Causes and Mitigations of Unreproducible Builds in Java

Canonicalization for
Unreproducible Builds in Java

Aman Sharma, Benoit Baudry, Martin Monperrus

Who am I?

Indian Institute of Technology Roorkee, India

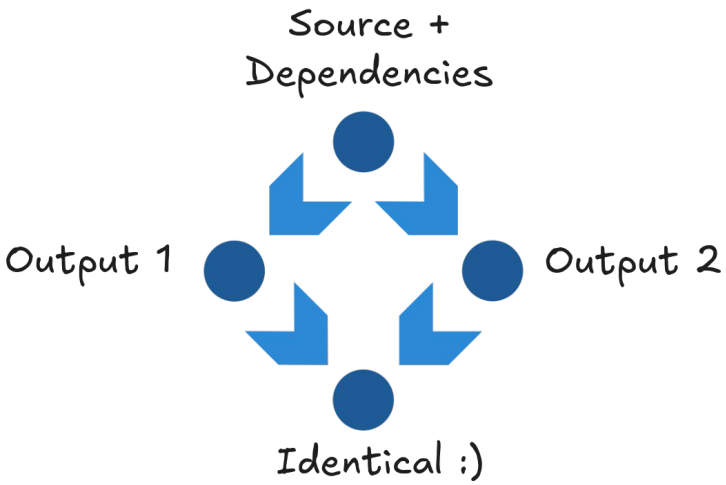
- Received Bachelor of Technology in 2021
- Was a part of Information Management Group

KTH Royal Institute of Technology, Sweden

- Joined as a Research Engineer in 2021
- Worked on Sorald, Collector-Sahab SBOM.exe, and other projects in the research group
- Switched to a PhD student in February 2023
- Funded by CHAINS to work on Software Supply Chain Security
- Supervised by Martin Monperrus and Benoit Baudry

What is Reproducible Builds?

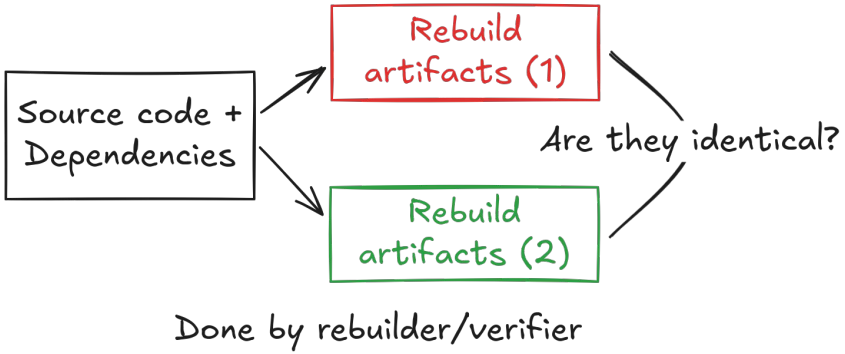
Build Reproducibility is a property of a software build process where the output artifact is bit-by-bit identical when built again, given a fixed version of source code and build dependencies, regardless of the environment [1].



[1] 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

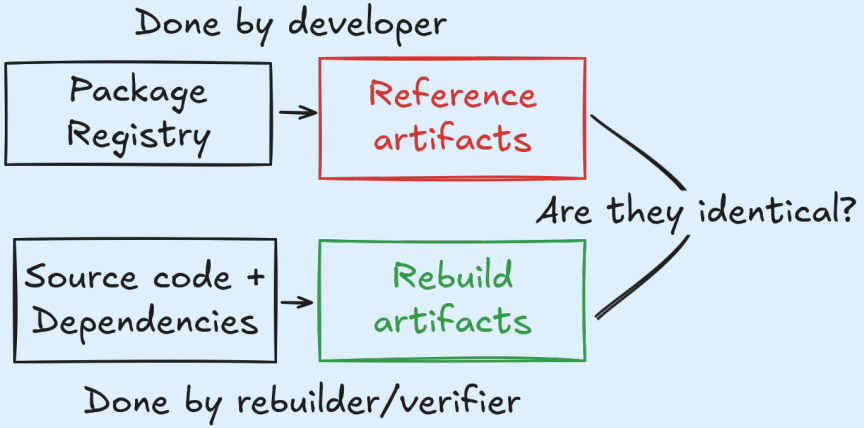
How to check for reproducible builds?

