Faculty of Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Reliable Bytecode-centric Detection of Vulnerable Open-Source Software Dependencies

Stefan Schott

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisors

Prof. Dr. Eric Bodden
Dr. Serena Elisa Ponta

Paderborn, May 9, 2026

Stefan Schott
Reliable Bytecode-centric Detection of Vulnerable Open-Source Software Dependencies
Dissertation, May 9, 2026
Advisors: Prof. Dr. Eric Bodden and Dr. Serena Elisa Ponta

Paderborn University
Research Group Secure Software Engineering
Department of Computer Science
Faculty of Computer Science, Electrical Engineering and Mathematics
Warburger Straße 100
33098 Paderborn

Abstract

The inclusion of Open-Source Software (OSS) into modern software projects has become ubiquitous. On average, 71% of the code in typical Java projects comes from OSS dependencies, making them the dominant component of modern software code bases. This high degree of OSS reliance comes with a considerable security risk of adding known security vulnerabilities to a code base. To remedy this risk, researchers and companies have developed various *dependency scanners*, which try to identify inclusions of known-to-be-vulnerable OSS dependencies. However, due to a reliance on metadata or source code, there are still challenges that modern dependency scanners do not overcome, especially when it comes to dependency modifications, such as re-compilations, re-bundlings or re-packagings, which are common in the Java ecosystem.

To overcome these challenges, we present JARALYZER, a *bytecode-centric* dependency scanner for Java. JARALYZER does not rely on the metadata or the source code of the included OSS dependencies being available but directly analyzes a dependency's bytecode. JARALYZER achieves this by incorporating the novel techniques presented in this thesis.

First, it leverages our *targeted compilation* approach JESS, which uses slicing and stubbing techniques to isolate and compile only targeted parts of a Java project's code base. Because compilation of large, real-world Java projects is notoriously challenging, this targeted strategy significantly improves compilation success rates and enables fix commits to be compiled in isolation. These isolated and compiled fix commits serve as the basis for JARALYZER 's dependency analysis. Afterwards, to remove compilation-induced differences, it applies our novel *bytecode normalization* technique JNORM to create a code representation independent of the compilation environment. This enables a subsequent comparison between included dependencies and vulnerability-fixing code, which JARALYZER conducts using a code property graph–based technique. This approach allows for fine-grained detection of vulnerability fixes, even when the code has evolved over time. By combining these techniques, JARALYZER is able to reliably detect known-to-be-vulnerable dependencies in Java projects, even when those dependencies have been modified.

Our evaluation of JESS on 347 OSS Java projects shows that it can successfully compile 72% of methods and constructors in isolation, of which 89% have bytecode equal to the original one. Furthermore, by applying JESS on the fix-commit database Project KB, which serves as the basis of JARALYZER, it is possible to successfully compile 90% of the entries in contrast to less than 11% when relying on provided build scripts. Moreover, our evaluation of JNORM on 322 OSS Java projects shows that solely the act of incrementing a compiler version may cause differences in 46% of all resulting bytecode files, even when the source code is identical. By applying bytecode normalization, one can remove 99% of compilation-induced differences, thus acting as a crucial enabler for bytecode-centric dependency scanning. Finally, our evaluation of JARALYZER across 56 popular OSS components demonstrates that it outperforms other state-of-the-art dependency scanners in detecting vulnerabilities within modified dependencies. It is the only scanner capable of reliably identifying known-to-be-vulnerable dependencies, even when modified. But even when applied to unmodified dependencies, JARALYZER outperforms the current state-of-the-art code-centric scanner Eclipse Steady by detecting 28 more true vulnerabilities and yielding 29 fewer false warnings.

Zusammenfassung

Die Einbindung von Open-Source-Software (OSS) in moderne Softwareprojekte ist allgegenwärtig geworden. Im Durchschnitt stammen 71% des Codes in typischen Java-Projekten aus OSS-Abhängigkeiten, was sie zum dominanten Bestandteil moderner Softwarecodebasen macht. Dieser hohe Grad an OSS-Abhängigkeit birgt ein erhebliches Sicherheitsrisiko, da dadurch bekannte Sicherheitslücken in eine Codebasis eingebracht werden können. Um dieses Risiko zu verringern, haben Forschende und Unternehmen verschiedene Dependency Scanner entwickelt, die versuchen, die Einbindung bekanntermaßen verwundbarer OSS-Abhängigkeiten zu identifizieren. Aufgrund ihrer Abhängigkeit von Metadaten oder Quellcode bestehen jedoch weiterhin Herausforderungen, die moderne Dependency Scanner nicht überwinden—insbesondere im Hinblick auf Modifikationen von Abhängigkeiten wie Rekompilationen, Neubündelungen oder Umbenennungen, die im Java-Ökosystem weit verbreitet sind.

Um diese Herausforderungen zu bewältigen, präsentieren wir JARALYZER, einen bytecode-zentrierten Dependency Scanner für Java. JARALYZER ist weder auf Metadaten noch auf den Quellcode der eingebundenen OSS-Abhängigkeiten angewiesen, sondern analysiert direkt den Bytecode einer Abhängigkeit. Dies erreicht JARALYZER durch die in dieser Arbeit vorgestellten neuartigen Techniken. Zunächst nutzt es unseren Ansatz der gezielten Kompilation JESS, der Slicing- und Stubbing-Techniken einsetzt, um nur gezielt ausgewählte Teile einer Java-Codebasis zu isolieren und zu kompilieren. Dies erhöht die Kompilationserfolgsrate erheblich, ein ansonsten notorisch komplexes Problem, und ermöglicht die Kompilation von Fix-Commits in Isolation, die die Grundlage der Dependency-Scans von JARALYZER bilden. Anschließend werden mit unserer neuartigen Bytecode-Normalisierung JNORM aus Kompilierung resultierende Unterschiede entfernt, um eine von der jeweiligen Kompilierungsumgebung unabhängige Code-Repräsentation zu erzeugen. Dies ermöglicht einen anschließenden Vergleich zwischen eingebundenen Abhängigkeiten und sicherheitsrelevanten Codeänderungen, den JARALYZER mittels einer auf Code Property Graphs basierenden Technik durchführt. Dieser Ansatz erlaubt eine fein granulare Erkennung von Sicherheitsfixes, selbst wenn sich der Code im Laufe der Zeit weiterentwickelt hat. Durch die Kombination dieser Techniken kann JARALYZER bekanntermaßen verwundbare Abhängigkeiten in Java-Projekten zuverlässig erkennen—selbst dann, wenn diese Abhängigkeiten modifiziert wurden.

Unsere Evaluation von JESS auf 347 OSS-Java-Projekten zeigt, dass JESS in der Lage ist 72% aller Methoden und Konstruktoren isoliert zu kompilieren, wovon bei 89% Bytecode erzeugt wird, der mit dem ursprünglichen vollständig übereinstimmt.

Darüber hinaus ermöglicht die Nutzung von JESS auf der Fix-Commit-Datenbank Project KB, die als Grundlage für JARALYZER dient, die Kompilation von 90% der Einträge, im Vergleich zu weniger als 11% bei Verwendung der mitgelieferten Build-Skripte. Unsere Evaluation von JNORM auf 322 OSS-Java-Projekten zeigt zudem, dass allein das Verändern der Compiler-Version zu Unterschieden in 46% aller resultierenden Bytecode-Dateien führen kann, selbst bei identischem Quellcode. Durch die Anwendung der Bytecode-Normalisierung lassen sich 99% dieser von Kompilation bedingten Unterschiede entfernen, was JNORM zu einem entscheidenden Faktor für bytecode-zentriertes Dependency Scanning macht.

Schließlich zeigt unsere Evaluation von JARALYZER an 56 populären OSS-Komponenten, dass es andere gängige Dependency Scanner bei der Erkennung von Schwachstellen in modifizierten Abhängigkeiten übertrifft. Es ist der einzige Scanner, der bekanntermaßen verwundbare Abhängigkeiten auch dann zuverlässig identifizieren kann, wenn sie modifiziert wurden. Doch selbst bei der Anwendung auf unmodifizierte Abhängigkeiten übertrifft JARALYZER den aktuellen Stand der Technik des code-zentrierten Scanners Eclipse Steady, indem es 28 zusätzliche Schwachstellen erkennt und gleichzeitig 29 falsche Warnungen weniger erzeugt.

