PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

الجمهـوريـــة الجزائـريـــة الديـمـقـراطيـــة الشـعـبيـــة

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

وزارة التعليم العالي والبحث العلمي

École Supérieure en Sciences et Technologies de l'Informatique et du Numérique

المدرسة العليا في علوم وتكنولوجيات الإعلام الآلي بجاية



Dissertation Submitted to the Department of 2nd Cycle Training in Partial Fulfillment of the Requirements for Master's Degree in Computer Science

Specialty: Cybersecurity

Submitted By:

Charaf Eddine HADJOUDJ

Diversity in Build Pipelines: A State-of-the-Art Survey on Dependency Resolution, Reproducibility, and Supply-Chain Security

Supervised by:

Pr. Benoit Baudry UdeM
Mr. Ali Hassaoui ESTIN

Members of Jury:

■ Dr. Bouternikh Salih	President	ESTIN
■ Dr. MEDJOUDJ Rafik	Examiner	ESTIN
■ Dr. Allama Oussama	Examiner	ESTIN

Academic year: 2025/2026

Abstract

Modern build pipelines rely on package managers and dependency resolvers to assemble large graphs of third-party components automatically. The internal algorithms these resolvers use, how they traverse the dependency graph and how they arbitrate between conflicting versions of the same artifact, are usually treated as implementation details, invisible to the developer and largely unexamined by the research literature. Yet these choices directly determine which bytes end up in a shipped artifact and, more importantly, which version of a dependency wins when a build pipeline is targeted by a supply-chain attack. This dissertation surveys the state of the art surrounding three adjacent research threads: build reproducibility, dependency-resolution algorithms, and software supply-chain security. We synthesize the literature on reproducible builds, on version-resolution semantics and dependency-confusion attacks, and on supply-chain integrity frameworks, and we identify a specific gap that connects them: resolver diversity, deliberately building the same source tree under multiple resolver configurations and comparing the outputs, is a natural, underexplored primitive for both measuring reproducibility and detecting tampering. No existing study empirically measures how often real resolver configurations actually diverge, or whether that divergence translates into a genuine security signal. This gap motivates the empirical study developed in the companion engineering dissertation.

Keywords: build reproducibility, dependency resolution, software supply chain security, Maven, SBOM, SLSA.

Résumé

Les pipelines de construction logicielle modernes reposent sur des gestionnaires de paquets et des résolveurs de dépendances pour assembler automatiquement de vastes graphes de composants tiers. Les algorithmes internes utilisés par ces résolveurs, la manière dont ils parcourent le graphe de dépendances et arbitrent les conflits de version, sont généralement considérés comme des détails d'implémentation, invisibles pour le développeur et peu étudiés par la littérature de recherche. Pourtant, ces choix déterminent directement les octets qui composent un artefact livré et, plus important encore, quelle version d'une dépendance l'emporte lorsqu'un pipeline de construction est la cible d'une attaque de la chaîne d'approvisionnement. Ce mémoire dresse l'état de l'art de trois axes de recherche connexes : la reproductibilité des constructions logicielles, les algorithmes de résolution de dépendances, et la sécurité de la chaîne d'approvisionnement logicielle. Nous synthétisons la littérature sur les constructions reproductibles, sur la sémantique de résolution de versions et les attaques par confusion de dépendances, ainsi que sur les cadres d'intégrité de la chaîne d'approvisionnement, et nous identifions une lacune précise : la diversité des résolveurs constitue un levier naturel et peu exploré pour mesurer la reproductibilité et détecter une falsification, mais aucune étude existante ne mesure empiriquement la fréquence à laquelle des configurations de résolveur réelles divergent. Cette lacune motive l'étude empirique développée dans le mémoire d'ingénieur associé.

Mots-clés : reproductibilité des constructions, résolution de dépendances, sécurité de la chaîne d'approvisionnement logicielle, Maven, SBOM, SLSA.

Acknowledgements

First and foremost, all praise is due to Allah Almighty, the Most Gracious and the Most Merciful, who granted me the strength, courage, health, and patience to successfully conduct and complete this research work.

I extend my deepest gratitude to my supervisors, Pr. Benoit Baudry (UdeM) and Mr. Ali Hassaoui (ESTIN), for their invaluable guidance, availability, and continuous support throughout the development of this dissertation.

I warmly thank the esteemed members of the jury for their time, interest, and thoughtful evaluation of my work. Their constructive feedback and academic perspective are deeply appreciated.

Finally, I extend my profound and heartfelt thanks to my family for their constant encouragement, moral support, and patience throughout this journey. I would also like to take this opportunity to thank everyone who contributed, directly or indirectly, to the successful completion of this thesis.

Charaf Eddine HADJOUDJ

Contents

Abstract i

Résumé ii

Acknowledgements iii

List of Figures ii

List of Tables iii

List of Abbreviations iv

General Introduction 1

1 Background 5

1.1 Introduction . . . 5

1.2 Software Build Systems and Dependency Resolution . . . 5

1.2.1 A Worked Example . . . 6

1.2.2 How Different Ecosystems Resolve This . . . 7

1.3 Maven and Maven Resolver . . . 9

1.3.1 Resolver 1.x (Maven 3.9.x) . . . 9

1.3.2 Resolver 2.x (Maven 4) . . . 9

1.4 Maven's Build Lifecycle . . . 10

1.5 Software Supply-Chain Security . . . 11

1.5.1 Three Real Incidents, and a Distinction Worth Making Precise . . . 12

1.5.2 Integrity Frameworks . . . 13

1.6 Conclusion . . . 14

2 State of the Art 15

2.1 Introduction . . . 15

2.2 Build Reproducibility . . . 15

2.2.1 Definitions and Motivation . . . 15

2.2.2 A Taxonomy of Non-Determinism . . . 16

2.2.3 What This Literature Does Not Address . . . 17

2.2.4 The Reproducible Central Corpus and AROMA 17
2.3 Dependency Resolution and Version-Conflict Semantics 18
2.3.1 Dependency Confusion as an Empirical Phenomenon 18
2.3.2 Three Real Incidents 18
2.3.3 The Specific Gap in Maven-Focused Work 19
2.3.4 Resolution Algorithms Beyond Maven 19
2.4 Supply-Chain Integrity Frameworks 20
2.4.1 SBOMs 20
2.4.2 SLSA 21
2.4.3 In-toto and Sigstore 21
2.4.4 A Structural Assumption Common to All Three 21
2.5 Build-Configuration Maintenance Research 22
2.6 Diversity as a Verification and Security Mechanism 22
2.6.1 N-Version Programming 22
2.6.2 Diverse Double-Compiling 23
2.6.3 Resolver-Configuration Diversity as a Direct Descendant 23
2.7 Systemic Limitations and Open Challenges 24
2.8 A Vision of the Domain's Evolution 25
2.9 Conclusion 25

3 Discussion and Critical Perspective 26
3.1 Introduction 26
3.2 Depth of Analysis versus Practical Adoption 26
3.2.1 The General Trade-off 26
3.2.2 Why Cheapness Matters for Adoption 27
3.3 Resolver Diversity as a Complement, Not a Replacement 27
3.4 Trustworthiness of Industrial versus Academic Approaches 28
3.4.1 Industrial Tools 28
3.4.2 Academic Tools and This Study's Position 28
3.4.3 Bridging the Gap 28
3.5 Threats to the Validity of a Purely Literature-Based Argument 29
3.5.1 Security Trade-offs of Conflict-Resolution Strategies 29
3.6 Recommendations for Future Frameworks 30
3.7 Conclusion 31

General Conclusion 32

Bibliography 36

A Glossary of Terms 43

List of Figures

1.1 Maven Resolver's three logical steps, and the Resolver-2.x configuration property governing each. Only the middle step, conflict resolution, is shown by the companion engineering dissertation's evaluation to have any effect on final build output; the other two are documented, correctly-implemented properties that either affect only build time (collection) or are computed correctly but never reach the build lifecycle (flattening), as Section 1.3 discusses in detail below. 6

1.2 A real diamond-conflict shape (`exec-maven-plugin`): two independent paths reach the same coordinate at different versions, at the same depth. Nearest-wins and highest-wins resolution genuinely disagree on this graph. 7

1.3 Two independent implementations of "flatten this graph into a list," in two different repositories. Only the hardcoded one, on the right, ever reaches the compiler. 11

List of Tables

1.1 Dependency-conflict handling across package ecosystems. 8
1.2 Maven Resolver 2.x's three configurable dimensions. 10
1.3 Supply-chain attack patterns relevant to dependency resolution. 12

2.1 Common sources of build non-determinism (reproducible-builds literature). 16
2.2 Three real compromised-maintainer incidents versus the companion dissertation's version-conflict threat model. 19
2.3 What each supply-chain integrity framework records, and what it does not. 20
2.4 Resolver-configuration diversity relative to prior diversity-based verification techniques. 23

List of Abbreviations

CI/CD Continuous Integration / Continuous Delivery

SBOM Software Bill of Materials

SLSA Supply-chain Levels for Software Artifacts

OIDC OpenID Connect

CVE Common Vulnerabilities and Exposures

JVM Java Virtual Machine

JDK Java Development Kit

JPMS Java Platform Module System

GAV groupId:artifactId:version (Maven coordinate)

POM Project Object Model (Maven's build descriptor)

RC Reproducible Central

CRC Cyclic Redundancy Check

SHA Secure Hash Algorithm

OSS Open-Source Software

npm Node Package Manager

PyPI Python Package Index

General Introduction

Background

Modern software engineering relies predominantly on automated composition, wherein developers integrate extensive graphs of third-party dependencies. A typical Java service built with Apache Maven declares a handful of direct dependencies in its project descriptor, and the build tool silently expands that handful into a graph of tens or hundreds of transitive dependencies, resolves version conflicts between them, downloads the winning artifacts, and links them into a final executable. This resolution step is almost entirely invisible to the developer: it happens automatically, in milliseconds, according to rules that most engineers have never read and that most build tools do not even expose as configurable. Yet the outcome of that resolution — which exact version of which exact artifact ends up on the classpath — is one of the most consequential and least scrutinized decisions in the entire software supply chain. A single popular library, transitively depended upon by thousands of downstream projects, can propagate a compromise to all of them the moment any of those projects' resolvers picks up its poisoned release; which resolver configuration each of those projects happens to run therefore matters far beyond the individual build it governs.

Two properties of dependency resolution motivate this dissertation.

