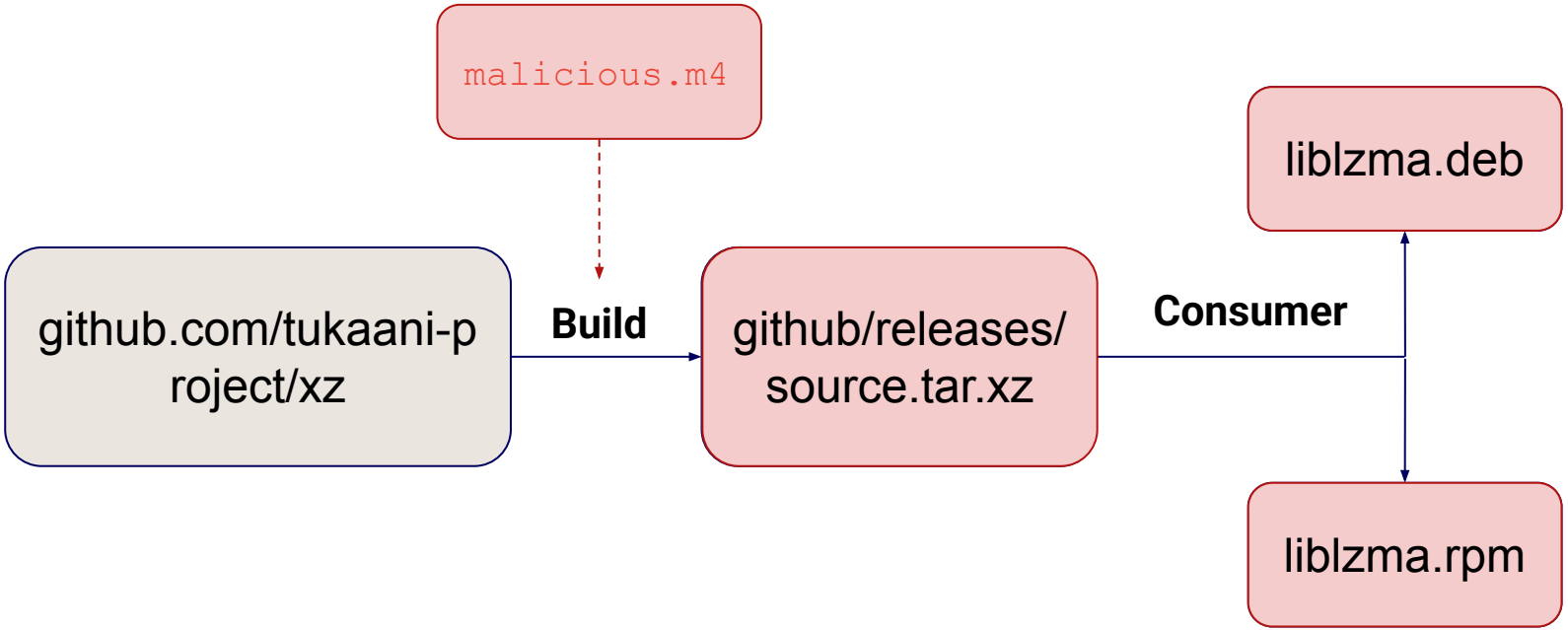
Causes and Canonicalization of Unreproducible Builds in Java