Acknowledgments

First, I would like to thank my supervisors, Eric Bodden and Serena Ponta. Eric gave me all the academic freedom I could wish for and encouraged me to explore my own ideas. He helped me learn how to navigate academia and its many challenges. Serena introduced me to the world of open-source security and supported me throughout all my research projects. Her frequent feedback, steady guidance and invaluable support always pushed me in the right direction.

Throughout the past years, I had the chance to work closely with SAP on several joint projects, where I met and collaborated with many great people. Besides my advisor Serena, I especially want to thank Wolfram Fischer, who contributed to many of my research efforts and always kept an eye on the technical details during our discussions. I also want to thank Henrik Plate, who worked with me during the early stages of my PhD, and Antonino Sabetta, who occasionally joined our meetings and offered helpful feedback.

I am very grateful to my long-time project partner and office mate, Jonas Klauke. Over the years, we supported each other in every possible way while working toward our shared goals. Jonas made sure that our PhD time was not only productive but also enjoyable and fun. I have many fond memories of our yearly stays at SAP's office in the south of France, including our frequent trips to the nearby beach.

Beyond the work presented in this thesis, I also had the chance to maintain and extend the open-source SootUp static analysis framework together with a few colleagues. I want to thank Kadiray Karakaya, Markus Schmidt, and Palaniappan Muthuraman for joining me in maintaining SootUp, organizing hackathons, and hosting tutorials in Copenhagen and Seoul.

My thanks also go to all my colleagues in the Secure Software Engineering research group. In particular, I want to thank Ashwin Prasad, Mugdha Khedkar, Michael Schlichting, Marcus Hüwe, Martin Mory, Faiza Tahir and my student assistant Michael Youkeim for being part of my PhD journey. I am also grateful to Vera Meyer, Nicole Graskamp, and Jürgen (Sammy) Maniera for handling many organizational matters so I could focus on research.

I would like to thank Felix Pauck for supervising my Master's thesis and later collaborating with me on my first publication. This work introduced me to static program analysis and sparked the interest that eventually led me to pursue a PhD.

My research was partially funded by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (GZ: SFB 901/3) under project number 160364472 (Projects Hektor and Reaktor), whose support I gratefully acknowledge. I also want to thank Ben Hermann, Juraj Somorovsky, and Simon Oberthür for serving on my defense committee.

I am deeply thankful to my parents, who always encouraged my curiosity and supported me throughout my education. I am grateful for my cats, Busy and Binky, whose constant companionship, welcome distractions, and timely reminders to take breaks sustained me through long days of research. Finally, I would like to thank my loving wife, Michelle, who stood by my side throughout my entire PhD journey, constantly encouraging me and celebrating every milestone with me.

Publications

This dissertation is an original work. However, parts of it have already been published in conference papers, of which the author of this thesis is also the lead author. In particular, this includes the following work:

- Stefan Schott, Wolfram Fischer, Serena Elisa Ponta, Jonas Klauke, and Eric Bodden. "Compilation of Commit Changes Within Java Source Code Repositories". In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*. IEEE, 2024, pp. 325–336
  Parts of Chapter 3 are taken from this paper.
- Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. "Java Bytecode Normalization for Code Similarity Analysis". In: *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 37:1–37:29
  Parts of Chapter 2 and Chapter 4 are taken from this paper.
- Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. "Bytecode-centric Detection of Known-to-be-vulnerable Dependencies in Java Projects". In: *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*. 2026
  Parts of Chapter 1, Chapter 2, Chapter 3 and Chapter 5 are taken from this paper.

Chapter Implementations and Data gives a complete overview of the available implementations and data sets.

Contents

1 Introduction **1**
- 1.1 Research Challenges . . . 2
- 1.2 Thesis Contributions . . . 5
- 1.3 Thesis Structure . . . 8

2 Background **9**
- 2.1 Dependency Management in Java . . . 9
- 2.2 Dependency Modifications . . . 10
- 2.3 Known Vulnerabilities . . . 11
- 2.4 Dependency Scanners . . . 12
- 2.5 Java Compilers and Bytecode . . . 14
- 2.6 SootUp and Jimple . . . 15
- 2.7 Code Property Graphs . . . 17

3 Targeted Compilation **19**
- 3.1 The Role of Compilation in Bytecode-centric Dependency Scanning . 21
- 3.2 Targeted Compilation with JESS . . . 22
  - 3.2.1 JESS Compilation Example . . . 24
  - 3.2.2 Slicing Exceptions . . . 26
  - 3.2.3 Stub Generation . . . 28
- 3.3 Compilation Heuristic for Compiling Commit Changes . . . 32
- 3.4 Evaluation . . . 35
  - 3.4.1 RQ1: How does JESS's compilation perform on popular and current Java projects? . . . 35
  - 3.4.2 RQ2: How similar is the bytecode obtained via JESS to the original bytecode? . . . 39
  - 3.4.3 RQ3: To what extent can JESS enable the successful compilation of fix-commit changes? . . . 41
  - 3.4.4 RQ4: To what extent can the compilation heuristic improve the success rate of compiling fix-commit changes? . . . 45
- 3.5 Threats to Validity . . . 46

3.6 Related Work . . . 47
3.7 Conclusion . . . 48

4 Bytecode Normalization **51**
4.1 Study: Usage of different Compilers and Target Levels in Java Projects 52
4.2 Java Bytecode Normalization with JNorm . . . 54
4.2.1 Investigation of Compilation Differences . . . 55
4.2.2 Overview of JNorm . . . 56
4.2.3 Jimple Transformation and Optimization . . . 57
4.2.4 Compilation Difference Transformation . . . 58
4.2.5 Standardization . . . 67
4.3 Evaluation . . . 68
4.3.1 Experimental Setup . . . 68
4.3.2 RQ5: Does the vendor of the JDK compiler influence bytecode generation? . . . 71
4.3.3 RQ6: How effective is JNorm in normalizing differences across JDK versions? . . . 72
4.3.4 RQ7: How effective is JNorm in normalizing differences across Java target levels? . . . 75
4.3.5 RQ8: How prevalent are the individual compilation difference transformations of JNorm? . . . 78
4.4 Threats to Validity . . . 79
4.5 Related Work . . . 80
4.6 Conclusion . . . 82

5 Bytecode-centric Dependency Scanning **85**
5.1 Bytecode-centric Dependency Scanning with Jaralyzer . . . 86
5.1.1 Knowledge-Base Creation . . . 88
5.1.2 Dependency Scanning . . . 90
5.1.3 Re-Packaging Detection . . . 92
5.2 Evaluation . . . 94
5.2.1 Experimental Setup . . . 95
5.2.2 RQ9: How does Jaralyzer compare to state-of-the-art dependency scanners in identifying *modified* known-to-be-vulnerable dependencies? . . . 96
5.2.3 RQ10: How does Jaralyzer compare to the state-of-the-art code-centric dependency scanner in identifying *unmodified* known-to-be-vulnerable dependencies? . . . 99
5.2.4 RQ11: How runtime-efficient is Jaralyzer? . . . 103

5.3 Threats to Validity 104
5.4 Related Work 105
5.5 Conclusion 106

6 Conclusion and Outlook **109**

Implementations and Data **113**

Bibliography **115**

Listings **131**

1 Introduction

The use of open-source software (OSS) has become ubiquitous in modern software development. OSS is now so prevalent in software projects that third-party dependencies account for the largest portion of the overall code base. According to a 2023 report by Endor Labs on the state of dependency management [Pla23], an average of 71% of the total code base in typical Java projects consists of OSS code. This also means that the use of vulnerable OSS dependencies poses a real threat to modern software projects as additionally highlighted by the OWASP Top Ten ranking of the most critical application security risks [Fou21]. The infamous *Log4shell* [ECZ22] and *Equifax Breach* [Lus18] incidents, which had a major impact on the cybersecurity world, further emphasized the risk associated with vulnerable OSS. To this end, various open-source [OWA24; Inc24c; Rer24; PPS20] but also commercial *dependency scanners* [Lim24; Inc24a; Men24; Lab24] have been developed. They all seek to help minimize the posed threat by identifying known-to-be-vulnerable OSS dependencies used within software projects. Most dependency scanners analyze *metadata* from dependency manifest files used by build tools or package managers to identify included OSS dependencies and their versions. They then compare this information against security advisories to determine whether any known-to-be-vulnerable components are present in the scanned project. However, as Ponta et al. [PPS20] have reported, relying solely on metadata has significant limitations. Security advisories often overstate the scope of vulnerable version ranges and frequently assign vulnerabilities to entire software projects, even when only a specific module is affected. Additionally, there are often substantial delays between when a vulnerability is fixed and when the corresponding advisory is updated. This leaves a window of increased risk for users during that time period [PM24]. To address this issue, Ponta et al. [PPS20] propose leveraging the actual *code changes* that fix vulnerabilities as the basis for dependency scanning. These code changes are available as fix commits in the project's source repository. They propose a *code-centric* approach that compares the code of included dependencies with the actual vulnerability fixes derived from fix commits in the project's source code repository.