Reproducibility. If the same source tree is built twice, do the two builds produce the same output? This question has occupied the *Reproducible Builds* community for over a decade, largely in the context of eliminating incidental non-determinism (timestamps, file ordering, locale-dependent output). Far less attention has been paid to a narrower but sharper version of the question: if the same source tree is built twice under two different, individually well-defined resolver configurations, does it produce the same output? Maven Resolver 2.x, bundled with Maven 4, is the first resolver generation to expose its internal dependency-collection algorithm and version-conflict-selection strategy as configuration properties rather than hard-coded behavior. This creates, for the first time on a widely

deployed build platform, an opportunity to measure resolver-induced non-determinism directly and at scale — an opportunity this dissertation's companion engineering study is, to our knowledge, the first to take.

Security. Modern supply-chain attacks increasingly target the dependency-resolution step itself rather than the human developer: dependency confusion attacks register a malicious package at a coordinate the resolver is likely to prefer; compromised-maintainer attacks push a malicious release of an already-trusted package, relying on downstream resolvers to pick it up automatically and silently. In both cases, the attacker's leverage is entirely mediated by the resolver's own conflict-resolution logic — whether it prefers the nearest declaration in the dependency graph or the numerically highest version, and in what order it visits graph nodes while collecting candidates. If two legitimate resolver configurations can be shown to disagree on which version wins a given conflict, that same disagreement is, in principle, a detection signal: an attacker's forged high version number should not survive a nearest-wins resolution the way it survives a highest-wins one. Real, well-documented incidents — the 2018 compromise of the `event-stream` npm package, the 2021 `ua-parser-js` compromise, and the 2024 `xz-utils` backdoor — establish that a compromised, previously-trusted maintainer or publishing pipeline is a realistic threat model, not merely a theoretical one, which is the threat model the empirical study's security experiment adopts directly.

Problem Statement

This dissertation is a state-of-the-art survey that situates an emerging empirical research question within the existing literature. A separate engineering dissertation studies the phenomenon experimentally; the present volume focuses on the evidence, concepts, and gaps established by prior research. We ask three questions:

- **RQ1 (Reproducibility).** What does the reproducible-builds literature already know about non-determinism in dependency resolution specifically, as opposed to compiler or packaging non-determinism more generally?
- **RQ2 (Resolution semantics).** How has prior work characterized the algorithms package managers use to resolve version conflicts, and what security consequences has that work already attributed to different resolution strategies?
- **RQ3 (Supply-chain security).** What do existing supply-chain integrity frameworks (SBOMs, SLSA, provenance attestation) assume about build reproducibility, and where do those assumptions leave a gap that resolver diversity could fill?

Contributions

This dissertation's contribution is a systematic survey answering these three questions and synthesizing them into a single, coherent gap, rather than an empirical measurement, which is the companion dissertation's contribution. Concretely, we:

- Situate Maven Resolver 2.x's newly-configurable behavior within the broader design space of how package ecosystems handle dependency conflicts, showing it is one deliberate design point among several, not an idiosyncratic implementation detail.
- Survey and synthesize four separately mature research threads — reproducibility, resolution semantics, supply-chain integrity, and build-configuration maintenance — and show, for the first time to our knowledge, that none of them individually addresses resolver-configuration diversity as a variable.
- Identify the diversity-verification principle (N-version programming, Diverse Double-Compiling) as the missing theoretical link between these threads and the empirical study's methodology.
- Trace, at the level of exact source-code locations in Maven Resolver and Maven Core, the documented implementation path by which one of Resolver 2.x's three configurable properties is discarded before it reaches a real build, thereby distinguishing documented configuration from effective build behavior.
- Offer concrete recommendations for how supply-chain integrity frameworks could evolve to account for resolver-configuration diversity.

Dissertation Structure

This dissertation is organized into three chapters. **Chapter 1: Background** introduces the necessary background: how modern build systems and package managers structure dependency resolution across several ecosystems, how Maven Resolver specifically implements it and where its documented configuration surface diverges from what a real build actually consults, and the anatomy of a software supply-chain attack. **Chapter 2: State of the Art** surveys the state of the art across the four research threads identified above and synthesizes their systemic limitations, closing with a short vision of how the field could evolve if that gap is closed. **Chapter 3: Discussion and Critical Perspective** offers a critical discussion of the trade-offs this body of work exposes — depth of analysis versus practical adoption, industrial versus academic tooling — and positions the empirical study against the identified gap, closing with concrete recommendations

for framework maintainers. The General Conclusion summarizes our findings and hands off to the engineering dissertation, where the study itself is designed, implemented, and evaluated.

Chapter 1

Background

1.1 Introduction

This chapter introduces the concepts the rest of the dissertation builds on. Section 1.2 describes, at a general level, how modern build systems resolve dependencies, and compares how several widely used package ecosystems approach the same problem differently. Section 1.3 narrows the discussion to Apache Maven and its resolver library, Maven Resolver (formerly "Aether"), whose configurable behavior is the object of the empirical study developed in the companion engineering dissertation. Section 1.4 places dependency resolution within Maven's broader build lifecycle, since several of the study's later findings depend on distinguishing what that lifecycle actually consults from what a configuration property merely claims to control. Section 1.5 introduces the software supply-chain security vocabulary — attack surface, provenance, attestation — used throughout the rest of the dissertation.

1.2 Software Build Systems and Dependency Resolution

A build system's job is to turn a declared set of dependencies into a concrete, ordered classpath (or, more generally, a concrete set of linked artifacts). This involves three logically distinct steps, illustrated in Figure 1.1, which different tools sometimes fuse together:

1. **Collection.** Starting from the project's direct dependencies, the tool recursively discovers transitive dependencies by reading each dependency's own manifest, producing a dependency *graph* (not a tree, in general, since the same coordinate can

be reached via multiple paths, at different depths, through different intermediate dependencies).

2. **Conflict resolution.** When two paths in the graph reach the same coordinate (the same group and artifact identifier, in Maven's terminology) at different versions, the tool must pick exactly one winning version — otherwise two incompatible copies of the same class could end up on the classpath simultaneously, a situation most JVM classloaders handle poorly and silently.

3. **Flattening / linking.** The resolved graph is turned into an ordered, flat list of artifacts (a classpath), which is what the compiler, test runner, or packager actually consumes.



Figure 1.1: Maven Resolver's three logical steps, and the Resolver-2.x configuration property governing each. Only the middle step, conflict resolution, is shown by the companion engineering dissertation's evaluation to have any effect on final build output; the other two are documented, correctly-implemented properties that either affect only build time (collection) or are computed correctly but never reach the build lifecycle (flattening), as Section 1.3 discusses in detail below.

Collection necessarily happens before conflict resolution: deciding a "winner" presupposes there is more than one candidate to choose between, and the set of candidates is exactly what collection discovers. Collection therefore always visits every reachable version of every coordinate in the graph, including versions that conflict resolution will later discard; the traversal order used to visit them governs only how much repeated work is done in a graph with shared sub-trees, not which candidates exist or which one ultimately wins. Flattening runs last, after a winner is picked for every coordinate, and its only remaining job is ordering: given a graph that already has exactly one version per coordinate, in what order do those artifacts appear on the classpath.

1.2.1 A Worked Example

Figure 1.2 illustrates the shape of graph that makes step 2 non-trivial: a real project (`exec-maven-plugin`, discussed at length in the companion engineering dissertation) whose dependency graph reaches the coordinate `commons-io:commons-io` through two separate paths, at the same depth, declaring two different versions (`2.2` and `2.5`). Nothing in the project's own declared dependencies is wrong or contradictory; the conflict arises purely because two unrelated transitive dependencies each happen to depend on the same shared library at different versions, which is common in any sufficiently large real-world project. Which version the build actually uses is decided entirely by the resolver's conflict-resolution strategy.

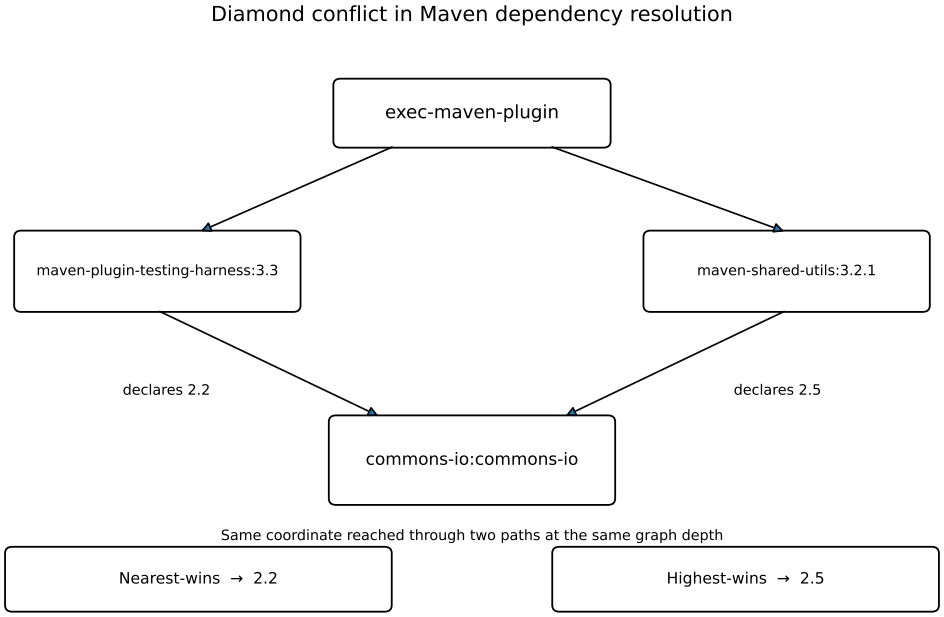


Figure 1.2: A real diamond-conflict shape (`exec-maven-plugin`): two independent paths reach the same coordinate at different versions, at the same depth. Nearest-wins and highest-wins resolution genuinely disagree on this graph.

1.2.2 How Different Ecosystems Resolve This

Package ecosystems differ substantially in how they implement conflict resolution, and the differences are informative because they show that Maven's choice — insist on exactly one winning version, chosen by a specific, nameable algorithm — is a design decision, not the only possible one. Table 1.1 summarizes five widely used ecosystems.

Maven's insistence on a single, deterministic winner is precisely what makes its conflict-resolution algorithm — and the fact that Maven 4 makes that algorithm configurable rather than fixed — interesting to study: unlike npm's coexistence model, a change in Maven's algorithm can change which *one* version of a library every part of the

Table 1.1: Dependency-conflict handling across package ecosystems.