Published in IEEE Transactions on Software Engineering

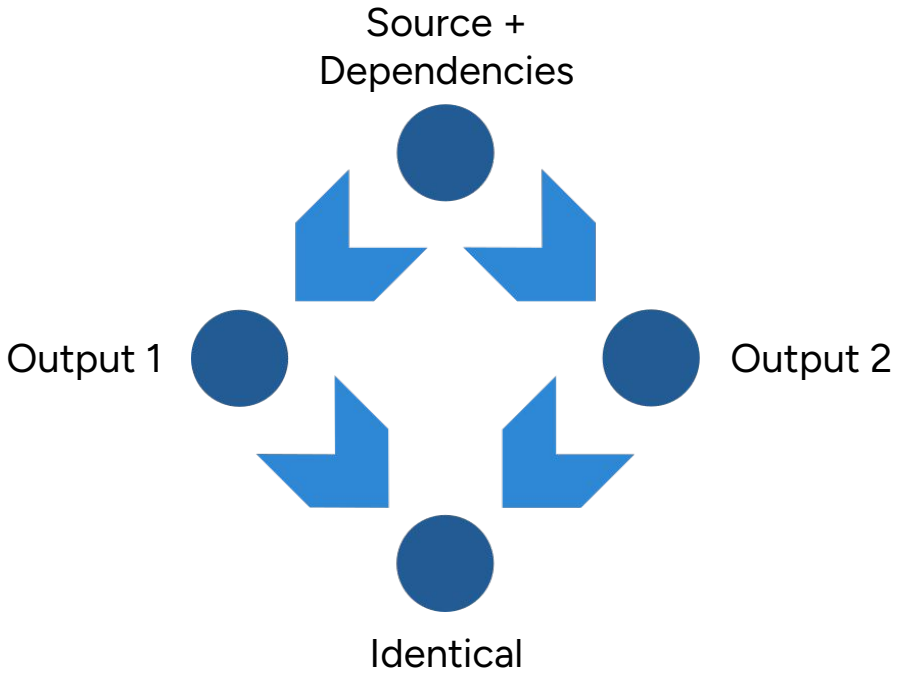
Aman Sharma, Benoit Baudry, Martin Monperrus

`xz-utils` attack



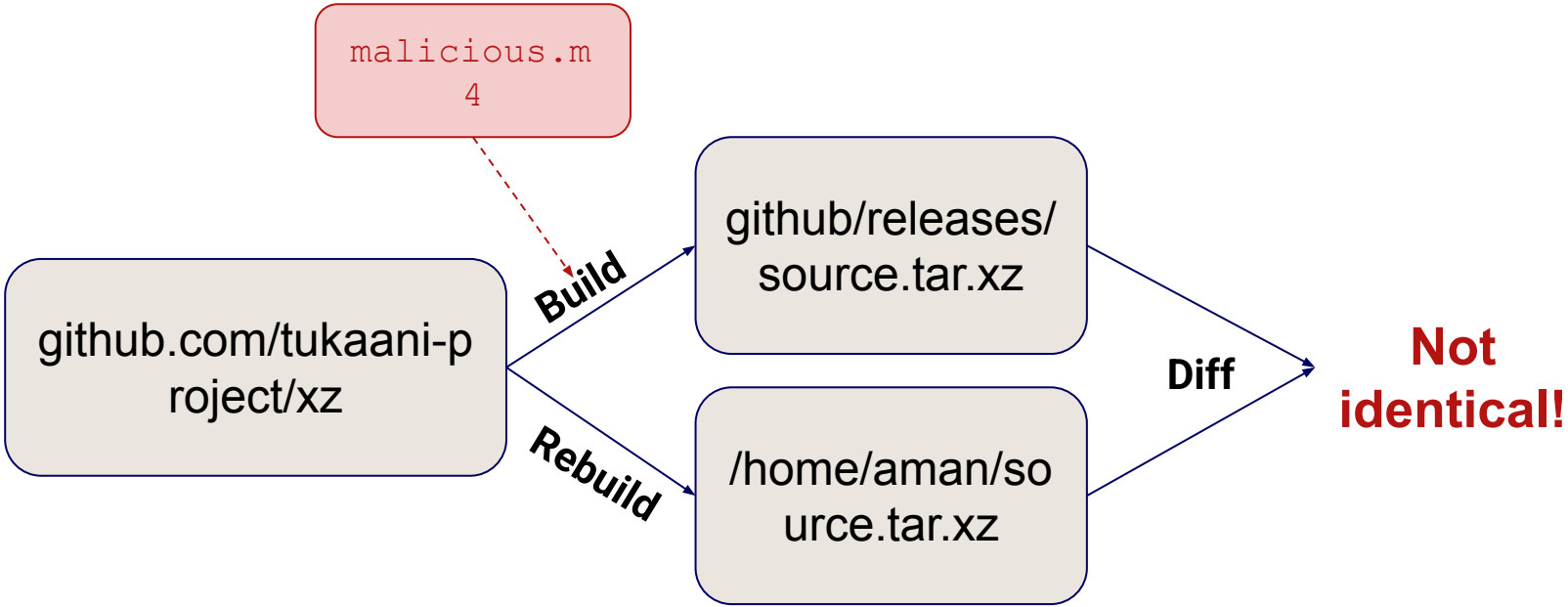
[1] J. Malka, 'How NixOS and reproducible builds could have detected the xz backdoor for the benefit of all'. Available: https://luj.fr/blog/how-nixos-could-have-detected-xz.html