Build twice and compare



[2] 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



[3] 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

Related Work: build twice and compare

[4] G. Benedetti *et al.*, 'An Empirical Study on Reproducible Packaging in Open-Source Ecosystems', presented at the Proceedings of the 47th International Conference on Software Engineering, 2025.
Summary: Checking if builds are reproducible for RubyGems, PyPI, Maven.

[5] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, 'Automated localization for unreproducible builds', in *Proceedings of the 40th International Conference on Software Engineering, 2018*
Summary: Localizing files that differ in two subsequent builds for Debian packages.

[6] O. S. Navarro Leija *et al.*, 'Reproducible Containers', in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020
Summary: Wrapper around the build process for debian packages.

Related Work: build and compare with package registry

[7] 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, 2025.

Summary

We like this approach more as it maximizes environment differences!

[8] R. Baja … o fix, causes, ar … 023.

Summary … Debian packages

[9] V. Andersson, Geth Rebuild : Verifiable Builds for Go Ethereum. 2024

Summary: Rebuilds ethereum client that enables connection to the main ethereum network.

Reproducible/Verifiable/Accountable builds

Verifiable Builds: Builds are verifiable if the build process of a software can be fixed to output bit-by-bit identical artifacts.

Accounta outputs unequal a

Basically, eproducible Builds in c

We always prefer “Reproducible Builds” over other terms.

[10] T. Pohl, P. Novák, M. Ohm, and M. Meier, ‘SoK: Towards Reproducibility for Software Packages in Scripting Language Ecosystems’, Mar. 27, 2025, arXiv: arXiv:2503.21705. doi: 10.48550/arXiv.2503.21705.

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 packages 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]

3. Patch updates: Only changes in source code (or dependencies) will lead to differences in the generated binaries thus reducing storage requirements. [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 is it important? XZ Utils backdoor example

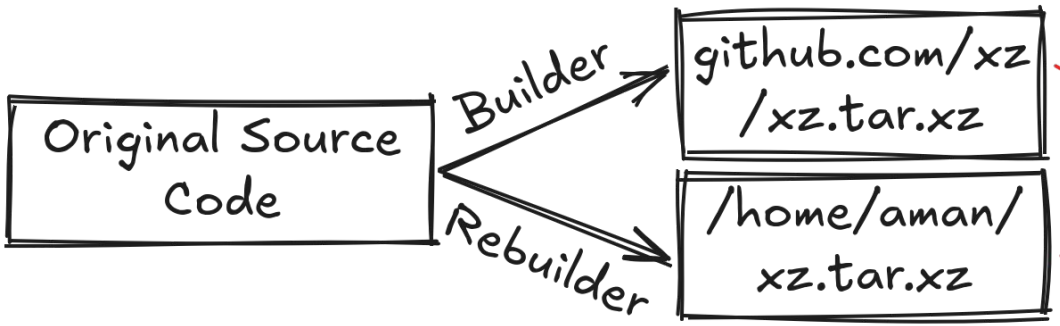
What?

Malicious code was introduced in the Linux build (tarball) of `xz` that enabled remote SSH access to the attacker.

How serious?

Extremely. Used in xz compression tool that outputs .tar.xz. But it was detected when new versions of Linux distros were in development phase.

How can reproducible builds detect such attacks?



[13] 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

Are builds reproducible yet?

Arch Linux is **87.7%** reproducible with **1849 bad** **4 unknown** and **13164 good** packages.