Ecosystem	Conflict strategy	Consequence
npm / Yarn (Node.js)	Nested `node_modules`: multiple versions of the same package coexist on disk.	Avoids the conflict entirely at install time, at the cost of disk space and, historically, a distinct attack surface (dependency confusion exploits exactly this permissiveness combined with resolution-order assumptions).
pip (Python, pre-2020)	No real conflict detection: the last-installed version of a conflicting package silently wins.	Conflicts fail silently; diagnosed only at import/runtime, if at all.
pip (Python, 2020+ resolver)	Backtracking SAT-like resolver: attempts to find a single consistent assignment across all constraints, failing loudly if none exists.	Conflicts are detected but the resolution algorithm itself is not user-configurable.
Cargo (Rust)	Permits multiple *major* versions of the same crate to coexist; only unifies identical versions.	Sidesteps most conflicts by design, relying on Rust's type system to keep incompatible versions from interacting unsafely.
Maven (Java)	Exactly one version per coordinate, chosen by a conflict-resolution algorithm applied to the whole build.	Maximizes predictability of the final classpath, at the cost of making the resolution algorithm itself a consequential, previously-fixed decision — and, since Maven 4, a configurable one.

build sees, which is a strictly stronger and more security-relevant kind of change than anything a coexistence-based ecosystem can express. It is also worth noting explicitly that Maven's single-winner arbitration is not the only alternative to coexistence: pip's post-2020 resolver and Dart's `pub` instead formulate resolution as a full satisfiability problem, searching the space of all mutually consistent version assignments rather than picking one winner per coordinate greedily [4]. That richer design space is out of scope for the empirical study, which studies Maven Resolver's actual, deployed single-winner behavior, but it is worth noting that the two-strategy space (nearest versus highest) this dissertation's companion study explores is itself a narrow slice of how dependency resolution could in principle be designed.

1.3 Maven and Maven Resolver

Maven Resolver is the dependency-resolution engine used by Apache Maven (and by other tools, such as certain IDEs and Eclipse Tycho) [64]. Two resolver generations are relevant here.

1.3.1 Resolver 1.x (Maven 3.9.x)

Resolver 1.x implements a single hard-coded conflict-resolution strategy, informally known as *nearest-wins*: among several candidate versions of the same coordinate, the one declared closest to the root of the dependency graph wins; ties at equal depth are broken by declaration order (the first-declared candidate, in document order, wins). Neither this strategy nor the graph-traversal order used to collect candidates is exposed as a configuration property — both are internal implementation details of the resolver library.

1.3.2 Resolver 2.x (Maven 4)

Resolver 2.x restructures this logic behind three separate, independently configurable properties, summarized in Table 1.2.

Two subtleties matter for the rest of this dissertation, both established empirically rather than merely documented, in the companion engineering dissertation.

A direct declaration always wins. A dependency the current project declares directly always wins outright, regardless of the configured selection strategy — conflict resolution only arbitrates between candidates that are both transitive. This has a direct methodological consequence: an experiment intending to compare nearest-wins against highest-wins must ensure the conflicting coordinate is reached transitively through *both*

Table 1.2: Maven Resolver 2.x's three configurable dimensions.

Property	Values	What it controls
`aether.dependencyCollector.impl`	`df` (depth-first), `bf` (breadth-first)	The order in which the raw dependency graph is *traversed* while collecting candidate versions, before any conflict is resolved.
`aether.conflictResolver.versionSelector.selectionStrategy`	`nearest`, `highest`	Which candidate *wins* a version conflict: the classic nearest-declaration behavior, or the numerically greatest version regardless of depth.
`aether.system.dependencyVisitor`	`preOrder`, `postOrder`, `levelOrder`	The order in which an already-resolved graph is *flattened* into a list, for callers that invoke a specific low-level resolver API directly.

candidate paths, never declared directly by the project under test, or the two strategies will trivially agree regardless of which is configured.

Not every documented property reaches a real build. The third property, `aether.system.dependencyVisitor`, is a genuine, correctly-implemented configuration property — Maven Resolver's own `DefaultRepositorySystem.resolveDependencies()` does read it, and does apply the corresponding traversal, and Maven Core's own `DefaultProjectDepende` does call exactly this method. The property's effect disappears one layer further in: the class that actually assembles a project's compile and runtime classpath, `LifecycleDependencyResolver` does not use that already-computed, correctly-ordered result at all. It discards it and independently re-derives the artifact set from the raw, unflattened dependency graph via a second, separate utility method, `RepositoryUtils.toArtifacts()`, which is a short, hand-written recursive traversal that is structurally always preorder and never reads any configuration property. Figure 1.3 traces this precisely: Maven Resolver and Maven Core each implement "flatten this graph into a list" independently, in two different code repositories, and only the configurable one is unused by a real build. This distinction — not merely that the property is unreachable, but precisely *which* of two independent graph-flattening implementations the build lifecycle actually uses, and why the correctly-computed configurable result is computed and then discarded rather than never computed at all — is not evident from Maven's own documentation and is, to our knowledge, traced here to exact source locations for the first time; the companion engineering dissertation independently confirms the resulting behavior empirically, down to the literal argument passed to the Java compiler.

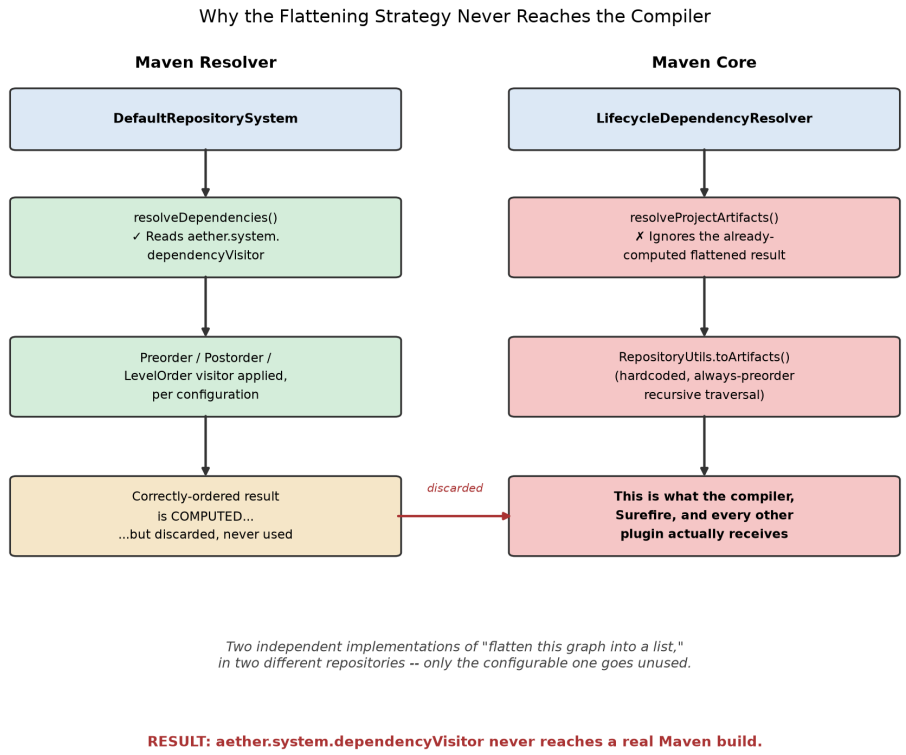


Figure 1.3: Two independent implementations of "flatten this graph into a list," in two different repositories. Only the hardcoded one, on the right, ever reaches the compiler.

1.4 Maven's Build Lifecycle

Maven organizes a build into a fixed sequence of *phases* (`validate`, `compile`, `test`, `package`, `verify`, `install`, `deploy`), each of which binds zero or more plugin goals. Dependency resolution is not itself a single phase; it is triggered lazily, the first time a phase needs a classpath (typically `compile`), and the resolved classpath is then reused by later phases in the same build invocation. This matters for interpreting build-time measurements in the companion dissertation: the cost of dependency collection and conflict resolution is paid once per build invocation, not once per phase, so comparing `clean package` build times across resolver configurations (as the companion dissertation does) directly reflects resolution cost rather than an artifact of which phases happened to run.

1.5 Software Supply-Chain Security

A software supply chain is the sequence of steps — source, dependencies, build, packaging, distribution — that turns source code into a deployed artifact. A supply-chain attack compromises one of these steps rather than the final target directly, so that the compromise propagates to every downstream consumer without the attacker needing

to breach each one individually. Table 1.3 summarizes the three attack patterns that recur most often in the literature and in real incidents, and that are directly relevant to dependency resolution specifically — as opposed to the broader space of supply-chain attacks (build-server compromise, source-repository tampering) that do not depend on resolver behavior at all.

Table 1.3: Supply-chain attack patterns relevant to dependency resolution.

Pattern	Mechanism	Resolver dependency
Dependency confusion	Attacker publishes a public package at the same name as an internal, privately-hosted one, betting the resolver prefers the public registry or the higher version number.	**Directly mediated** by whichever version-selection strategy the resolver uses.
Compromised maintainer / compromised transitive dependency	Attacker gains publish access to an already-trusted package and pushes a release containing a backdoor or a manipulated dependency declaration.	**Directly mediated**: the attacker's leverage over which declaration wins a downstream conflict is exactly the mechanism this dissertation studies.
Typosquatting	Attacker publishes a package whose name closely resembles a popular one, hoping for an accidental, manual install.	Not resolver- mediated: exploits human error in dependency *declaration*, not resolver conflict resolution.

1.5.1 Three Real Incidents, and a Distinction Worth Making Precise

Three well-documented compromised-maintainer incidents establish that gaining publish access to an already-trusted package is a realistic, repeatedly demonstrated attacker capability. In the 2018 `event-stream` [61] incident, an attacker was granted commit and publish access to a popular npm package by its original maintainer, then introduced a new dependency carrying obfuscated, encrypted malicious code that activated only inside one specific downstream application. In the 2021 `ua-parser-js` [62] incident, an attacker instead compromised the maintainer's account credentials directly and published versions containing a cryptocurrency miner and a credential-stealing trojan. In the 2024 `xz-utils` [63] incident, an account built social-engineering pressure over roughly two years to earn co-maintainer status on a widely used compression library, then hid a backdoor inside build-system metadata rather than in the reviewable source history itself.

It would overstate the connection to claim that resolver-configuration diversity would have prevented any of these three specific incidents: in all three, the malicious code shipped *as* the only available release at its coordinate, with no competing, legitimate version at a different point in the dependency graph for a resolver strategy to prefer instead. The mechanism the companion engineering dissertation's hijack simulation tests is narrower, and, to our knowledge, not yet documented in a real, disclosed incident: an attacker who compromises a *transitive* dependency and uses that foothold to declare a fraudulent, inflated version of a *separately-owned* shared library, creating an artificial version conflict that a resolver must then arbitrate. What the three incidents *do* establish, and what motivates testing this narrower mechanism synthetically rather than waiting for a real instance of it, is that gaining publish access to a trusted coordinate is a realistic capability real attackers have repeatedly demonstrated; whether that capability, redirected specifically at a version conflict rather than a content compromise, is equally effective is exactly the question the companion dissertation's hijack simulation answers.