Reproducible Builds



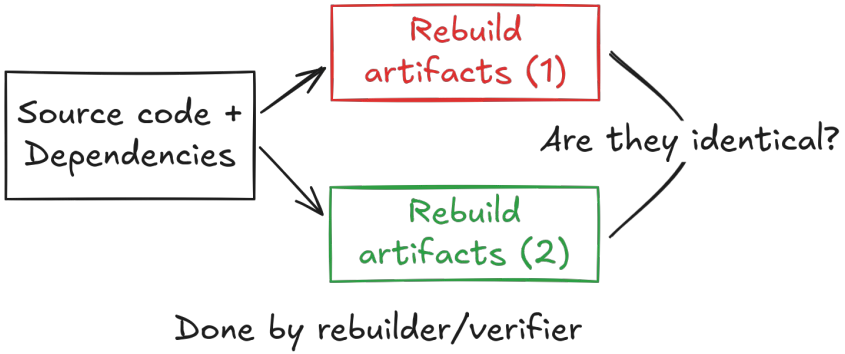
[2] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. IEEE Software 39, 2 (March 2022), 62–70. https://doi.org/10.1109/MS.2021.3073045

`xz-utils` mitigation



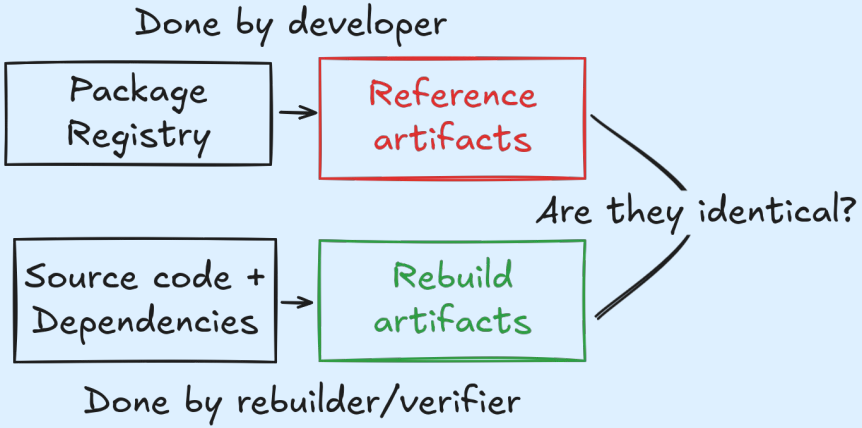
How to check for reproducible builds?

Build twice and compare



[3] G. Benedetti et al., 'An Empirical Study on Reproducible Packaging in Open-Source Ecosystems', 57th International Conference on Software Engineering, 2025.

Build and compare with package registry



[1] J. Malka, S. Zacchiroli, and T. Zimmermann, 'Does Functional Package Management Enable Reproducible Builds at Scale? Yes', in 22nd International Conference on Mining Software Repositories, Ottawa

Why is it important?