The wide array of proposed tools and approaches, both in research and commercial contexts, underscores the significant risks associated with using OSS dependencies and highlights the importance of being able to reliably detect vulnerable OSS.

1.1 Research Challenges

While dependency scanners have gradually improved over the past years, important challenges remain, which considerably impair the detection performance of state-of-the-art tools. Dann et al. [Dan+21] conducted a study evaluating the performance of six state-of-the-art dependency scanners—both open-source and commercial—on Java projects where dependencies were included in *modified* forms. These modifications include re-compilations, re-bundlings and re-packagings, which as determined by Dann et al. are widespread modifications in the Java ecosystem. In their investigation of a sample of 254 vulnerable classes, they identified 67,196 artifacts on Maven Central that contained these classes in a modified form, highlighting the prevalence of such modifications. These modifications lead to changes or removals of dependency identifiers, timestamps, package structures and even changes within the dependency's bytecode itself. In such cases, the lack of precise metadata prevents the investigated scanners from consistently detecting known-to-be-vulnerable dependencies. A similar experiment conducted by Dietrich et al. [Die+24] further confirmed these results and emphasized the prevalence of dependency modifications that can transparently be applied by the popular Maven Shade Plugin, unbeknown to developers. Such modifications can even be found in the Java standard library. In these scenarios, developers may not realize that their projects include modified instances of known-to-be-vulnerable dependencies. Because state-of-the-art dependency scanners often fail to report such cases, this can lead to a false sense of security despite the presence of potentially severe vulnerabilities.

To overcome these challenges one cannot rely upon the availability of metadata corresponding to the included dependencies. Although the code-centric approach introduced by Ponta et al. [PPS20] demonstrates potential for identifying modified, known-to-be-vulnerable dependencies, their tool implementation, Eclipse Steady, still falls short in the experiments performed by Dann et al. A prerequisite of the code-centric approach is that both fix commits and scanned dependencies are available in the same code representation to be comparable. This assumption generally holds for languages like Python or JavaScript. In contrast, Java poses a challenge, as dependencies are usually distributed in compiled *bytecode*, whereas fix commits modify the original *source code*. To circumvent the issue of mismatched code representations, Eclipse Steady attempts to retrieve the source code of dependencies from Maven Central, the most popular Java artifact repository. However, when dependencies are modified, source code is typically unavailable. This prevents Eclipse Steady from consistently detecting known-to-be-vulnerable dependencies included in modified forms.

To perform a reliable dependency scan, regardless of any modifications, one must directly analyze the *bytecode* of the included dependencies. It is the only information consistently available. However, since the information about vulnerability fixes is typically not available on a bytecode level, but only in form of *fix commits* in an OSS project's source code repository, several requirements need to be fulfilled to facilitate a *bytecode-centric* detection of vulnerable OSS dependencies. Such an approach, which operates independently of metadata and source code, can detect known vulnerabilities even in modified dependencies.

Requirement 1: Transforming a dependency's bytecode and corresponding fix commits into a common representation

To compare a dependency's bytecode with its corresponding source code fix, both must first be transformed into a common, comparable representation. Originally, a dependency's bytecode has been obtained by compiling the project's source code. Vulnerability fixes are often fine-grained in nature, modifying only few code statements, in some cases a single instruction. Thus, to reliably and precisely identify vulnerabilities in a dependency's bytecode, one must reproduce the same process used to generate the original dependency's bytecode, which requires compiling the source code modified in the corresponding fix commits. However, this comes with multiple challenges.

Compiling, possibly old, repository snapshots is a complex and notoriously error-prone task [Tuf+17; Has+17; SP16]. This means that in many cases it is not even possible to obtain the bytecode in the first place. Furthermore, if one is able to obtain the bytecode of the fix commits, one does not know in which compilation environment (compiler, version and settings) the dependency has been compiled originally. This is problematic since different compilation environments will produce different bytecode for the exact same source code [DHB19].

Requirement 2: Addressing differences induced during compilation

As previously mentioned, compiling the exact same source code in different compilation environments will result in different bytecode, thus complicating a subsequent comparison [DHB19]. Coupled with the fine-grained nature of vulnerability fixes, compilation induced differences can easily mask the changes performed in a fix. One thus needs to *normalize* the resulting bytecodes to create a common representation

that is independent from the used compilation environment. However, this is challenging due to the multitude of existing Java compilers, compiler versions and Java Development Kit (JDK) versions [Cor25d; IBM14; Pro17; Fou25a; Ora25].

A first step towards normalizing bytecode for subsequent comparison has been proposed by Dann et al [DHB19]. They propose SootDiff, an approach that converts the bytecode into an intermediate representation and applies optimizations to normalize it. While this approach shows promising initial results, it still has several shortcomings such as only supporting up to JDK version 8 and having only been evaluated on simple programs.

However, to enable large-scale, bytecode-centric detection of known-to-be-vulnerable dependencies, this approach must be extended. First, it is essential to investigate which compilers and versions are actually used in practice. Furthermore, applying optimizations alone is insufficient, as several compilation-induced differences cannot be resolved through common optimization techniques. It is therefore necessary to devise *code transformations* that normalize the bytecode into a representation independent of the compilation environment.

Requirement 3: Avoiding reliance on fully qualified names

Having access to the normalized bytecode of the vulnerability fix is not yet sufficient for reliably detecting known-to-be-vulnerable dependencies across all types of modifications.

Scanning for the presence of the fix's bytecode across the entire code base of all included dependencies is both inefficient and potentially inaccurate. To overcome this, code-centric approaches like Eclipse Steady rely on the fully qualified names (FQN) of constructs (classes, interfaces or methods) to identify constructs potentially affected by a vulnerability [PPS18; PPS20]. The FQN uniquely identifies each construct by specifying its full package and class name. However, since modifications like re-packagings alter the FQNs of constructs, one cannot solely rely on them for the identification of potentially affected constructs and instead needs to identify them based on their actual code.

Requirement 4: Considering the evolution of software projects

Code-centric dependency scanning does not rely on manual classifications of vulnerable components and versions, but instead operates directly on the code. However,

software evolves over time with newer versions of projects being released continuously. This means that the project's code base, including vulnerability fixes, changes over time. Thus, to detect the presence of vulnerability fixes effectively, even as the fixed code evolves over time, a flexible comparison approach, which not only considers syntactical changes, is required to verify their presence.

1.2 Thesis Contributions

Overall, this thesis contributes a novel technique to reliably detect known-to-be-vulnerable OSS dependencies, even when included in a modified form. To achieve this, we introduce approaches for compiling targeted code changes within commits in source code repositories and normalizing bytecode by eliminating compilation-induced differences. Although these techniques were developed to support our bytecode-centric detection of known vulnerabilities in OSS dependencies, they are designed and implemented to be broadly applicable and can be used independently across different Java analysis contexts. A high-level overview of the individual contributions of this thesis is presented in Figure 1.1.