1.5.2 Integrity Frameworks

In response to these attack patterns, the ecosystem has produced several integrity frameworks, introduced here and revisited critically in Section 2.4.

Software Bill of Materials (cro:sbomSoftware Bill of Materials (SBOM)). A machine-readable inventory of every component — direct and transitive — in a built artifact, intended to let a consumer cross-reference it against vulnerability databases without needing to re-derive the dependency graph themselves. The two dominant formats, CycloneDX [60] and SPDX [68], both represent a flat or graph-structured component list with associated metadata (license, hash, known-vulnerability references) but neither format includes a field recording which resolver configuration produced the listed versions.

Supply-chain Levels for Software Artifacts (cro:slsaSupply-chain Levels for Software Artifacts (SLSA)). A graduated framework [56] defining build-provenance guarantees in four levels, from "the build process is documented and scripted" up to "the build runs on a hardened, ephemeral, non-interactive platform whose output is cryptographically signed and independently verifiable." Each level adds requirements about the build platform's isolation and the tamper-evidence of its provenance record, but — like SBOM formats — says nothing about whether an equally legitimate, differently configured build of the same source would have produced the same provenance claim.

In-toto and Sigstore. In-toto [57] defines a framework for attesting to each step of a supply chain, chaining cryptographic evidence from source to deployment. Sigstore removes long-lived signing-key management as a prerequisite for adopting in-toto-style attestation [58], using cro:oidcOpenID Connect (OIDC)-backed short-lived signing iden-

tities (popularized by the `cosign`/`fulcio`/`rekor` toolchain) so that a build pipeline can sign its own output without provisioning and protecting a permanent private key.

1.6 Conclusion

This chapter established the vocabulary the rest of the dissertation relies on: dependency collection, conflict resolution, and flattening as three distinct steps a build system performs, illustrated with a real diamond-conflict graph; a comparison showing Maven's single-winner design is one point in a wider design space, itself narrower than the satisfiability-based resolvers some other ecosystems use; Maven Resolver's specific, newly-configurable implementation of conflict resolution, including a precise, source-level account of why not every documented property reaches a real build; Maven's build-lifecycle context for interpreting resolution cost; and the supply-chain attack patterns and integrity frameworks that give resolver behavior its security relevance, together with a careful distinction between what three real compromised-maintainer incidents actually demonstrate and the narrower mechanism this dissertation's companion study tests synthetically. The next chapter surveys how the existing literature has studied each of these threads in isolation, and identifies where they have not yet been studied together.

Chapter 2

State of the Art

2.1 Introduction

The literature base used in this survey includes peer-reviewed work on package dependency networks, dependency versioning, constraint solving, Maven build systems, reproducibility, and software supply-chain security [4, 5, 6, 7, 8, 9, 10, 11, 12, 13].

This chapter surveys the literature across the four threads introduced in the General Introduction and Chapter 1: build reproducibility (Section 2.2), dependency-resolution semantics and their security consequences (Section 2.3), supply-chain integrity frameworks (Section 2.4), and build-configuration maintenance research (Section 2.5). Section 2.6 then looks specifically at prior work on using *diversity* — building the same input multiple ways and comparing outputs — as a verification or security mechanism, since this is the precise idea the empirical study operationalizes for dependency resolution. Section 2.7 synthesizes the systemic limitations that emerge once these threads are read together, and Section 2.8 sketches how the field could evolve to close them.

2.2 Build Reproducibility

Empirical work on reproducibility and historical buildability also provides important context for distinguishing reproducibility of a fixed configuration from variation introduced by a deliberately changed build input [1, 2, 37, 38, 40, 41, 42, 39].

2.2.1 Definitions and Motivation

The *Reproducible Builds* project [59], and the parallel effort within Debian, established the now-standard definition of reproducibility: given the same source and the same build

instructions, an independent party should be able to produce a bit-for-bit identical binary [1]. The motivation is trust: if a rebuild by an independent party matches the binary a vendor distributes, that binary is verifiably derived from the published source, closing the gap a compromised or coerced build server could otherwise exploit invisibly. This is a narrower, and more mechanically checkable, guarantee than code review of the source alone, since source review cannot detect a discrepancy introduced purely at build time.

2.2.2 A Taxonomy of Non-Determinism

The reproducible-builds literature and tooling converge on a short, well-characterized list of non-determinism sources, summarized in Table 2.1.

Table 2.1: Common sources of build non-determinism (reproducible-builds literature).

Source	Example	Typical fix
Embedded timestamps	Zip-entry modification times; "build date" strings compiled into binaries.	Normalize to a fixed value derived from version control (e.g. last commit date).
Filesystem iteration order	Archiving files in the order `readdir()` happens to return them.	Sort file lists deterministically before archiving.
Locale / timezone	Log messages or generated documentation whose wording depends on the build machine's locale.	Pin `LC_ALL` and `TZ` for the build process.
Parallel-build nondeterminism	Output ordering that depends on which of several concurrently-running compiler invocations finishes first.	Force deterministic output ordering downstream of parallel execution.
Absolute build paths	Debug symbols or embedded strings containing the build machine's absolute filesystem path.	Use relative or normalized paths throughout the build.

Each of these has a largely mechanical fix once identified, and the Reproducible Builds project's own tooling (`diffoscope` [66] for recursive artifact comparison, `disorderfs` for filesystem-order fuzzing, `faketime` for timestamp control) is built specifically around detecting and eliminating them. `reprotest` [67] automates the natural verification template these tools support: rebuild a project multiple times while deliberately varying one environmental condition at a time, and compare the results. Maven's own `project.build.outputTimestamp` property, used throughout the companion engineering dissertation's methodology, is a direct product of this line of work, standardized as part of Maven's broader adoption of reproducible-build practices.

2.2.3 What This Literature Does Not Address

What the taxonomy in Table 2.1 does *not* include is non-determinism introduced by the dependency *resolver itself* choosing a different, but individually well-defined and individually deterministic, dependency version. A rebuild that pins every timestamp, sorts every file list, and fixes locale and timezone can still legitimately produce a different artifact if the resolver, run twice with two different (but individually deterministic) configurations, selects a different dependency version. This is not incidental non-determinism to be eliminated by better normalization — each individual build *is* reproducible in the standard sense, twice with the same configuration — it is a genuine, by-design difference in build *input* that reproducible-builds verification methodology, which reports a single `ok`/`ko` outcome per project, is not designed to distinguish from ordinary non-determinism.

2.2.4 The Reproducible Central Corpus and AROMA

The dataset this ambiguity affects is public and substantial: *Reproducible Central* [69] tracks, for several hundred real Java projects, a verified record of whether an independent rebuild reproduces a released artifact, together with the exact build command and JDK version used, encoded per-project in machine-readable `.buildspec`/`.buildcompare` file pairs. This corpus's selection methodology — filtering to Maven-built projects with a verified reproducibility record — is reused directly as the project-corpus selection procedure for the empirical study in the companion dissertation, precisely because a pre-verified reproducibility record rules out ordinary build flakiness as a confound before resolver-strategy comparison even begins.

The most directly relevant recent use of this corpus is AROMA [2], which automates recovering a Maven project's source repository, release tag, and JDK version directly from release metadata, generating the same kind of `.buildspec` record Reproducible Central otherwise curates by hand, and reports up to 99.5% field-by-field agreement against that manually curated data — in some cases identifying errors, such as broken repository links, in the manual record itself. The companion engineering dissertation uses the same corpus AROMA's evaluation draws on, but asks a different question of it. AROMA's goal is reconstructing, and automating the reproduction of, *one specific*, already-known-good build configuration for a given release. The companion dissertation's goal is the converse: holding the source tree and build environment fixed, and asking whether *varying resolver configuration* within the same build produces a different, still-legitimate result. The two are complementary rather than competing: an ecosystem that adopted AROMA's approach to automate reproducibility verification at scale would still benefit from the companion dissertation's finding, since AROMA's methodology reproduces a single configuration

faithfully but does not itself ask whether an equally legitimate alternative configuration would have produced a materially different, and possibly security-relevant, output.

2.3 Dependency Resolution and Version-Conflict Semantics

Empirical dependency research has examined dependency conflicts, version constraints, technical lag, update practices, dependency bloat, and Maven dependency graphs across multiple ecosystems [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 53].

2.3.1 Dependency Confusion as an Empirical Phenomenon

A smaller, more incident-driven body of work looks specifically at how package managers resolve version conflicts and what happens when that resolution is manipulated. The canonical example is dependency confusion, first demonstrated at scale against several large technology companies' internal package namespaces [54]. The attacker's contribution was not a novel resolver exploit in the technical sense; it was the empirical observation that several ecosystems' *default* resolution behavior — prefer the public registry, or prefer the higher version number — could be relied upon as a working attack primitive across many organizations simultaneously, without needing to compromise anything beyond registering a public package at a predictable coordinate. Subsequent academic and industry work has characterized the scale of this exposure across registries (npm, PyPI, RubyGems) via large-scale scanning of internal-looking package names, but this line of work generally treats the resolver as a fixed oracle whose default behavior is measured, rather than as a variable whose configuration could itself be changed as a mitigation and evaluated as such.

2.3.2 Three Real Incidents

Table 2.2 summarizes three well-documented compromised-maintainer incidents against this same backdrop, and the distinction Section 1.5.1 already draws between what they demonstrate and the narrower, version-conflict-specific mechanism the companion dissertation's hijack simulation tests.

Table 2.2: Three real compromised-maintainer incidents versus the companion dissertation's version-conflict threat model.

Incident	Year	Access mechanism	Relevance to resolver strategy
`event-stream`	2018	Granted commit/publish access via social engineering	None directly — malicious code shipped as the only available release; establishes attacker capability, not the specific mechanism tested here
`ua-parser-js`	2021	Stolen maintainer account credentials	None directly — same reasoning as `event-stream`
`xz-utils`	2024	Earned co-maintainer status over ∼2 years	None directly — backdoor hidden in build metadata, not a version conflict
Companion dissertation's simulation	—	Synthetic: compromised transitive dependency	Direct and central — the entire mechanism is a resolver-arbitrated version conflict

2.3.3 The Specific Gap in Maven-Focused Work

Maven's own resolver has been the subject of comparatively little targeted academic attention, in part because, prior to Resolver 2.x, its conflict-resolution behavior was not configurable and therefore not a variable an empirical study could manipulate at all — there was only one behavior to measure. Community documentation, mailing-list discussions, and issue trackers record the nearest-wins default and its declaration-order tie-break as long-standing, well-known behavior, but we found no peer-reviewed study that empirically measures, across a large real-world corpus, how frequently nearest-wins and highest-wins resolution actually diverge once both are legitimately configurable, nor one that traces a divergence to a specific dependency and verifies whether it survives into compiled bytecode as opposed to remaining a tree-level artifact. This is precisely the empirical gap the companion engineering dissertation closes, reporting a 46% divergence rate across 359 real projects.