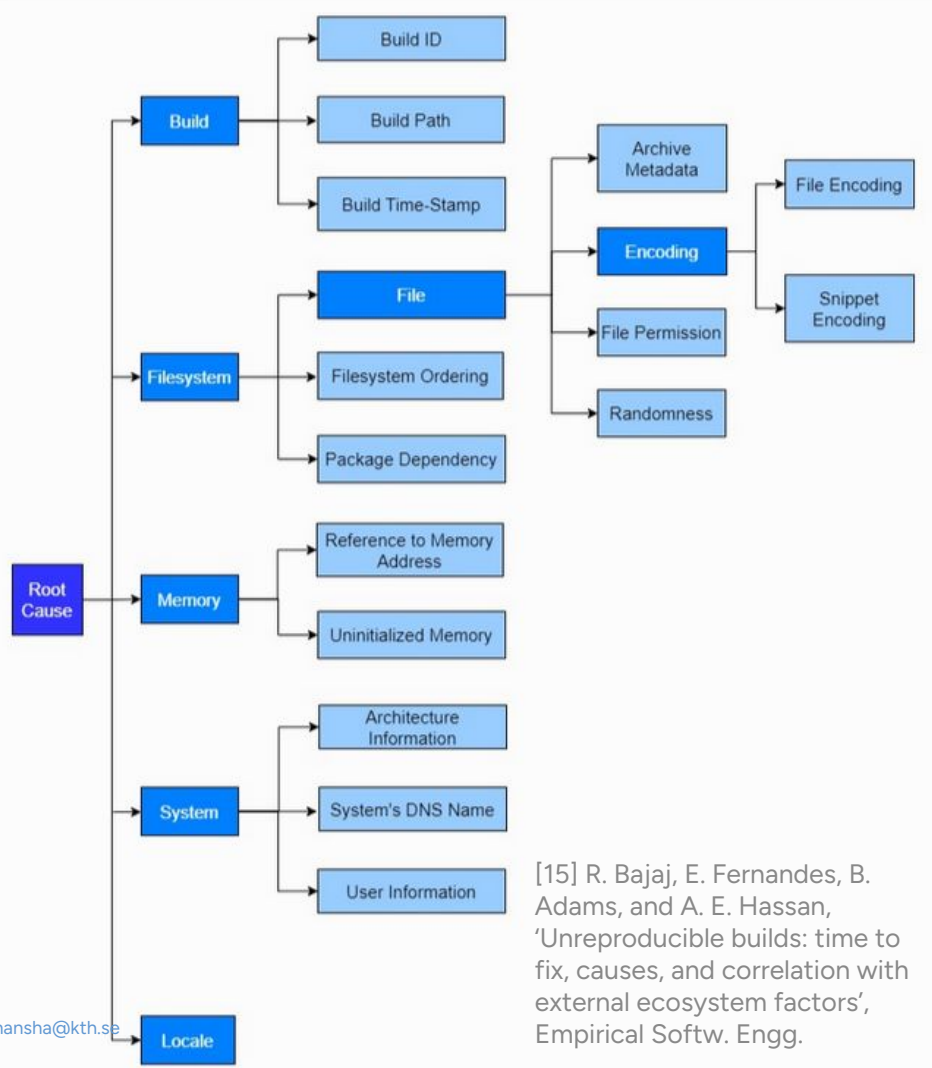
1. Ensuring Integrity: Reproducible builds ensure that the executable corresponds to the source code (assuming source code can be audited) and hence is not tampered with. [11]

2. Faster builds: Dependent sources do not need to be rebuilt and dependent tasks do not need to be rerun if a rebuild of a package does not yield different results. [12]

[11] 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/
[12] https://reproducible-builds.org/

Why aren’t builds reproducible?

Short answer: Timestamps, signature embedded



[15] R. Bajaj, E. Fernandes, B. Adams, and A. E. Hassan, ‘Unreproducible builds: time to fix, causes, and correlation with external ecosystem factors’, Empirical Softw. Engg.

Aman Sharma | amansha@kth.se

Our first goal

To build a taxonomy of reasons why unreproducible builds occur in Java and propose a fix for them

Dataset

We build our dataset on top of Reproducible Central - it is an infrastructure to verify whether Maven projects are reproducible.

