Canonicalization for Unreproducible Builds in Java

Aman Sharma
KTH Royal Institute of Technology
Stockholm, Sweden
amansha@kth.se

Benoit Baudry
Université de Montréal
Montréal, Canada
benoit.baudry@umontreal.ca

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
monperrus@kth.se

ABSTRACT

The increasing complexity of software supply chains and the rise of supply chain attacks have elevated concerns around software integrity. Users and stakeholders face significant challenges in validating that a given software artifact corresponds to its declared source. Reproducible Builds address this challenge by ensuring that independently performed builds from identical source code produce identical binaries. However, achieving reproducibility at scale remains difficult, especially in Java, due to a range of non-deterministic factors and caveats in the build process. In this work, we focus on reproducibility in Java-based software, archetypal of enterprise applications. We introduce a conceptual framework for reproducible builds, we analyze a large dataset from Reproducible Central, and we develop a novel taxonomy of six root causes of unreproducibility. We study actionable mitigations: artifact and bytecode canonicalization using OSS-Rebuild and jNorm respectively. Finally, we present Chains-Rebuild, a tool that raises reproducibility success from 9.48% to 26.89% on 12,283 unreproducible artifacts. To sum up, our contributions are the first large-scale taxonomy of build unreproducibility causes in Java, a publicly available dataset of unreproducible builds, and Chains-Rebuild, a canonicalization tool for mitigating unreproducible builds in Java.

KEYWORDS

Reproducible Builds, Software Supply Chain, Canonicalization, Java

1 INTRODUCTION

The growing complexity of software supply chains [9, 54], coupled with the increasing frequency of supply chain attacks [1], raises concerns about software integrity. In such a fragmented ecosystem, relying solely on assumptions about the identity of the distributor [56] is no longer sufficient - what is needed is verifiable evidence that the software one installs corresponds exactly to its declared source. This challenge is especially acute in open source environments, where binaries are often distributed separately from their source code [11], making it difficult for users to independently validate what they are running. As a result, the focus is shifting toward techniques that can guarantee that what is built matches what was intended, regardless of who performs the build.

This technique is formally called "Reproducible Build" [10, 59]: builds are considered "reproducible" if and only if the build process is deterministic, where the same binary can be computed from the same source code by independent parties [24]. It helps prevent attacks on the build process, where an attacker modifies it to insert backdoors or malicious code into the software application to be built [39]. Any backdoor or malicious code would be detected by the verifier as the built artifact would not match the original artifact.

This approach has already seen adoption in security-critical environments, such as in certain federal government contracts, where vendors are required to supply source code so the agencies can perform their own builds and verify the integrity of the software [2].

Despite the promises of Reproducible Builds, ensuring and verifying reproducibility at scale remains technically challenging due to the multitude of spurious differences in build outputs (e.g., timestamps, file ordering). Those differences make binaries appear different even when built from the same source, with an untampered build pipeline [17]. Spurious differences hinder reproducibility and make it difficult to justify the differences between two builds. XZ Utils is a widely used compression library that is used in almost all Linux distributions. In 2024, it is found that the XZ Utils has a backdoor to give remote code execution to the attacker [42]. However, this backdoor is only observed in the official tarballs on the registry and do not exist in the git repository. Such supply chains attacks can be prevented if build process is reproducible. In the case of XZ Utils, if the tarball on the package registry does not match the one built from the source code, then the user can be sure that the tarball is tampered with [30].

In this work, we contribute to solving the problem of achieving build reproducibility in the context of enterprise software in Java. Recall that Java has consistently been one of the top 5 programming languages in the world for the past 5 years [3], underscoring its widespread use and relevance. Its importance in the context of finance [34, 54], government services [49], and military applications [4] [5] highlights the need for reproducible builds in Java. We aim at building the most comprehensive taxonomy of unreproducible builds in Java, and the corresponding mitigation strategies. Furthermore, we perform a deep study how canonicalization - removal of non-deterministic differences - is a good solution for mitigating unreproducibility.

Our approach for analyzing unreproducible builds in Java is shown in Figure 1. We first propose an original framework for build reproducibility where we clearly define the roles of the builder and rebuilder and how both of them contribute to reproducible build verification. Next, we build a dataset of unreproducible builds in Java by leveraging Reproducible Central [4], the leading project for rebuilding and verifying Maven applications. We systematically analyze the dataset to identify and classify the causes of unreproducibility into an original taxonomy of unreproducibility. Finally, we leverage the dataset to evaluate the effectiveness of bytecode and artifact canonicalization. Bytecode canonicalization focuses on internal representation of program logic and eliminates compiler introduced variations. While artifact canonicalization eliminates

---

[1]https://www.sonatype.com/hubfs/SSCR-2024/SSCR_2024-FINAL-10-10-24.pdf

[2]https://news.ycombinator.com/item?id=43492115
[3]https://innovationgraph.github.com/global-metrics/programming-languages
[4]http://pdf.cloud.opensystemsmedia.com/vita-technologies/Aonix.Jun07.pdf
[5]https://tsri.com/news-blog/press/u-s-department-of-defense-mainframe-cobol-to-java-on-aws
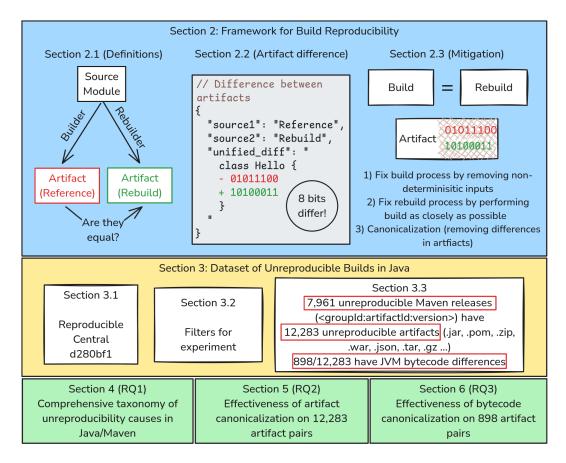
Figure 1: The problem space of unreproducible builds in Java, and our novel mitigation strategies.

spurious variations in metadata that can influence how the artifact is interpreted or processed by downstream build tools. Together, these two levels of canonicalization address both semantic and non-semantic sources of non-determinism, significantly improving overall build reproducibility.

Our study reveals 6 original categories of unreproducibility causes in Java, each with the underlying root causes and the best practice mitigations. For example, changes in the bytecode in Java artifacts. Our large-scale experiments empirically measure the effectiveness of two state-of-the-art canonicalization techniques. First, we show that bytecode canonicalization with JNorm [50] can successfully canonicalize 267 out of 898 unreproducible artifacts with JVM bytecode. We study the reasons why JNorm fails to canonicalize the remaining artifacts, which are promising areas of future work. Next, we evaluate artifact canonicalization using Google's OSS-Rebuild and find that it can successfully canonicalize 9.48% (1,165 / 12,283) of unreproducible artifacts. By implementing new canonicalization patterns informed by our taxonomy, our new tool Chains-Rebuild increases the verification success rate to 26.89% (3,303 / 12,283). Our paper is the first to demonstrate, at a large scale, that canonicalization is a promising mitigation strategy for fixing build unreproducibility in the Java ecosystem.

There has been prior work on reproducible builds in Java focusing on bytecode canonicalization [14] and fixing build process [60]. However, we are the first to analyze Maven projects hosted on Maven Central at scale and to propose a comprehensive taxonomy of unreproducibility causes and their mitigations. In addition to bytecode canonicalization, we also study artifact canonicalization and show that it is a promising solution to mitigate unreproducibility in Java.

To summarize, our contributions are:

- A sound conceptual framework of build reproducibility, founded on the novel roles of the "builder" and "rebuilder" and on the powerful concept of artifact canonicalization.
- A novel, comprehensive taxonomy of 6 unreproducibility causes and their mitigation in Java, helping practitioners to improve build reproducibility in their projects.
- A large-scale experiment on the effectiveness of canonicalization, showing that it is a promising solution to mitigate unreproducibility in Java. Artifact canonicalization is more effective at scale than bytecode canonicalization.
- A publicly available, curated dataset of 7,961 unreproducible Maven releases for future research.

2 FRAMEWORK FOR BUILD REPRODUCIBILITY

2.1 Definitions

Developers often release reusable software in package registries, making them widely available for integration into third-party software projects. For our work on build reproducibility, we need to introduce key terminology. We work in the context of the Java programming language and its Maven build system.

Package Registry: A package registry is a centralized repository of software packages that can be reused by other applications. In this paper Maven Central, hosted by a company called Sonatype, is the considered package registry.

Module: A module is a uniquely addressable component in a software system. In Maven, a module is referenced by a combination of Group Identifier, Artifact Identifier, and Version. Group ID is a unique identifier for the organization, eg. "org.apache" [6]. Artifact ID is the unique name of the module and is unique within the group ID. For example, `com.google.guava:guava:32.0.0-jre` is a Maven module where `com.google.guava` is the group ID, `guava` is the artifact ID, and `32.0.0-jre` is the version. There are two types of modules: a **source module** is a module that is hosted on a version control system (eg. Github), not necessarily published on a package registry; a **released module** is one published on a package registry, Maven Central in this paper. Both released and source modules can be identified by the combination of group ID, artifact ID, and version to refer to both types of module. In the latter case, we assume the presence of a mapping from version to Git tag or Git commit.

Project: A Maven project is a collection of source modules in the same source repository. It contains build configuration for all the modules in a so-called 'POM file'. For example, Maven project guava contains 6 source modules - `guava-parent`, `guava-tests`, `guava-testlib`, `guava-gwt`, `guava-bom`, and `guava`.

Artifact: An artifact is any file that is part of the released module. In Maven, there are mostly a JAR or a ZIP archive. For example, in released `com.google.guava:guava:32.0.0-jre`, `guava-sources.jar` and `guava.jar` are JAR file in the released module. In this paper, we have two types of artifacts: The **reference artifact** is the artifact of the released module that is available on the package registry; The **rebuild artifact** is the artifact obtained by rebuilding the source module.

Builder: The builder is the entity that publishes a released module on a package registry, from the source module. A builder is often an automated script implemented in a CI/CD pipeline, like GitHub Actions. For example, the released module `com.google.guava:guava:32.0.0-jre` has been deployed on Maven Central by the Google builder.

Rebuilder: The rebuilder is the entity that verifies the build reproducibility of the released module, with information available from the package registry. In other words, the rebuilder 1) builds the projects again 2) verifies the reproducibility of all the rebuild artifacts against the artifacts in the released module. For example, the Reproducible Central project [4] is a rebuilder that verifies the

reproducibility of the released modules of 700+ open-source Java libraries.

Canonicalization Canonicalization is the process of converting different representations of the same data into a single, standard representation. For example, in the domain of XML parsing [5], canonicalizing means encoding the XML document in a single charset (UTF-8), adding default attributes, and many more changes as listed in the specification [5].

Canonicalization is useful in the context of build reproducibility as it allows to convert the output artifacts of the build process into a standard representation that is independent of the machine and environment where the build has been performed. Assume that the output artifact contains details of the operating system or the user who built the artifact. These details are not relevant to the functionality of the software and can be removed as part of canonicalization or the artifact.

2.2 Build Reproducibility

Build Reproducibility is a property of a software build process where the output artifact is bit-by-bit identical when built again, given a fixed version of source code and build dependencies, regardless of the environment [24]. This is shown in the equation below, where source module is built twice by builder and rebuilder to produce reference and rebuild artifacts, respectively. The build is reproducible if the reference and rebuild artifacts are identical.

Reproducible builds help prevent software supply chain attacks at the build time by ensuring that the software is not tampered with during the build process. It also allows users to verify that the executable provided to them is free of backdoors as they can build the software themselves and compare the output with the provided executable.

$$ReferenceArtifact \leftarrow Build(SourceModule) \quad (1)$$

$$RebuildArtifact \leftarrow Rebuild(SourceModule) \quad (2)$$

$$ReferenceArtifact = RebuildArtifact \quad (3)$$

There are 2 other terms in the literature that are used interchangeably with build reproducibility - "verifiable builds" and "accountable builds" [40]. Builds are verifiable if the build process can be modified in order to make the output artifact bit-by-bit identical. Accountable builds are the same as verifiable builds, but instead of removing the differences, these differences are explained and documented so that unreproducibility can be understood upon inspection. In our paper, we stick with the term "build reproducibility" as the other two terms are extensions of build reproducibility with extra steps.

2.3 Mitigation

We claim that any cause of unreproducibility can be mitigated in three ways - fixing the build process, fixing the rebuild process, or canonicalizing the output of the rebuild process.

Fixing the build process and **Fixing the rebuild process** involve modifying inputs such as source code, dependencies, or build tools (e.g., build scripts and configurations) to the build or rebuild workflows in order to eliminate sources of non-determinism. The builder ensures that the build process is deterministic and easy

---

to reproduce for rebuilder. The rebuilder tries to perform the build process as closely as possible to the original build process. However, some build tools generate non-deterministic information or there is a difference in the environment which makes these mitigation techniques not always feasible neither practical.

Canonicalizing the output of the rebuild process as defined above means removing non-deterministic information from output artifacts. Some non-deterministic information is spurious, but it is sometimes perfectly legitimate such as a cryptographic signature with one-time key [12]. Canonicalization is generally not done by the builder for this reason and it is more the responsibility of the rebuilder. It requires careful removal of all non-deterministic information from the output artifact. If done incorrectly, it may inadvertently remove or obscure meaningful differences in the output artifacts. This can lead to false positives, where builds that are genuinely unreproducible are incorrectly reported as reproducible. Such errors would undermine the reliability of reproducibility verification and can mask underlying issues that need to be addressed. Therefore, it is crucial to ensure that canonicalization is performed with precision, preserving all relevant differences while eliminating only the spurious, non-deterministic variations. Canonicalization can address the unreproducibility issues of past releases by removing non-deterministic information from its output artifacts. It can also help as last resort to fix unreproducibility issues when build process and rebuild process are embedding non-deterministic information which is not in control (like cryptographic signatures). Moreover, techniques related to fixing build process have also been studied in prior literature [22, 28, 36, 44, 47, 60]. Although we suggest these techniques as mitigation strategies, we focus deeply on the third mitigation strategy (canonicalization) due to the fact that it is not well evaluated in the literature.