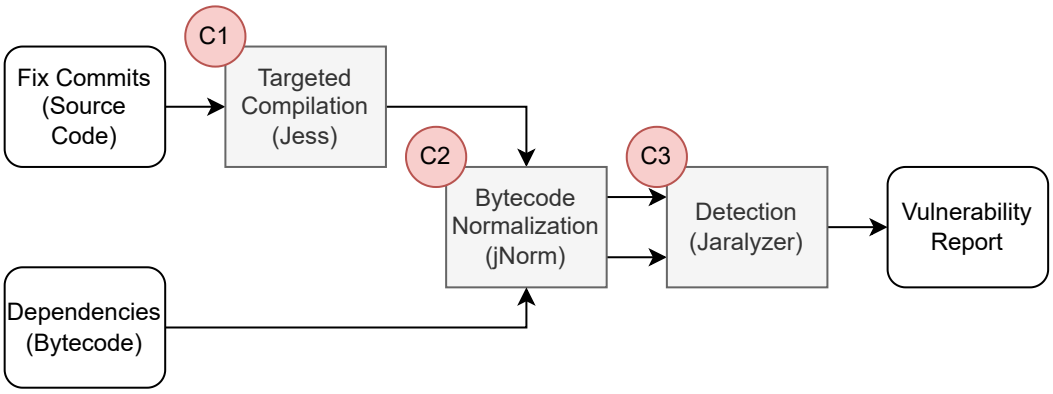


Figure 1.1: High-level overview of thesis contributions

Contribution 1: Targeted Compilation of Commits in Source Code Repositories

In our first contribution (see C1 in Figure 1.1) we present our approach, JESS, that allows for a *targeted compilation* of isolated code parts (e.g. single methods) changed within one or multiple source code repository commits. This step converts fix commits from source code to bytecode, the format in which dependencies are included, to obtain a common representation (*Requirement 1*).

Since fix commits form the basis of bytecode-centric dependency scanning and correspond to its first step, as depicted in Figure 1.1, it is essential to compile as many of them as possible. Reliable detection of vulnerable dependency inclusions is only possible when the corresponding fix commits can be successfully compiled into bytecode. Our approach specifically combats the challenges that are presented when trying to compile (old) source code repository snapshots and aims at improving compilation success rates. Using our technique, one can avoid the need to invoke the provided build scripts for compilation, which frequently results in failure [Tuf+17; Has+17; SP16].

JESS uses slicing to remove parts of the code base not required for compiling the areas of code modified within the provided commits. It then uses type inference techniques for generating empty class and method stubs to satisfy missing compilation dependencies. We conduct a large-scale evaluation of JESS on 32,970 methods from 347 popular Java projects, demonstrating that JESS can successfully compile up to 72% of these methods in isolation. Furthermore, we show that the bytecode obtained via JESS on average only differs by 0.66%, in 89% of cases even equals the original bytecode.

We further increase the success rate of JESS by integrating it in a custom compilation heuristic specifically designed for compiling commit changes. The heuristic extracts identifiers from the project's build configuration scripts to determine and retrieve necessary compilation dependencies. By applying JESS integrated within the compilation heuristic, we achieve a 90% compilation success rate across the 728 Java entries in the Project KB [Pon+19] fix-commit database—a comparatively high success rate for compiling old Java repository snapshots.

Contribution 2: Bytecode Normalization for Subsequent Code Similarity Analysis

In our second contribution we investigate compilation induced differences to design a bytecode normalization approach that is able to eliminate these differences (*Requirement 2*).

First, we conduct a large-scale study on compiler usages in real-world Java projects. Based on these findings we uncover and present 16 difference classes that different compilation environments may induce when compiling the exact same source code.

We then present our approach, JNORM, designed to remove the uncovered compilation differences and effectively normalize the bytecode (see C2 in Figure 1.1). We conduct a large-scale evaluation of JNORM on more than 300 popular Java projects, demonstrating that solely the act of incrementing the compiler version may cause differences in 46% of all resulting bytecode files. Using JNORM, one can remove more than 99% of these differences, effectively normalize the resulting bytecode and enable a subsequent similarity analysis.

Contribution 3: Bytecode-centric Detection of Known Vulnerabilities in OSS Dependencies

Finally, to overcome the challenges posed by modified dependencies to state-of-the-art dependency scanners, we propose JARALYZER, our approach to *bytecode-centric* Java dependency scanning (see C3 in Figure 1.1). JARALYZER does not rely on the availability of metadata or source code, but directly compares a dependency's bytecode against a fix-commit database. To be able to do so, JARALYZER employs the techniques from our first two contributions to compile fix commits in isolation and normalize the obtained bytecode. This enables a comparison independent of the used compilation environment. Subsequently, it employs a code property graph (CPG)-based comparison technique that considers syntax, control and data flow to determine whether known-to-be-vulnerable dependencies are present in the analyzed software project and whether the corresponding fixes have been applied. This CPG-based approach enables JARALYZER to not solely rely on the constructs' FQNs (*Requirement 3*) and offers the needed flexibility to detect the presence of fixes even when the original fix has evolved over time (*Requirement 4*).

Our evaluation of JARALYZER across 56 popular Java OSS dependencies shows that, when it comes to detecting known-to-be-vulnerable dependencies included in modified form, it outperforms five other dependency scanners, four open-source and one commercial. With a miss rate of at most 6% for vulnerabilities in repackaged dependencies compared to unmodified ones, it is the only scanner capable of handling all types of modifications identified by Dann et al [Dan+21]. But even when applied to unmodified dependencies, when directly compared to the state-of-the-art code-centric dependency scanner Eclipse Steady, JARALYZER reported 28 more true vulnerabilities and 29 fewer false warnings.

1.3 Thesis Structure

The remainder of this thesis is structured as follows: In Chapter 2, we introduce important terminology and concepts regarding Java dependency management and dependency modifications. Furthermore, we introduce current dependency scanners and the approaches behind them. Finally, we give insights into Java compilers, bytecode and code property graphs.

In Chapter 3, we present our approach towards targeted compilation of commit changes (*Contribution 1*). This includes a custom compilation heuristic and our approach JESS, which applies slicing and stubbing to compile individual code parts in isolation.

In Chapter 4, we present an in-depth study of compiler usage in real-world Java projects and highlight the differences induced by varying compilation environments. Furthermore, we present JNORM, our approach designed to eliminate compilation differences and effectively normalize the bytecode (*Contribution 2*).

In Chapter 5, we present JARALYZER, our bytecode-centric dependency scanner (*Contribution 3*). JARALYZER builds upon targeted compilation and bytecode normalization to be able to reliably detect known vulnerabilities in OSS dependencies.

Finally, in Chapter 6, we conclude the thesis with a summary of our contributions and an outlook on future research in the field of detecting known-to-be-vulnerable OSS dependencies.

2 Background

In this chapter we introduce important topics and terminology regarding dependency management in Java, modifications that may be applied to dependencies, current dependency scanning approaches and core concepts essential to bytecode-centric dependency scanning.

2.1 Dependency Management in Java

A Java project P can be defined as a tuple $P = (C, D)$, where C represents the application code written by the project's developers, and D denotes the set of *dependencies*, i.e., third-party components originating from external sources. JARALYZER focuses exclusively on analyzing the set of dependencies D, and does not examine the application code C.

The term *dependency* refers to a separately distributed software library or framework that is included in a software project [Dan+21]. In the case of Java, these software libraries are typically distributed as JAR archives containing the *bytecode* of the library. Bytecode is a machine-readable code representation Java source code is compiled to, allowing it to be executed by the Java Virtual Machine.

To manage the inclusion of dependencies, developers typically use build-automation tools such as Maven or Gradle. In the case of Maven, the developer specifies the `groupId`, `artifactId` and `version` of the desired library within a `pom.xml` file, which contains the build information for the software project. These three properties, jointly referred to as *GAV*, are used to uniquely identify a library.

During build time, Maven automatically retrieves the specified dependencies from configured artifact repositories, typically Maven Central. In addition, Maven also resolves possible dependency version conflicts and retrieves *transitive* dependencies, which are the dependencies of the project's *direct* dependencies.

2.2 Dependency Modifications

Dependencies in Java projects are not always distributed and included in the straightforward way described in Section 2.1. Dann et al. [Dan+21] have identified four types of modifications developers frequently apply to OSS components, which are then in turn included by other developers as dependencies. In the remainder of this thesis, 'modifications' refers to these four types.