Snapshot Reproducible Central on 8th October, 2024 (d280bf1) and we got **8,292 Maven releases** which gives us around 12K pairs of artifacts.

```
groupId=io.github.chains-project
artifactId=maven-lockfile
version=5.10.0
gitRepo=https://github.com/chains-
project/maven-lockfile.git
gitTag=${artifactId}-${version}
tool=mvn
jdk=17
newline=lf
command="mvn clean package
-DskipDepClean -DskipTests
-Dmaven.javadoc.skip -Dgpg.skip"
```

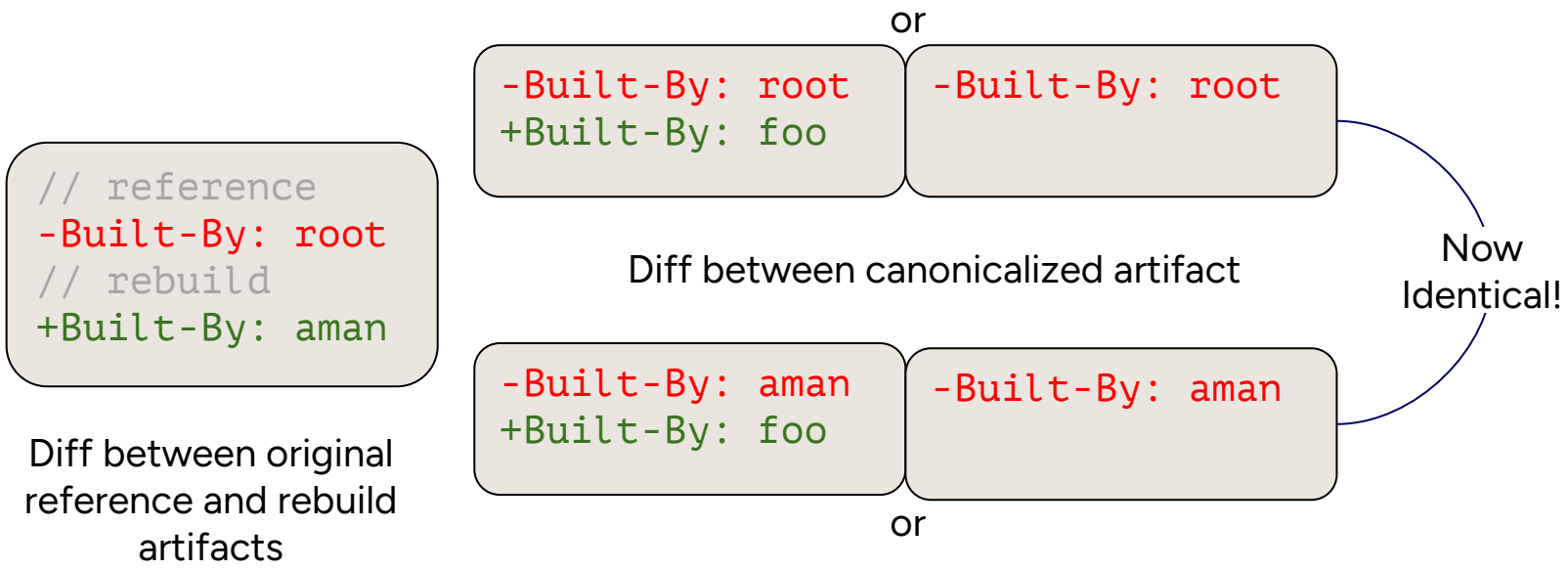
Example: Build Manifest

```
// org.apache.camel:came-ahc-ws:3.13.0
// Built-By: name of user that built the
artifact
-Built-By: root
+Built-By: aman
```

Mitigation: Fix build process or canonicalize

```
// org.apache.any23:apache-any23-csvutils:
2.7
-Specification-Title: Apache Any23 :: CSV Utilities
-Specification-Version: 2.7
-Specification-Vendor: The Apache Software Foundation
-Implementation-Title: Apache Any23 :: CSV Utilities
-Implementation-Version: 2.7
-Implementation-Vendor: The Apache Software Foundation
... some more attributes
```

