

Maven-Lockfile: High Integrity Rebuild of Past Java Releases

Larissa Schmid
Elias Lundell
Martin Monperrus
{lgschmid,ellundell,monperrus}@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Yogya Gamage
Benoit Baudry
first.lastname@umontreal.ca
Université de Montréal
Montréal, Canada

Abstract

Modern software projects depend on many third-party libraries, complicating reproducible and secure builds. Several package managers address this with the generation of a lockfile that freezes dependency versions and can be used to verify the integrity of dependencies. Yet, Maven, one of the most important package managers in the Java ecosystem, lacks native support for a lockfile. We present MAVEN-LOCKFILE to generate and update lockfiles, with support for rebuilding projects from past versions. Our lockfiles capture all direct and transitive dependencies with their checksums, enabling high integrity builds. Our evaluation shows that MAVEN-LOCKFILE can reproduce builds from historical commits and is able to detect tampered artifacts. With minimal configuration, MAVEN-LOCKFILE equips Java projects with modern build integrity and build reproducibility, and fosters future research on software supply chain security in Java.

Demonstration: <https://youtu.be/eGgR3toBgxU>

CCS Concepts

• **Security and privacy** → **Software security engineering**.

Keywords

Lockfiles, Build Integrity, Software Supply Chain Security, Java

1 Introduction

Modern software projects depend on a large number of third-party libraries. These dependencies make development faster, but they also introduce new risks [5]. In the face of these risks, developers need tools to verify the integrity of dependencies, in order to have reproducible build results, avoid incompatible changes between dependency versions, and detect possible tampering of artifacts. Without such guarantees, tracking down errors due to wrong dependencies is hard and build results cannot be trusted.

Lockfiles are a practical way to address these problems. They record the exact versions and checksums of direct and transitive dependencies, making builds both reproducible and transparent. With lockfiles, developers can confidently compare dependency trees across builds and precisely control the effect of version ranges. Prior work shows that developers perceive lockfiles as helpful for improving the efficiency and security of dependency management [2]; for enabling reproducible installs [4], for preventing unwanted updates to untested or compromised versions [8], and for supporting the reliable generation of software bills of materials (SBOMs) [9]. Maven, one of the most widely used build tools in the Java ecosystem, does not provide native lockfile support. This gap limits the reproducibility and security of Maven-based builds, and prevents

the research community from investigating high-integrity builds for Java.

We here present MAVEN-LOCKFILE, which adds state-of-the-art lockfile support for Maven. Our tool generates lockfiles, validates and rebuilds projects based on them, bringing essential integrity features to Maven. Lockfiles produced by MAVEN-LOCKFILE contain versions of dependencies and their checksums, covering both direct and transitive dependencies. MAVEN-LOCKFILE is compatible with continuous development practices through a GitHub Action that automates the generation, validation, and update of lockfiles in Continuous Integration pipelines.

To sum up, our contributions are:

- A blueprint architecture for high-integrity builds in the context of Maven – one of the most important build systems in enterprise contexts.
- MAVEN-LOCKFILE, a tool¹ for ensuring the integrity of a Maven build and for guarding the build against malicious tampering with the artifacts.
- An experimental evaluation showing how MAVEN-LOCKFILE enables rebuilding old versions of a non-trivial, real-world Java project.

2 Lockfiles and Pinning

Modern software depends on many external libraries, which are managed and fetched using a package manager. To control the versions of these dependencies, one option is to explicitly fix the version of a direct dependency in the project’s manifest file, a practice sometimes referred to as “pinning”. However, pinning in manifest files has a clear limitation: it does not control the versions of transitive dependencies. These may still change silently, making builds unpredictable and harder to reproduce [4].

Lockfiles extend the idea of pinning by freezing the entire dependency graph, recording the exact versions of both direct and transitive dependencies as they are resolved by the package manager [2]. Lockfiles may also contain additional details, such as the version of the package manager used and checksums for each dependency package.