3 DATASET OF UNREPRODUCIBLE BUILDS

3.1 The Reproducible Central Project

The Reproducible Central project [4] is a rebuild infrastructure to verify whether Maven projects are reproducible. It specifies list of steps to build a Java project from source in a `buildspec` file [7]. Then, the Reproducible Central script takes this `buildspec` as input to rebuild a project. It produces `buildinfo` and `buildcompare` files; `buildinfo` records the output of the build in terms of the artifact names, sizes, and checksums [6]; `buildcompare` reports which artifacts are reproducible and which are unreproducible.

We use the Reproducible Central project dataset as a starting point to study the causes of unreproducible builds in Java. We take a snapshot of the Reproducible Central dataset on 8th October, 2024 whose commit is d280bf1. At this date, Reproducible Central contains 706 Maven projects, their corresponding build scripts, and a history of 4,956 releases. It is the largest dataset attempting rebuilds of Java projects and is also maintained by maintainers of Apache Maven.

Our goal is to curate a dataset of unreproducible released Maven modules in order to study the reasons why unreproducible builds occur. In order to curate the dataset of unreproducible released

modules, we first apply 2 filters before running the rebuilds and 1 filter afterwards.

3.2 Filtering

3.2.1 Maven Projects. First, we filter out Maven projects that require manual intervention to build so that we can run the entire rebuild process automatically. This is indicated in `buildspec` file either by the presence of build tool or when the rebuild command is prefixed with 'SHELL'. The rebuild script would spawn an interactive terminal when any of the above conditions are met. We exclude projects that do not have a POM file at the root directory as it also requires manual intervention to get the path to rebuild artifacts in build directory. Second, we only include projects that use Maven as their build tool (some projects use Gradle or SBT). This is because Maven is the most mature build tool for Java projects, most widely used in the Java ecosystem [55]. Moreover, Maven builds success rate is higher than Gradle and Ant [29]. Once all rebuilds are done, we exclude projects where the unreproducible artifacts are not a subset of the all artifacts released on Maven Central. This can happen when not all artifacts produced by the build process are released on Maven Central. For example, `content/org.infinispan: protostream-aggregator:5.0.7.Final` produces an artifact `rotostream-integrationtests-5.0.7.Final-test-sources.jar` which is not released on Maven Central. After applying the filters, we end up with 3,927 releases over 665 Maven projects.

3.2.2 Released Modules. Next, we need to identify the released modules to compare against. In order to map each unreproducible project to corresponding released modules, we first identify the Maven modules in each project. `org.apache.aries.cdi:org.ap ache.aries.cdi.executable:1.1.5` is an example of a source module that does not have a corresponding URL on Maven Central. We parse the POM file located at the root directory of the project as it contains the required information about the modules. For example, jpmml-python is a Maven project hosted on GitHub [8] which has 3 Maven modules with version `1.2.2` - `org.jpmml:jp mml-python`, `org.jpmml:jpmml-python:1.2.2`, and `org.jpmml: pmml-python`. Thus, jpmml-python Maven project on Reproducible Central corresponds to 3 Maven modules in our dataset. 2 out of 3 released modules in this project contain unreproducible artifacts. `org.jpmml:pmml-python-testing:1.2.2` has `pmml-python-testing-1.2.2-sources.jar` and `org.jpmml:pmml-python: 1.2.2` has `pmml-python-1.2.2-sources.jar`. Finally, there are 4 cases where unreproducible reference and rebuild artifact are not mapped correctly. Maven module `org.apache.ratis:ratis-assembly:3.1.0` and 3 more of its versions have an artifact with `.pom` extension. The rebuild script maps this artifact to the Jar file produced during rebuild. This is a bug in the Reproducible Central rebuild script and have reported this to the author [9].

At the end of the pipeline, we end up with 7,961 Maven modules which have a total of 34,271 artifacts. Out of 34,271 artifacts, there are 12,283 unreproducible artifacts (35.8%).

3.3 Description of the Dataset

The dataset contains groupId, artifactId, version, and the URL of all 7,961 unreproducible released modules on Maven Central. It also contains the reference and rebuild artifact pair of all 12,283 unreproducible artifacts. It is available at https://github.com/chains-project/reproducible-central/. To our knowledge, this is the largest ever curated dataset of unreproducible artifacts in the context of Java. Based on the dataset, we can answer the following research questions:

- RQ1 (Causes): What are the causes of unreproducible builds in Java?
- RQ2 (Artifact Canonicalization): To what extent does artifact canonicalization using OSS-Rebuild make a Maven release reproducible?
- RQ3 (Bytecode canonicalization): To what extent does bytecode canonicalization with jNorm mitigate unreproducible builds in Java?

4 RQ1: TAXONOMY OF UNREPRODUCIBILITY CAUSES

4.1 Objective

The goal of this section is to provide a comprehensive taxonomy of unreproducibility causes in Java. We first report the cause based on our analysis and then propose a mitigation strategy for each cause.

4.2 Methodology

We have a dataset of 12,283 unreproducible artifacts and for each of them we have the reference artifact from Maven Central and its rebuilt version from our own rebuild. We run diffoscope 285 on each pair of reference and rebuild artifacts to get the content diff. DIFFOSCOPE is a state-of-the-art tool for computing the difference between various types of files, archives, and directories [10]. This gives us a total of 12,283 diffoscope files. The median size of diffoscope files is 102 KB, the maximum is 3.40 GB, and the minimum is 0.38 KB.

Given the scale of the dataset, we first analyze a random sample of the diffoscope files to understand the reasons of unreproducibility. There are two attributes in the diffoscope file that are of interest to us - `source1` and `unified_diff`, both of them indicating a reproducibility problem. The `source1` attribute either contains the name of the file or the command that is run to get the textual output of the file. For example, if `source1` includes `cyclonedx.json`, we can infer that the difference is in the Software Bill of Materials (SBOM). Another example is if `source1` is `javap -verbose -constants -s -l -private {}`, it means that the Java class file is being compared and the reason of unreproducibility is due to differences in the bytecode. The `unified\_diff` attribute contains the diff of the file in the unified format, for textual files. We then write regular expressions to match these attributes to categories. For example, if regex

`^[+-].*\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}\.\d{3}Z$`

is matched, we infer that the reason of unreproducibility is due to timestamps in the file. We define regular expressions, which we match against all 12,283 diffoscope files.

Finally, we manually go through samples that remain unmatched to the regular expressions and create expressions to map them too. This refines the different root causes of unreproducibility. Then, we report on mitigation strategies from the literature that could fix the reproducibility problem.

4.3 Results

In this section, we present a comprehensive taxonomy of unreproducibility in Java, based on our large-scale, systematic protocol presented in Figure 1. Table 1 summarizes the taxonomy, before deep-diving into each of them. The first column lists the reason for unreproducibility. The second column further breaks down the reason into root causes. The third column indicates if the root cause is novel and has not been reported in the literature before, to our knowledge. The fourth column gives an example of the root cause that appears in our dataset. Finally, the last column lists the main mitigation strategy to fix the unreproducibility, one of the three presented in subsection 2.3.

For example, consider the row about JVM bytecode, where unreproducibility is due to changes in the JVM bytecode between the reference and rebuild artifacts. One of its root cause is due to embedded data in the bytecode, this cause has not been discussed before. The main mitigation strategy for this problem is to fix the build process by preventing build tools to embed branch names in the JVM bytecode. Branch names are mutable and can change over time. Hence embedding them in the bytecode can lead to difference in subsequent builds. The other two mitigation strategy would also work, but they have cons - 1) third-party rebuilder would have to debug the build process to understand what build tool is responsible for embedding and 2) branch names have no conventional names and hence canonicalization would risk removing meaningful differences in the bytecode.

Next, We describe each reason and its root cause in details in the following subsections.

4.3.1 Unreproducible Build Manifests. We consider two manifest files in our analysis - `MANIFEST.MF` and `pom.properties`. The MANIFEST.MF file is a mandatory metadata file in a JAR file. It potentially contains information on the entry point of the JAR file, signatures of the classes, and details of the version control system [11]. pom.properties is an automatically generated file by Maven that contains the group ID, artifact ID, and version of the Maven module.

The first set of differences in manifests is due to a change in environment. For example, the `Built-By` attribute in the `MANIFEST.MF` file is sometimes updated to the username of the user who built the JAR file. **Mitigation:** This non-deterministic information should be stripped out from the JAR file by rebuilder before comparison. See appendix A.1, appendix A.4, appendix A.5, and appendix B.3 for more attributes that follow the same problem and solution.

The next set of differences in manifests is due to differences in the script that configures the rebuild. For example, the `Build-Jdk`

---

[10]https://diffoscope.org/

[11]https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html

Reason for Unreproducibility	Root Cause of Unreproducibility	Novelty	Example	Main Mitigation
Build Manifests	Environment	-	`Built-By` attribute	Canonicalization by rebuilder
	Rebuild Process	-	`Build-Jdk` attribute	Fix rebuild process
	Non-deterministic configuration	✓	Embedded branch names	Fix build process
SBOM	Java Vendor	✓	Different checksum algorithms	Fix rebuild process
	Custom release configuration	✓	Releasing a subset of artifacts	Fix rebuild process
	Non-deterministic information	✓	Serial Number	Canonicalization by rebuilder
Filesystem	Environment	-	Permissions	Canonicalization by rebuilder
	Custom release configuration	✓	Generated binaries are not included	Fix rebuild process
JVM Bytecode	JDK Version	-	Ordering of constant pool entries	Fix rebuild process
	Embedded data	✓	Git branch embedded	Fix build process
	Build time generated code	-	Lambda functions	Canonicalization by rebuilder
Versioning Properties	-	-	Number of tags embedded in Jar	Canonicalization by rebuilder
Timestamps	-	-	Embedded in shell script in Jar files	Fix build process

Table 1: The taxonomy of unreproducibility causes based on the analysis of 12,283 unreproducible artifacts. We are the first to state the most appropriate mitigation strategy.

attribute depends upon the version of JDK used to build the JAR file. **Mitigation:** All environment specifications must be exactly specified in the the build script by rebuilder. Ideally, the build script file extracts information from the MANIFEST to be reproduced. See appendix A.2, appendix A.3, appendix A.6, appendix A.7, and appendix A.8 for concrete examples.

Finally, there are reasons of differences in the manifests MANI FEST.MF and pom.properties that are due to non-deterministic behavior in the build script of the builder. For example, build manifests pom.properties and MANIFEST.MF can differ in order of the attributes if they are not sorted. **Mitigation:** The builder should depend upon on plugins that generate sorted data [12], should not embed non-deterministic data (eg branch names) in the artifact, should ignore timestamp, and stay up to date with the latest version of plugins. appendix A.9, appendix A.10, appendix A.11, appendix B.1, and appendix B.2 describe these reasons and solutions in detail.

4.3.2 Unreproducible Software Bill of Materials. Software bill of material (SBOM) is a formal, machine-readable inventory of software components, metadata about those components, and the hierarchical relationships between them [13]. Its goal is to enable transparency of the software supply chain of an application, for its consumers. In the case of Java, the best practice is to push SBOMs as part of the Maven release [19]. Our experiment finds that this poses reproducibility challenges.

The first set of changes are due to incorrect Java vendor used by the rebuilder. For example, old JDK versions do not have newer hash algorithms like SHA-3. However, vendors of some JDKs backport these algorithms and if builder uses a JDK with backported algorithms, the resulting SBOM will have hashes computed using more algorithms. If rebuilder does not use the same JDK, there will be differences in number of hash algorithms used. **Mitigation:** The rebuilder should use the same JDK vendor and version as the builder. See appendix C.1 for more details.

The next set of changes are due to a tailored release configuration by the builder. For example, the builder can choose to release only a

subset of the artifacts produced by the build process which can lead to less components in the released SBOM. **Mitigation:** The builder should use a conventional build process - which is to streamline the build process only using Maven commands. This helps the rebuilder to reproduce the build process exactly without knowing all ad hoc scripting in the build. See appendix C.2 and appendix C.3 for concrete examples.

Finally, the last set of changes are due to non-deterministic information in the SBOM. CycloneDX attributes like timestamp and serialNumber are non-deterministic and hence they differ in each build. **Mitigation:** There are two possible mitigations. First, the builder can generate the values of these attributes so that they are deterministic. Second, the rebuilder can strip out these attributes from the SBOM before comparing the outputs. See appendix C.5, appendix C.4, and appendix C.6 which outlines the implementation on how to either generate deterministic values or strip out the non-deterministic values.

Reproducible SBOMs is an open problem in the field, we are the first to report about it.

4.3.3 Unreproducible Filesystem. We observe unreproducible Maven releases due to presence or absence of files, changes in the absolute file paths and their related metadata such as permissions, ownership, size and type. We classify them into 2 reasons below.

The first set of changes are due to an influence from the environment. For example, the permissions of the file can vary depending on the umask of the environment of rebuilder appendix D.2. The ownership and timestamp of the file can also vary depending who and when the artifact is built appendix D.3. File sizes can vary depending on the file system used to build the artifact appendix D.6. The names of the files can vary if absolute paths are used to refer to the files in artifact appendix D.7. These names can be of generated files or they could be embedded in JVM bytecode and configuration files which makes the artifact unreproducible appendix D.8. Finally, the order of files in Jar file appendix G.5 can vary as the order depends upon the filesystem layout [14]. **Mitigation:** The build script should either set a fixed value or strip this information from reference and rebuild artifacts before comparison.

---

[12]https://github.com/apache/maven-archiver/commit/763a940540eefad74f9ba73cb5 eed288dc4e639d

[13]https://www.ntia.gov/sites/default/files/publications/sbom_at_a_glance_apr2021 _0.pdf

[14]https://reproducible-builds.org/docs/archives/

Finally, there are some causes due to differences in the build process by the builder and rebuilder. For example, some binaries are not included in the rebuild version as they are not generated by the rebuilder build process. **Mitigation:** The rebuilder process should replicate the build process of the builder exactly. For the above example, the rebuilder should incorporate commands to generate the missing binaries. See appendix D.1 and appendix D.5 for more concrete examples.

*4.3.4 **Unreproducible JVM bytecode**.* We observe 898 unreproducible Maven releases due to changes in the JVM bytecode when rebuilding. The reasons are as follows.