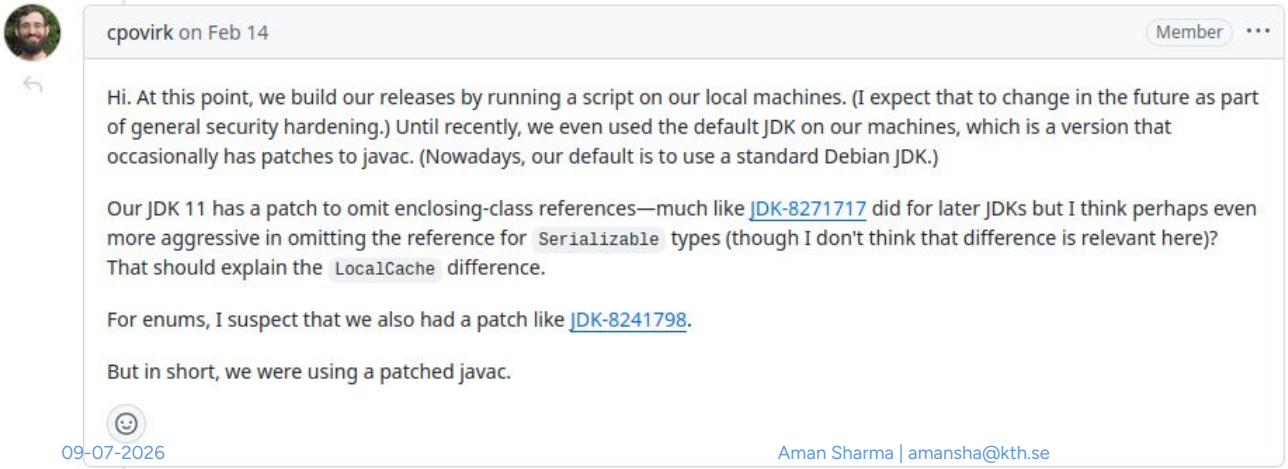
Build Manifest mitigation



JVM bytecode

```
private static com.google.common.base.Functions$IdentityFunction[] $values();
  descriptor: ()[Lcom/google/common/base/Functions$IdentityFunction;
  flags: (0x100a) ACC_PRIVATE, ACC_STATIC, ACC_SYNTHETIC
  Code:
    stack=4, locals=0, args_size=0
       0: iconst_1
       1: anewarray     #1                  // class com/google/common/base/Functions$IdentityFunction
       4: dup
       5: iconst_0
       6: getstatic     #2                  // Field INSTANCE:Lcom/google/common/base/Functions$IdentityFuncti
       9: aastore
      10: areturn
    LineNumberTable:
      line 90: 0
```

Mitigation: Hard to find JDK in such cases. It is better to consider two implementations equivalent (canonicalize).



cpovirk on Feb 14 — Member

Hi. At this point, we build our releases by running a script on our local machines. (I expect that to change in the future as part of general security hardening.) Until recently, we even used the default JDK on our machines, which is a version that occasionally has patches to javac. (Nowadays, our default is to use a standard Debian JDK.)

Our JDK 11 has a patch to omit enclosing-class references—much like JDK-8271717 did for later JDKs but I think perhaps even more aggressive in omitting the reference for `Serializable` types (though I don't think that difference is relevant here)? That should explain the `LocalCache` difference.

For enums, I suspect that we also had a patch like JDK-8241798.

But in short, we were using a patched javac.

[19] https://github.com/google/guava/issues/6321

First Goal: Causes of Unreproducibility

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

Table 1: Summary of the taxonomy of unreproducibility causes

First Goal: Causes of Unreproducibility

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
				process
				alization by rebuilder
Filesystem				alization by rebuilder
				ild process
JVM Bytecode				ild process
				process
				alization by rebuilder
Versioning Properties				alization by rebuilder
Timestamps	-	-	Embedded in shell script in Jar files	Fix build process

Table 1: Summary of the taxonomy of unreproducibility causes

Takeaway: 6 main reasons of unreproducibility in Java

Our second goal

Analyze effectiveness of canonicalization in mitigating unreproducible builds

What is this term “canonicalization”?

Inspired by canonical URL. To choose the preferred, representative URL for a web page, helping search engines understand which version of a page should be indexed [20].