Type 1 (patched)

This type of modification frequently occurs when developers fork the source code of an OSS and modify or patch it. Dependencies modified this way do not include the original bytecode but the bytecode obtained when *re-compiling* the modified source code. This also entails that classes' digests and timestamps change. Furthermore, one typically modifies the GAV by appending a suffix like `fix` or `patch` to it.

Type 2 (Uber-JAR)

In this type of modification, developers *re-bundle* multiple OSS into a single JAR file, usually referred to as *Uber-JAR*. This type of modification is sometimes indicated by the JAR file containing `jar-with-dependencies` or `uber` in its file name. To do so, developers frequently use plugins such as the Maven Shade Plugin [Fou25i] or the Maven Assembly Plugin [Fou25d]. In contrast to type 1, the original bytecode, digests and timestamps of each individual OSS are preserved.

Type 3 (bare Uber-JAR)

Type 3 modifications are similar to type 2 in the sense that multiple OSS are re-bundled into a single JAR file. However, in contrast to type 2, the metadata contained within the JAR file (`pom.xml`, `META-INF` folder and file timestamps) is removed. This type of modification can mostly be observed in legacy JAR files, which have been created before the advent of assembly plugins.

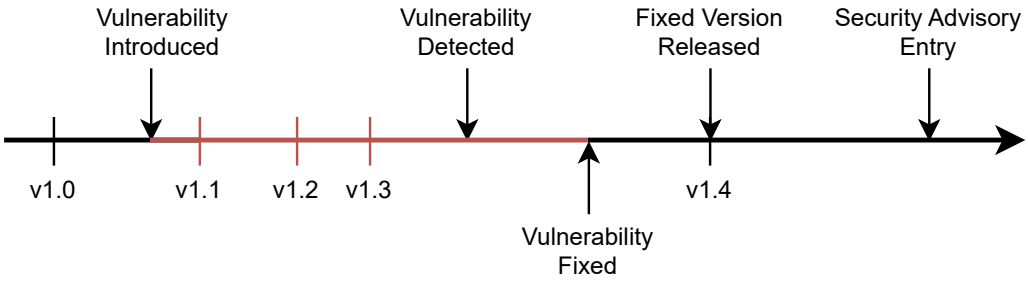


Figure 2.1: Exemplary project timeline of a vulnerability affecting an OSS library

Type 4 (re-packaged Uber-JAR)

Type 4 modifications are also similar to type 2. However, instead of just re-bundling multiple OSS into a single JAR file, the OSS are additionally *re-packaged*. This involves modifying their original package names, either by prepending a string or by replacing them entirely with new names [Fou25i]. In such cases the original bytecode of the OSS is changed substantially, because all references within the code need to be adjusted. Furthermore, classes' digests and timestamps are also changed. Re-packagings can be configured using the Maven Shade Plugin [Fou25i] and are often applied to avoid version conflicts and name clashes between multiple dependencies.

2.3 Known Vulnerabilities

In contrast to unknown vulnerabilities, which developers usually uncover in their own code through testing, reviews or static analysis tools, *known vulnerabilities* are security issues identified in specific versions of OSS libraries. These are typically listed in *security advisories*. Among the most popular security advisories are the National Vulnerability Database (NVD) [ST25], the GitHub Advisory Database [Inc25a] and the OSV database [Goo25b]. However, there are many more free and commercial security advisories. Each advisory assigns a unique identifier to every vulnerability affecting an OSS library and comes with a different set of information. Additionally, advisories specify the versions of the library that are impacted, along with the first version in which the vulnerability has been patched.

Figure 2.1 depicts an exemplary project timeline of an OSS library that was affected by a vulnerability. Such a vulnerability is often not present from the project's inception but introduced at some later point in time. Typically, such a vulnerability is

eventually detected and reported to the maintainers of the OSS project. At this stage, information about the vulnerability's existence is often not yet publicly available. Afterwards, the vulnerability is fixed in the project's source code repository by creating appropriate commits fixing the vulnerability. At this point, due to the public nature of the fix commits, the existence of the vulnerability becomes public knowledge. Sometime after the fix commits are applied, a patched version of the OSS library is released to a public artifact repository. Eventually, following an additional delay, information about the vulnerability, along with the affected version range, is collected and published in various security advisories. The median delay between public patch availability and advisory publication is 25 days [PM24]. In the Maven ecosystem, the average delay extends to 41 days.

One factor that contributes towards this long delay is the need to manually map the vulnerability to its impacted version range. As shown in the example in Figure 2.1, the vulnerability was first detected in version 1.3 of the project. However, it originated in version 1.1, thereby affecting versions 1.1 through 1.3. This is often further complicated by multiple versions of the OSS project being maintained in parallel.

2.4 Dependency Scanners

Dependency scanners are tools that serve the purpose of detecting inclusions of known-to-be-vulnerable dependencies in a scanned software project. Due to the high risk associated with the heavy usage of OSS components, many open-source and commercial dependency scanners have been created. Over the years, the term *software composition analysis* (SCA) has emerged to describe dependency scanners, particularly in commercial contexts. In general, dependency scanners can be classified into two primary categories.

Metadata-based Dependency Scanners

Metadata-based dependency scanners conduct vulnerability scans by leveraging the metadata associated with dependency inclusions. One common method involves generating a *Software-Bill-of-Materials* (SBOM) for the project under analysis. The SBOM lists all dependencies included in the software project along with their identifiers and versions. This information is then compared against a security advisory of known-to-be-vulnerable dependency versions, often represented as Common Vulnerability Enumeration (CVE) entries [Cor24b]. In the case of a Java

Maven project this means creating an SBOM from the dependency information obtained within the project's `pom.xml` files and comparing it to some advisory. Scanners accomplish this by using existing standard SBOM formats and tools, such as CycloneDX [Fou25c] or SPDX [Fou23b], or by relying on a custom internal SBOM format.

Metadata-based dependency scanners are often not limited to using only dependency identifiers. Some approaches also leverage the file structure of included dependencies, or even use digests and timestamps associated with the dependency's class files [Goo25a]. Metadata-based dependency scanners often support multiple programming languages and typically feature fast scans due to their code-agnostic nature.

However, as Ponta et al. [PPS18; PPS20] have reported, this code-agnostic nature results in multiple shortcomings. For one, metadata-based dependency scanners heavily rely on the quality of the underlying security advisories. Often Security advisories do not contain precise information about which *modules* of a specific OSS are affected by a certain CVE entry, causing metadata-based scanners to report false positives [PPS20]. For instance, if one considers the popular Spring framework for Java, advisories often assign a specific CVE entry to the whole framework, even though only one of its 20 modules is actually affected. Even projects using only unaffected modules will nonetheless receive vulnerability alerts. On the other hand, metadata-based dependency scanners will not report vulnerabilities and patches until the advisories they rely on have been updated. As noted in Section 2.3, the median delay for this update is 25 days, in the Maven ecosystem even 41 days [PM24]. This timeframe does not even account for the period between a fix becoming available in the project's source code repository and its eventual release to a public artifact repository. This creates a timeframe during which it is publicly known that certain versions of an OSS library are affected by a vulnerability, yet metadata-based scanners do not report it, making this period particularly risky.

Popular open-source or free-to-use metadata-based dependency scanners include OWASP DependencyCheck [OWA24], OSV Scanner [Inc24c] and GitHub Dependabot [Inc24b]. Prominent examples for commercial dependency scanners include Endor Labs SCA [Lab24], Snyk Open Source SCA [Lim24], Mend SCA [Men24] and Black Duck SCA [Inc24a].

Code-centric Dependency Scanners

The *code-centric* approach to dependency scanning has been originally proposed by Plate et al. [PPS15] and Ponta et al. [PPS18; PPS20] to directly address the shortcoming of metadata-based dependency scanners. The code-centric approach does not rely on metadata and information about known-to-be-vulnerable dependencies in advisories but directly compares the *code* of included dependencies to a database of *fix commits*. These fix commits represent the real code changes performed within the OSS respective Git repositories that fix a specific vulnerability. Directly relying on the fix commits, one does not have to manually map the vulnerability to a list of affected OSS versions first. Furthermore, having access to the exact code changes comprising the fix, one exactly knows which module of the OSS is affected by a vulnerability. Moreover, it allows for a subsequent reachability analysis to check whether the vulnerable code segment is even executed within the context of the including software project. This enables the approach to reduce false positives in the detection of vulnerabilities when compared to metadata-based approaches.