The first set of changes are due to the use of different versions of the JDK or JDK flags used in the build and rebuild process. This differences are due to core compilation changes such as refactoring, ordering changes in the bytecode (specifically methods, fields, static initializers, entries in the constant pool, and array type values in annotation appendix G.1), optimization flags, debug flags. For example, `java.nio.ByteBuffer.flip()` returns a different type in JDK 8 and JDK 11. For `io.dropwizard.metrics:metrics-collectd:4.1.20` rebuilt with JDK 11, all `flip` calls return `java.nio.ByteBuffer` which is different from the reference version built with JDK 8 where the return type is `java.nio.Buffer`. [15] See appendix E.1, appendix E.2, and appendix E.3 for more examples. **Mitigation:** The rebuilder should use the same JDK version and flags as the builder. We propose that the builder should always embed the precise vendor and version of JDK used to build the artifact in the MANIFEST file under the attribute `Build-Jdk`. This way, the rebuilder can use the same version and vendor of JDK to rebuild the artifact by reading the information instead of guessing it. If the builder is using a custom JDK as done in Google Guava [33], this should be made available to the rebuilder for verification.

The next set of changes are due to embedding information related to environment. We observe that the bytecode has git branch names, user name of the builder, and absolute paths. For example, `org.apache.hive:hive-exec:4.0.0-alpha-2` embeds git properties and timestamps in `package-info.class` file using a shell script that adds a Java annotation to `package-info.java`. The Java compiler also embeds the exact Java version used to compile the source code in the bytecode of module-info. However, this hinders reproducibility even if the patch version of the JDK is different. **Mitigation:** The builder should avoid embedding environment specific information in the bytecode. This information changes with each rebuilder and hence the bytecode will be different making the artifact unreproducible. See appendix E.4 for example of a release where it happens.

Finally, the last set of changes are due to build time code generation or modification. The Java compiler produces synthetic code during build in order to handle access to type members [48]. The compiler also names lambda functions [16] and anonymous classes [32] which are not present in the source code. In addition to these capabilities, Maven provides plugins to generate or modify code during the build process. All of this generated code can be different across different builds and hence the build is unreproducible. For example, we observe a difference in names of lambda

function in `org.apache.helix:zookeeper-api:1.0.3`. Another example is `org.apache.hive:kafka-handler:4.0.0-alpha-2` which uses `maven-shade-plugin` to rename package names from `kafkaesqueesque` to `kafkaesque` but only when a Maven profile is activated. **Mitigation:** The names of the synthetic code can be canonicalized by the rebuilder before comparison. Bytecode canonicalization [51] is a technique to strip out non-deterministic information from the bytecode. Differences caused by Maven plugins can be mitigated by running the same plugins as in the original build process. Maven plugins can be configured to run only for certain build profiles. The rebuilder should document the build profiles used by the builder to reproduce the build process. See appendix E.5 for more examples.

*4.3.5 **Unreproducible Versioning Properties**.* Git properties are embedded in Jar files in `git.json` or `git.properties` files. Maven plugin `git-commit-id-maven-plugin` [16] is commonly used to do that. These properties include timezone of commit, remote URL, number of git tags in repository, name of build host, time of build, name of builder, email of builder, branch name, number of commits, local branch name, and total commits in the repository. All of them sometimes diverge. For example, value of `git.tags` vary for Maven release `org.apache.drill:drill-opentsdb-storage:1.21.0` due to additional number of tags in the repository during rebuild. **Mitigation:** The rebuilder should strip out the git properties from the Jar file before comparison or avoid comparison of the Git properties. Another solution is to fix the values for each git property. For example, total commits should be calculated up to the commit that is used to build the artifact. See issue on our repository [17] to see more examples of differences in the git properties.

*4.3.6 **Unreproducible Timestamps**.* Timestamps are the most common cause of unreproducibility in our dataset as we observe them embedded in multiple ways spread across artifacts in Maven release. They are a widely known cause of unreproducibility [2] [20] [3].

In Maven, one solution exists to prevent timestamps from breaking reproducibility. The Maven POM file can be configured to set the timestamp to a fixed value using property `project.build.outputTimestamp` [18]. The value of this property is then used by Maven plugins like `maven-jar-plugin` to set the timestamp in the JAR file.

However, we observe that there are 10 other ways timestamps can be embedded in the JAR file. In our dataset, we observe that: 1) they can be embedded in the properties file appendix F.1, 2) generated documentation appendix F.2, 3) shell scripts appendix F.3, 4) executable binaries appendix F.4, 5) software bill of materials appendix F.5, 6) JVM bytecode appendix F.6, 7) file metadata appendix F.7, 8) MANIFEST.MF appendix F.8, and 9) NOTICE appendix F.9. 10) servlets created by Jasper JSP compiler appendix F.10. **Mitigation:** The solution proposed by Maven is partial, more engineering is needed to support `project.build.outputTimestamp` in all Maven plugins that create the above mentioned files.

---

[15]This is reported in the bug tracking system of OpenJDK [18] which shows that this change is introduced in JDK 9.

[16]https://github.com/git-commit-id/git-commit-id-maven-plugin
[17]https://github.com/chains-project/reproducible-central/issues/19
[18]https://maven.apache.org/guides/mini/guide-reproducible-builds.html

*4.3.7 **Miscellaneous Unreproducibility Reasons**.* We report reasons of reproducibility that are too specific to be categorized in the above sections.

The first reason is due to the use of different build tools. For example, the Maven build tool is available as standalone tool and it also integrated into the Eclipse IDE. They embed information differently. `m2e.projectLocation` and `m2e.projectName` is embedded in `pom.properties` of `org.spdx:spdx-maven-plugin:0.7.0` and `org.owasp.antisamy:antisamy:1.7.3` appendix B.3. Even if the build tool is same, there are differences in rebuild due to usage of versions ranges of a plugin. For example, `org.apache.aries: org.apache.aries.jax.rs.whiteboard:2.0.2` uses `[1.2.5,)` as the version range for `flatten-maven-plugin` in the POM file. This causes difference in the ouput of the plugin. **Mitigation:** The rebuilder should use the same build tool and its precise version as the builder. Moreover, the builder should avoid using version ranges for plugins in the POM file.

The next reason is due to a difference in the operating system. The reference version of `org.apache.helix:helix-front: 1.0.3` produces a Mach-O object file format (Mac) which is different from the ELF binary format produced by the rebuilt version (Linux). See https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2583944777 for detailed diff. **Mitigation:** The builder should ensure that 1) the OS is specified in the build specification or 2) that the binary format is independent of the operating system used to build the artifact.

Usage of Universally Unique Identifier (UUID) is another reason of unreproducibility. We observe that Maven release `org.apache.k araf.examples:karaf-docker-example-static-dist:4.4.0` uses UUID to generate the configuration files. **Mitigation:** The builder should use seed values to generate the UUID so that the UUID can be reproduced.

Maven releases `org.apache.helix:helix-front:1.0.3`, `io.wcm:io.wcm.caconfig.editor:1.9.0`, and `org.apache.i sis.viewer:isis-viewer-restfulobjects-viewer:2.0.0-M7` embed bundled JavaScript files in the Jar file. These files are subject to changes in variable names and new lines. **Mitigation:** The changes in variable names can be fixed by pinning the version of the bundler. For example, Goswami et al. [20] show that UglifyJS can produce output with different variable names for the same input if the version of UglifyJS is different. The difference in new lines can be fixed by the rebuilder by canonicalizing them.

There is one instance where microbenchmarking results are embedded in the Jar file appendix G.4. **Mitigation:** The builder should not embed microbenchmarking results in the Jar file as they are solely for the purpose of improving performance [8] of the application and not needed for the end user.

Finally, there are some indices files created by tools like `Jandex` [19] - a space efficient Java class file indexer, `EFXToolkit` [20] - a toolkit for expressing eForms business rules, and `SpotlessFormatter` [21] - a code formatter for Java. **Mitigation:** The builder should not embed these index files in the artifact as they are not required for the end user and they serve only as cache files for plugins that use them.

*4.3.8 **Mysteries**.* We observe changes in `NOTICE`, POM, and other configuration files that are not due to any of the above reasons and thus, we are unable to propose a mitigation strategy for them. Since these reasons of unreproducibility are so diverse, we discuss them one by one in appendix H.

> **Answer to RQ1: "What are the causes of unreproducible builds in Java?"**
> We identify 6 main reasons of unreproducibility in Java and their mitigation: 1) build manifests, 2) software bill of materials, 3) filesystem, 4) JVM bytecode, 5) versioning properties, and 6) timestamps. They represent the key challenges that researchers will address. For practitioners, they provide clear guidelines to diagnose and mitigate their unreproducible builds in Java.

5 RQ2: UNREPRODUCIBILITY MITIGATION WITH ARTIFACT CANONICALIZATION

5.1 Objective

The goal of this research question is to investigate if artifact canonicalization can make the build reproducible.

5.2 Methodology

Artifact canonicalization means that the entire artifact is transformed by removing non-deterministic and spurious changes, especially in metadata. This helps downstream build tools to interpret the artifact in a consistent way. Multiple reasons can prevent the reproducibility of an artifact(section 4), such as the presence of non-deterministic information in files like MANIFEST.MF, `pom.properties`, and `git.properties` files. Moreover, the artifact may contain non-deterministic information in the form of timestamps or file order. Artifact canonicalization is a process of transforming the artifact into a representation that is independent of these spurious changes. For example, the `pom.properties` file is a file generated by Maven that records the group ID, artifact ID, version, and timestamp of the build in the JAR file. Artifact canonicalization can also remove the timestamp and set a fixed order to the declaration of properties.

OSS-REBUILD[22] is a tool by Google to canonicalize software artifacts in order to improve reproducibility. In OSS-REBUILD, "canonicalizing" is called "stabilizing". It is capable of canonicalizing tarballs, ZIP files, and GZIP files. For example, it canonicalizes file order, modification timestamp, compression algorithm, encoding, and other miscellaneous attributes from a ZIP archive.

CHAINS-REBUILD[23] is our own fork of OSS-REBUILD, with crucial improvements dedicated to Maven reproducibility, which is unfortunately missing in the upstream project. We add support for canonicalizing build manifests and embedded versioning properties in JAR files as suggested in taxonomy (Table 1). For example, the `Built-By` attribute in the MANIFEST.MF file is sometimes updated to the username of the user who built the JAR file, a spurious change which is removed by CHAINS-REBUILD. For versioning properties,

---

[19]https://smallrye.io/jandex/jandex/3.2.1/index.html
[20]https://github.com/OP-TED/efx-toolkit-java
[21]https://github.com/diffplug/spotless

[22]https://github.com/google/oss-rebuild/commit/4ef4c013fe6903cda40a9ee4244e3b6 5b5834325
[23]https://github.com/chains-project/chains-rebuild/commit/6dd67d5c7ac4db112f341 9b5132d8f80a22cbe65

we remove the `git.properties` files embedded in the JAR file. Finally, we remove the `pom.properties` file.

We run OSS-Rebuild and Chains-Rebuild on the 12,283 reference and rebuild artifact pairs in order to analyze how effectively they make the build reproducible, that is when artifacts are bit-by-bit identical after canonicalization as shown below.

$$canonicalize(ReferenceArtifact) = canonicalize(RebuildArtifact)$$

5.3 Results

Tool	Successful Canonicalization	Failed Canonicalization
OSS-Rebuild (4ef4c01)	1165 (9.48%)	11118 (90.52%)
Chains-Rebuild (6dd67d5)	3036 (24.72%)	9247 (75.28%)

Table 2: Effectiveness of OSS-Rebuild and Chains-Rebuild on mitigating 12,283 unreproducible artifacts via canonicalization.

Table 2 shows the results of OSS-Rebuild and Chains-Rebuild on 12,283 unreproducible artifacts. The first column is the name of the tool and its commit hash we use to canonicalize the artifacts. The second column shows the number of artifact pairs that are successfully canonicalized by OSS-Rebuild and Chains-Rebuild. The third column shows the number of artifact pairs that are not properly canonicalized.

OSS-Rebuild, with generic archive stabilizers, is little effective, with only 9.48% of the artifacts becoming reproducible. Chains-Rebuild is significantly more effective, with 2.5x more success, thanks to dedicated features for canonicalizing JAR files. Recall that, we add support for canonicalizing MANIFEST, embedded versioning properties, and pom.properties files in JAR files. For example, artifact `slf4j-ext-2.0.6.jar` of Maven release or `g.slf4j:slf4j-ext:2.0.6` is successfully canonicalized by Chains-Rebuild. Originally, there are differences in zip metadata and MANIFEST.MF. OSS-Rebuild only canonicalizes the zip metadata. Chains-Rebuild takes a step further and canonicalizes the MANIFEST.MF file as well. It fixes the ordering of values in MANIFEST.MF under the `Export-Package` attribute. As a minified example, the fix changes `Export-Package:org.slf4j.ext;version="2.0.6",org.slf4j.agent;version="2.0.6"` to `Export-Package:org.slf4j.agent`. Here arranging the order of the values in `Export-Package` attribute leads to a canonicalized output and a reproducible build.

Overall, this experiment is evidence that artifact canonicalization is a valid solution to mitigate some unreproducible builds. However, it is only a partial workaround and does not replace the mitigation fixing build and rebuild process (see subsection 2.3). As shown in Table 1, there are many other causes of unreproducibility that can be fixed by canonicalization. For example, non-deterministic information in SBOMs and environment influences in files can be fixed by canonicalization. This includes SBOM, generated JVM bytecode, and ordering of sections in JVM bytecode. We leave this as future work as it is purely an engineering task.

Answer to RQ2: "To what extent does artifact canonicalization using OSS-Rebuild make a Maven release reproducible?"
Our publicly available prototype Chains-Rebuild can canonicalize (3036) 24.71% of the artifacts. Our study is the first large-scale experiment on artifact canonicalization, demonstrating that it is a promising mitigation of build unreproducibility in Maven.

6 RQ3: UNREPRODUCIBILITY MITIGATION WITH BYTECODE CANONICALIZATION

6.1 Objective

In section 4, we observe that changes in JVM bytecode is one of the causes of unreproducible builds. In this research question, we investigate to what extent these changes can be mitigated by using bytecode canonicalization.