In our context, we create a preferred representation of the artifact which abstracts away non-deterministic differences.

1. Removing parts of artifacts
2. Setting fixed values

[20] https://support.google.com/webmasters/answer/10347851?hl=en

Artifact Canonicalization

It means that the entire artifact is transformed by removing non-deterministic and spurious changes, especially in metadata.

Eg. ordering of files in archive is made consistent

Tool used: OSS-Rebuild [21]

Bytecode Canonicalization

It is a process of transforming the bytecode of a program into a representation that is independent of specific implementation details inserted by the compiler.

Eg. implementation of string concatenation pre and post Java 9

Tool used: jNorm [22]

[22] https://github.com/google/oss-rebuild
[23] S. Schott, S. E. Ponta, W. Fischer, J. Klauke, and E. Bodden, 'Java Bytecode Normalization for Code Similarity Analysis'

Second Goal: Effectiveness of Canonicalization

12,803 = artifacts out of which 936 have bytecode.

# Artifact	
# Bytecod	

Takeaway: Only a few percentage of artifacts are actually reproducible.

Reasons of f

It is hard to know what is equivalent and what to remove

Third goal

Integrate improvements in state of the art tool OSS Rebuild

google/**oss-rebuild**

Securing open-source package ecosystems by originating, validating, and augmenting build attestations.