Ponta et al. have implemented the code-centric approach in the dependency scanner Eclipse Steady [PPS18; PPS20]. A prerequisite of the code-centric approach is that both fix commits and scanned dependencies are available in the same code representation to be comparable, an assumption that generally holds for languages like Python or JavaScript [Chi+20]. In contrast, Java poses a challenge, as dependencies are usually distributed in compiled *bytecode*, whereas fix commits modify the original *source code*. To circumvent the issue of mismatched code representations, Eclipse Steady attempts to retrieve the source code of dependencies from Maven Central, the most popular Java artifact repository. However, the source code may or may not be available alongside the compiled artifacts and, even when present, there is no guarantee that it actually matches the compiled artifact. If Eclipse Steady cannot retrieve the source code, it is unable to compare bytecode to source code [SE21], and thus requires costly and error-prone manual analysis to determine if a dependency is vulnerable.

2.5 Java Compilers and Bytecode

In contrast to other compiled programming languages like C, Rust or Go, Java applications are not directly compiled into machine-executable code, but into *bytecode*. This bytecode is executed by the Java virtual machine (JVM), which comes with a

just-in-time (JIT) compiler that compiles the bytecode into executable machine code at runtime. Because of this, the Java bytecode has the following characteristics:

1. **Platform independence**: The JVM architecture aims at platform independence. The generated bytecode is independent from the platform it is intended to run on [Cor25e].
2. **Unoptimized bytecode**: Optimizations are performed during runtime by the JIT compiler, therefore compiled bytecode is typically not optimized [Cor25c].

Because of these characteristics, since there are no different optimization levels or differences due to the targeted platform, the amount of variance across generated bytecode is not as high when compared to machine-code.

There exist multiple different Java compilers like e.g. Oracle's JDK or OpenJDK's compiler [Cor25d], Eclipse's JDT compiler [Fou25a], IBM's Jikes compiler [IBM14], or the GNU Compiler for Java (GCJ) [Pro17]. While IBM Jikes and GCJ are deprecated nowadays, Oracle JDK's, OpenJDK's and Eclipse JDT's compilers are still being actively maintained and updated with each new Java version being released. The JDK's and OpenJDK's compilers can be invoked programmatically or through the command-line application *javac* that comes pre-shipped with each JDK. Given the same source code as input, in many cases these compilers produce *different* Java bytecode.

In general, Java compilers support source codes that adhere to different versions of the language specification and can generate bytecode for different JVM versions lower than the compiler's version. This backwards compatibility can be used by setting the compiler's *target level*. For example, bytecode compiled with a JDK11 compiler with target level set to 8 can be executed by a JVM only supporting up to Java 8. The set of available target levels for a compiler is usually limited to a subset of earlier versions.

2.6 SootUp and Jimple

Jimple is an intermediate representation (IR) of JVM bytecode that was designed for providing a format that allows for simplified analysis, optimization and code transformations. Jimple maps the more than 200 Java bytecode instructions to only 15 different Jimple instructions in a *three-address* based representation. Three-address based representation means, that each instruction generally contains at most three different operands, e.g., one used for the left-hand side of an assignment

```
1 public int add(int a, int b) {
2   int c = Addition.add(a, b) + 1;
3   return c;
4 }
```

Listing 2.1: Java method example

```
1 public int add(int, int) {
2   int i0, i1, $i2, i3;
3   i0 := @parameter0: int;
4   i1 := @parameter1: int;
5
6   $i2 = staticinvoke <Addition: int
        add(int,int)>(i0, i1);
7   i3 = $i2 + 1;
8   return i3;
9 }
```

Listing 2.2: Jimple method example

and two used for binary operations on the right-hand side. This restriction greatly simplifies the processing of individual IR statements, which is why three-address IRs are commonplace nowadays. During the transformation Jimple retains all the type information present in the bytecode.

Jimple is the IR of the popular Java bytecode analysis framework SootUp [Kar+24]. Alongside various code optimization options, SootUp provides an API to conveniently transform Jimple instructions. SootUp can automatically convert JVM bytecode to Jimple via its bytecode frontend.

Listing 2.1 depicts a Java function that adds two provided integers and increments the result by one. To obtain its corresponding Jimple representation, one needs to compile the source code first and then provide the corresponding bytecode to SootUp. Listing 2.2 shows the corresponding Jimple representation of the function. One can immediately see that the identifiers of all variables and parameters are missing and replaced by a generic letter and number combination. This is due to the compilation step, in which the compiler removes most identifier information in its default configuration. A $-sign in front of the variable name means that it is a temporary stack variable that has been generated by SootUp and is not part of the original code. The instruction in Line 2 of Listing 2.1 is split up into two instructions in Jimple, due to the three-address based representation, which splits the “complex” expression, containing a method call and an addition, into two simple ones. The call of the static method `add` of the `Addition` class is made more explicit in the Jimple function: in Line 6 of Listing 2.2 the full signature of the method (return type and both parameter types being `int`) can be seen alongside the type of invocation. Since the `add` method is static, a `staticinvoke` instruction is used to invoke the method.

2.7 Code Property Graphs

A *code property graph* (CPG) is a representation, first proposed by Yamaguchi et al. [Yam+14], that combines syntax, control flow, data dependence and control dependence relations of a given method into a single graph representation. To achieve this, it combines the following three graph representations into a single one:

- **Abstract Syntax Tree (AST)**: The AST models the syntactic structure of a given method. It shows how individual statements and expressions are nested within the method, while abstracting away inessential information such as grouping parentheses or braces. The leaf nodes of an AST represent operands, such as constants, identifiers or literals, while the inner nodes represent operators, such as arithmetic operations or assignments.

- **Control Flow Graph (CFG)**: The CFG describes the execution order of statements within a given method. Furthermore, it also models the conditions that need to be met for a certain execution path to be taken. This means that e.g. in case of an if-else statement, the CFG describes which execution path will be taken if the condition evaluates to true or false.

- **Program Dependence Graph (PDG)**: The PDG models data and control dependencies between statements and predicates of a given method. The data dependencies explicitly model the influence the value of one variable has on the value of another variable, while the control dependencies model the influence the value of a predicate has on the value of a variable. This representation has originally been created for applications in program slicing and particularly represents directly connected parts of a given method.

```
1 public void qux() {
2   int x = foo();
3   if (x < 42) {
4     int y = 2 * x;
5     bar(y);
6   }
7 }
```

Listing 2.3: Example of a Java method for code property graph creation

Listing 2.3 shows a Java method `qux` that invokes a method `foo` and assigns its return value to a variable `x`. If the value of `x` is less than 42, the method doubles

the value of `x`, assigns it to a variable `y` and provides it as argument to the invoked method `bar`.

Figure 2.2 depicts the CPG corresponding to method `qux`, shown in Listing 2.3. This graph combines the individual AST, CFG and PDG representations of the method into a unified multigraph. A multigraph is a type of graph that allows for multiple edges between the same pair of nodes.

From the CPG, one can derive not only the syntactic structure of the method, but also the execution order of individual statements, including alternative execution paths depending on whether the if-expression (see Listing 2.3, line 3) evaluates to `true` or `false`. Additionally, the graph represents data dependencies, such as the dependence of variable `y` on the value of variable `x`, as well as control dependencies, indicating that `y` is only declared and method `bar` is only invoked if the if-expression evaluates to `true`.