6.2 Methodology

Bytecode canonicalization is a process of transforming the bytecode of a program into a representation that is independent of specific implementation details inserted by the compiler. This is either done by converting the bytecode to an intermediate form or by rewriting the bytecode. For example, subtraction of two integers is implemented as addition of a positive and negative integer in Java 6 and higher versions. Meanwhile, in Java 5 and lower versions, it is implemented as subtraction of two integers. Hence, if the program is compiled with Java 5 and 6, the two bytecode representations differ even though the program is semantically the same - performing subtraction of two integers. Canonicalization removes these differences by converting the bytecode to an intermediate form which is free from the implementation details that vary across different versions of the compiler.

We consider the state-of-the-art Java bytecode canonicalization tool by Schott et al. [50], called jNorm. jNorm's primary goal is to remove the parts from Java bytecode that may differ due to changes in JDK version and output the canonicalized version of the bytecode. jNorm only removes the differences that do not change the semantic behavior of the artifact. Note that they refer to "canonicalization" as "normalization" in their tool.

We run jNorm version `1.0.0` on the 898 reference rebuild artifact pairs that have differences in the bytecode. For maximum canonicalization, we run the tool with all the command line options as listed in the README [24] of the repository and tool's help output. As a result of canonicalizing the artifacts, jNorm produces files in the Jimple [43] format. We save the canonicalized Jimple files for both the reference and the rebuild artifacts. There are cases where jNorm crashes and canonicalization fails, which we consider as a failure. We do not analyze them further as they are bugs in tool, but we upload the logs for future debugging. We run the Linux diff tool over the canonicalized pairs of Jimple files. If there is no difference, it means that canonicalization is a success, for reproducibility. In other words, successful canonicalization by jNorm produces two

---

[24]jNorm tool - https://github.com/stschott/jnorm-tool/tree/cec4645c5c9b52f73c347349 bf14945b0eb55c87

identical artifacts. We report the number of cases where the diff between the pair is empty. Finally, we analyze the cases where JNORM is unable to fully canonicalize the bytecode in order to understand its fundamental limitations and identify required future work.

6.3 Results

Out of 898 artifacts, JNORM's execution leads to the following distribution: it successfully canonicalizes 267 (29.7%) artifacts which means it produces two identical artifacts; it leaves 478 (53.2%) artifacts with a diff after canonicalization; and there are 153 (17.1%) crashes, which are ignored.

Our results are in contrast with the results of Schott et al. [50] where the authors report that JNORM is able to canonicalize up to 99% of pairs of Java classfiles when JDK version differs. However, our success ratio is lower (29.7%). This can be attributed to the different methodology used to generate the dataset. JNORM is evaluated on Java classfiles built on the same machine but varying JDK versions. In our case, the reference and rebuild artifacts are built on different machines and based on our results in section 4, we know that the build process is different.

Next, we study the reasons why JNORM fails to produce identical artifacts when run on 478 cases. We categorize them based on the features in the bytecode that are not properly canonicalized by JNORM. Note that some unreproducible artifacts may fall into multiple categories as it is possible JNORM is unable to canonicalize multiple problems in the same artifact.

Structural Change Limitation. Recall from subsubsection 4.3.4 that the order of methods, fields, static initializers, entries in the constant pool, and array type values in annotation sometimes differ between the reference and rebuilt artifacts. We find that JNORM is unable to canonicalize most of those problems: the order of fields, methods, static initializers, array type values in annotation, removal of implicit modifier from fields. The only mitigated problem by JNORM is the constant pool order problem, because the Jimple format abstracts away the constant pool [57]. We now discuss concrete examples of failed canonicalization:

- `org.apache.shiro:shiro-aspectj:1.11.0` is not canonicalized due to the changed order of static initializers in the bytecode.
- `antisamy-1.7.3.jar` has a difference in implicit visibility modifier between the reference and rebuild artifact, not handled by JNORM.
- `org.apache.helix:zookeeper-api:1.0.3` has one unreproducible artifact which is due to the naming of lambda functions in the bytecode. Since the names of lambda functions are suffixed by integers determined non-deterministically by the compiler, JNORM is unable to canonicalize them.

Control Flow Limitation. There are two cases of this category where JNORM is unable to canonicalize changes in control flow.

- The first case is change in control flow of if statements as illustrated in Listing 1 and Listing 2. `eu.maveniverse.maven.mima:2.4.2:standalone-static-uber` have changes in the control flow of if conditions. The reference checks for the negative condition of the if statement while the rebuild checks for the positive condition first, because

the two builds use different compilers. This is shown in the diff generated by DIFFOSCOPE in Listing 1. JNORM does not canonicalize control flow there is still a diff in Jimple format, as seen in Listing 2.

```
1  -    99: ifne        106
2  -   102: iconst_1
3  -   103: goto        107
4  -   106: iconst_0
5  -   107: ireturn
6  +    99: ifeq        104
7  +   102: iconst_0
8  +   103: ireturn
9  +   104: iconst_1
10 +   105: ireturn
```

Listing 1: javap diff for control flow difference.

```
1  -if v != 0 goto label;
2  -v = 1;
3  -goto label;
4  -label:
5  -v = 0;
6  +if v == 0 goto label;
7  +return 0;
8  label:
9  -return v;
10 +return 1;
```

Listing 2: JNORM diff for control flow differnce.

- Second, Maven release `io.fabric8:kubernetes-model-apiextensions:6.4.0` contains generated code where the order of if-else blocks is different across different builds. This is not an issue with JNORM, this is an issue with the unreproducibility of the code generator employed during build.

Embedded Data Limitation. In section 4, we discussed that absolute file paths, timestamps, Java version, and project version are embedded in the bytecode. JNORM is able to canonicalize the Java version and project version as it deletes the `module-info.class` file that stores this information. However, it is unable to canonicalize absolute file paths and timestamps[25].

Optimization Limitation. We find 5 cases of optimizations or deoptimizations that are not canonicalized by JNORM. We present the different features in Java that are not canonicalized by JNORM. For a more detailed discussion, refer to the Appendix I.

- JDK 17 creates a lookup table to store the values of the enum while JDK 22 simply stores it based on the order of the enum values.
- JDK 17 uses the `invokevirtual` bytecode instruction when handling invocations to `toString`, `equals`, and `getClass`, while JDK 21 uses `invokeinterface`. This is due to a change proposed in JDK 18 [26].
- String concatenation expressions where one of the operands requires a cast to String.
- Implementation of try and try-with-resource statements and finally clauses vary between Java 5, 8, and 11.
- Reference to outer class from inner class is handled differently in patch versions of Java 17.

We do not delegate fixing all of these problems to JNORM. Recall that we mentioned in section 4 that rebuilder can either fix the build process or canonicalize the output in order to mitigate the unreproducibility. JNORM can fix problems in bytecode structure,

---

[25]JNORM is able to canonicalize timestamp if they are declared in a static final field.
[26]https://github.com/openjdk/jdk/pull/5165

control flow, and embedded data. However, unreproducible optimization and de-optimization can be mitigated by fixing the Java version in the build process.

We note that multiple canonicalization tools can be used in conjunction. For instance, CHAINS-REBUILD and JNORM can be used together to canonicalize the JVM bytecode and the JAR files of a Maven artifact. Since CHAINS-REBUILD can canonicalize 3,036 artifacts and JNORM can canonicalize 267 artifacts, we can combine the 26.89% (3,303 / 12,283). Out of 3,303 artifacts, 2,578 artifacts are canonicalized such that 2,263 / 7,961 (28.43%) releases become fully reproducible.

> **Answer to RQ3: "To what extent does bytecode canonicalization with jNorm mitigate unreproducible builds in Java?"**
>
> JNORM is able to canonicalize 267 (29.7%) out of 898 bytecode artifacts. However, there are 478 (53.2%) artifacts that are still unreproducible after canonicalization which calls for improvements. We present 4 limitation of bytecode canonicalization related to structural change, control flow, embedded data, and optimization. Finally, our experiments demonstrate that combining artifact and bytecode canonicalization successfully mitigates unreproducibility in 26.89% (3,303 / 12,283) artifacts.

7 RELATED WORK

7.1 Reproducibility in Applications

Reproducibility is a key feature in state of the art systems. One of the first software to become reproducible was Bitcoin [27] ensuring that none of the nodes in the Bitcoin peer-to-peer network contains backdoors. Tor Browser[28], a web browser that anonymizes the user's web traffic, is also reproducible since 2013 [26]. Tails, an operating system that protects against surveillance and censorship, also announced in 2017 that their ISO images are reproducible [29]. Both the Tor Browser and Tails operating system emphasize that Reproducible Builds are necessary for software geared towards privacy and security.

The Reproducible Builds [30] project works on the reproducibility of coreboot [31] - an open-source firmware for x86 and ARM systems, Debian [32] - a Linux distribution, FreeBSD [33] and NetBSD - a UNIX based operating systems, and OpenWrt [34] - operating system for embedded devices.

In the Java ecosystem, Eclipse Temurin, a vendor for Java Development Kit, provides reproducible builds for their JDK [25]. The build of Apache Maven is also reproducible [35]. Our work aims to enable ever more Java software applications to become reproducible by providing comprehensive and effective mitigation strategies.

---

[27]https://github.com/bitcoin/bitcoin
[28]https://gitlab.torproject.org/tpo/core/tor
[29]https://tails.net/news/reproducible_Tails/
[30]https://reproducible-builds.org/
[31]https://tests.reproducible-builds.org/coreboot/coreboot.html
[32]https://tests.reproducible-builds.org/debian/reproducible.html
[33]https://tests.reproducible-builds.org/freebsd/freebsd.html
[34]https://tests.reproducible-builds.org/openwrt/openwrt.html
[35]https://github.com/jvm-repo-rebuild/reproducible-central/blob/master/content/org/apache/maven/maven/README.md

7.2 Verifying Reproducibility

In the literature, we have found three ways to check reproducibility. First approach is to build once and compare the generated artifacts with the artifacts on package registry. Second is to build twice with variations in environment and compare the generated artifacts. Finally, third approach is to verify that the projects stand the test of time by ensuring that they build successfully over time.

Comparison between reference and rebuild artifacts. One way to verify build reproducibility is to build once and compare the generated artifacts with the artifacts on package registry. This is what we do in this paper.

Outside the Java ecosysem, Malka et al. [31] investigate reproducibility in functional package manager Nix. The paper proves that functional package managers help reproduciblity by showing 91% of the packages in Nix are reproducible. Similarly, Bajaj et al. [2] investigate reproduciblity in Debian and Arch Linux packages. Apart from uncovering the causes of unreproducibility, they also do a survival analysis of packages to see how long it takes to fix the unreproducibility and how long the package stays reproducible. They show that Arch Linux pacakges become reproducible sooner than Debian packages. However, a Debian package is likely to stay reproducible for longer before it becomes unreproducible again. Moritz [35] proposes a decentralized protocol on Hyperledger Fabric to verify reproducibility. The protocol involves comparing the locally generated buildinfo file with the one on Debian registry. Linderud [27] also proposes a protocol to verify buildinfo files, but a layer of transparency log is added in order to ensure reproducibility status for specific builds are immutable. Drexel et al. [15] contributes with an independent rebuilder that verifies reproducibility of Arch Linux packages based on this approach. Vu et al. [58] focus on finding the most common files and APIs that differ between source repository and Python wheel published on PyPI. Goswami et al. [20] investigate reproducibility in NPM packages. They compare the minified artifacts generated by JavaScript bundlers to report the reasons of unreproducibility. Although there are overlap in taxonomies reported by these works, our analysis focuses on Java ecosystem and proposes novel reasons for unreproducibility. For example, variations in MANIFEST, JVM bytecode, and versioning properties are first seen in our work. Even for reasons such as timestamps, we are the first to report 9 ways that it can be embedded in a Maven release.

Some works analyze single projects rather than the entire ecosystems. Shi et al. [52] investigate reproducibility for 4 commercial systems running in Huawei and CentOS - a production grade Linux operating system. Pöll et al. [41] discover reproducibility issues while rebuilding Android 5 to 12 images. Carné de Carnavalet et al. [12] investigate reproducibility for 16 versions of the TrueCrypt encryption tool. Finally, Andersson [1] investigates reproducibility for Go Ethereum binary. None of the subjects in these related work are Java based. The work of Pöll et al. [41] is Java based, but Android images do not compile to JVM bytecode like traditional Java applications. Hence, ours paper is the first one to investigate reproducibility of normal Java applications at scale.

The most closely related work here is the work by Xiong et al. [60] that compares 59 Maven projects on Maven Central and internal registry of Huawei with the artifacts generated by the

build process. They discover build reproducibility problems due to environment, JDK, multithreading, and other tools. Our work differs in the scale of the dataset (ours is 4× larger) and the taxonomy of the root causes of unreproducibility. Our taxonomy has more categories than the one proposed by Xiong et al. incl. new essential categories such as variations in filesystem and software bill of materials.

Comparison between two subsequent builds. Benedetti et al. [3] verify the reproducibility of 12,000 Ruby, PyPI, and Maven packages using this approach. They keep all inputs same except the environment which is changed for each build using reprotest [36]. reprotest can change locale, timezone, and number of CPUs for example. Ren et al. [45, 47] and Leija et al. [37] propose a technique to localize and fix the root cause of unreproducibility in Java artifacts. They build the source files and dependencies for Debian packages twice and compare the artifacts using DIFFOSCOPE [37]. Lagnöhed [23] propose a tool rmake built on top of make with support for building projects twice and comparing the artifacts. These works differ from our work in the approach they take to verify reproducibility. We claim that the first approach uncovers more reasons of unreproducibility as it compares artifacts generated by build process which have maximum differences in environment. Carné de Carnavalet et al. [12] also emphasizes that there should be maximum difference between the environments to detect unreproducibility.

Build Outcome Analysis. Build success is a prerequisite to study reproducibility. Maes-Bermejo et al. [29], Hassan et al. [21], [55] and Yasi et al. [61] show that only 40-60% of the Maven projects in their respective datasets build successfully. This indicates that around half of the projects don't even generate artifacts as their build process fails. All of these works are different from ours as they focus on the success of the build process only and do not look at the generated artifacts. Our work focuses on artifacts to identify the reasons of unreproducibility. Technically, Maes-Bermejo et al. [29] use mvncleancompile-X to build all the Maven projects. In our work, our build commands in the dataset are tailored to each project so that the build process succeeds.