Lockfiles make builds reproducible and transparent, no matter when or where they are executed. They provide several important benefits: (i) **Deterministic builds**: The same dependency graph is installed every time the project is built, avoiding silent upgrades [8], (ii) **Integrity and security**: Checksums in lockfiles allow the verification of downloaded artifacts, reducing the risk of tampering [2], (iii) **Transparency and auditing**: Lockfiles enable comparing builds, enabling a full audit trail of changes in dependencies.

¹<https://github.com/chains-project/maven-lockfile/>

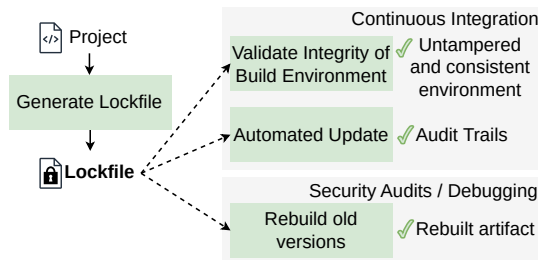


Figure 1: Overview of the unique, novel integrity features of MAVEN-LOCKFILE.

Maven is one of the most popular package managers in the Java ecosystem. In Maven, dependencies are declared in the project’s `pom.xml` file. Each dependency is identified by three attributes: `groupId`, `artifactId`, and `version`. The version may be fixed, but developers can also choose to declare it as the latest available version; or as a version range. When building a project, Maven resolves the dependencies of those packages: it determines which specific version to use for every dependency of a project. This process is fundamentally not deterministic. Even if a project’s own `pom.xml` specifies exact versions, the transitive dependencies may still rely on ranges or unspecified versions. As a result, different builds at different points in time may resolve to different versions, leading to builds that behave differently. This problem is compounded by the fact that resolved dependencies are largely invisible to developers. They are not aware that the versions of transitive dependencies have changed, which makes debugging and reproducing builds more difficult. In addition, Maven does not provide a built-in mechanism to ensure that downloaded artifacts have not been tampered with during distribution. These fundamental limitations highlight the need for proper lockfile support in Maven to guarantee determinism, integrity, and security of Maven builds.

3 Maven-Lockfile

MAVEN-LOCKFILE brings lockfiles to Maven (cf. Figure 1): It enables generating lockfiles (subsection 3.1), validating the integrity of build environments based on them (subsection 3.2), rebuilding old versions of a project (subsection 3.3), and automatically updating lockfiles (subsection 3.4), equipping Java projects with high integrity builds (subsection 3.5).

3.1 Generate Lockfiles

MAVEN-LOCKFILE can generate a lockfile that records the complete dependency graph of a Maven project, including checksums of all artifacts, by executing a single command. This allows developers to freeze and verify dependencies with both direct and transitive coverage. The lockfile is written in a human-readable JSON format and contains the full dependency tree, similar to a software bill of materials (SBOM). Figure 2 shows an excerpt of an example lockfile. MAVEN-LOCKFILE supports multi-module projects, creating one lockfile per module. For each artifact, the lockfile stores the hashes

```
"dependencies": [ { "groupId": "A", "artifactId": "A",
                    "version": "1.2", "checksum": "85bc8591c87" },
                  { "groupId": "D", "artifactId": "D",
                    "version": "3.4", "checksum": "f92d72b8720" }, ],
"mavenPlugins": [],
"metadata": { "environment": {
              "osName": "Linux", "mavenVersion": "3.8.2", "javaVersion": "21.0.5" }
},
"config": { "checksumMode": "local" }
```

Figure 2: Excerpt of an example lockfile.

of all transitive dependencies. This makes it possible to validate not only direct dependencies but also the entire chain of dependencies.

MAVEN-LOCKFILE also offers the possibility to add Maven plugins to the lockfile. Maven plugins are themselves artifacts downloaded from external repositories, forming part of the software supply chain and influencing the build result. If a plugin is replaced or tampered with, it could compromise the entire build process. Including plugins in the lockfile ensures that their versions and checksums are validated, protecting not only the project’s dependencies but also the integrity of the build system itself.