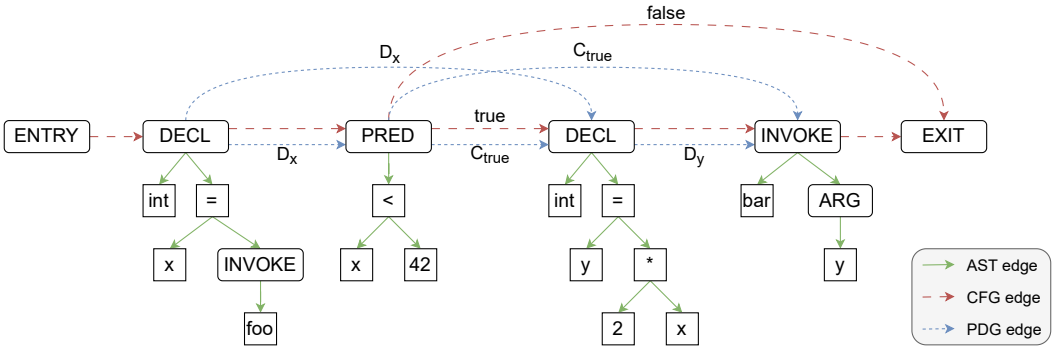


Figure 2.2: Code property graph for the method shown in Listing 2.3 [Yam+14]

3 Targeted Compilation

Compiling Java programs is not a trivial task. This has been shown in multiple independent studies [Has+17; SP16; Tuf+17]. Especially if one needs to compile a specific source code repository snapshot from the change history of a project, this often leads to failure. Compiling repository snapshots corresponding to fix commits however, is crucial for enabling bytecode-centric dependency scanning. Even though common build tools try to provide a reproducible build of applications, there is no guarantee that the compilation will succeed. As Tufano et al. [Tuf+17] have shown, across 100 investigated Java projects from the Apache Software Foundation, only 38% of the change history was compilable. This not only hinders the possibility to analyze the runtime behavior of the application, but also prevents one from obtaining the compiled bytecode that corresponds to the source code.

The causes for build failures are manifold and include parsing errors or version compatibility issues. However, the most common reason for build failures is the incomplete resolution of artifacts when a third-party dependency, which was available when the commit was originally created, has been removed from the repository it has been hosted in [Tuf+17]. Even a single unavailable external artifact breaks the whole compilation process. However, when one is only interested in the bytecode of a specific part of the program, e.g. security-relevant methods, even if all information needed for compiling the program's part of interest is present in the project's source code repository, the unavailability of some external artifacts prevents the build from succeeding.

To address these compilation challenges, we propose JESS, an approach for compiling *targeted areas* of Java source code without requiring the resolution of external artifacts or the execution of build scripts. While other approaches have been developed to compile partial Java programs, they either transform the source code into a representation other than bytecode [DH08], or they modify the original code and focus solely on generating *some* bytecode [SAH21; GMP20], without aiming to produce bytecode that is identical, or at least similar, to what would be generated by the original build.

JESS receives source code of a Java project as input and slices away all information from the code base that is not required to compile the specific areas of interest, but

would only interfere with the compilation. It then infers the signatures of types that are required for the compilation but are not available within the code base, and generates type stubs corresponding to the inferred types to allow for a targeted compilation of only classes and methods of interest.

Furthermore, for the use case of compiling the changes introduced in commits within a source code repository, we have devised a dedicated *compilation heuristic* that leverages JESS to further improve compilation success rates.

We have evaluated JESS on 354 of the most popular Java projects on GitHub. Our evaluation showed that of the 32,970 methods and constructors we had randomly sampled from the projects, JESS was able to compile more than 72%. Furthermore, our evaluation showed that the bytecode obtained via JESS is highly similar to the original bytecode and, on average, only differs by 0.66%, in 89% of cases even equals the original bytecode. This is critical for bytecode-centric dependency scanning, as we must compare the bytecode produced by JESS with the original bytecode of dependency artifacts obtained through full compilation. Moreover, our evaluation on the Project KB fix-commit database [Pon+19] showed that the provided build scripts only allowed for a compilation of 11% of the provided fix commits, while JESS enabled the full compilation of the modified code belonging to 50% of Java fix commits and at least a partial compilation of 75% of fix commits. A targeted compilation via JESS made it possible to obtain the bytecode of 71% of files modified in the fix commits, in contrast to only 8% of files that can be obtained when relying on provided build scripts for compilation. Using the proposed compilation heuristic specifically dedicated to compiling commit changes we were even able to obtain the bytecode of 91.5% of all fix commits in Project KB.

The remainder of this chapter is structured as follows: First, in Section 3.1, we explain the role of compilation in bytecode-centric dependency scanning. In Section 3.2 we then describe the concept and details behind targeted compilation via JESS. We describe the compilation heuristic for compiling commit changes in Section 3.3. Then, in Section 3.4 we evaluate targeted compilation using JESS on a large set of Java GitHub projects and the Project KB fix-commit database. We give an overview over the threats to validity to our work on targeted compilation in Section 3.5. Afterwards we present related work in Section 3.6 before we conclude in Section 3.7.

3.1 The Role of Compilation in Bytecode-centric Dependency Scanning

A dependency that is included in a Java project is usually obtained as a pre-compiled JAR-archive from a software repository like Maven Central. Thus, to compare a dependency's bytecode to its corresponding source code fix, available in the form of fix commits in its respective source code repository, one needs to transform both representations into a common or comparable representation (see Section 1.1). To achieve this, three possible approaches can be followed, each with its own set of advantages and disadvantages:

- **Decompiling the dependency's bytecode:** One possible way of unifying both representations is to *decompile* the dependency's bytecode. During the decompilation step the bytecode is transformed back into Java source code. While there exists a plethora of tools capable of performing such a decompilation, none of the tools is able to transform the bytecode back into its *original* source code [HD09; Har+19]. This is due to the information loss that occurs during the initial compilation. In such cases the decompilers have to estimate the original source code constructs. This frequently results in the decompiled source code being vastly different from the original source code making a subsequent comparison imprecise.

- **Compiling the Java source code modified within the fix commits:** The bytecode of a dependency has been originally obtained by compiling its respective source code. Thus, compiling the fix commits to bytecode allows for the most precise subsequent comparison. However, this comes with multiple challenges. Compiling, possibly old, repository snapshots is a notoriously error-prone task [Tuf+17; Has+17; SP16]. Consequently, in many cases, the bytecode cannot be obtained through a normal compilation at all. Furthermore, even if one is able to successfully compile the fix commits, one does not know in which compilation environment (compiler, version and settings) the *original* dependency artifact has been compiled. This is problematic because different compilation environments can produce different bytecode from the same source code [DHB19], making bytecode comparison challenging.

- **Extracting common information from bytecode and Java source code:** Some comparison approaches rely on extracting common information from source and bytecode. These approaches extract feature sets from the source and bytecode that are stable across the compilation and compare them to

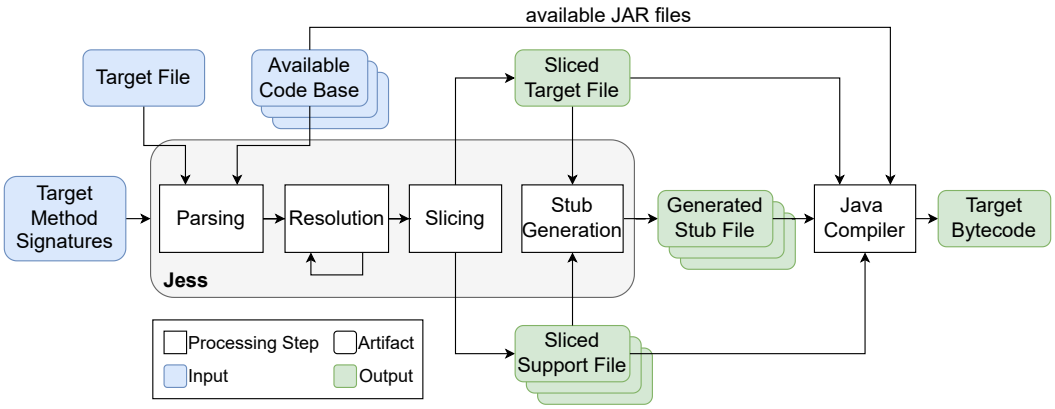


Figure 3.1: Overview of JESS

each other [Dai+20; Pan+24]. Such approaches do not rely on compilation or decompilation and can typically be applied in most cases. However, these approaches come at the cost of potentially missing vulnerabilities due to their often coarse-grained feature representation and their lack of missing control-flow, control-dependence and data-dependence information.