7.3 Fixing Unreproducibility

We now review the related work on fixing causes of unreproducibility.

Ren et al. [45–47] propose techniques to localize and fix unreproducibility in Debian packages. They localize the file in the Debian package that causes unreproducibility and then suggest a patch based on a history of patches for fixing unreproducible builds. This approach is specific for Debian packages and does not apply to Java artifacts. Moreover, their approach to detect unreproducibility is different from ours as mentioned in section 7.2.

Mukherjee et al. [36] propose a technique to fix unreproducible Python builds caused by dependency errors. They report that 71% of Python builds in their dataset fail; they fix these builds by analyzing the logs. This work is only comparess success of builds, not artifacts, and it is focussed on the Python ecosystem.

Xiong et al. [60] classifies fixing unreproducibility into 3 categories - 1) remediation by patching source code, third-party dependency, or build script 2) controlling which means interception of non-deterministic build instructions and returning pre-defined values for them 3) interpretation which means documenting the reason for unreproducibility. The authors perform these fixes automatically using a tool called JavaBEPFix which is not available to the public. Hence, we cannot evaluate the effectiveness of their tool on our dataset.

Keshani et al. [22] automates the process of reproducing Maven projects by generated buildspec files. They show that their approach simplifies generated buildspec files and also fixes issues in the currently existing buildspec files in Reproducible Central dataset. Our work also finds that a mitigation strategy to fix unreproducibility issue is by fixing the build script. In addition, we perform the first analysis of fixing by canonicalization.

Randrianaina et al. [44] identifies the configuration options in that cause unreproducibility. The authors select highly configurable systems such as Linux, BusyBox, and ToyBox to study the unreproducibility of their builds. They find options for module signing, debug information, and profiling that cause unreproducibility. In our work, configuration options is one of the causes of unreproducibility. Especially, when the build process involves setting Maven profiles subsubsection 4.3.4.

Dietrich et al. [14] show that only a 25% (out of 14,156) of the binaries in Maven ecosystem are actually bit by bit identical. These binaries come from 4 different registries - Maven, Google Assured Open Source Software, RedHat, and Oracle build-from-source. They propose 4 levels of binary equivalence to compare binaries which becomes increasingly lenient as we go from level 1 to level 4. First is to compare binaries bit by bit and this is the strictest level of binary equivalence check. Second is to compare binaries and ignore differences that have no semantic effect (for example, @Deprecated). OSS-REBUILD fits into the second level of checking binary equivalence. Third is to convert binaries into a common format and then compare them. JNORM fits into the third level of checking binary equivalence Finally, fourth is to compare binaries using a similarity metric. In the paper, they use tlsh [38] to compare hash distance between two binaries; lesser the hash distance, more similar the binaries are. This work is similar to ours as they also evaluate effectiveness of JNORM over thousands of pairs of JVM bytecode [13] to fix unreproducibility issues. However, we consider more causes of unreproducibility other than ones in JVM bytecode.

Finally, strip-nondeterminism [38] is a tool the Debian project that strips out timestamps and file system order from archives. However, it only takes care of these 2 causes of unreproducibility and does not cover the other causes of unreproducibility that we have discussed in this paper.

8 CONCLUSION

In this paper, we have presented the first large-scale study of build unreproducibility in Java. We have identified the six main causes of unreproducibility with underlying root causes: 1) build manifest, 2) software bill of materials, 3) filesystem, 4) JVM bytecode, 5) versioning properties, and 6) timestamps. Leveraging the same

---

[36]https://salsa.debian.org/reproducible-builds/reprotest
[37]https://diffoscope.org/

[38]https://salsa.debian.org/reproducible-builds/strip-nondeterminism

dataset, we have evaluated the effectiveness of canonicalization as a mitigation strategy to fix unreproducibility. Our results have shown that 26.89% of unreproducible artifacts can be canonicalized and become reproducible after canonicalization.

As future work, we want to formalize the notion of acceptable canonicalization by verifying what transformations can be applied to build artifacts without obscuring meaningful differences. This is important to ensure that the technique eliminates non-deterministic noise while preserving semantic integrity. Build reproducibility verification via canonicalization must be sound and precise.

9 ACKNOWLEDGEMENTS

REFERENCES

[1] Vivi Andersson. 2024. *Geth Rebuild : Verifiable Builds for Go Ethereum.* https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-355285

[2] Rahul Bajaj, Eduardo Fernandes, Bram Adams, and Ahmed E. Hassan. 2023. Unreproducible Builds: Time to Fix, Causes, and Correlation with External Ecosystem Factors. *Empirical Softw. Engg.* 29, 1 (Nov. 2023). https://doi.org/10.1007/s10664-023-10399-4

[3] Giacomo Benedetti, Oreofe Solarin, Courtney Miller, Greg Tystahl, William Enck, Christian Kästner, Alexandros Kapravelos, Alessio Merlo, and Luca Verderame. 2025. An Empirical Study on Reproducible Packaging in Open-Source Ecosystems. In *Proceedings of the 47th International Conference on Software Engineering.*

[4] Hervé Boutemy. 2024. Jvm-Repo-Rebuild/Reproducible-Central. jvm-repo-rebuild. https://github.com/jvm-repo-rebuild/reproducible-central

[5] John Boyer and Glenn Marcy. 2008. Canonical XML Version 1.1. https://www.w3.org/TR/xml-c14n11/

[6] Reproducible Builds. 2024. JVM — Reproducible-Builds.Org. https://reproducible-builds.org/docs/jvm/

[7] Iris Clark and Mark Reinhold. 2014. JEP 223: New Version-String Scheme. https://openjdk.org/jeps/223

[8] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. 2021. What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering* 47, 7 (July 2021), 1452–1467. https://doi.org/10.1109/TSE.2019.2925345

[9] Russ Cox. 2019. Surviving Software Dependencies: Software Reuse Is Finally Here but Comes with Risks. *Queue* 17, 2 (April 2019), Pages 80:24–Pages 80:47. https://doi.org/10.1145/3329781.3344149

[10] Russ Cox. 2025. Fifty Years of Open Source Software Supply Chain Security: For Decades, Software Reuse Was Only a Lofty Goal. Now It's Very Real. *Queue* 23, 1 (April 2025), Pages 20:84–Pages 20:107. https://doi.org/10.1145/3722542

[11] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. 2011. Software Bertillonage: Finding the Provenance of an Entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11).* Association for Computing Machinery, New York, NY, USA, 183–192. https://doi.org/10.1145/1985441.1985468

[12] 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).* Association for Computing Machinery, New York, NY, USA, 16–25. https://doi.org/10.1145/2664243.2664288

[13] Jens Dietrich, Tim White, Mohammad Abdollahpour, Elliott Wen, and Behnaz Hassanshahi. 2024. BinEq-A Benchmark of Compiled Java Programs to Assess Alternative Builds. https://www.researchgate.net/publication/383666359_BinEq-A_Benchmark_of_Compiled_Java_Programs_to_Assess_Alternative_Builds

[14] Jens Dietrich, Tim White, Behnaz Hassanshahi, and Paddy Krishnan. 2024. Levels of Binary Equivalence for the Comparison of Binaries from Alternative Builds. https://doi.org/10.48550/arXiv.2410.08427 arXiv:2410.08427

[15] Joshua Drexel, Esther Hänggi, and Iyán Méndez Veiga. 2024. Reproducible Builds and Insights from an Independent Verifier for Arch Linux. (2024). https://doi.org/10.18420/SICHERHEIT2024_016

[16] Ben Evans. 2020. Behind the Scenes: How Do Lambda Expressions Really Work in Java? https://blogs.oracle.com/javamagazine/post/behind-the-scenes-how-do-lambda-expressions-really-work-in-java

[17] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. 2023. It's like Flossing Your Teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security. In *2023 IEEE Symposium on Security and Privacy (SP).* 1527–1544. https://doi.org/10.1109/SP46215.2023.10179320

[18] Neal Gafter. 2002. [JDK-4774077] Use Covariant Return Types in the NIO Buffer Hierarchy - Java Bug System. https://bugs.openjdk.org/browse/JDK-4774077

[19] Yogya Gamage, Nadia Gonzalez Fernandez, Martin Monperrus, and Benoit Baudry. 2025. Software Bills of Materials in Maven Central. https://doi.org/10.48550/arXiv.2501.13832 arXiv:2501.13832 [cs]

[20] Pronnoy Goswami, Saksham Gupta, Zhiyuan Li, Na Meng, and Daphne Yao. 2020. Investigating The Reproducibility of NPM Packages. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 677–681. https://doi.org/10.1109/ICSME46990.2020.00071

[21] Foyzul Hassan, Shaikh Mostafa, Edmund S. L. Lam, and Xiaoyin Wang. 2017. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '17).* IEEE Press, Markham, Ontario, Canada, 38–47. https://doi.org/10.1109/ESEM.2017.11

[22] Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch. 2024. AROMA: Automatic Reproduction of Maven Artifacts. *Proc. ACM Softw. Eng.* 1, FSE (July 2024), 38:836–38:858. https://doi.org/10.1145/3643764

[23] Felix Lagnöhed. 2024. *Integration of Reproducibility Verification with Diffoscope in GNU Make.* Ph. D. Dissertation. https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-205473

[24] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39, 2 (March 2022), 62–70. https://doi.org/10.1109/MS.2021.3073045

[25] Andrew Leonard. 2024. Eclipse Temurin Reproducible Verification Builds for Secure Supply Chain Validation | Adoptium. https://adoptium.net/blog/2024/08/adoptium-reproducible-verification-builds/

[26] Holger Levsen, kpcyrd, and Jelle van der Waa. 2025. A Tale of Several Distros Joining Forces for a Common Goal: Reproducible Builds. https://reproducible-builds.org/_lfs/presentations/2025-02-02-a-tale-of-several-distros-joining-forces-for-a-common-goal-reproducible-builds/#/

[27] Morten Linderud. 2019. *Reproducible Builds:Break a Log, Good Things Come in Trees.* Master's thesis. The University of Bergen. https://bora.uib.no/bora-xmlui/handle/1956/20411

[28] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. 2021. The Nature of Build Changes. *Empirical Software Engineering* 26, 3 (March 2021), 32. https://doi.org/10.1007/s10664-020-09926-4

[29] Michel Maes-Bermejo, Micael Gallego, Francisco Gortázar, Gregorio Robles, and Jesus M. Gonzalez-Barahona. 2022. Revisiting the Building of Past Snapshots — a Replication and Reproduction Study. *Empirical Software Engineering* 27, 3 (March 2022), 65. https://doi.org/10.1007/s10664-022-10117-6

[30] Julien Malka. 2025. How NixOS and Reproducible Builds Could Have Detected the Xz Backdoor for the Benefit of All. https://luj.fr/blog/how-nixos-could-have-detected-xz.html

[31] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. 2025. Does Functional Package Management Enable Reproducible Builds at Scale? Yes. In *22nd International Conference on Mining Software Repositories.* Ottawa, France. https://hal.science/hal-04913007

[32] Maverik. 2016. Why getClass Returns the Name of the Class + $1 (or $*). https://stackoverflow.com/q/7172581/11751642

[33] Kyle Moore. 2023. Guava Artifacts Are Not Bitwise Reproducible · Issue #6321 · Google/Guava. https://github.com/google/guava/issues/6321

[34] Alexa Morales. 2021. Java Still Rocks the Finance Industry. Here's Why Java 16 Makes It Even Better. https://blogs.oracle.com/javamagazine/post/finance-quant-forex-java16

[35] Johan Moritz. 2023. *Decentralized Validation of Reproducible Builds : A Protocol for Collaborative and Decentralized Validation of Package Reproducibility.* Ph. D. Dissertation. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-329759

[36] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing Dependency Errors for Python Build Reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, Virtual Denmark, 439–451. https://doi.org/10.1145/3460319.3464797

[37] 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).* Association for Computing Machinery, New York, NY, USA, 167–182. https://doi.org/10.1145/3373376.3378519

[38] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH – A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop.* 7–13. https://doi.org/10.1109/CTC.2013.9

[39] Mike Perry. 2013. Deterministic Builds Part One: Cyberwar and Global Compromise | Tor Project. https://blog.torproject.org/deterministic-builds-part-one-

cyberwar-and-global-compromise/

[40] Timo Pohl, Pavel Novák, Marc Ohm, and Michael Meier. 2025. SoK: Towards Reproducibility for Software Packages in Scripting Language Ecosystems. https://doi.org/10.48550/arXiv.2503.21705 arXiv:2503.21705 [cs]

[41] Manuel Pöll and Michael Roland. 2022. Automating the Quantitative Analysis of Reproducibility for Build Artifacts Derived from the Android Open Source Project. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '22)*. Association for Computing Machinery, New York, NY, USA, 6–19. https://doi.org/10.1145/3507657.3528537

[42] Piotr Przymus and Thomas Durieux. 2025. Wolves in the Repository: A Software Engineering Analysis of the XZ Utils Supply Chain Attack. https://doi.org/10.48550/arXiv.2504.17473 arXiv:2504.17473 [cs]

[43] Raja Vallée-Rai and Laurie J. Hendren. 1998. Jimple: Simplifying Java Bytecode for Analyses and Transformations. http://www.sable.mcgill.ca/publications/techreports/#report4

[44] Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, and Mathieu Acher. 2024. Options Matter: Documenting and Fixing Non-Reproducible Builds in Highly-Configurable Systems. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR '24)*. Association for Computing Machinery, New York, NY, USA, 654–664. https://doi.org/10.1145/3643991.3644913

[45] 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*. ACM, Gothenburg Sweden, 71–81. https://doi.org/10.1145/3180155.3180224

[46] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. 2019. Root Cause Localization for Unreproducible Builds via Causality Analysis over System Call Tracing: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019* (Nov. 2019), 527–538. https://doi.org/10.1109/ASE.2019.00056

[47] Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. 2022. Automated Patching for Unreproducible Builds. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 200–211. https://doi.org/10.1145/3510003.3510102

[48] Donato Rimenti. 2018. Synthetic Constructs in Java | Baeldung. https://www.baeldung.com/java-synthetic

[49] David Robinson. 2017. Trends in Government Software Developers - Stack Overflow. https://stackoverflow.blog/2017/07/12/trends-government-software-developers/

[50] Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. 2024. Java Bytecode Normalization for Code Similarity Analysis. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ECOOP.2024.37*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2024.37

[51] Aman Sharma, Martin Wittlinger, Benoit Baudry, and Martin Monperrus. 2024. SBOM.EXE: Countering Dynamic Code Injection Based on Software Bill of Materials in Java. https://doi.org/10.48550/arXiv.2407.00246 arXiv:2407.00246 [cs]

[52] Yong Shi, Mingzhi Wen, Filipe R. Cogo, Boyuan Chen, and Zhen Ming Jiang. 2022. An Experience Report on Producing Verifiable Builds for Large-Scale Commercial Systems. *IEEE Transactions on Software Engineering* 48, 9 (Sept. 2022), 3361–3377. https://doi.org/10.1109/TSE.2021.3092692

[53] Aleksey Shipilev. 2024. JEP 280: Indify String Concatenation. https://openjdk.org/jeps/280

[54] César Soto-Valero, Martin Monperrus, and Benoit Baudry. 2022. The Multibillion Dollar Software Supply Chain of Ethereum. *Computer* 55, 10 (Oct. 2022), 26–34. https://doi.org/10.1109/MC.2022.3175542

[55] Matúš Sulír and Jaroslav Porubän. 2016. A Quantitative Study of Java Software Buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2016)*. Association for Computing Machinery, New York, NY, USA, 17–25. https://doi.org/10.1145/3001878.3001882

[56] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (Aug. 1984), 761–763. https://doi.org/10.1145/358198.358210

[57] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, Mississauga, Ontario, Canada, 13.