2.3.4 Resolution Algorithms Beyond Maven

It is worth situating Maven's algorithm relative to resolution research outside the Java ecosystem, to make clear that "nearest-wins versus highest-wins" is not an idiosyncratic detail unique to one tool but an instance of a general algorithmic choice every non-coexistence-based package manager must make. Backtracking, SAT-based resolvers [4] (used, in different forms, by modern pip and by Dart's `pub`, and documented for Gradle's own dependency resolution model in its user manual [65]) reformulate dependency resolution as a constraint-satisfaction problem: rather than a single deterministic traversal rule,

they search the space of possible version assignments for one that satisfies every declared constraint, failing loudly when no such assignment exists rather than silently picking a winner. This class of resolver is not configurable in the same sense Maven Resolver 2.x is — there is no analogous "nearest versus highest" switch — because the algorithm's job is to find *a* valid assignment, not to arbitrate between several already-valid but conflicting ones the way Maven's traversal-order-based algorithm does.

2.4 Supply-Chain Integrity Frameworks

Peer-reviewed security research complements the framework documentation by characterizing dependency exposure, outdated libraries, malicious packages, and open-source supply-chain attack patterns [45, 46, 47, 48, 49, 50, 51, 52, 44, 43].

Table 2.3 summarizes what each of the three major integrity frameworks introduced in Section 1.5 records, and what none of them records.

Table 2.3: What each supply-chain integrity framework records, and what it does not.

Framework	What it records	What it does not record
SBOM (SPDX, CycloneDX)	The final, resolved list of components and versions that ended up in a build	*Why* that specific version won over any competing candidate; the resolver configuration used
SLSA provenance	The build platform, the build steps, cryptographic hashes of inputs and outputs, whether the build was hermetic and reproducible	The internal conflict-resolution logic the build tool applied while assembling those inputs
in-toto	A signed, verifiable chain of custody across each step of a multi-party supply chain (who did what, in what order)	Whether an alternative, equally legitimate resolver configuration at any step would have selected different inputs
Sigstore	Cryptographic proof of *who* signed an artifact and *that* it has not been altered since	Nothing about how the artifact's own dependency graph was resolved before signing

2.4.1 SBOMs

A SBOM, standardized through formats such as CycloneDX and SPDX, is a structured inventory of every component — direct and transitive — in a built artifact. In principle, an SBOM lets a consumer answer "does this deployed artifact contain a known-

vulnerable version of library X" without needing to re-derive the dependency graph themselves. In practice, an SBOM is only as accurate as the tooling that generated it, and that tooling almost always means running the project's own build system (or a component-scanning tool layered on top of it) once, under whatever resolver configuration that specific invocation happens to use.

2.4.2 SLSA

SLSA formalizes provenance guarantees in four graduated levels, moving from "the build process is documented" to "the build runs on a hardened, ephemeral platform whose output is signed and independently verifiable." Each level adds requirements about the build platform's isolation, non-interactivity, and the tamper-evidence of the resulting provenance record, but — notably for this dissertation's argument — no SLSA level requires recording *which resolver configuration* was used to produce a given build, nor does any level require that an alternative, equally legitimate resolver configuration would have produced a consistent (or at least explicably different) result.

2.4.3 In-toto and Sigstore

In-toto defines a framework for attesting to each step of a supply chain, and Sigstore removes long-lived signing-key management as a prerequisite for adopting it via OIDC-backed short-lived signing identities. Both frameworks strengthen the *chain of custody* around a build's output — who ran it, on what platform, and whether the resulting artifact has been tampered with after the fact — without strengthening any claim about the build's *internal* determinism relative to an alternative, equally legitimate configuration of the same tool.

2.4.4 A Structural Assumption Common to All Three

Reading SBOMs, SLSA, and in-toto/Sigstore together surfaces a structural assumption that recurs across all three: each treats the dependency- resolution step that produced a given SBOM or provenance attestation as a single, fixed, effectively deterministic function of the source tree. None of the three frameworks records which resolver configuration produced a given SBOM or attestation, and none requires cross-checking it against an alternative configuration. This is precisely the assumption the empirical study probes: if resolver strategy can change which dependency version is selected — which Section 2.3 establishes is plausible but had not, prior to the companion dissertation, been measured at scale — then an SBOM or a SLSA attestation is only as trustworthy as the specific

resolver configuration that generated it, a caveat none of these frameworks currently surfaces to a consumer.

2.5 Build-Configuration Maintenance Research

The build-maintenance literature further establishes that build specifications, dependency changes, and build failures are recurring empirical phenomena rather than purely implementation details [32, 33, 34, 35, 36].

A separate empirical literature studies Maven and Gradle build *configuration* itself as a maintenance artifact, rather than as a resolver-diversity signal. Build specifications require real, recurring maintenance effort, unevenly distributed across the build lifecycle [32]; the changes developers make to build files fall into a rich, classifiable taxonomy rather than being uniformly minor [33]; a meaningful fraction of build failures are specifically dependency-related and can, in some cases, be repaired automatically by pattern-matching against how similar failures were fixed historically in the same project [34, 35]. This literature treats resolver behavior as a fixed, given part of the build tool rather than as a variable to compare across; it studies *how build configuration changes over time*, not *whether varying the resolver's own configuration at a single point in time produces a security-relevant signal*, which is the question this dissertation and its companion pursue instead. The two threads are complementary: a mature account of how build configuration evolves and breaks would make an even stronger case for recording resolver configuration explicitly, per the recommendation in Chapter 3, since it establishes that build behavior is already known to drift for reasons unrelated to source-code changes.

2.6 Diversity as a Verification and Security Mechanism

2.6.1 N-Version Programming

The idea of deliberately producing several independent versions of the same computation and comparing them for agreement predates software supply-chain security by decades. N-version programming [3], originating in fault-tolerant systems research, proposed building several independently-implemented versions of the same specification (ideally by separate teams, to minimize correlated mistakes) and voting on their outputs at runtime, as a defense against faults specific to one implementation — a guarantee about fault tolerance, not security, but built on the same underlying logical move

resolver-configuration diversity makes: independent agreement is evidence of correctness; independent disagreement is itself informative. The move is narrowed here from whole reimplementations to a single configuration switch within one implementation.

2.6.2 Diverse Double-Compiling

Closer to the present problem, *Diverse Double-Compiling* [55] addressed a specific instance of the general verification question this dissertation is concerned with: given a compiler binary and its claimed source code, how can an independent party verify that the binary was really produced from that source, without a compromised compiler being able to hide a self-propagating backdoor (Thompson's classic "trusting trust" attack) from the check? Wheeler's answer was to recompile the compiler's source using a *second, independently-sourced* compiler and compare outputs — diversity of build *tool*, rather than diversity of build *configuration*, but the same underlying logical structure: if two independently-arrived-at builds of the same input agree, that agreement is evidence, not proof, of integrity; if they disagree, the disagreement is itself the signal worth investigating.

2.6.3 Resolver-Configuration Diversity as a Direct Descendant

Resolver-configuration diversity, as operationalized in the companion dissertation, is a direct descendant of this idea, narrowed to a single, specific, and previously unstudied axis: not a different compiler, and not a different machine, but a different (equally standard, equally legitimate) dependency-resolution strategy applied to the identical source tree. Table 2.4 situates it relative to its two ancestors.

Table 2.4: Resolver-configuration diversity relative to prior diversity-based verification techniques.

Technique	What varies	What must agree	Cost
N-version programming	Independent implementations (ideally separate teams)	Runtime output, voted at execution time	Very high: full reimplementation
Diverse Double-Compiling	Compiler binary (independently sourced)	Compiled output of the compiler's own source	Moderate: a second, trusted compiler must exist
Resolver-configuration diversity (this work)	A single resolver configuration property, same tool	Resolved dependency version / build output	Low: re-run an existing build with a different flag

To our knowledge, no prior published work has proposed or measured this specific mechanism, though its logical ancestry in N-version programming and Diverse Double-

Compiling is direct enough that we treat it as an application of an established idea to a previously unexamined variable, rather than as an entirely novel verification principle.

2.7 Systemic Limitations and Open Challenges

Reading the four threads above together surfaces a coherent, cross-cutting gap rather than four unrelated ones.

- **Reproducibility research treats resolver output as a black box.** It asks whether a build is reproducible, not whether the *resolution* step specifically is a source of legitimate, by-design divergence distinct from the non-determinism its own methodology (Table 2.1) targets.
- **Resolution-semantics research treats the resolver as fixed.** Dependency-confusion literature documents how default resolver behavior can be weaponized, but does not evaluate configuring the resolver differently as a countermeasure, nor measure how often legitimate configurations already disagree on real projects.
- **Supply-chain frameworks treat resolution as deterministic.** SBOMs and SLSA attestations are generated once, by whatever resolver configuration a given pipeline happens to run, with no mechanism to record that choice or to cross-check it against an alternative.
- **Similarly, empirical build-maintenance research abstracts dependency resolution as a static primitive, omitting resolver policy variance as an independent variable.** It documents that build behavior drifts and requires maintenance, without treating resolver-configuration diversity as one of the axes worth measuring.
- **The diversity principle exists but has not been applied here.** N-version programming and Diverse Double-Compiling establish that comparing independently-produced outputs is a valid verification strategy, but neither the literature nor practice has applied it to resolver configuration specifically, at the scale needed to know whether the resulting signal is common enough, and clean enough, to be useful.

Two further, more practical gaps compound these. First, essentially no public dataset exists that records, for a large number of real projects, build outputs under multiple resolver configurations side by side — the Reproducible Central corpus records a single verified rebuild per project, not a matrix of configurations. Second, no published methodology exists for distinguishing a resolver-driven divergence from ordinary non-determinism

(embedded metadata, timestamps) at scale, which is a prerequisite for any of the above gaps to be closed empirically rather than anecdotally — the companion dissertation's own metadata-normalization methodology (developed after an initial naïve comparison proved unusable) is a direct, first-hand illustration of how non-trivial this prerequisite turns out to be in practice.

