the build process. For example, a build process generates 100 deliverable packages (e.g., 100 `jar` files). If there is at least one `jar` file which is not verifiable, we do not consider the build verifiable. Our approach is similar to the prior work in this area [30–32].

7 CONCLUSIONS

Build verifiability is essential for software security and trustworthiness. While various prior work has been done to ensure verifiable build for C/C++ systems, there is a lack of systematic solution for Java-based systems. In this paper, we propose a systematic approach towards build verifiability in Java-based systems. Our approach includes a unified process and two main techniques: a tool JavaBEPEnv, which controls non-deterministic behavior from the build tools, and another tool JavaBEPFix, which interprets non-determinism by post-processing non-equivalences in build artifacts. Case studies show that among 59 OSSs which are not build verifiable, 55 (93%) projects are now build verifiable by applying our approach. We also present a mitigation guideline, which includes all the sources of non-determinism we encountered and the corresponding mitigation strategies. Last, we discuss some challenges and provide some open research problems.

REFERENCES

[1] 2017 (accessed September, 2021). *Popular approaches to preventing code injection attacks are dangerously wrong.* https://infocondb.org/con/owasp/appsec-usa-2017/popular-approaches-to-preventing-code-injection-attacks-are-dangerously-wrong

[2] 2021 (accessed August, 2021). *Preventing Supply Chain Attacks like SolarWinds.* https://www.linuxfoundation.org/blog/preventing-supply-chain-attacks-like-solarwinds

[3] 2021 (accessed August, 2021). *Sonatype Stops Software Supply Chain Attack Aimed at the Java Developer Community.* https://blog.sonatype.com/malware-removed-from-maven-central

[4] 2021 (accessed July, 2021). *Configuring for Reproducible Builds - Maven.* https://maven.apache.org/guides/mini/guide-reproducible-builds.html

[5] 2021 (accessed July, 2021). *Rebuilding artifacts from (Maven) Central Repository.* https://github.com/jvm-repo-rebuild/reproducible-central

[6] 2021 (accessed July, 2021). *Reproducible Build Maven Plugin.* https://github.com/Zlika/reproducible-build-maven-plugin

[7] 2021 (accessed July, 2021). *Reproducible Builds - JVM.* https://reproducible-builds.org/docs/jvm/

[8] 2021 (accessed July, 2021). *Reproducible builds in Gradle.* https://docs.gradle.org/current/userguide/working_with_files.html#sec:reproducible_archives

[9] 2021 (accessed July, 2021). *Reproducible Builds in Maven and Gradle.* https://github.com/jfrog/reproducible-build

[10] 2021 (accessed October, 2021). *CipherKit reproducible builds.* https://guardianproject.info/2015/09/21/cipherkit-reproducible-builds/

[11] 2021 (accessed October, 2021). *Huawei Cyber Security Evaluation Centre (HCSEC) Oversight Board – Annual Report.* https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/1004291/2021_HCSEC_OB_REPORT_FINAL__1_.pdf

[12] 2021 (accessed October, 2021). *javadoc - The Java API Documentation Generator.* https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

[13] 2021 (accessed October, 2021). *Package java.lang.instrument.* https://docs.oracle.com/en/java/javase/11/docs/api/java.instrument/java/lang/instrument/package-summary.html

[14] 2021 (accessed October, 2021). *Reproducible/Verifiable Builds - Maven.* https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=74682318

[15] 2021 (accessed October, 2021). *TIOBE Index for October 2021.* https://www.tiobe.com/tiobe-index/

[16] 2021 (accessed October, 2021). *WebLogic JSP Reference.* https://docs.oracle.com/cd/E13222_01/wls/docs81/jsp/reference.html

[17] 2021 (accessed September, 2021). *Reproducible Build Maven Plugin.* http://zlika.github.io/reproducible-build-maven-plugin/

[18] 2021 (accessed September, 2021). *Reproducible Builds.* https://reproducible-builds.org

[19] 2021 (accessed September, 2021). *Sources of non-determinism: Constant pool table.* https://github.com/jvm-repo-rebuild/reproducible-central/blob/master/content/io/fabric8/docker-maven-plugin/docker-maven-plugin-0.36.0.diffoscope

[20] 2021 (accessed September, 2021). *Sources of non-determinism: Git related information.* https://github.com/jvm-repo-rebuild/reproducible-central/blob/master/content/com/github/ldapchai/ldapchai-0.8.1.diffoscope

[21] 2021 (accessed September, 2021). *Sources of non-determinism: JavaDoc.* https://bugs.openjdk.java.net/browse/JDK-8013887

[22] 2021 (accessed September, 2021). *Sources of non-determinism: JDK version.* https://github.com/jvm-repo-rebuild/reproducible-central/blob/master/content/io/dropwizard/metrics/metrics-servlets-4.1.22.diffoscope

[23] 2021 (accessed September, 2021). *Sources of non-determinism: Lambda.* https://bugs.openjdk.java.net/browse/JDK-8067422

[24] 2021 (accessed September, 2021). *Sources of non-determinism: LineNumberTable.* https://github.com/jvm-repo-rebuild/reproducible-central/blob/master/content/io/fabric8/docker-maven-plugin/docker-maven-plugin-0.36.1.diffoscope

[25] Xavier de Carné de Carnavalet and Mohammad Mannan. 2014. Challenges and Implications of Verifiable Builds for Security-Critical Open-Source Software. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. 16–25.

[26] Chris Lamb and Stefano Zacchiroli. 2021. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* (2021), 0–0. https://doi.org/10.1109/MS.2021.3073045

[27] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 167–182.

[28] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[29] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 23–43.

[30] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated Localization for Unreproducible Builds. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*.

[31] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie. 2019. Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE'19)*.

[32] Young Shi, Mingzhi Wen, Filipe Roseiro Cogo, Boyuan Chen, and Zhen Ming (Jack) Jiang. 2021. An Experience Report on Producing Verifiable Builds for Large-Scale Commercial Systems. *IEEE Transactions on Software Engineering* (2021).

[33] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* (1984).

[34] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards Using Source Code Repositories to Identify Software Supply Chain Attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*. 2093–2095.