The checksums for lockfile generation can be resolved or retrieved using two different modes: *Remote* downloads the checksum directly from the Maven repository. *Local* calculates the checksum directly from the artifact stored in the local Maven cache. Remote checksums provide stronger guarantees that the downloaded artifact is the same as the one published by the repository, but this trust relies on the repository itself not being compromised. Local checksums ensure consistency with the artifact already downloaded, but they cannot detect tampering that happened before the artifact was downloaded.

Moreover, MAVEN-LOCKFILE can include environment metadata in the lockfile, which allows warnings when the build environment changes. As a change in the build environment can affect the outcome of a build, recording this information helps explain differences in build results and supports reproducibility across systems.

3.2 Validate Integrity of Build Environments

After generating a lockfile, MAVEN-LOCKFILE can validate the integrity of a Maven repository against an existing lockfile. This ensures that all dependencies used in the build are the same as when the lockfile was originally created, and that the local repository has not been tampered with. The validation process checks both versions and checksums of dependencies, covering the entire dependency graph. If the `validate` command runs successfully, it means that the build environment is valid; i.e., all defined dependencies match the lockfile. If the validation fails, it means that one or more dependencies have changed since the lockfile was generated, which could indicate either a silent update or a tampering. MAVEN-LOCKFILE also enables skipping specific modules to disable validation in selected parts of a multi-module project.

By validating dependencies against a lockfile, developers have guarantees that the build environment is untampered and consistent with the reference dependencies, no matter whether it is today or far in the future.

3.3 Rebuild Old Versions

MAVEN-LOCKFILE supports rebuilding projects from a lockfile with the freeze feature, making it possible to reproduce older versions of a build with the same dependency resolution. This is done by generating a new POM file, `pom.lockfile.xml`, that incorporates all dependency information from the lockfile. In the generated POM file, every version of the direct dependencies from the original POM is replaced with the versions recorded in the lockfile. In addition, all transitive dependencies are added to the `dependencyManagement` section, with both their version and scope taken from the lockfile. In this way, the new POM represents a complete and reproducible dependency specification, complying with Maven dependency resolution semantics. Once `pom.lockfile.xml` is created, Maven is invoked with it using the `-f` flag to rebuild the project exactly as described by the lockfile, including both direct and transitive dependencies.

With this functionality, developers can reliably reproduce historical builds, making it easier to investigate bugs, validate security concerns, or recreate past releases.

3.4 Automated Update of Lockfile

To support the integration of lockfiles into development workflows, MAVEN-LOCKFILE provides a CI pipeline for validating and updating lockfiles automatically. The pipeline is implemented as a Github Action, and works as follows: if a `pom.xml` or `lockfile.json` file has been modified in a pull request, the action automatically generates an updated lockfile and adds it as a commit to the pull request, creating a full audit trail of dependency changes. If no changes are detected in these files, the action validates the existing lockfile and fails the build if the lockfile is incorrect (cf. Section 3.2). Per the best practices for lockfiles, the MAVEN-LOCKFILE action enforces all CI to remain consistent with their declared lockfiles and supports keeping them up-to-date.

3.5 Novelty of MAVEN-LOCKFILE

When generating a lockfile for a project, one needs to include four key fields [2]: resolved package versions, package checksums, package source, and a way to distinguish between resolved direct and indirect dependencies. Among the most popular package managers, only Go and Cargo include all these elements. MAVEN-LOCKFILE is the first system to equip Java projects with state-of-the-art build integrity.

The other popular build automation tool for Java, Gradle, includes a built-in solution to generate lockfiles. However, in practice, Gradle lockfiles are rarely used because their configuration is not user-friendly. First, the lockfile generation is not the default behavior. In addition to limited usability, Gradle lockfiles omit important details that should be part of a lockfile, such as checksums of resolved dependencies. The key novelty of MAVEN-LOCKFILE compared to Gradle is that it requires minimal configuration effort from developers and includes the most important elements to ensure integrity and reproducibility. MAVEN-LOCKFILE finally brings high integrity builds to Java developers.