[core] repository is **95.1%** reproducible with **13 bad** **0 unknown** and **252 good** packages.
[extra] repository is **87.3%** reproducible with **1800 bad** **0 unknown** and **12392 good** packag
[extra-testing] repository is **92.8%** reproducible with **36 bad** **4 unknown** and **519 good** packa
[core-testing] repository is **100.0%** reproducible with **0 bad** **0 unknown** and **1 good** packages.

every 4.1 months from 2017 to 2023. Our findings show that bitwise reproducibility in nixpkgs is very high and has known an upward trend, from 69% in 2017 to 91% in 2023. The mere ability to rebuild packages (whether bitwise reproducibly or

Finding 1: *1,303 out of 3,390 studied versions (38%) are non-reproducible. With such a large portion of non-reproducible package versions, developers should not blindly trust the verifiability of NPM packages.* [14]

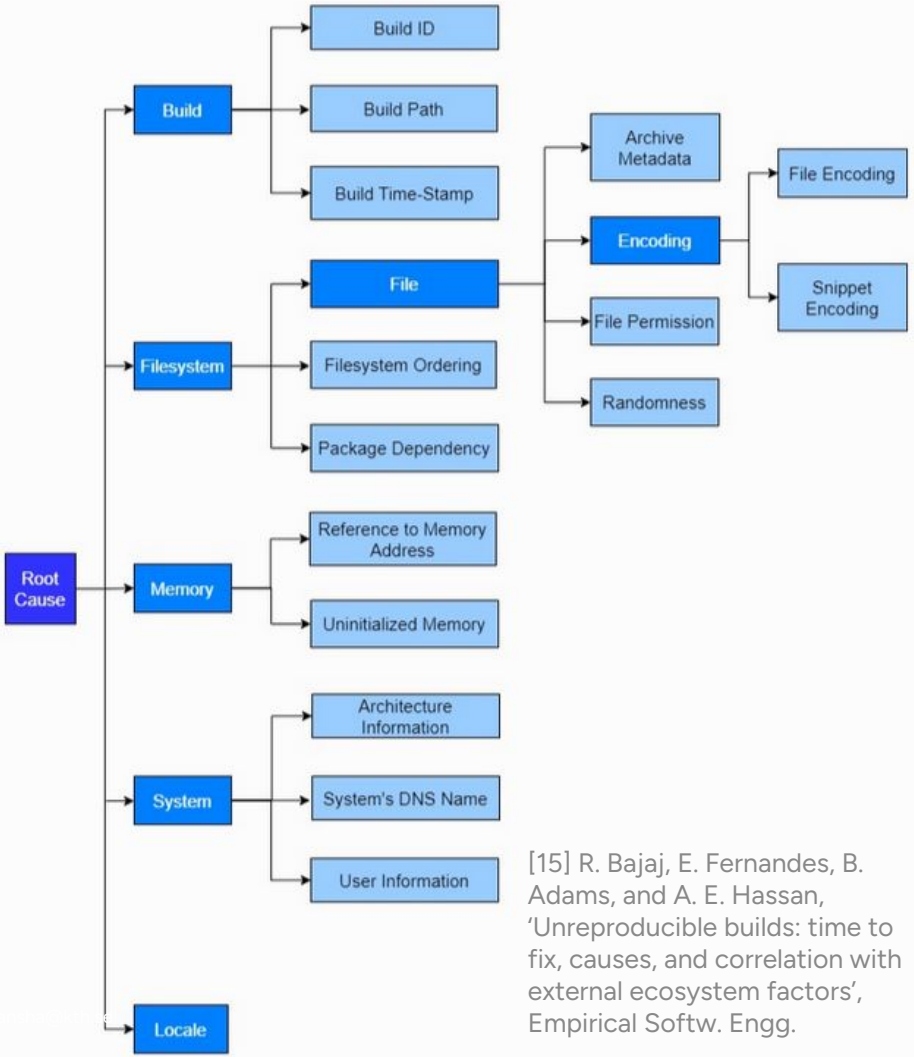
rebuilding **7080 releases** of **869 projects**:

- **5112** releases are confirmed **fully reproducible** (100% reproducible artifacts ✅),
- 1968 releases are only partially reproducible (contain some unreproducible artifacts ⚠️)
- on 869 projects, 755 have at least one fully reproducible release, 114 have none

[14] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao, 'Investigating The Reproducibility of NPM Packages', in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)

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.

What is the status of reproducible builds in Java?