[58] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the Discrepancy between Sources and Packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 780–792. https://doi.org/10.1145/3468264.3468592

[59] Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, Michel Cukier, Christian Kästner, Alexandros Kapravelos, Dominik Wermke, and William Enck. 2025. Research Directions in Software Supply Chain Security. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). https://doi.org/10.1145/3714464

[60] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R. Cogo, and Zhen Ming (Jack) Jiang. 2022. Towards Build Verifiability for Java-based Systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 297–306. https://doi.org/10.1145/3510457.3513050

[61] Tommy Yasi and Jingzhong Qin. 2022. *Automatic Building Of Java Projects On GitHub: A Study On Reproducibility*. Ph. D. Dissertation. https://urn.kb.se/resolve?urn=urn:nbn:se:miun:diva-45928

A UNREPRODUCIBLE MANIFEST.MF

The MANIFEST.MF file is a mandatory metadata file that contains information about the JAR file.

A.1 Built-By

This attribute is used to indicate the name of the user who built the JAR file. This can be different for environments as the user name is different.

Example Maven release `org.apache.camel:came-ahc-ws:3.13.0` has an unreproducible artifact `camel-ahc-ws-3.13.0.jar`. This JAR file has a different `Built-By` attribute in the MANIFEST.MF file. In the reference version of the JAR file, the `Built-By` attribute is set to `root` while in the rebuild version of the JAR file, the `Built-By` attribute is set to `aman`.

Solution Remove the `Built-By` attribute from the MANIFEST.MF file.

A.2 Implementation-Build-Java-Vendor

This attribute records the vendor of the JDK used to build the JAR file. There are 17 different open source distributions of JDK [39] - GraalVM, OpenJDK, Temurin to name a few. Although all of them follow the same Java specification by Oracle, the implementation details could differ.

Example MANIFEST.MF of `ldapchai-0.8.0.jar` of Maven release `com.github.ldapchai:ldapchai:0.8.0` shows difference in vendor when rebuilt. It changes from `AdoptOpenJDK` to `Oracle Corporation`. AdoptOpenJDK, now known as `Adoptium`, indicates that the JAR file is built using `Temurin`. We use `OpenJDK` for the rebuild version, so the vendor is set to `OracleCorporation` as OpenJDK is maintained by Oracle.

Solution The `buildspec` file should document the exact vendor of the JDK.

A.3 Build-Jdk or Implementation-Build-Java-Version

This attribute reports the exact version of JDK used to build the JAR file. The JDK version given a fixed vendor is based on Semantic Versioning. Ensuring the exact same JDK version across different environments depends upon the major, minor, patch version and if this version has a pre-release flag or not. These four factors are in addition to the vendor of the JDK as discussed above.

Example Artifact `camel-cloud-3.13.0.jar` of Maven release `org.apache.camel:camel-cloud:3.13.0` is built with different versions of JDK. The reference version of the JAR file has the `Build-Jdk` attribute set to `1.8.0\_292` while the rebuild version of the JAR file has the `Build-Jdk` attribute set to `1.8.0\_422`. The difference in the JDK version is due to the update number [7].

Solution The `buildspec` file should document the exact version of JDK and provide an automated way to rebuild. There are some cases where the JDK version is precisely mentioned in Reproducible Central dataset. However, we remove them in our dataset filtering process as they require manual intervention to build.

A.4 Signed JARs

Java provides a way to sign each file in a JAR to ensure its users that the file has not been tampered with [40].

Example The JAR file `xmlchai-0.1.0.jar` of Maven release `org.jrivard.xmlchai:xmlchai:0.1.0` shows that signatures are removed in the rebuild version of the JAR file.

Solution Strip the signature from the JAR file as signatures can not be replicated.

A.5 Os-Version

This attribute is used to indicate the operating system version used to build the JAR file.

Example The artifact `jandex-maven-plugin-3.1.0.jar` from Maven release `io.smallrye:jandex-maven-plugin:3.1.0` has different `Os-Version` attribute in the MANIFEST.MF file. It seems that the kernel version is different as the value changes from `5.15.0-1035-azure` to `5.15.0-118-generic`.

Solution This attribute should be stripped from the MANIFEST.MF file before comparison as it is environment specific and Reproducible Builds are environment independent.

A.6 Created-By

These attribute is used to indicate the name and version of the build tool or the Java version used to produce the JAR file. This can be created by Java `jar` tool or any build tool like Maven, Gradle, or Ant.

Example `jline-console-3.22.0-sources.jar` of Maven release `org.jline:jline-console:3.22.0` has different `Created-By` attribute in the MANIFEST.MF file. The reference version reports `ApacheMaven` while the rebuild version reports `MavenSourcePlugin3.2.1`.

Solution The `buildspec` file should document the exact variant and version of the build tool used to build the JAR file as this ensures the same build tool is used across different environments.

A.7 Originally-Created-By

This attribute is added by `maven-bundle-plugin` when the JAR file is created again without cleaning the build directory. The plugin updates the JAR with this attribute to indicate the original build plugin that created the JAR file.

Example Maven release ch.qos.logback:logback-access:1.3.0-alpha14 has an artifact `logback-access-1.3.0-alpha14.jar` which does not have the `Originally-Created-By` when rebuilt.

Solution The `buildspec` file should document prerequisites to build the JAR file for the same reasons as `Created-By` attribute.

A.8 SCM-Revision

MANIFEST file can contain information related to version control system like Git. `SCM-Revision` attribute is used to indicate the commit hash of the source code used to build the JAR file [41].

Example The JAR file `io.wcm.tooling.commons.crx-packmgr-helper-2.0.2.jar` of Maven release `io.wcm.tooling.commons:io.wcm.tooling.commons.crx-packmgr-helper:2.0.2` has

---

[39]https://sdkman.io/jdks/

[40]https://docs.oracle.com/javase/tutorial/deployment/jar/intro.html
[41]https://www.mojohaus.org/buildnumber-maven-plugin/usage.html

different `SCM-Revision` attribute in the MANIFEST.MF file. The value changed from `c48c286dat2022-01-06T11:46:12+01:00` to `bc9cc174at2024-10-17T02:14:22Z`. There are two reasons of differences here. First, the commit hash is different which indicates that commit hash corresponding to the git tag has changed. Second, the timestamp is different and this is due to embedding build time in the same attribute.

Solution The commit hash should remain the same across different builds as it guarantees that the source code is the same across different builds. `buildspec` files currently relies on Git tags to fetch source code, but they are mutable and can be moved to different commits. Thus, the `buildspec` file should document the exact commit hash of the source code.

A.9 `SCM-Git-Branch`

This attribute is used to indicate the branch of the source code used to build the JAR file.

Example MANIFEST.MF of `ldapchai-0.8.0.jar` of Maven release `com.github.ldapchai:ldapchai:0.8.0` shows that the value changes from `master` to `338023a44e4dc62aff8985ca42c2f6743258b1c0`.

Solution Branches are mutable in Git so they should not be embedded in the MANIFEST.MF file. Commit hash should be used instead as they are fixed.

A.10 `Bnd-LastModified`

This attribute is part of the OSGi specification to create bundles.

Example `shiro-cas-1.9.0.jar` is an artifact of `org.apache.shiro:shiro-cas:1.9.0` where the value of this attribute changes from `1647431421514` to `1735860648961`.

Solution Apache Maven provides a property `${project.build.outputTimestamp}` that fixes the timestamp and can be read from again in subsequent rebuilds [42]. This property should can either be reused to set the value of `Bnd-LastModified` attribute or the attribute should be stripped from the MANIFEST.MF file.

A.11 Order of values for attributes

If the value of an attribute is a comma separated list, the order of the values can differ across different builds.

Example `slf4j-api-2.0.6.jar` of Maven release `org.slf4j:slf4j-api:2.0.6` has an attribute `Export-Package` in the MANIFEST.MF file. This attribute is part of OSGi specification and is used to export packages that can be imported by other bundles. The value of this attribute is a list of packages whose order differs across the different builds.

Solution The attributes where this problem exists are `Include-Resource`, `Private-Package`, and `Provide-Capability`. The solution to depend upon the latest version of `maven-bundle-plugin` as it makes the order deterministic by sorting the values [43].

A.12 Addition or removal of attributes

Presence or absence of an attribute in the MANIFEST.MF file can also cause unreproducibility.

Example `apache-any23-csvutils-2.7-sources.jar` of Maven release `org.apache.any23:apache-any23-csvutils:2.7` removed 10 attributes from the MANIFEST.MF file. These attributes are `Specification-Title`, `Specification-Version`, `Specification-Vendor`, `Implementation-Title`, `Implementation-Version`, `Implementation-Vendor`, `Implementation-Build`, `Implementation-Build-Date`, `X-Compile-Source-JDK`, and `X-Compile-Target-JDK`.

Solution We suggest to canonicalize these attributes. We cannot confirm what build tool embeds these attributes.

B UNREPRODUCIBLE POM.PROPERTIES

The `pom.properties` file is an automatically generated file that contains information about the Maven project.

B.1 Order of properties

The file contains properties `groupId`, `artifactId`, `version`. However, their order can differ across different builds.

Example The `pom.properties` file of Maven release `io.dropwizard.metrics:metrics-annotation:4.1.20` has the property `version` in different order [44].

Solution The order of the properties should be fixed in the `pom.properties` file or the file should be removed from the JAR file as it is not required for the JAR file to run.

B.2 Timestamp

While generating `pom.properties` file, Maven uses the current time to set the time of generation.

Example `ch.qos.logback.db:logback-classic-db:1.2.11.1` embeds timestamp in one of the comments in the `pom.properties` file.

```
1   #Generated by Apache Maven
2   #Wed Apr 20 20:27:33 CEST 2022
3   #Wed Jan 29 08:18:40 UTC 2025
```

Solution The timestamp should either be set to the value of `SOURCE\_DATE\_EPOCH` or removed from the `pom.properties` file.

B.3 Eclipse properties

If a Maven module is built using, m2e - Maven build tool for Eclipse IDE, it adds some properties to the `pom.properties` file other than the ones added by standalone Maven build tool.

Example Maven release `org.spdx:spdx-maven-plugin:0.7.0` has `m2e.projectLocation` and `m2e.projectName` properties in the `pom.properties` file.

Solution These attributes can be removed from the `pom.properties` file as they are not required for the JAR file to run.

C UNREPRODUCIBLE SOFTWARE BILL OF MATERIALS

C.1 Removal of hash algorithms

CycloneDX supports multiple hash algorithms to calculate the hash of components. Our dataset shows that some rebuilds omits some of the hash algorithms.

---

[42] https://maven.apache.org/guides/mini/guide-reproducible-builds.html
[43] https://github.com/apache/felix-dev/commit/d885d99a6a16660f655a4fd18e8a1a39beef0a15
[44] https://github.com/chains-project/reproducible-central/issues/17#issuecomment-2576045821

Example `ratis-3.1.1-cyclonedx.json` artifact of Maven release `org.apache.ratis:ratis:3.1.1` has different hash algorithms in the CycloneDX SBOM. The rebuild version of the SBOM omits the `SHA3-384`, `SHA3-256`, and `SHA3-512` hash algorithms.

Solution These hash algorithms are not present in JDK 1.8.0_422 which is used to rebuild the artifact in our dataset. The solution is to use a JDK that has backported these hash algorithms. This has already been solved by modifying the `buildspec` file to use Azul JDK [45].

C.2 Addition or removal of `components`

SBOM records the components used to build the software. In the context of Maven, this can either be a dependency or a Maven submodule. Difference in components in SBOM mean that the dependencies are different across different builds and hence the build is unreproducible.

Example Maven release `net.sourceforge.pmd:pmd:7.0.0` has an artifact `pmd-7.0.0-cyclonedx.json` which shows that the `components` attribute is different across reference and rebuild versions of the SBOM. The rebuild version documents two more Maven submodules which are `net.sourceforge.pmd:pmd-cli:7.0.0` and `net.sourceforge.pmd:pmd-dist:7.0.0`.

Solution The build script here [46] shows ad-hoc configuration of the build process. This should not be done as it hinders automation when checking for reproducibility. The fix here is to stick to conventional builds.

Example Maven release `io.dropwizard.metrics:metrics-core:4.1.32` has a CycloneDX SBOM, `metrics-core-4.1.32-cyclonedx.json`, where all `components` are deleted except `org.slf4j:slf4j-api:1.7.36`.

Solution It is unclear why all components are deleted. The original dataset does not have the same result [47] asked here:https://github.com/dropwizard/metrics/issues/4702. The solution that Reproducible Central uses is to ignore checking for generated SBOMs.

C.3 Modification of `components`

Even if the components are the same across different builds, the content of the components should exactly match across different builds.