27 Contributors

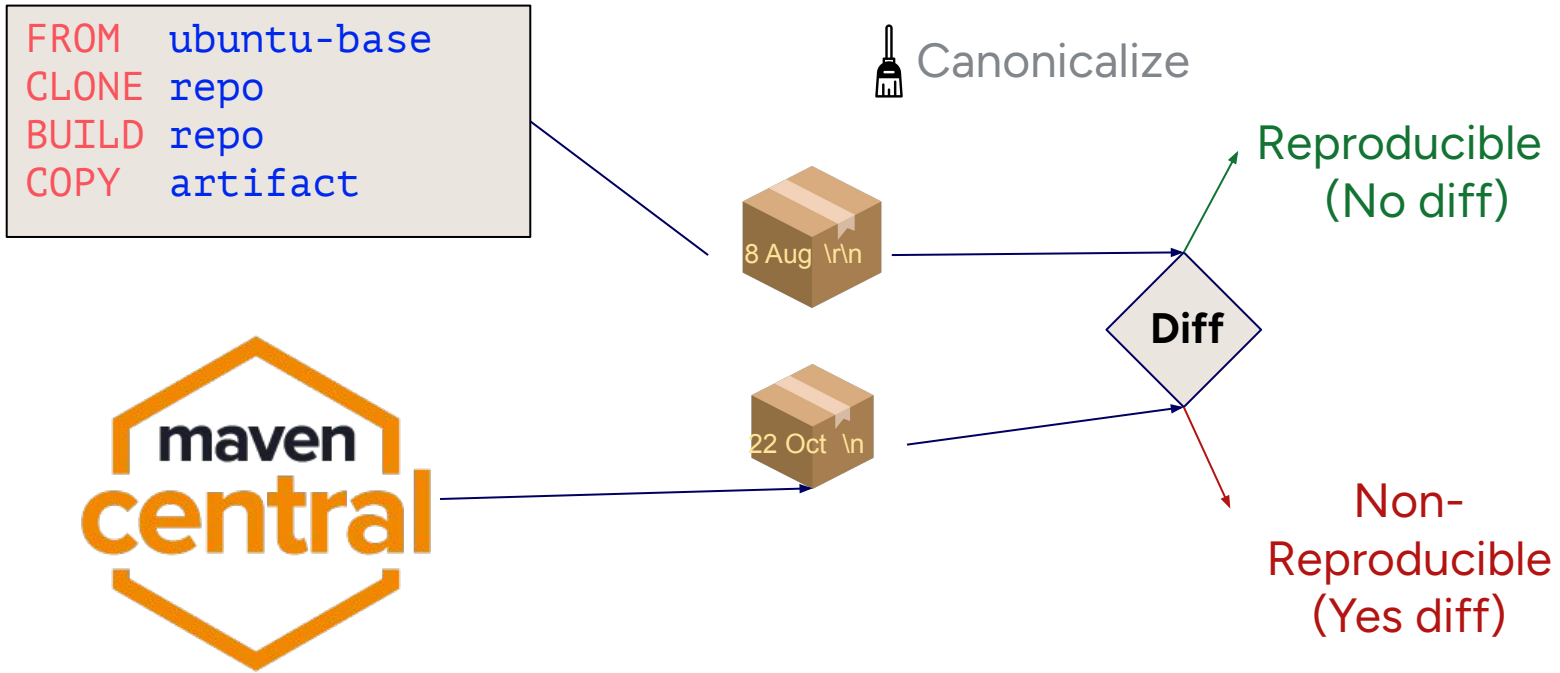
73 Issues

704 Stars

59 Forks

What is OSS Rebuild?

Rebuilder infrastructure for applying reproducible builds at scale.



Third Goal: Effectiveness after Improvments

# Before	
# After	

Takeaway: Improved the state of the art of canonicalization.

Future Work

1. Correctness of canonicalization.
2. Distributed architecture for rebuild attestations.
3. Asking LLMs to make builds reproducible.

```
8   var WrongToolchain = &genai.Part{
9       Text: `The category of the issue is "Wrong Toolchain".
10  This happens because the build logs complain
11  The logs can convey this error is multiple ways.
12  1) "Cannot find a Java installation on your machine matching "
13  2) "No matching toolchains found for requested specification"
14  3) "Java compilation initialization error invalid source/target release:"
15  4) "requires Java X to run. You are currently using Java Y."
16  5) maven-toolchains-plugin errors "Cannot find matching toolchain definitions"
17  6) "Detected JDK Version: X is not in the allowed range [Y,)."
18  7) "Source option 5 is no longer supported. Use 6 or later." This happens because
    Java version used in the build logs.`,
19  }
```

[18] J. Malka and A. Engelen, 'Lila: Decentralized Build Reproducibility Monitoring for the Functional Package Management Model', Jan. 28, 2026

Conclusion

1. A comprehensive taxonomy of unreproducible builds in Java and their mitigation.

2. Artifact and bytecode canonicalization in conjunction can be used to mitigate unreproducibility.

Paper:

[Causes and Canonicalization of Unreproducible Builds in Java]

Published in IEEE Transactions on Software Engineering

https://ieeexplore.ieee.org/document/11223991

Causes and Canonicalization of 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 (improvements to OSS-Rebuild), a tool that raises reproducibility success from 9.48% to 26.60% on 12,803 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, 56], 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 [58] 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, 61]: 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 [25]. 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 [41]. 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 and privacy critical environments, such as Bitcoin and Tor Browser [28]. Ensuring that clients for both of them are free of any backdoors injected by the build system.

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 create a new attack surface. Several high-profile supply chain attacks have exploited unreproducible builds. For instance, in 2024 it was discovered that the official tarballs of XZ Utils, compression library used in almost all Linux distributions, contained a backdoor that enabled remote code execution, even though the backdoor was absent in the project's Git repository [44]. Similarly, the Ultralytics Python package, a widely used library for AI models, was compromised when attackers injected malicious code into the build pipeline, resulting in a PyPI release with a backdoor [26]. A related incident targeted the Solana ecosystem as well [7]. These incidents share a common vector: the artifacts distributed through registries or release tarballs differed from what could be obtained by building directly from the publicly available source code. If the build processes for these projects had been reproducible, users or third party rebuilders could have verified that the distributed binaries and packages matched the source. Any discrepancies would have immediately signaled tampering, thereby preventing such attacks [32].

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 [2], underscoring its widespread use and relevance. Its importance in the context of finance [36, 56], government services [51], and military applications [3] [4] 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

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

[2] https://innovationgraph.github.com/global-metrics/programming-languages
[3] http://pdf.cloud.opensystemsmedia.com/vita-technologies.com/Aonix.Jun07.pdf
[4] https://tsri.com/news-blog/press/u-s-department-of-defense-mainframe-cobol-to-java-on-aws