Most closely related work:

[16]J. Xiong, Y. Shi, B. Chen, F. R. Cogo, and Z. M. (Jack) Jiang, 'Towards build verifiability for Java-based systems', in Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, in ICSE-SEIP '22

1) Follows "build twice and compare" 2) Small and close sourced dataset

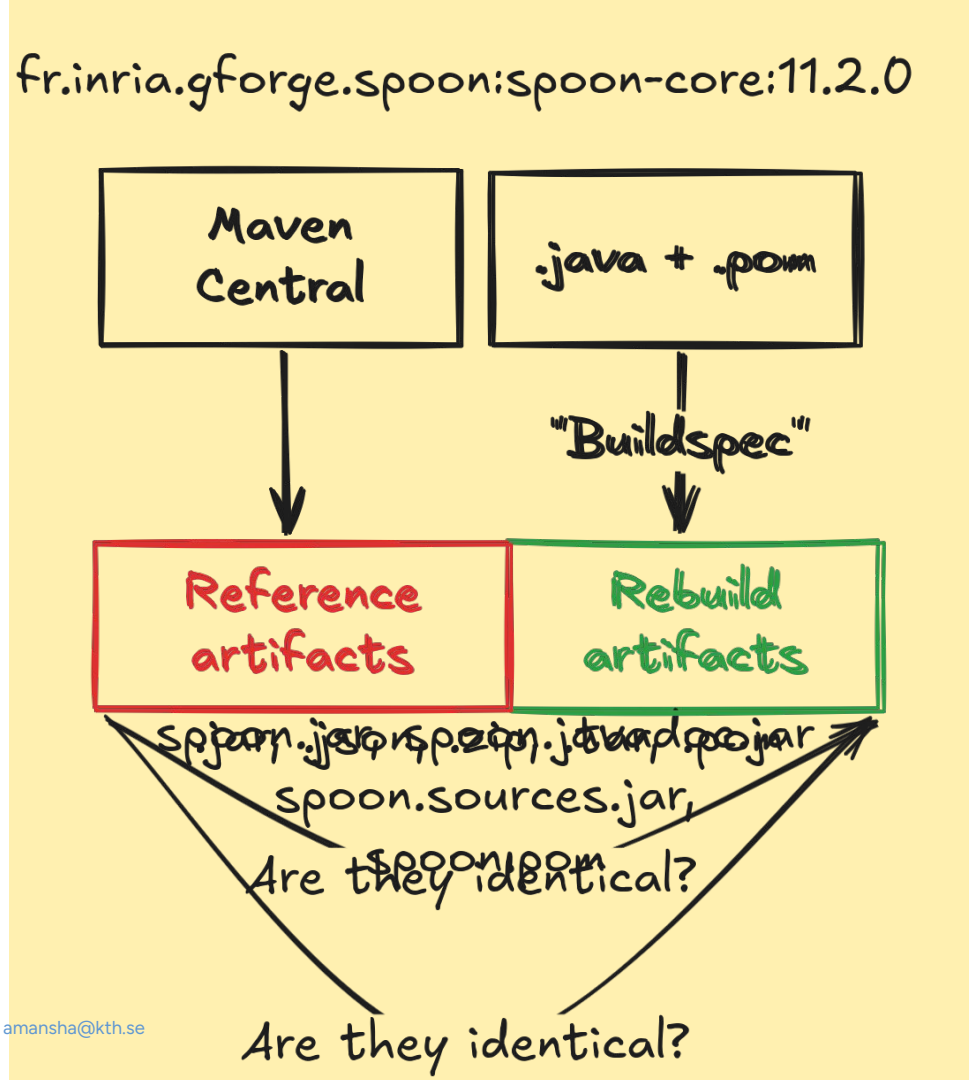
rebuilding **7080 releases** of **869 projects**:

- **5112** releases are confirmed **fully reproducible** (100% reproducible artifacts ✅),
- 1968 releases are only partially reproducible (contain some unreproducible artifacts ⚠️)
- on 869 projects, 755 have at least one fully reproducible release, 114 have none

[16] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao, 'Investigating The Reproducibility of NPM Packages', in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)

Reproducible Central

1. It is a GitHub repository: https://github.com/jvm-repo-rebuild/reproducible-central/
2. Artifacts like jar, .json, .zip files are downloaded from Maven Central.
3. Source code is downloaded from hosting services like GitHub.
4. The source code along with all of its build requirement is built using a “buildspec”.
5. Buildspec file is specification that lists how the build should done
   a. Build command
   b. Java version
   c. Environment variables
6. If corresponding reference and rebuild artifacts are identical, project is marked reproducible.