Example Maven release `org.apache.bcel:bcel:6.8.1` has an artifact `bcel-6.8.1-cyclonedx.json` where one of the components `org.apache.commons:commons-lang3:3.14.0` has different hashes. SHA1 hash in reference versions is `29a8e03` while in the rebuild version is `1ed4711`. The rebuild version's hash is also consistent with the one on Maven Central [48]. This happens because the local `.m2` folder has locally installed version which is referenced in the SBOM. However, the rebuild version uses the version from Maven Central.

Solution The release command should clean the local Maven repository before building the artifact. This can be done using `purge-local-repository` goal of `maven-dependency-plugin` [49].

C.4 `metadata.timestamp`

An attribute in the CycloneDX SBOM is used to indicate the time when the SBOM is generated. This can be different across different builds as the build time is different.

Example The CycloneDX SBOM, `cyclonedx-core-java-7.3.2-cyclonedx.json` of Maven release `org.cyclonedx:cyclonedx-core-java:7.3.2` has different `metadata.timestamp` attribute.

Solution It should respect the Maven property `${project.build.outputTimestamp}` to set the value of `metadata.timestamp` attribute. Although this has been addressed in pull request [50] in release `3.0.8`, but it does not seem to resolve the issue until release `8.0.0`. Perhaps, a better way to deal with this is to canonicalize the timestamp attribute.

C.5 `serialNumber`

A unique serial number is assigned to an SBOM even if the content of the SBOM is the same [51].

Example Maven release `org.cyclonedx:cyclonedx-maven-plugin:2.7.5` produces `cyclonedx-maven-plugin-2.7.5-cyclonedx.xml` where difference in `serialNumber` attribute is observed.

Solution The `serialNumber` attribute should be removed before comparison as it is not relevant to the content of the SBOM. Else, the generation should depend upon some fixed content of the SBOM to ensure the same serial number across different builds. This has been addressed by computing `serialNumber` on the basis of groupId, artifactId, and version [52].

C.6 System Package Data Exchange (SPDX)

SPDX is a standard to document the licenses of the components used to build the software. It has non-deterministic attributes like `created` and `licenseListVersion` which can cause unreproducibility. `created` attribute is used to indicate the time when the SPDX file is created. `licenseListVersion` attribute is used to indicate the version of the SPDX license list used to document the licenses of the components. This list is updated externally and can vary with time.

Example We only see one instance of unreproducibility that falls under this category. Maven release `org.apache.commons:commons-parent:60` has an artifact `commons-parent-60.spdx.json` where value of `created` and `licenseListVersion` attribute is different across different builds.

Solution These attributes should be removed before comparison as they are non-deterministic.

---

[45] https://github.com/jvm-repo-rebuild/reproducible-central/commit/ff0cc3037b7a1d0f009d4e4461c10f817c8658c7

[46] https://github.com/pmd/pmd/blob/main/.ci/build.sh#L81

[47] https://github.com/jvm-repo-rebuild/reproducible-central/blob/55c06c8c4a080e66267f573560166c42071b2814/content/io/dropwizard/metrics/metrics-parent-4.1.32.diffoscope

[48] https://repo1.maven.org/maven2/org/apache/commons/commons-lang3/3.14.0/commons-lang3-3.14.0.jar.sha1

[49] https://maven.apache.org/plugins/maven-dependency-plugin/examples/purging-local-repository.html

[50] https://github.com/CycloneDX/cyclonedx-core-java/pull/63

[51] https://cyclonedx.org/docs/1.6/json/#serialNumber

[52] https://github.com/CycloneDX/cyclonedx-maven-plugin/pull/425

D UNREPRODUCIBLE FILESYSTEM

D.1 Files are removed or added

If archives have different number of files, then the archive cannot be reproducible as the content of the archive is different.

Example Maven release `commons-daemon:commons-daemon:1.4.0` has an unreproducible artifact `commons-daemon-1.4.0-bin-windows.zip` which does not contain 3 Windows executable binary in the rebuild version of the archive. There is custom non-Maven command that has to be executed during the build process to generate and place these binaries under the build folder. Since the command is not documented in the `buildspec` file, the rebuild version of the archive does not contain these binaries.

Solution The build process should be fixed to incorporate generation of other binaries along with the Maven build command. If this is not possible, the rebuilder should document the steps for generating the binaries in the `buildspec` file.

Example Maven release `org.apache.rat:apache-rat-project:0.16.1` has an unreproducible artifact `apache-rat-0.16.1-src.tar.bz2` which has an empty directory `apache-rat-0.16.1/apache-rat-api/` compressed in the archive. This directory has been deleted in this version[53]. However, it existed in the earlier versions and while checking out only the files are deleted but directory is kept as git does not track directories.

Solution The rebuilder should remove the empty directories from the source code before building the artifact. `gitclean-df` can be used to remove the empty directories from the source code.

Example Maven release `org.apache.activemq:apache-artemis:2.28.0` has source archive, `apache-artemis-2.28.0-source-release.tar.gz`, as its artifact. It has pushed the node modules in the archive which are not present in the rebuild version.

Solution The builder should avoid pushing the node modules in the source archive and should only push the files that are tracked by the version control system. Ideally, the node modules are always untracked to reduce the size of git repository.

D.2 File permissions

The file in the archive can be created with different permissions across different builds. `umask` utility controls the permission of newly created files and directories and it can be different across different environments.

Example Maven release `io.github.albertus82:unexepack:0.2.1` has an unreproducible artifact `unexepack-0.2.1-bin.zip` which has different file permissions for the file `unexepack-0.2.1/unexepack.jar` in the rebuild version of the archive.

Solution The rebuilder can either set the file permissions to a fixed value or set `umask` in the rebuilder script.

D.3 File ownership

The file in the archive can be created with different ownership across different builds.

Example Maven release `org.apache.accumulo:accumulo:1.10.2` has an artifact `accumulo-1.10.2-src.tar.gz`. The file ownership has changed from `christopher` to `aman`.

Solution File ownership should be set to a fixed value by the rebuilder from all artifacts before comparison.

D.4 File timestamps

Example `org.apache.zookeeper:parent:3.8.1` upon rebuilding produces `parent-3.8.1.tar.gz` where the timestamp of the files in the tar ball is set to the time of rebuild.

Solution The timestamp can either be set to a fixed value or stripped from the file by rebuilder before comparison.

D.5 File type

The file type of archive are different across different builds. This causes the size of the archives to be different as different compression algorithms could be used.

Example Consider Maven release `io.github.git-commit-id:git-commit-id-maven-plugin:6.0.0` which has an unreproducible artifact `git-commit-id-maven-plugin-6.0.0-sources.jar`. It shows that the type of the file changes from `Ziparchivedata,atleastv1.0toextract,compressionmethod=store` to `Javaarchivedata(JAR)`[54]. This seems to happen because outdated plugins are used to generated the source archive as it has attributes like `Built-By` in the MANIFEST.MF file which has been removed in the newer version of `maven-archiver` plugin[55].

Solution The builder should always use the specified version of the plugins to generate the archives. To help with this, the local repository can be cleaned before building the artifact using `purge-local-repository`[56].

Example

D.6 File size

Example `org.apache.maven:apache-maven:3.8.1` has the executable `mvn` whose size is different across different builds. It changes from 5741 bytes to 5940 bytes because of different new lines in both executables.

Solution The rebuilder should replace the new lines from all files to either LF or CRLF before comparison based on the operating system of the builder. Builder should not replace the new lines as it could break behavior of the executable on different platforms[57].

D.7 File names

Example `org.apache.jena:jena-permissions:4.3.2` has an unreproducible artifact `jena-permissions-4.3.2-sources.jar` where the file name is different in the rebuild version of the archive, but the content is identical.

Solution The builder should not embed system specific paths in any artifact [1]. Go ecosystem has a tool `trimpath` which removes the system specific paths.

---

[53]https://github.com/apache/creadur-rat/blob/c5a31fecc3b6d3697e20cb867e82c55de cf969be/RELEASE-NOTES.txt#L22

[54]Seems different from RC https://github.com/jvm-repo-rebuild/reproducible-central/blob/329fd7e4d7721cf7ea85fe203fadfb884be13fa9/content/io/github/git-commit-id/git-commit-id-maven-plugin-6.0.0.diffoscope#L129-L130

[55]https://issues.apache.org/jira/browse/MSHARED-799

[56]https://maven.apache.org/plugins/maven-dependency-plugin/examples/purging-local-repository.html

[57]https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=74682318

D.8 File paths embedded

If absolute file paths are embedded in the archive, then the build is unreproducible. In our dataset, we observe that the absolute file paths are embedded in JVM bytecode and configuration files.

Example Maven release `org.apache.cxf.fediz:common:1.6.1` has an artifact `common-1.6.1.jar` which has a file path embedded in the JVM bytecode.

Example Maven release `org.owasp:dependency-check-cli:7.0.4` embeds absolute paths in `dependencycheck-cache.properties`.

Solution The builder should not embed absolute paths in the configuration files and stick to relative paths.

E UNREPRODUCIBLE JVM BYTECODE

E.1 Debug information

This corresponds to the information in the JVM bytecode that is used for debugging the Java program. There are three types of debug information in the JVM bytecode - `SourceFile`, `LineNumberTable`, and `LocalVariableTable`.

Example In release `io.dropwizard.metrics:metrics-httpclient:4.1.27`, the artifact `metrics-httpclient-4.1.27.jar` shows an extra line in LineNumberTable which corresponds to the `return` JVM bytecode instruction for a void method. This could be because the precise version and vendor of JDK is not used.

Solution The rebuilder should remove the information related to debugging from the JVM bytecode before comparison.

E.2 Optimizations or de-optimizations

The JVM bytecode can be optimized or de-optimized across different builds if the JDK version used to build the artifact is different. Newer versions of JDK can have better optimizations while older versions can have less performant code. We have found the following types of differences in the JVM bytecode across different builds:

- try-with-resource is simplified with less `goto` instructions and `close()` invocations. Example: `io.dropwizard.metrics:metrics-servlets:4.1.20`.
- return type in `invokevirtual` change from `java/nio/Buffer` to `java/nio/ByteBuffer`. Example: `io.dropwizard.metrics:metrics-collectd:4.1.20`.
- `invokeinterface` is changed to `invokevirtual`. Example: `com.github.ldapchai:ldapchai:0.8.6`.
- implementation `finally` is changed from using subroutines (`jsr` and `ret`) to using more inline code. Example: `net.bytebuddy:byte-buddy-dep:1.14.5`.
- `static` and `final` modifiers are added or removed from anonymous classes. Example: `dev.langchain4j:langchain4j-core:0.26.0`.
- invocations to `Objects.requireNonNull` are added in rebuild version of the JVM bytecode. Example: `dev.langchain4j:langchain4j-core:0.26.0`.
- `checkcast` instructions are reduced. Example: `org.apache.sling::org.apache.sling.servlets.resolver:2.8.2`.

- implementation of nested enums differ. This is due to a commit in OpenJDK 22 [58] which changes how switch map tables are generated.
- synthetic method, `values()`, has bytecode under a separate method in reference version and under the static initializers in rebuild version [59]. Example: `com.google.guava:guava:32.0.0-jre`
- synthetic field for inner class is added in rebuild version [60]. Example: `com.google.guava:guava:32.0.0-jre`
- local variables in `LocalVariableTable` are in the scope for a more number of instructions. Example: `org.apache.helix:helix-common:1.0.3`.
- `ldc\_w` instructions are replaced with `ldc` instructions. Example: `org.codehaus.mojo:jaxb2-maven-plugin:3.1.0`.
- strings are concatenated which reduces the number of `ldc` instructions. Example: `org.apache.sling.event:org.apache.sling.event:4.2.18`.
- `StackMapTable` is modified. `org.apache.sling:org.apache.sling.servlets.resolver:2.8.2`

Solution The rebuilder should use the precise version of JDK to build the artifact as the reference version. This is a non-trivial task as the exact version of JDK is not always mentioned and there are a myriad of JDK versions and vendors available. As an example, we found that building `com.github.ldapchai:ldapchai:0.8.6` with JDK 22 solves all differences related to JVM bytecode.

E.3 Refactoring

These changes simply change the structure of the JVM bytecode without altering the semantics or performance. Changes in order of sections in JVM bytecode is an example of refactoring. The JVM bytecode contains sections that are order independent, but they can differ across different builds and fail the build reproducibility check. In our dataset, we observe that the order of methods, fields, static initializers, entries in the constant pool, array type values in annotation, method declaration in interfaces, and order of inner classes can differ across different builds.

Example In Maven release `org.apache.activemq:activemq-runtime-config:5.16.4`, the artifact `activemq-runtime-config-5.16.4.jar` uses `jakarta.xml.bind.annotation.XmlElementRefs` which takes an array of `XmlElementRef` annotations. The order of these values in the array is different across different builds.

Solution The rebuilder should sort the order independent sections of the JVM bytecode before comparison.

Apart from order, we observe 2 types of refactoring that did not change the order.

- `private` modifier is removed from a synthetic field in an anonymous class in artifact `antisamy-1.7.3.jar`. We consider this as a refactoring as fields in anonymous classes are implicitly `private`.
- The positive and negative branches of an `if` statement are swapped in the JVM bytecode of Maven release `eu.maven iverse.maven.mima:2.4.2:standalone-static-uber`

---

[58]https://github.com/openjdk/jdk/pull/10797
[59]https://bugs.openjdk.org/browse/JDK-8241798
[60]https://bugs.openjdk.org/browse/JDK-8271717

Solution The rebuilder should canonicalize the JVM bytecode before comparison.

E.4 Embedded data

Absolute file paths, timestamp, java version, project version and other environment related variables are embedded in the JVM bytecode.

Example Maven release `org.apache.hive:hive-exec:4.0.0-alpha-2` embeds git branch name, user name, timestamp, and git repository path in `hive-exec-4.0.0-alpha-2.jar` which makes it unreproducible.

Solution The builder should embed such environment related variables in the JVM bytecode.

E.5 Build time generated/modified code

The Java compiler produces synthetic code during build in order to handle access to members [48]. The compiler also adds names to lambda functions [16] and inner classes [32] which are not present in the source code. In addition to these capabilities, Maven provides plugins to generate or modify code during the build process. All of this generated code can be different across different builds and hence the build is unreproducible.