4 Evaluation

We evaluate MAVEN-LOCKFILE based on two research questions:

Version	Date	Generate	Validate	Rebuild
2.1.0	2023-06-05	✓	✓	✓
2.2.0	2023-06-07	✓	✓	✓
3.0.1	2023-06-12	✓	✓	✓
3.1.0	2023-06-12	✓	✓	✓
4.1.0	2023-08-29	✓	✓	×
5.1.0	2024-04-30	✓	✓	✓
5.3.2	2024-12-19	✓	✓	✓
5.3.4	2024-12-20	✓	✓	✓
5.5.0	2025-04-23	✓	✓	✓
5.6.2	2025-09-10	✓	✓	✓

Table 1: Results of generating new lockfiles, validating old lockfiles and rebuilding from them.

RQ1 Can MAVEN-LOCKFILE rebuild old versions thanks to the lockfile?

RQ2 Can MAVEN-LOCKFILE detect tampered dependency artifacts?

4.1 Methodology

To answer RQ1, we use MAVEN-LOCKFILE to rebuild ten randomly chosen previous releases of MAVEN-LOCKFILE itself, starting from the first release supporting rebuilding from 2023-06-05. For each release, we checkout the corresponding commit and the corresponding lockfile. We first reproduce the build environment by downloading the Java and Maven versions specified in the lockfile. We then use the chosen release of MAVEN-LOCKFILE to produce a frozen POM from the committed lockfile of the old release (cf. Section 3.3). Finally, we validate the build environment and rebuild MAVEN-LOCKFILE using the frozen POM.

To answer RQ2, we first generate a lockfile for a project and validate the resulting build based on it (cf. Section 3.2). For this, we use the latest version of MAVEN-LOCKFILE, version 5.7.1. Then, we modify one of the dependencies of the project by changing the locally downloaded artifact used for the build through a random binary perturbation. We then validate the environment again based on the lockfile, with the expectation that MAVEN-LOCKFILE raises an error.

4.2 Results

Table 1 shows the results of RQ1. All releases contain a valid lockfile, demonstrating that the generation of lockfiles behaves correctly and all could be used to validate the build environment. In nine out of ten cases, it is possible to rebuild the project based on this environment. Major version 4 fails to rebuild due to a bug in the dependency resolution used by MAVEN-LOCKFILE. Specifically, MAVEN-LOCKFILE incorrectly added transitive dependencies with the test scope to the frozen `pom.xml`, instead of the same transitive dependency without the test scope. Dependencies with the test scope are not available in non-test classpaths and thus, the compilation fails as Maven cannot access the expected classes. This was fixed in version 5.1.0, after which rebuilding has been stable.

Validating and rebuilding version 3.0.1 and 3.1.0 required the use of MAVEN-LOCKFILE version 2. This is because the major version 3.x of MAVEN-LOCKFILE added new fields to the lockfile which

were not backwards compatible. As no dependencies changed with the release of version 3, the lockfile in the repository was still generated by version 2 of MAVEN-LOCKFILE, and thus required the older version to operate on them. Using the latest version 2 release of MAVEN-LOCKFILE, validating and rebuilding behaves as expected.

When validating versions up to and including 5.1.0, the lockfiles contained the previous -SNAPSHOT version instead of the version specified in the pom at the release tag. Thus, the commit before the release tag was used when validating. Version 5.5.0, similar to major version 3, introduced new fields requiring the use of the previous version of MAVEN-LOCKFILE (in this case 5.4.2) to successfully validate. Moreover, prior to version 4.1.0, optional parameters such as inclusion of maven plugins have to be specified manually for validation.

For RQ2, generating the lockfile for MAVEN-LOCKFILE itself and running `validate` against it gives no errors, which is a prerequisite. We then modify the first dependency listed in the lockfile, the `jar` for `com.google.code.gson:gson:2.13.2` by extracting and repackaging it, thus changing access times in the zip. Running `validate` against the lockfile again, MAVEN-LOCKFILE correctly raises an error, reporting that the checksum of the `gson` dependency has changed with respect to the lockfile. After rebuilding using the modified dependency, the test suite on the newly built artifact passes, illustrating that some adversarial malicious modification of `gson` can go undetected under normal Maven operations of a CI/CD pipeline.

Overall, our evaluation shows that MAVEN-LOCKFILE enables rebuilding old releases with reproduced dependency resolution, as frozen in the lockfile.