[17] H. Boutemy, jvm-repo-rebuild/reproducible-central. (Oct. 31, 2024). jvm-repo-rebuild.

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.

1. Snapshot Reproducible Central on 8th October, 2024 (d280bf1)
   a. 706 Maven projects and their buildspec files

2. We apply four filters that resulted in exclusion of projects
   a. Projects that require manual intervention (eg. setting precise Java version).
   b. Projects that have more artifacts built locally than on package registry (eg. some jars are are built locally but never released).
   c. Projects that do not use Maven as build tool.
   d. Projects that mapped artifacts incorrectly (eg. pom file is compared with jar).

3. Finally, we have
   a. 7,961 Maven releases **of submodules**
   b. 12,283 unreproducible artifact pairs

RQ1: 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

Mitigation Strategies

There are 3 mitigation strategies

1. Fix the build process: repairing the build script used by developer to publish package
   a. Pros: anyone can replicate the build with 0 ad-hoc configuration
   b. Cons: sometimes you have to embed non-deterministic information like signatures

2. Fix the rebuild process: repairing the buildspec file
   a. Pros: contributes to fixes in the build process
   b. Cons: can be hard to adapt for future releases

3. Canonicalization: conversion of output artifacts into a standard representation that is free of non-deterministic changes
   a. Pros: past releases can also be verified, no need to rebuild, “i can’t find a jdk 14 nowadays”
   b. Cons: careful removal as it may mask meaningful differences

We try to evaluate canonicalization since few work in the literature

[18] J. Dietrich, T. White, B. Hassanshahi, and P. Krishnan, ‘Levels of Binary Equivalence for the Comparison of Binaries from Alternative Builds’

Example: Build Manifest

```
// rg.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

```
// reference
-Built-By: root
// rebuild
+Built-By: aman
```

Diff between original reference and rebuild artifacts

or

```
-Built-By: root
+Built-By: foo
```

```
-Built-By: root
```

Diff between canonicalized artifact

Now Identical!

```
-Built-By: aman
+Built-By: foo
```

```
-Built-By: aman
```

or

Example: Software Bill of Materials

```
// org.apache.commons:commons-parent:61

-           <hash alg=\"SHA3-384\">289f952[TRUNCATED]</hash>
-           <hash alg=\"SHA3-256\">32b50a7[TRUNCATED]</hash>
-           <hash alg=\"SHA3-512\">1daf813[TRUNCATED]</hash>
```

Reason: Azul JDK had backported these hash algorithms.

Mitigation: Fix build process by correcting the Java vendor

Example: Filesystem

```
// io.github.albertus82:unexepack:0.2.1
// change in file permissions and owner
--rw-r--r--  root unexepack-0.1.1/unexepack.jar
+-rw-rw-r--  aman unexepack-0.1.1/unexepack.jar
```

Mitigation: Set permissions and owner to fixed value (canonicalization)

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.

Versioning Properties

```
// com.github.ldapchai:ldapchai:0.8.6
// Embedded in MANIFEST.MF
-SCM-Git-Branch: master
+SCM-Git-Branch: 338023a
```

```
// org.apache.drill:drill-opentsdb-storage:1.21.0
// Embedded in git.properties
-git.tags=drill-1.21.0
+git.tags=drill-1.21.0,drill-1.21.1,drill-1.21.2
```

Mitigation: Remove such attributes (canonicalization).

Timestamps

```
-#Wed Apr 20 20:27:41 CEST 2022
+#Fri Oct 18 03:03:44 UTC 2024
```

1. they can be embedded in the properties file
2. generated documentation
3. shell scripts appendix
4. executable binaries
5. software bill of materials
6. JVM bytecode
7. file metadata
8. MANIFEST.MF
9. NOTICE
10. servlets created by Jasper JSP compiler

Mitigation: Add `project.build.outputTimestamp` property to pom so that plugins can respect it (fixing build process)

RQ1: 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 [19].

Also, used in previous work to detect allowlisted classes at runtime.

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

Bytecode Canonicalization

- Classnames could change across different executions.
- The type references change.
- The order of method is not fixed.