2.8 A Vision of the Domain's Evolution

If the gap identified above is closed — which is precisely what the companion engineering dissertation sets out to do — three developments plausibly follow. First, supply-chain integrity frameworks could extend SBOM and SLSA provenance records with a resolver-configuration field, letting a consumer request or verify a second, differently-configured build as a cheap corroborating check, analogous to how Diverse Double-Compiling corroborates a compiler binary. Second, build tools beyond Maven could follow Resolver 2.x's lead in exposing previously hard-coded conflict-resolution behavior as configuration, since doing so is a prerequisite for this entire class of diversity check to exist at all in a given ecosystem. Third, and most speculatively, package registries could begin publishing, alongside a release, the resolved dependency version it would receive under more than one resolution strategy, shifting part of the verification burden from the consumer's build pipeline to the registry itself. None of these developments is guaranteed to follow from a positive empirical result; they are offered here as directions the field could plausibly take, not as predictions.

2.9 Conclusion

This chapter has shown that build reproducibility, dependency-resolution semantics, supply-chain integrity frameworks, and build-configuration maintenance research are four separately mature research threads that have not yet been read together against a single, concrete, newly-measurable variable: Maven Resolver 2.x's exposed collector and version-selector configuration. The diversity principle that would let us treat that variable as a verification and security signal is well established in an adjacent literature but has not been applied to it. The next chapter discusses the trade-offs this gap raises and positions the empirical study explicitly against it.

Chapter 3

Discussion and Critical Perspective

3.1 Introduction

This chapter steps back from the literature survey to discuss the trade-offs it exposes, positions the empirical study explicitly against the gap identified in Section 2.7, and offers recommendations for how the frameworks surveyed in Section 2.4 could evolve to account for resolver diversity.

3.2 Depth of Analysis versus Practical Adoption

3.2.1 The General Trade-off

A recurring tension in the reproducible-builds and supply-chain-security literatures is between analytical depth and practical cost. Full bit-for-bit reproducibility verification, or a complete SLSA level 4 pipeline, gives the strongest guarantee but requires infrastructure most projects do not have: hardened, ephemeral build platforms; signing-key or OIDC identity management; and, for reproducibility specifically, a second independent party willing and able to perform the verification rebuild. In contrast, resolver diversity checking imposes minimal operational overhead, requiring only alternate configuration parameter execution within existing CI/CD pipelines. It does not inherently require new identity or key-management infrastructure, nor an independent verification party, because the same pipeline can execute the comparison locally.

3.2.2 Why Cheapness Matters for Adoption

This asymmetry is worth dwelling on because adoption, not theoretical strength, is usually the binding constraint on whether a security technique has any real-world effect. A technique that is maximally rigorous but requires infrastructure investment reaches only the organizations that can afford that investment — typically large, already well-resourced ones, which are not the organizations dependency- confusion and typosquatting attacks (Table 1.3) most often target in practice, since smaller projects and organizations are disproportionately represented among victims precisely because they lack this kind of infrastructure. A technique that is both cheap and only partially effective is, from an ecosystem-wide risk-reduction standpoint, plausibly more valuable than one that is maximally effective but reaches only well-resourced adopters. This is the practical case, independent of its novelty as a research question, for measuring whether resolver diversity produces a usable signal at all.

3.3 Resolver Diversity as a Complement, Not a Replacement

It is important to be precise about what resolver diversity can and cannot claim to provide, relative to the frameworks in Section 2.4. SLSA provenance and SBOMs attest to *what was built and how*; resolver diversity, at best, offers a cheap, local signal about *whether an alternative, equally legitimate resolution of the same source tree would have produced a different result.* It cannot replace cryptographic attestation — it proves nothing to a third party who does not trust the pipeline running the comparison, since there is no signature or external witness involved — and it says nothing about a compromise that does not manifest as a version-conflict outcome (a backdoor injected directly into a compromised package's own code, rather than mediated through a manipulated dependency declaration, would not be caught by this technique at all — as Section 1.5.1's three real incidents illustrate directly). Its value, if the empirical study confirms it has any, is narrow and specific: cases where an attacker's leverage is mediated by resolver version-selection behavior, which is exactly the mechanism dependency-confusion attacks rely on (Table 1.3).

3.4 Trustworthiness of Industrial versus Academic Approaches

3.4.1 Industrial Tools

Industrial supply-chain tooling (dependency scanners, SBOM generators, registry-side abuse detection) is typically evaluated by its vendor against its own detection rate on a proprietary or semi-public benchmark, optimizes heavily for low false-positive rates (since a security tool that interrupts every build with false alarms is quickly disabled by its users), and is usually a black box whose internal resolution assumptions are not published.

3.4.2 Academic Tools and This Study's Position

Academic work, including the empirical study developed in the companion dissertation, optimizes differently: for a precisely stated, falsifiable claim (here, a measured divergence rate and a measured hijack-detection outcome, both reported with their exact denominators and with the methodology that produced them fully disclosed), rather than for production-readiness or a polished false-positive rate. This is a genuine trade-off, not merely a difference in maturity: the companion dissertation's own methodology chapter documents, in detail, engineering problems (a stdin-sharing bug that silently dropped roughly half a project list; a batching bug that corrupted result rows) that a mature industrial tool would very likely have already encountered and fixed silently, without a corresponding published record of what was wrong or why. This transparency is itself a research contribution distinct from the headline results, since it lets a future study or tool avoid re-discovering the same pitfalls independently — a discipline the build-configuration-maintenance literature (Section 2.5) suggests is broadly under-documented across the ecosystem, not merely in this one study.

3.4.3 Bridging the Gap

The most productive relationship between the two traditions, in this specific case, is sequential rather than competing: an academic study measures whether a signal exists and is frequent enough to matter (this dissertation's contribution), and, if it is, industrial tooling is better positioned to productionize it robustly, at the false-positive rate and integration ergonomics real adoption requires. This dissertation and its companion make the first claim; whether the second follows is future work, not a claim made here.

3.5 Threats to the Validity of a Purely Literature-Based Argument

Because this dissertation is a state-of-the-art survey, its central claim — that resolver diversity is an underexplored but promising signal — is necessarily an argument from absence: we did not find prior work measuring it, which is weaker evidence than prior work having tried and failed. Three considerations temper this. First, the underlying mechanism (Resolver 2.x's configurable strategies) is recent enough, tied to Maven 4's release, that the absence of prior study is at least partly explained by novelty rather than by the idea having been tried and found unfruitful. Second, the adjacent diversity literature (Section 2.6) gives the idea a plausible theoretical basis rather than treating it as speculative. Third, and most importantly, the companion engineering dissertation does not rest on this literature-based argument alone: it tests the claim directly, on 572 real projects and through a controlled dependency-hijack simulation, and reports the actual divergence rate and the actual detection outcome rather than an assumed one.

3.5.1 Security Trade-offs of Conflict-Resolution Strategies

Resolver diversity should not be interpreted as evidence that one conflict-resolution strategy is universally safer than another. Nearest-wins can reduce exposure to attacks that exploit artificially inflated versions because a higher version located deeper in the dependency graph does not automatically override a nearer declaration. However, the same property can produce an opposite security effect when a nearer dependency specifies an older vulnerable version while a deeper transitive path provides a patched release. In such cases, nearest-wins can suppress the patched version and preserve vulnerability exposure. This creates a form of *vulnerability shadowing*: the resolution policy determines not only which version satisfies the graph, but also which security update becomes effective in the final classpath.

The trade-off is consistent with the dependency-maintenance literature, which shows that projects often retain outdated dependencies and that security-fix adoption can lag substantially behind release. In this setting, a resolver policy that favors graph proximity may preserve a vulnerable nearer version even when a patched transitive version is available [16, 18, 21]. Conversely, a policy that favors the highest compatible version can expose a build to version-inflation or compatibility risks when a deeper dependency introduces a newer release than the root project intended [29, 4, 5].

Accordingly, the security value of a resolver strategy is threat-dependent. Rather than prescribing nearest-wins as a universal security default, build tools should make

the selected resolution policy explicit, record it as part of build provenance, and provide checks that flag when the selected version is older, known vulnerable, or materially less secure than an available compatible alternative. A practical policy may therefore combine the resolver's normal selection rule with vulnerability-aware validation, so that conflict resolution and security assessment remain distinct but coordinated decisions.

3.6 Recommendations for Future Frameworks

Drawing the discussion above together, we offer four concrete recommendations, aimed at the maintainers of the frameworks surveyed in Section 2.4 and at future researchers extending this line of work.

Record the resolver configuration. SBOM formats and SLSA provenance schemas should add an optional field recording the exact resolver configuration (collector algorithm, version-selection strategy, and tool version) used to produce a given build, at negligible schema cost, enabling exactly the kind of cross-check this dissertation argues is currently invisible to consumers of those artifacts.

Treat resolver-configurability as a security property, not merely a build-engineering conv
Build-tool maintainers considering whether to expose previously hard-coded resolution behavior as configuration (as Maven 4 did) should weigh the diversity-verification use case documented here alongside the more conventional flexibility and performance motivations that typically drive such changes.

Evaluate resolution strategies as security trade-offs, not as universal defaults. Resolver diversity should not be interpreted as evidence that one conflict-resolution strategy is universally safer than another. Nearest-wins can reduce exposure to attacks that exploit artificially inflated versions because a higher version located deeper in the dependency graph does not automatically override a nearer declaration. However, the same property can produce an opposite security effect when a nearer dependency specifies an older vulnerable version while a deeper transitive path provides a patched release. In such cases, nearest-wins can suppress the patched version and preserve vulnerability exposure. This creates a form of *vulnerability shadowing*: the resolution policy determines not only which version satisfies the graph, but also which security update becomes effective in the final classpath. The security value of a strategy is therefore threat-dependent. Build tools should make the selected policy explicit, record it as part of build provenance, and provide checks that flag when the selected version is older or associated with a known vulnerability than an available compatible alternative.

Extend diversity checking beyond resolver configuration. The companion dissertation's own evaluation identifies POM declaration order as a factor resolver-configuration diversity alone does not cover; future tooling implementing this idea should treat the specific dimensions varied here as a starting point, not a complete list.

3.7 Conclusion

The literature makes a coherent case that resolver-configuration diversity sits at an intersection — of reproducibility research, resolution- semantics research, supply-chain integrity frameworks, and build- configuration maintenance research — that no single existing thread fully covers, that the general principle of comparing diverse, independently-produced outputs is well established enough to motivate testing it here specifically, and that its low cost relative to existing integrity frameworks is itself a reason to test it rather than dismiss it as redundant with more rigorous techniques. Whether the signal this produces is frequent enough, and clean enough, to be practically useful is an empirical question the literature alone cannot answer. The General Conclusion summarizes this dissertation's contribution and hands off to the engineering dissertation, where that empirical question is answered.