4.3 Discussion

There is scarce literature on longitudinal build reproducibility. We know some factors that break reproducibility, such as missing version information or nondeterministic transitive dependencies, but we lack tools for verifying reproducibility. To the best of our knowledge, MAVEN-LOCKFILE is the first system to enable systematic longitudinal reproducibility in real-world Java projects.

Moreover, we have shown that MAVEN-LOCKFILE detects artifacts that have been tampered with. Future research can explore how lockfiles can be extended and integrated with existing security mechanisms, such as digital signatures of artifacts or transparency logs, to provide even stronger guarantees for the software supply chain.

Beyond Java, the design of MAVEN-LOCKFILE provides a blueprint for bringing lockfile support to Maven and other ecosystems that currently lack it. Our paper informs general strategies for reproducible builds and supply chain protection across different build and dependency management systems.

5 Related Work

Several studies mention lockfiles in the context of dependency management and recognize their impact. He *et al.* [3] recommend lockfiles instead of pinning. Bogart *et al.* [1] identify the use of lockfiles as a way to stabilize projects by locking indirect dependency versions. Venturini *et al.* [7] show that lockfiles help avoid in-range breaking updates. Vaidya *et al.* [6] suggest that the `npm CLI` lockfile `npm-shrinkwrap` can mitigate attacks in the `npm` ecosystem. Kabir

et al. [4] consider committing lockfiles one of the best practices. To the best of our knowledge, none of these studies do action research as we do: injecting lockfiles support in real-world projects. MAVEN-LOCKFILE provides a foundation for future research on integrity and build determinism in the context of Java projects.

6 Conclusion

We have introduced MAVEN-LOCKFILE, a state-of-the-art tool to validate the integrity of Maven builds, reproduce old builds, and automatically integrate with modern software engineering processes. Our evaluation shows that MAVEN-LOCKFILE can successfully rebuild previous project versions using pinned dependencies from the lockfile, and is able to detect tampered artifacts. The novelty of MAVEN-LOCKFILE lies in its full coverage of the essential elements needed for integrity and reproducibility, equipping Java projects with modern build integrity at minimal cost for developers. Future research is needed to study how lockfiles can integrate with other software supply chain security mechanisms, such as transparency logs.

Acknowledgments

We would like to thank Martin Wittlinger for his contributions to MAVEN-LOCKFILE. This work is supported by the CHAINS project funded by the Swedish Foundation for Strategic Research (SSF), and by IVADO and the Canada First Research Excellence Fund.

References

- [1] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems. *ACM TSE* (July 2021).
- [2] Yogya Gamage, Deepika Tiwari, Martin Monperrus, and Benoit Baudry. 2025. The Design Space of Lockfiles Across Package Managers. arXiv:2505.04834 [cs.SE]
- [3] Hao He, Bogdan Vasilescu, and Christian Kästner. 2025. Pinning Is Futile: You Need More Than Local Dependency Versioning to Defend against Supply Chain Attacks. *Proc. ACM Softw. Eng.* 2, FSE (June 2025). doi:10.1145/3715728
- [4] Md Mahir Asef Kabir, Ying Wang, Danfeng Yao, and Na Meng. 2022. How Do Developers Follow Security-Relevant Best Practices When Using NPM Packages?. In *2022 IEEE SecDev*. 77–83. doi:10.1109/SecDev53368.2022.00027
- [5] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1509–1526.
- [6] Ruturaj K Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2019. Security issues in language-based software ecosystems. (2019). arXiv:1903.02613 [cs.CR]
- [7] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A. Gerosa, and Igor Scalante Wiese. 2023. I Depended on You and You Broke Me: An Empirical Study of Manifesting Breaking Changes in Client Packages. *ACM TSE* (May 2023).
- [8] 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 TSE* 34, 5 (May 2025). doi:10.1145/3714464
- [9] Sheng Yu, Wei Song, Xunchao Hu, and Heng Yin. 2024. On the Correctness of Metadata-Based SBOM Generation: A Differential Analysis Approach. In *2024 IEEE/IFIP DSN*. 29–36. doi:10.1109/DSN58291.2024.00018