Unreproducible Builds in Java: Causes and Mitigations

Aman Sharma, Benoit Baudry, Martin Monperrus

What is Reproducible Builds? Why is it important?

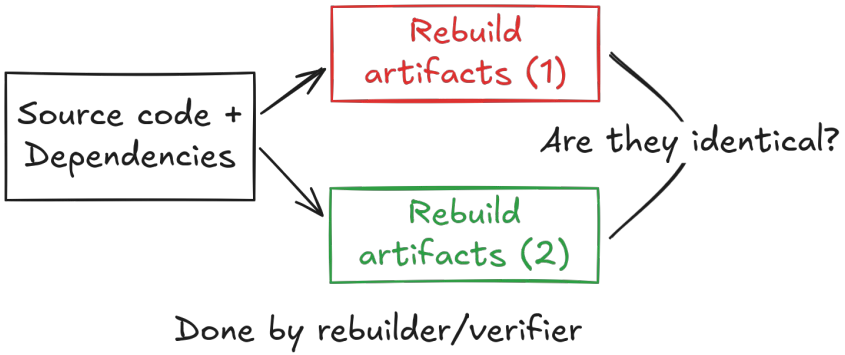
Build Reproducibility is a property of a software build process where the output artifact is bit-by-bit identical when built again, given a fixed version of source code and build dependencies, regardless of the environment [1].

1. 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. [2]
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. [3]
3. Patch updates: Only changes in source code (or dependencies) will lead to differences in the generated binaries thus reducing storage requirements. [3]

[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
[2] 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/
[3] https://reproducible-builds.org/

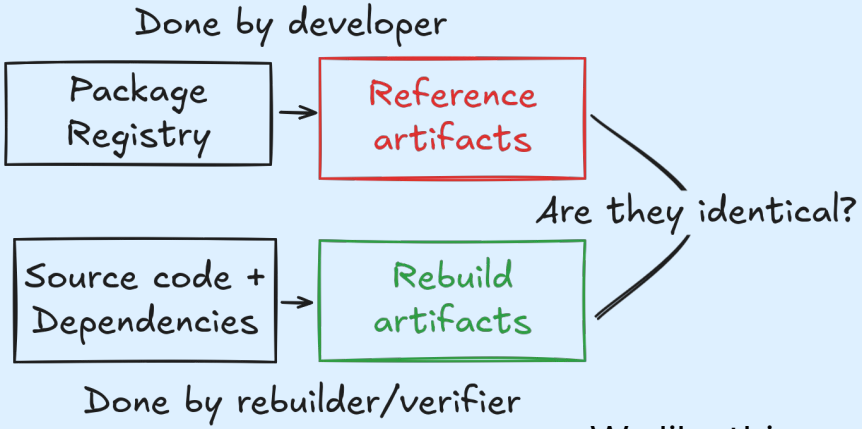
How to check for reproducible builds?

Build twice and compare



[4] 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



[5] 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

We like this approach more as it maximizes environment differences!

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.* [6]

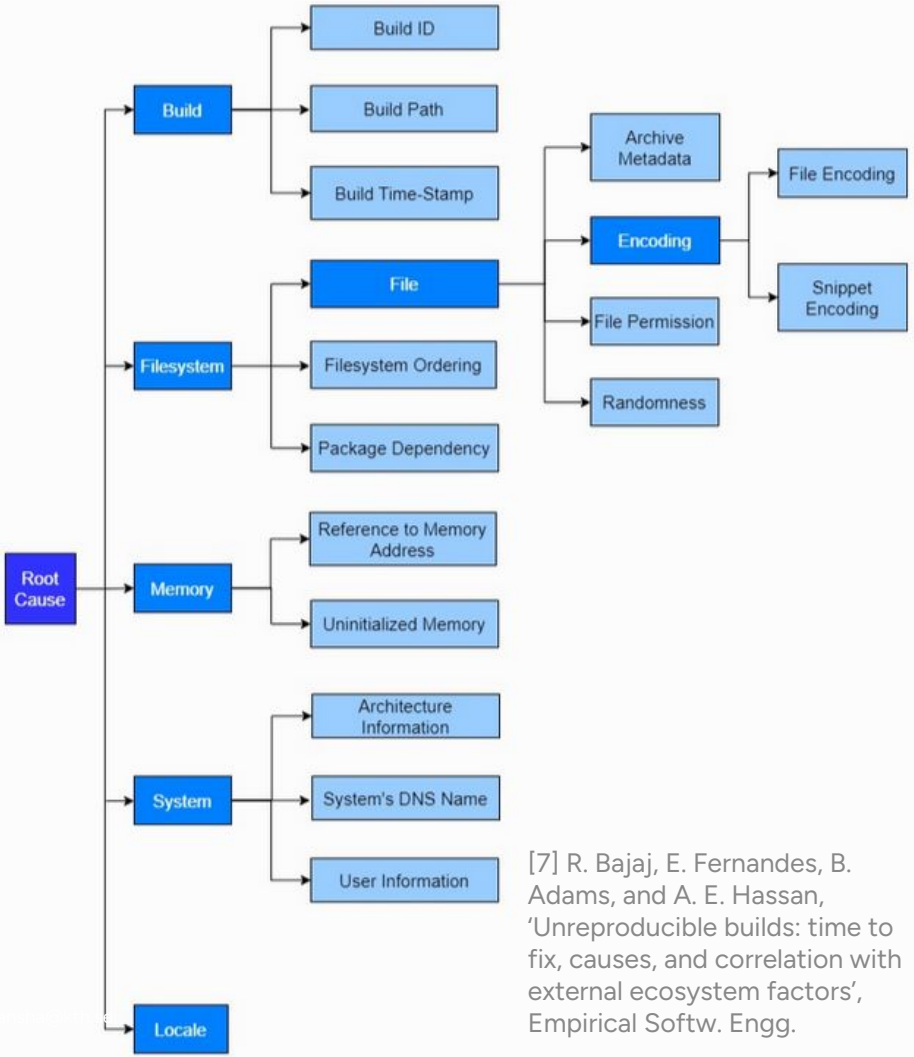
rebuilding **7080 releases** of **869 projects**:

- **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

[6] 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



[7] 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?

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

Most closely related work:

[8]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

Small and close sourced dataset

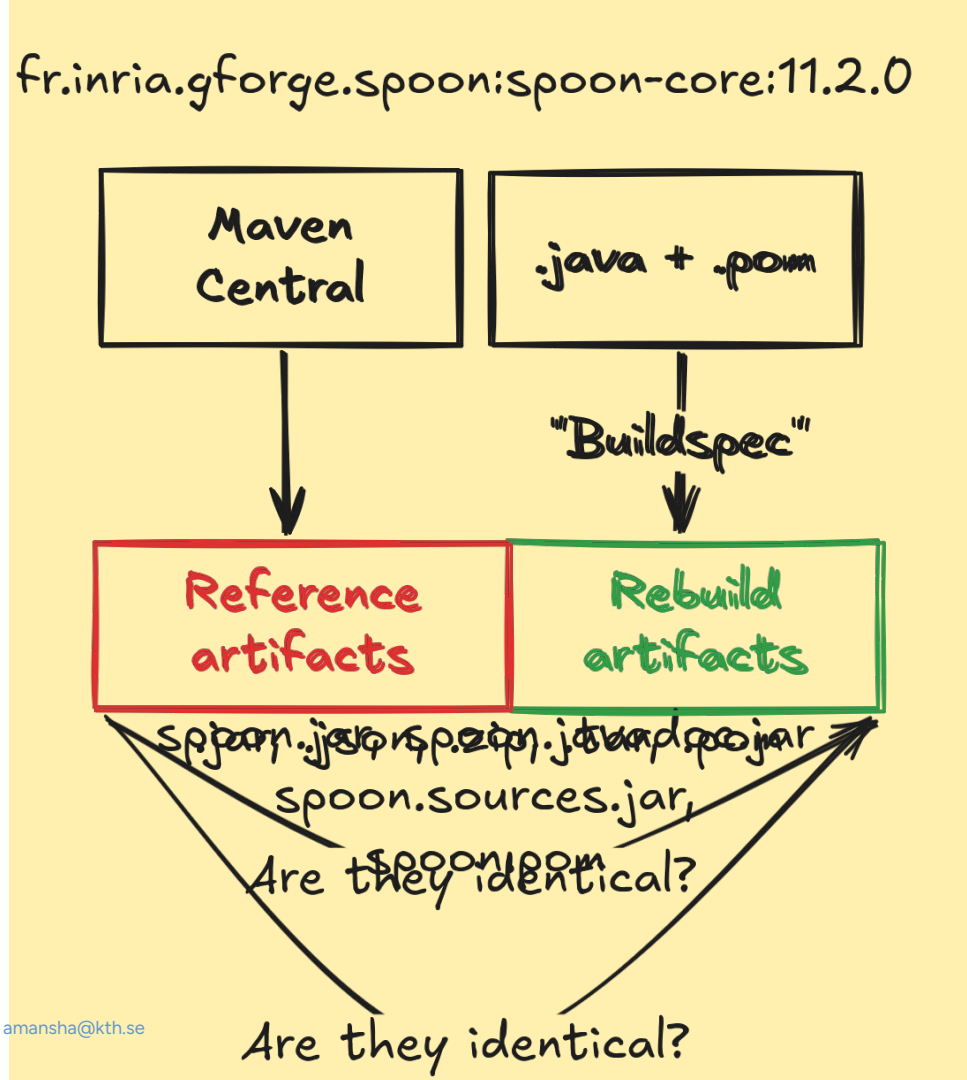
[8] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao, 'Investigating The Reproducibility of NPM Packages', in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)

Our first goal

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

Reproducible Central

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



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

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 releases).
   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 artifact pairs

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
	Custom release configuration	✓	Releasing a subset of artifacts	Fix build process
	Non-deterministic information	✓	Timestamp	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: Summary of the taxonomy of unreproducibility causes

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

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

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

Our second goal

Analyze effectiveness of canonicalization in mitigating reproducible builds

Aman Sharma | amansha@kth.se

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 [10]

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 [11]

[10] https://github.com/google/oss-rebuild
[11] 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
#	267 (29.7%)	487 (53.2%)

Reasons of failure:

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

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. More features for canonicalization in OSS-Rebuild.
3. Integration of canonicalization into Reproducible Central infrastructure?

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 Draft:
https://algomaster99.github.io/publications/by-the-pool/paper.pdf