General Conclusion

This dissertation surveyed four research threads — build reproducibility, dependency-resolution semantics, software supply-chain integrity frameworks, and build-configuration maintenance research — and showed that, read together, they leave a specific, concrete gap: Maven Resolver 2.x's newly-configurable collector and version-selector strategies are a variable no existing study has measured at scale, despite sitting exactly at the point where all four threads overlap. Chapter 1 showed that Maven's single-winner conflict-resolution design is a deliberate point in a wider space of ecosystem choices, and that Resolver 2.x's three configurable dimensions are not uniformly meaningful: one of the three, traced here to exact source locations in Maven Resolver and Maven Core rather than merely observed as ineffective, is computed correctly by the resolver library and then silently discarded by a second, independent, hardcoded implementation inside Maven Core, so it never reaches a real build regardless of how it is configured. Chapter 2 showed that the reproducible-builds literature treats resolver output as an opaque part of the build it verifies rather than as a distinct source of legitimate divergence; that the dependency-confusion literature treats the resolver's default behavior as a fixed attack surface rather than a configurable one; that supply-chain integrity frameworks assume resolution is deterministic without recording or cross-checking the configuration that produced a given attestation; and that the adjacent build-configuration-maintenance literature, while establishing that build behavior already drifts for reasons unrelated to source code, has not itself treated resolver configuration as a variable worth measuring. The general principle needed to close this gap — comparing independently-produced, diverse outputs of the same computation as a verification signal — is well established in the N-version programming and Diverse Double-Compiling literature, but has not previously been applied to dependency-resolver configuration. Chapter 3 argued that this gap is worth closing specifically because the technique it points to is unusually cheap relative to the infrastructure existing integrity frameworks require, and closed with concrete recommendations for how those frameworks could record and expose resolver-configuration choice.

From Literature to Evidence

This dissertation's claim — that resolver diversity is an underexplored but theoretically well-motivated signal — is, by construction, an argument from the literature's silence rather than from a measurement. That silence is informative because the resolver mechanism examined here became configurable only recently, but it is not, by itself, evidence that the signal is frequent, stable, or operationally useful. These properties therefore remain empirical questions for the companion engineering study and for future replication work. This dissertation deliberately does not present the companion study's measurements as evidence generated by the present survey.

Threats to the Validity of a Purely Literature-Based Argument

Because this dissertation is a state-of-the-art survey, its central claim is necessarily an argument from absence: we did not find prior work measuring resolver-configuration diversity, which is weaker evidence than prior work having tried and failed. Chapter 3 tempers this in three ways — the underlying mechanism's novelty, the plausible theoretical basis the adjacent diversity literature provides, and the explicit separation between the literature-based contribution of this dissertation and the empirical evaluation conducted in the companion engineering study. A further, narrower limitation specific to this dissertation is that our search of the literature, while systematic across the four threads identified above, was not a formal systematic literature review with a pre-registered search protocol; it is possible that a narrow, unindexed, or very recent piece of work measuring this exact variable exists and was not located.

Handoff

Where this dissertation asks *whether the literature has already answered these questions*, the companion engineering dissertation asks *what the data shows*. Its experimental results are outside the evidentiary scope of this survey and are therefore not treated here as findings of the present dissertation. The appropriate role of the companion study is to test, on real projects, whether the gap identified by this literature review corresponds to a measurable and practically useful phenomenon.

Future Work

Beyond the concrete recommendations already made to framework maintainers in Chapter 3, four directions follow from the gap this dissertation identifies and the companion engineering dissertation's empirical findings, aimed at future researchers rather than at tool maintainers.

- **Extend diversity checking to other build ecosystems.** This dissertation and its companion restrict themselves, deliberately, to Apache Maven, the only widely deployed build tool whose conflict-resolution behavior is configurable enough to make the experiment possible at all. Gradle, npm, and pip each implement their own, differently-structured conflict-resolution defaults; whether the general pattern observed here — one dimension of resolver behavior mattering for build output and security, others not — is specific to Maven Resolver's particular design or a more general property of how conflict-resolution algorithms tend to be structured is an open question this dissertation's method, but not its data, could be extended to answer.

- **Test deeper, multi-hop compromise chains.** The companion dissertation's hijack simulation models a single-hop compromise: an attacker one edge away from the shared, contested dependency. Real supply-chain compromises can in principle propagate through longer chains of transitive trust; whether resolver-strategy choice remains as decisive a factor at two or more hops of separation, or whether its protective effect weakens with distance from the root project, is untested here.

- **Establish whether POM-order manipulation is attacker-exploitable.** The companion dissertation's evaluation identifies POM declaration order, not resolver configuration, as the actual determinant of split-package class-shadowing outcomes, but the experiment demonstrating this edited the root project's own POM directly. Whether an attacker who has compromised only an intermediate dependency, one the root project declares directly but which itself declares further dependencies, can influence that ordering without root-level access remains an open, untested question, and the answer determines whether this finding is a second confirmed attack vector or merely a technical curiosity about how Maven resolves same-named classes.

- **Standardize resolver-configuration recording in SBOM and SLSA metadata.** Chapter 3 already recommends this to framework maintainers as a matter of practice; the corresponding research question is whether such a field, once standardized, actually changes detection outcomes in the field — that is, whether consumers who cross-check recorded resolver configuration against an independent

rebuild catch anything in practice that they would otherwise have missed, which would require a follow-up study of real, deployed pipelines rather than the controlled corpus used here.

Bibliography

[1] C. Lamb and S. Zacchiroli, "Reproducible Builds: Increasing the Integrity of Software Supply Chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2022, doi: 10.1109/MS.2021.3073045.

[2] M. Keshani, T.-G. Velican, G. Bot, and S. Proksch, "AROMA: Automatic Reproduction of Maven Artifacts," *Proceedings of the ACM on Software Engineering*, vol. 1, FSE, 2024, doi: 10.1145/3643764.

[3] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491–1501, 1985.

[4] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Dependency Solving: A Separate Concern in Component Evolution Management," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2228–2240, 2012, doi: 10.1016/j.jss.2012.02.018.

[5] P. Abate, R. Di Cosmo, G. Gousios, and S. Zacchiroli, "Dependency Solving Is Still Hard, but We Are Getting Better at It," in *2020 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 547–551, 2020, doi: 10.1109/SANER48275.2020.9054837.

[6] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "OPIUM: Optimal Package Install/Uninstall Manager," in *29th International Conference on Software Engineering (ICSE)*, pp. 178–187, 2007, doi: 10.1109/ICSE.2007.59.

[7] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Why Do Software Packages Conflict?" in *2012 9th IEEE Working Conference on Mining Software Repositories*, pp. 141–150, 2012, doi: 10.1109/MSR.2012.6224274.

[8] M. Janota, I. Lynce, V. Manquinho, and J. Marques-Silva, "PackUp: Tools for Package Upgradability Solving," *Journal of Satisfiability, Boolean Modeling and Computation*, vol. 8, no. 1–2, 2012, doi: 10.3233/SAT190090.

[9] P. Trezentos, I. Lynce, and A. L. Oliveira, "Apt-pbo: Solving the Software Dependency Problem Using Pseudo-Boolean Optimization," in *25th IEEE/ACM Interna-*

tional Conference on Automated Software Engineering (ASE), pp. 427–436, 2010, doi: 10.1145/1858996.1859087.

[10] C. Michel and M. Rueher, "Handling Software Upgradeability Problems with MILP Solvers," *Electronic Proceedings in Theoretical Computer Science*, vol. 29, pp. 1–10, 2010, doi: 10.4204/EPTCS.29.1.

[11] J. Argelich, D. Le Berre, I. Lynce, J. Marques-Silva, and P. Rapicault, "Solving Linux Upgradeability Problems Using Boolean Optimization," *Electronic Proceedings in Theoretical Computer Science*, vol. 29, pp. 11–22, 2010, doi: 10.4204/EPTCS.29.2.

[12] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "A Modular Package Manager Architecture," *Information and Software Technology*, vol. 55, no. 2, pp. 459–474, 2013, doi: 10.1016/j.infsof.2012.09.002.

[13] R. Di Cosmo and S. Zacchiroli, "Feature Diagrams as Package Dependencies," in *Software Product Lines: Going Beyond*, LNCS 6287, pp. 476–480, Springer, 2010, doi: 10.1007/978-3-642-15579-6_40.

[14] P. Trezentos, "Comparison of PBO Solvers in a Dependency Solving Domain," *Electronic Proceedings in Theoretical Computer Science*, vol. 29, pp. 23–31, 2010, doi: 10.4204/EPTCS.29.3.

[15] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache Community Upgrades Dependencies: An Evolutionary Study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015, doi: 10.1007/s10664-014-9325-9.

[16] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration," *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018, doi: 10.1007/s10664-017-9521-5.

[17] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a Library: A Study of the Latency to Adopt the Latest Maven Release," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 520–524, 2015, doi: 10.1109/SANER.2015.7081869.

[18] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the Release, Adoption, and Propagation of npm Vulnerability Fixes," *Empirical Software Engineering*, vol. 26, article 47, 2021, doi: 10.1007/s10664-021-09951-x.

[19] A. Decan, T. Mens, and P. Grosjean, "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019, doi: 10.1007/s10664-017-9589-y.

[20] A. Decan, T. Mens, and E. Constantinou, "On the Evolution of Technical Lag in the npm Package Dependency Network," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 404–414, 2018, doi: 10.1109/ICSME.2018.00050.

[21] A. Zerouali, T. Mens, J. M. González-Barahona, A. Decan, E. Constantinou, and G. Robles, "A Formal Framework for Measuring Technical Lag in Component Repositories—and Its Application to npm," *Journal of Software: Evolution and Process*, vol. 31, no. 8, article e2157, 2019, doi: 10.1002/smr.2157.

[22] S. Mirhosseini and C. Parnin, "Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-Date Dependencies?" in *2017 IEEE/ACM 32nd International Conference on Automated Software Engineering (ASE)*, 2017, doi: 10.1109/ASE.2017.8115621.

[23] A. J. Jafari, D. E. Costa, E. Shihab, and R. Abdalkareem, "Dependency Update Strategies and Package Characteristics," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, article 149, pp. 1–29, 2023, doi: 10.1145/3603110.

[24] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem," *Empirical Software Engineering*, vol. 26, article 45, 2021, doi: 10.1007/s10664-020-09914-8.

[25] C. Soto-Valero, T. Durieux, and B. Baudry, "A Longitudinal Analysis of Bloated Java Dependencies," in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1021–1031, 2021, doi: 10.1145/3468264.3468589.