Example Maven release `io.github.chains-project:maven-lockfile-github-action:3.4.0` generates a lot of code during the build process. The code is generated by two dependencies of maven-lockfile, `o.quarkiverse.githubaction:quarkus-github-action:2.0.1` and `io.quarkus:quarkus-arc:3.1.1.Final` in order to make the application compatible with GitHub actions. The generated code differs in order of fields, different constant values, and embedded system properties. Another example is `org.apache.hive:kafka-handler:4.0.0-alpha-2` which uses `maven-shade-plugin` to rename package names from `kafkaesqueesque` to `kafkaesque`.

Example Maven release `org.apache.helix:zookeeper-api:1.0.3` has an artifact `zookeeper-api-1.0.3.jar`. The lambda function changes from `lambda\$parseRoutingData\$19` to `lambda\$parseRoutingData\$1`.

Example Maven release `org.apache.hive:hive-parser:4.0.0-alpha-2` has an artifact `hive-parser-4.0.0-alpha-2-sources.jar` that has sources generated by ANTLR3 Maven plugin. The generated code has comments where order of values differs across builds. This issue is fixed by sorting the values in the comments [61].

Example Maven release `org.apache.shiro:shiro-aspectj:1.11.0` produces an artifact where AspectJ metadata is embedded in the JVM bytecode. The metadata does not exist in the reference build. This issue has been fixed by forcing the compilation of AspectJ classes [62].

Example Maven release `io.fabric8:kubernetes-model-apiextensions:6.4.0` contains build time generated code where the order of if-else blocks is different across different builds. We do not classify them under refactoring as the order of if-else blocks is relevant to the semantics of the code.

---

[61]https://github.com/antlr/antlr4/pull/3809
[62]https://github.com/apache/shiro/commit/2d92460cd8b7a621be0b82725b7a5cd6952734c1

Solution The generated code should be deterministic across different builds. This calls for fixes in the Java compiler or the Maven plugins that generate code during the build process.

F UNREPRODUCIBLE TIMESTAMPS

F.1 Properties file

The properties files in a JAR file stores configuration related to the project. This can be either a file manually created by the developer or generated by the build tools. In our dataset, we observe that the timestamps in the properties files are different across different builds.

Example Maven release `com.adobe.acs:acs-aem-commons-bundle:5.2.0` has files embedded in the JAR file that have different timestamps of generations across different builds.

Solution The plugin that generates the properties file should respect the Maven property `${project.build.outputTimestamp}` to set the value of the timestamp in the properties file.

F.2 Documentation

The generated documentation can have timestamps embedded in it.

Example Maven release `org.apache.synapse:synapse-documentation:3.0.2` has an artifact `synapse-documentation-3.0.2.jar` which has a timestamp embedded in the documentation. `org.apache.karaf:manual:4.4.0` is another example.

Solution The plugin that generates the documentation should respect the Maven property `${project.build.outputTimestamp}` to set the value of the timestamp in the documentation.

F.3 Shell scripts

Example Maven release `org.apache.sling:org.apache.sling.feature.cpconverter:1.3.4` has an artifact `org.apache.sling.feature.cpconverter-1.3.4.tar.gz` that packs a shell script `bin/cp2feature`. The script takes build timestamp as one of the arguments and the timestamp is different across different builds.

Solution The shell script should respect the Maven property `${project.build.outputTimestamp}` or the standard `SOURCE\_DATE\_EPOCH` environment variable to set the value of the timestamp in the shell script.

F.4 Binaries

Example Maven release `io.github.albertus82:unexepack:0.2.1` embeds a Windows executable that embeds the timestamp.

Solution The build tool for binary should respect `SOURCE\_DATE\_EPOCH` environment variable to set the value of the timestamp in the binary.

F.5 SBOM

Unreproducibility due to attribute `timestamp` in the SBOM as explained in appendix C.4.

F.6 JVM bytecode

JVM bytecode can have timestamps embedded as values of variables.

Example package-info.class in `4.0.0-alpha-2:hive-stan` `dalone-metastore-server:4.0.0-alpha-2` has timestamp embedded in an annotation.

Solution This should respect the Maven property `${project.` `build.outputTimestamp}` to set the value of the timestamp in the JVM bytecode.

F.7 File timestamps

Files have timestamp for creation or modification time as shown in appendix D.4.

F.8 MANIFEST

Attribute `Bnd-LastModified` tends to be non-deterministic as documented in appendix A.10.

F.9 NOTICE

Example Maven release `org.apache:apache:24` has a sources Jar where the year of build is embedded in the `apache-24/NOTICE` file.

Solution The plugin that generates the NOTICE file should respect the Maven property `${project.build.outputTimestam` `p}` to set the value of the timestamp in the NOTICE file.

F.10 Servlets

Example Maven release `org.apache.nifi:nifi-web-ui:1.27` `.0` has embedded timestamps. These timestamps are created by Jasper which compiles JSP files to servlets.

Solution The plugin that generates the servlets should respect the Maven property `${project.build.outputTimestamp}` to set the value of the timestamp in the servlets.

G UNREPRODUCIBLE ORDER

G.1 Ordering in JVM bytecode

The JVM bytecode contains sections that are order independent, but they can differ across different builds and fail the build reproducibility check. In our dataset, we observe that the order of methods, fields, static initializers, entries in the constant pool, array type values in annotation, method declaration in interfaces, and order of inner classes can differ across different builds.

Example In Maven release `org.apache.activemq:activemq-` `runtime-config:5.16.4`, the artifact `activemq-runtime-conf` `ig-5.16.4.jar` uses `jakarta.xml.bind.annotation.XmlEleme` `ntRefs` which takes an array of `XmlElementRef` annotations. The order of these values in the array is different across different builds.

Solution The rebuilder should sort the order independent sections of the JVM bytecode before comparison

appendix E.3

G.2 pom.properties

The attributes in the `pom.properties` file are not in the same order across different builds.

Example `io.dropwizard.metrics:metrics-jcache:4.1.20` has an artifact `metrics-jcache-4.1.20.jar` which has a `version` attribute that appears at line 1 in the reference version and at line 3 in the rebuild version. `io.takari.maven:takari-smart-`

`builder:1.0.0` is another example where order of `artifactId` is different.

Solution The rebuilder should sort the attributes in the `pom.pr` `operties` file before comparison. However, this file is most commonly generated by `maven-archiver-plugin` and this ordering issue has been fixed [63].

G.3 XML files

Example Order of `remoteResources` in file generated by `maven-` `remote-resources-plugin` is different across different builds. This is seen in `org.apache.axis2:axis2-resource-bundle:` `1.8.2`.

Solution The rebuilder should sort the order of elements in the XML files before comparison. This won't affect the future releases as this unstable order has been fixed in `3.3.0` version of `maven-` `remote-resources-plugin` [64].

Example `components.xml` generated by `plexus-containers` has different order of elements across different builds.

Solution The rebuilder should sort the order of elements in the XML files before comparison. This has been fixed recent releases of `plexus-containers` [65].

G.4 JMH benchmark

JMH is a microbenchmarking framework for Java.

Example `org.apache.hive:hive-metastore-benchmarks:` `4.0.0-alpha-2` publishes output from JMH benchmarks in the JAR file. Even though the contents remain the same, the order of results is different across different builds.

Solution Builder should not publish the output from JMH benchmarks in the JAR file. It is a metric that is used to improve the performance of the code and should not be part of the released artifact.

G.5 Files in archive

Jar file is a zip file and the order of files can be in any order.

Example Order of files in `org.apache.synapse:synapse-` `package-archetype:3.0.2` do not stay the same across builds.

Solution The order of zip files can be canonicalized by rebuilder before comparison. For future releases, this has been fixed [66] [67].

G.6 Generated files

Example Maven release `org.apache.hive:hive-parser:4.0.` `0-alpha-2` generates a file using ANTLR3 Maven plugin which creates a different order of values in the comments across different builds.

Solution The plugin that generates the file should sort the order of values in the comments before comparison. ANTLR3 is not maintained anymore but this issue has been fixed in ANTLR4 [68].

Example `net.sourceforge.pmd:pmd-cli:7.0.0-rc3` generates a shell completion script which has different order of flags across different builds.

[63]https://github.com/apache/maven-archiver/commit/763a940540eefad74f9ba73cb5eed288dc4e639d
[64]https://github.com/apache/maven-remote-resources-plugin/pull/74
[65]https://github.com/codehaus-plexus/plexus-containers/issues/8
[66]https://issues.apache.org/jira/browse/MJAR-263
[67]https://issues.apache.org/jira/browse/MSHADE-347
[68]https://github.com/antlr/antlr4/pull/3809

Solution The plugin that generates the file should sort the order of flags in the shell completion script before comparison. This has been fixed in later version of `pmd-cli`.

Example Apache Tomcat JspC creates `web.xml` file with Java Server Pages configuration. We observe that the order declaration is different across builds in Maven release `org.apache.hive:hive-service:4.0.0-alpha-2`.

Solution The plugin that generates the file should sort the order of declarations in the `web.xml` file before comparison.

G.7 Attribute values in MANIFEST

Some values are comma separated lists whose order can differ as shown in appendix A.11.

H MYSTERIES

- Target property is changed from 8 to 1.8 and JVM option removed in dependency-reduced-pom.xml https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2581417522
- Change in generated documentation https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2582710398
- URL to Nexus repository is replaced with a new one in `io.dropwizard.metrics:metrics-parent:4.2.6` https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2583498935.
- Configuration for `maven-failsafe-plugin` is deleted in `org.apache.maven.plugins:maven-gpg-plugin:3.1.0`. https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2583617364
- URL change in license headers org.apache.drill:distribution:1.20.0. https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2583664806
- Change in `registry.json` in `org.apache.drill:drill-common:1.21.1`. https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2583691367
- Changes in META-INF files https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2583778244
- Differences in encoding https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2583612194
- Changes in POM.xml https://github.com/chains-project/reproducible-central/issues/20#issuecomment-2584322171
- `groupId` under `exclusion` varies. https://github.com/chains-project/reproducible-central/issues/17#issuecomment-2576071584
- `org.apache.turbine:turbine-webapp-6.0:3.0.0` changes 2011-1969 to 2011-1970 in copyright header.
- Content of NOTICE changes in `org.apache.paimon:paimon-bundle:0.9.0`.
- Order of flags of commands is flipped autocompletion script. Maven release `net.sourceforge.pmd:pmd-cli:7.0.0-rc3`. Artifact `pmd-cli-7.0.0-rc3-completion.sh`

- Trailing slash is missing from one of the URLs in properties file https://github.com/chains-project/reproducible-central/issues/8#issuecomment-2566465525.
- Interface is changed from `Writable` to `Observable` in `org.apache.isis:isis-core-metamodel:2.0.0-M7` [69]
- Extra `Recipe` is returned in the list in `tech.picnic.error-prone-support:error-prone-contrib:0.15.0` [70]
- The name of method parameter of `java.lang.Thread` is changed from `arg0` to `name` in `dubbo-2.7.12.jar` [71]
- Change in project.build.outputTimestamp https://github.com/chains-project/reproducible-central/issues/18#issuecomment-2581344963
- Maven release `org.mybatis:mybatis:3.5.16` has an artifact `mybatis-3.5.16.jar` which has different `X-Compile-Release-JDK` attribute in the MANIFEST.MF file. It changes from 8 to 16. This is strange because source and target JDK are 16. `X-Compile-Source-JDK` and `X-Compile-Target-JDK` are contradicting `X-Compile-Release-JDK` attribute.

I OPTIMIZATIONS NOT CANONICALIZED BY JNORM

The first case is how implementation enums differs between JDK 17 and JDK 22 compiled artifacts. JDK 17 creates a lookup table to store the values of the enum while JDK 22 simply stores it based on the order of the enum values [72]. We observe this difference in `com.github.ldapchai:ldapchai:0.8.6` as the reference version is built with JDK 22 and the rebuilt version is built with JDK 17.

The second case is the usage of invokevirtual in JDK 17 and invokeinterface in JDK 21 bytecode instructions to execute `toString`, `equals`, and `getClass` methods. They differ in the time the method is resolved. The invokeinterface instruction resolves the method at runtime while invokevirtual resolves the method at compile time. We observed this difference in our dataset in `dev.langchain4j:langchain4j:0.28.0` when `toString` method is invoked on instance of class.

The third case that JNORM is unable to canonicalize the implementation of string concatenation when its operands require a cast to String. For example, `"ErrordeterminingJDBCtypefor column"+column+".Cause:"+e` is a string concatenation expressions in artifact `mybatis-3.5.16.jar`. JNORM identifies that there are two ways to implement string concatenation [53] - using `StringBuilder` (JDK < 9) and using `invokedynamic` (JDK ≥ 9) instruction and canonicalizes the expression to the second one. However, it fails to canonicalize the string conversion of e which is an instance of `java.sql.SQLException` before concatenation using `invokedynamic`.

Implementation of try and try-with-resource statements and finally clauses also differ across Java versions and is also not canonicalized by JNORM. For example, reference and rebuild artifacts of `io.dropwizard.metrics:4.1.20:metrics-servlets-4.1.20`

---

[69]https://github.com/chains-project/reproducible-central/issues/6#issuecomment-2734000008

[70]https://github.com/PicnicSupermarket/error-prone-support/issues/1595

[71]https://github.com/chains-project/reproducible-central/issues/6#issuecomment-2734158834

[72]https://github.com/openjdk/jdk/pull/10797

are compiled with JDK 8 and JDK 11 respectively. The rebuild artifact is optimized in the way it handles closing of resource. Another example is `net.bytebuddy:1.14.5:byte-buddy-1.14.5.jar` that uses deprecated bytecode instructions `jsr` and `ret` to implement closing of resource.

jNorm is unable to handle differences in how outer class reference is handled in the bytecode. A synthetic field called `this\$0` is created in the bytecode to refer to the outer class from inner class. The reference version of `com.google.guava:guava:32.0.1-jre` is built using custom JDK which omits this synthetic field [73] as it is not needed. We also observe that patch versions of JDK 17 handle this differently. They refer to the outer class reference using either `this.this\$0` or `this\$0`. The former adds another `getfield` instruction to the bytecode. `psi-probe-tomcat85-3.7.0-tests.jar` is an example of artifact that contains this difference.

[73] https://bugs.openjdk.org/browse/JDK-8271717