```
- public class $Proxy10 {
+ public class $Proxy7 {
-    private static $Proxy10.x;
+    private static $Proxy7.x;
-   m1 () {}
+   m3 () {}
-   m3 () {}
+   m1 () {}
}
```

[20]

[19] https://support.google.com/webmasters/answer/10347851?hl=en
[20] A. Sharma, M. Wittlinger, B. Baudry, and M. Monperrus, ‘SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java’, Jun. 28, 2024, arXiv: arXiv:2407.00246.

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]

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

RQ2: Effectiveness of Artifact Canonicalization

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: Results of OSS-Rebuild and Chains-Rebuild on 12,283 artifacts.

OSS-Rebuild: Tool by Google to canonicalize software artifacts.

CHAINS-Rebuild: Fork of OSS-Rebuild with support for canonicalizing build manifests and embedded versioning properties.

RQ2: Effectiveness of Artifact Canonicalization

Tool	Successful Canonicalization	Failed Canonicalization
OSS-R…		
CHAIN…		

Table 2… …on
12,283

Takeaway: 2.5x increase in canonicalized artifacts with only a few improvements.

OSS-Rebuild…

CHAINS-Rebuild: Fork of OSS-Rebuild with support for canonicalizing build manifests and embedded versioning properties.

RQ3: Effectiveness of Bytecode Canonicalization

We select a subset of unreproducible artifacts that have JVM bytecode changes.

This is 898/12,283 artifact pairs.

	Successful canonicalization	Failure in canonicalization	Error in canonicalization
#	267 (29.7%)	478 (53.2%)	153 (17.1%)

Reasons of failure:

1. Structural Changes limitation (eg. changes in order of methods, fields, etc)
2. Control flow limitation (eg. ifne and ifeq)
3. Embedded data (eg. absolute file paths)
4. Optimization limitation (eg. string concatenation)

Example diff for jNorm and oss-rebuild

```
-    99: ifne        106
-   102: iconst_1
-   103: goto        107
-   106: iconst_0
-   107: ireturn
+    99: ifeq        104
+   102: iconst_0
+   103: ireturn
+   104: iconst_1
+   105: ireturn
```

Listing 1: javap diff for control flow difference.

```
-if v != 0 goto label;
-v = 1;
-goto label;
-label:
-v = 0;
+if v == 0 goto label;
+return 0;
label:
-return v;
+return 1;
```

Listing 2: JNORM diff for control flow differnce.

RQ3: Effectiveness of Bytecode Canonicalization

We select a subset of unreproducible artifacts that have JVM bytecode changes.

This is 898/12,283 artifact pairs.

#	

Takeaway: 4 categories of improvements for bytecode canonicalization.

Reasons of f

1. Structu
2. Control flow problems (eg. ifne and ifeq)
3. Embedded data (eg. absolute file paths)
4. Optimization problem (eg. string concatenation)

Future Work

1. Formalize the notion of acceptable canonicalization; what to remove and what to keep?
2. Correctness of canonicalization via running tests.
3. More features for canonicalization in OSS-Rebuild.
4. Integration of canonicalization into Reproducible Central infrastructure?

Central Repository groupId	artifactId(s)	versions	result: reproducible?
biz.aQute.bnd	bnd-maven-plugin	10	10 ✅
ch.galinet	reproducible-build-maven-plugin	5	5 ✅
ch.qos.logback	logback	59	44 ✅ / 15 ⚠️

After canonicalization
10 5 59/0

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.

Thank you!

Aman Sharma

amansha@kth.se

Paper:

Canonicalization for Unreproducible Builds in Java

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.com/AonixJun07.pdf
[5] https://tsri.com/news-blog/press/u-s-department-of-defense-mainframe-cobol-to-java-on-aws