[26] N. Harrand, A. Benelallam, C. Soto-Valero, F. Bettega, O. Barais, and B. Baudry, "API Beauty Is in the Eye of the Clients: 2.2 Million Maven Dependencies Reveal the Spectrum of Client–API Usages," *Journal of Systems and Software*, vol. 184, article 111134, 2022, doi: 10.1016/j.jss.2021.111134.

[27] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 344–348, 2019, doi: 10.1109/MSR.2019.00060.

[28] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The Emergence of Software Diversity in Maven Central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 333–343, 2019, doi: 10.1109/MSR.2019.00059.

[29] J. Dietrich, D. J. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency Versioning in the Wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 349–359, 2019, doi: 10.1109/MSR.2019.00061.

[30] E. Wittern, P. Suter, and S. Rajagopalan, "A Look at the Dynamics of the JavaScript Package Ecosystem," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 351–361, 2016, doi: 10.1145/2901739.2901743.

[31] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm," in *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 385–395, 2017, doi: 10.1145/3106237.3106267.

[32] C. Désarmeaux, A. Pecatikov, and S. McIntosh, "The Dispersion of Build Maintenance Activity across Maven Lifecycle Phases," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 492–495, 2016, doi: 10.1145/2901739.2903498.

[33] C. Macho, S. Beyer, S. McIntosh, and M. Pinzger, "The Nature of Build Changes: An Empirical Study of Maven-Based Build Systems," *Empirical Software Engineering*, vol. 26, article 32, 2021, doi: 10.1007/s10664-020-09926-4.

[34] C. Macho, S. McIntosh, and M. Pinzger, "Automatically Repairing Dependency-Related Build Breakage," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 106–117, 2018, doi: 10.1109/SANER.2018.8330201.

[35] F. Hassan and X. Wang, "HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts," in *40th International Conference on Software Engineering (ICSE)*, 2018, doi: 10.1145/3180155.3180181.

[36] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An Empirical Study of Bugs in Software Build Systems," in *13th International Conference on Quality Software (QSIC)*, pp. 200–203, 2013, doi: 10.1109/QSIC.2013.60.

[37] M. Maes-Bermejo, M. Gallego, F. Gortázar, G. Robles, and J. M. González-Barahona, "Revisiting the Building of Past Snapshots—a Replication and Reproduction Study," *Empirical Software Engineering*, vol. 27, article 65, 2022, doi: 10.1007/s10664-022-10117-6.

[38] M. Maes-Bermejo, M. Gallego, F. Gortázar, G. Robles, and J. M. González-Barahona, "Testing the Past: Can We Still Run Tests in Past Snapshots for Java Projects?" *Empirical Software Engineering*, vol. 29, article 127, 2024, doi: 10.1007/s10664-024-10530-z.

[39] M. Maes-Bermejo, A. Serebrenik, M. Gallego, F. Gortázar, G. Robles, and J. M. González-Barahona, "Hunting Bugs: Towards an Automated Approach to Identifying Which Change Caused a Bug through Regression Testing," *Empirical Software Engineering*, vol. 29, article 66, 2024, doi: 10.1007/s10664-024-10479-z.

[40] J. M. González-Barahona and G. Robles, "On the Reproducibility of Empirical Software Engineering Studies Based on Data Retrieved from Development Repositories," *Empirical Software Engineering*, vol. 17, pp. 75–89, 2012, doi: 10.1007/s10664-011-9181-9.

[41] J. M. González-Barahona and G. Robles, "Revisiting the Reproducibility of Empirical Software Engineering Studies Based on Data Retrieved from Development Repositories," *Information and Software Technology*, vol. 164, article 107318, 2023, doi: 10.1016/j.infsof.2023.107318.

[42] S. Butler, J. Gamalielsson, B. Lundell, C. Brax, A. Mattsson, T. Gustavsson, J. Feist, B. Kvarnström, and E. Lönroth, "On Business Adoption and Use of Reproducible Builds for Open and Closed Source Software," *Software Quality Journal*, vol. 31, pp. 687–719, 2023, doi: 10.1007/s11219-022-09607-z.

[43] D. Jayasuriya, S. Ou, S. Hegde, V. Terragni, J. Dietrich, and K. Blincoe, "An Extended Study of Syntactic Breaking Changes in the Wild," *Empirical Software Engineering*, vol. 30, article 42, 2025, doi: 10.1007/s10664-024-10563-4.

[44] R. Paramitha and F. Massacci, "Technical Leverage Analysis in the Python Ecosystem," *Empirical Software Engineering*, vol. 28, article 139, 2023, doi: 10.1007/s10664-023-10355-2.

[45] A. Gkortzis, D. Feitosa, and D. Spinellis, "Software Reuse Cuts Both Ways: An Empirical Analysis of Its Relationship with Security Vulnerabilities," *Journal of Systems and Software*, vol. 172, article 110653, 2021, doi: 10.1016/j.jss.2020.110653.

[46] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web," in *Network and Distributed System Security Symposium (NDSS)*, 2017, doi: 10.14722/ndss.2017.23414.

[47] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2020)*, LNCS 12223, pp. 23–43, Springer, 2020, doi: 10.1007/978-3-030-52683-2_2.

[48] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in *28th USENIX Security Symposium*, pp. 995–1010, 2019.

[49] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains," in *2023 IEEE Symposium on Security and Privacy*, pp. 1509–1526, 2023, doi: 10.1109/SP46215.2023.10179304.

[50] P. Ladisa, S. E. Ponta, A. Sabetta, M. Martinez, and O. Barais, "Journey to the Center of Software Supply Chain Attacks," *IEEE Security & Privacy*, vol. 21, no. 6, pp. 34–49, 2023, doi: 10.1109/MSEC.2023.3302066.

[51] S. Nocera, S. Vegas, G. Scanniello, and N. Juristo, "Software Composition Analysis and Supply Chain Security in Apache Projects: An Empirical Study," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 103–115, 2025, doi: 10.1109/MSR66628.2025.00027.

[52] B. Gokkaya, L. Aniello, and B. Halak, "Software Supply Chain: A Taxonomy of Attacks, Mitigations and Risk Assessment Strategies," *Journal of Information Security and Applications*, vol. 97, article 104324, 2026, doi: 10.1016/j.jisa.2025.104324.

[53] A. Decan, T. Mens, and M. Claes, "An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, doi: 10.1109/SANER.2017.7884604.

[54] A. Birsan, "Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies," *Medium*, 2021. [Online]. Available: `https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610`

[55] D. A. Wheeler, *Fully Countering Trusting Trust through Diverse Double-Compiling*, Ph.D. dissertation, George Mason University, 2009.

[56] OpenSSF, *SLSA: Supply-chain Levels for Software Artifacts*. [Online]. Available: `https://slsa.dev`

[57] in-toto Community, *in-toto: A Framework to Secure the Integrity of Software Supply Chains*. [Online]. Available: `https://in-toto.io`

[58] Sigstore Project, *Sigstore*. [Online]. Available: `https://www.sigstore.dev`

[59] Reproducible Builds Project, *Reproducible Builds*. [Online]. Available: `https://reproducible-builds.org`

[60] OWASP Foundation, *CycloneDX: A Full-Stack Bill of Materials Standard*. [Online]. Available: https://cyclonedx.org

[61] npm, Inc., "Details about the event-stream Incident," 2018. [Online]. Available: https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident

[62] GitHub Security Lab, "Malicious Versions of ua-parser-js Package," 2021. [Online]. Available: https://github.com/github/securitylab/issues/175

[63] A. Freund, "Backdoor in upstream xz/liblzma leading to SSH server compromise," oss-security disclosure, 2024. [Online]. Available: https://www.openwall.com/lists/oss-security/2024/03/29/4

[64] The Apache Software Foundation, *Maven Resolver Documentation*. [Online]. Available: https://maven.apache.org/resolver/index.html

[65] Gradle Inc., *Gradle User Manual: Dependency Resolution*. [Online]. Available: https://docs.gradle.org/current/userguide/dependency_resolution.html

[66] Reproducible Builds Project, *diffoscope*. [Online]. Available: https://diffoscope.org

[67] Debian/Reproducible Builds Project, *reprotest*. [Online]. Available: https://salsa.debian.org/reproducible-builds/reprotest

[68] Linux Foundation / SPDX Workgroup, *SPDX: Software Package Data Exchange Specification*. [Online]. Available: https://spdx.dev

[69] jvm-repo-rebuild, *Reproducible Central*. [Online]. Available: https://github.com/jvm-repo-rebuild/reproducible-central

Appendix A

Glossary of Terms

This appendix collects the domain-specific terms used throughout this dissertation that are not already covered by the List of Abbreviations. Terms are defined once here rather than re-explained at each use in the body chapters.

Collector The dependency-graph traversal algorithm used during Maven's collection phase (recursively discovering the dependency graph from each project's manifest), configured via `aether.dependencyCollector.impl`. Maven Resolver 2.x exposes it as depth-first (`df`) or breadth-first (`bf`).

Version-selector strategy The policy Maven Resolver uses to pick exactly one winning version when the dependency graph reaches the same artifact coordinate via multiple paths, configured via `aether.conflictResolver.versionSelector.selectionStrategy`: either `nearest` (the declaration closest to the root project wins) or `highest` (the numerically highest version wins).

Visiting (graph-flattening) strategy The order in which a resolved dependency graph is traversed to produce the final, ordered classpath, configured via `aether.system.dependencyVis` (`preOrder`, `postOrder`, or `levelOrder`). Traced in Section 1.3 to two independent implementations inside Maven Resolver and Maven Core, only one of which a real build actually uses.

outputTimestamp The Maven property (`project.build.outputTimestamp`) that pins a build's embedded timestamps to a fixed value, following standard reproducible-builds practice, so that ordinary build-to-build timestamp non-determinism cannot masquerade as resolver-driven divergence.

Split-package collision A situation in which two separate dependencies each define a class with the same fully-qualified name; whichever class is placed first on the resolved classpath silently shadows the other at runtime.

Dependency confusion An attack technique in which an attacker registers a malicious package under a coordinate a resolver is likely to prefer (typically a public registry, or a higher version number), without needing to compromise the legitimate package's own publish access.

Resolver diversity The primitive this dissertation surveys and the companion engineering dissertation measures empirically: deliberately building the same source tree under multiple, independently legitimate resolver configurations and comparing the outputs, as a signal for both measuring reproducibility and detecting tampering.

Reproducible Central A dataset recording, for several hundred real Java projects, a verified independent-rebuild outcome together with the exact build command and JDK used to achieve it. The source of the project corpus used in the companion engineering dissertation.

Diamond conflict A dependency-graph shape in which two independent paths from the root project reach the same coordinate, at the same depth, declaring different versions — the specific shape required for `nearest`-wins and `highest`-wins resolution to genuinely disagree on an outcome.