Vulnerability fixes are often fine-grained in nature, modifying only few code statements, sometimes even a single instruction. To reliably and precisely identify such vulnerabilities in a dependency's bytecode, one must be able to compile the source code modified in the corresponding fix commits, as alternative approaches introduce limitations that hinder accurate comparison. Unfortunately, compiling the specific snapshot version corresponding to the fix commit is often not possible—a problem that becomes more prevalent the older the snapshot becomes [Tuf+17].

However, for bytecode-centric dependency scanning, it is sufficient to obtain the bytecode of only those program parts that were actually modified by the fix commit. We can thus use JESS to compile those parts in isolation and increase our compilation success rates.

3.2 Targeted Compilation with JESS

Figure 3.1 shows our approach behind targeted compilation via JESS. The goal is to obtain bytecode identical or at least highly similar to the one that would be generated by invoking the original build scripts with every single dependency fully resolved. JESS takes as input the signatures of the methods to be compiled (referred to as target in the following), the Java source code files containing them, all other

Java source code files available in the hosting code base and all dependencies that are still retrievable in form of JAR files. For the sake of brevity we describe the approach using a single method as target, however JESS supports handling of single or multiple members (e.g., methods, fields, constructors, and initializers) of classes and inner classes.

JESS follows a mark-and-sweep approach [McC60] for its processing. Initially, JESS parses all available files and marks the compilation target with the annotation *Keep-All* to denote that it must not be altered (e.g., the body of methods must not be sliced or modified).

In the resolution step, JESS identifies the parts of the code base that are relevant for compiling the target. It inspects all references in the target, e.g., field accesses, method invocations, or object creations, and tries to find the corresponding declaration. Such references may be part of the target file itself, a different file within the code base, or external files. Referenced members are marked with the annotation *Keep* to denote that only the signature of the member should be kept. All newly marked members are recursively resolved until no further references are found or the found references are not part of the available code base. While references in marked signatures are always resolved, those in the body of methods and constructors only need to be resolved when annotated as *Keep-All*. For marked fields, the type definition is always resolved, whereas the field initializer is resolved only in case of *Keep-All*.

In the slicing step, JESS removes all class members that are not needed for the compilation of the target: class members that have not been marked, bodies of methods and constructors marked as *Keep* only, and the initializers of fields marked as *Keep* only. Additionally, JESS removes classes and inner classes without any marked member, now unused import declarations, and JavaDoc as it is not reflected within the bytecode. Exceptions that need to be considered while slicing are described in Section 3.2.2. Slicing the body of methods invoked from the target method, while keeping its signature, allows to minimize the amount of transitively referenced members, while obtaining compilable code. After slicing, JESS outputs the sliced target file, which contains the target method and the class members it references, and a set of sliced support files, which contain class members that are either referenced directly or transitively from the target method.

Due to slicing away code unrelated to the target methods compilation, it may enable a successful compilation of the target method, even when the snapshot contains broken source code (e.g. due to syntax errors), as long as the the broken code is not

```java
import org.example.X;

public class A {
  @Keep // 2
  int num = D.getNumber();

  int otherNum = 12;

  @KeepAll // 1
  public int visit() {
    B b = new B();
    b.inc(num);

    X x = new X();
    String s = "five";
    int val = x.getVal(s);
    boolean con = X.conv;

    return val;
  }
}
```

Listing 3.1: Compilation target file

```java
public class B {
  @Keep // 2
  public C inc(int a){...}
  public E dec(int a){...}

  @Keep // 3
  class C {...}
}
```

Listing 3.2: Compilation support file

```java
// Generated Stub
package org.example;

public class X {
  static boolean conv = true;

  public int getVal
    (String arg) {return 0;}
}
```

Listing 3.3: Generated stub file

contained within the target method itself. After slicing, the compilation may still fail due to references to classes outside of the available code base.

Finally, the stub generation step infers the signature of unresolved references based on the symbol's usage context and synthesizes a *type stub*, i.e., definitions of classes and class members with full signatures, yet no content. Similar to interfaces, which describe the structure of a class without describing the specific implementations, JESS generates classes that contain the full member signatures without any actual functional implementation. The stub generation is described in detail in Section 3.2.3.

Finally, a Java compiler can then compile the target method with the sliced target file, sliced support files, generated stub files and available JAR files.

3.2.1 JESS Compilation Example

Listings 3.1, 3.2 and 3.3 illustrate an example of how JESS performs the processing for compiling commit changes. Listing 3.1 shows the target file with the `visit` method being the target method (e.g. modified within a fix commit). Only the classes `A` and `B` (cf. Listings 3.1, 3.2) are assumed to be within the available code

base. After parsing, in the first iteration of the resolution step, JESS marks the target method as *Keep-All* (e.g. method changed in the commit) to state that the method body should be kept intact. The numbers next to the markings within the listings indicate the resolution iteration in which the corresponding member has been marked. In the next iteration, JESS resolves all references within the `visit` method. In line 11 (Listing 3.1), a new object of class `B` is created and in line 12, the `inc(int a)` method of class `B` is invoked. Due to this invocation, JESS marks the method declaration (line 3 in Listing 3.2) as *Keep*, which indicates that the method signature needs to be kept intact, while the body can be sliced away. This method invocation receives the field `num` as argument and therefore JESS marks the field declaration in line 5 of Listing 3.1 also with a Keep-Marking. The remaining references within the target method's body or signature all reference a class `X`, which is assumed to not be part of the code base. Now JESS looks at all members that have been marked in the previous step. As all added markings are Keep-Markings, only the member signatures are of interest, since their content will be sliced away in the next step. Thus, the only additional reference is in line 3 of Listing 3.2 where the `inc` method specifies a return type of `C`. Because of this, a Keep-Marking is added to the inner class C (cf. line 6 of Listing 3.2). After all relevant references for the compilation of the target method have been considered, JESS removes all non-marked members within the classes and the contents of the members marked with a Keep-Marking. In class `A` JESS will remove the declaration of the field `otherNum` and the initialization of the field `num`. In class `B` JESS will remove the declarations of the method `dec`, as well as the body of method `inc` (and add a corresponding dummy return statement, which will be explained in detail in Section 3.2.2). Finally, in inner class `C`, JESS will remove all class members. In doing so, JESS eliminates references to the classes `D` and `E`, which might be originating from an external library that might not be available any longer and would cause a normal compilation of the full snapshot to fail.

After slicing, not all references to classes outside of the available code base could be removed. As the target method contains multiple references to members of the unavailable class `X`, JESS now generates a stub file, which can be seen in Listing 3.3. When JESS processes the creation of the non resolvable object (line 14 in Listing 3.1), it tries to infer all necessary information to generate an appropriate stub file from its usage context. In this case, since no argument is given in the object creation, JESS does not need to generate any constructor. To generate the correct bytecode, however, the simple name of the referenced class is insufficient—the fully qualified name (FQN) is required instead. To determine the FQN, JESS considers the import declaration within the target file (line 1 in Listing 3.1) and sets the package

definition of the generated stub file to the determined package path (`org.example` in this example). The next unresolvable reference can be seen in line 16 of Listing 3.1, where the method `getVal` is invoked. Whenever JESS processes a method invocation, it needs to determine four properties:

1. the object or class that the method is being invoked on,
2. the return type of the method,
3. the parameter types of the method,
4. whether the method is static or non-static.

In the usage context within Listing 3.1, it is possible to determine all four properties. The method is invoked on an object with the name `x`, which can be traced back to the object instantiation in line 14. The return value of the method is assigned to a variable of type `int`. As its only argument it receives the variable `s`, which can be traced back to its definition site in line 15, where it is defined as type `String`. Since the method is invoked on the previously instantiated object `x`, it is likely that the method is non-static. Due to this information, JESS generates the corresponding method stub in lines 7–8 in Listing 3.3. Due to syntactic ambiguities (which will be described in detail in Section 3.2.3) it is not possible to infer with certainty the definite return and parameter types and whether the method is static or non-static. However, the generated stub will still enable a successful compilation. Finally, the reference within the target method can be observed in line 17 in Listing 3.1. Here a field of class `X` is accessed and assigned to the `boolean` variable `con`. Due to the usage context, JESS is able to generate the field stub in line 5 of Listing 3.3, by applying the same inference procedure as for the previous method invocation. When the sliced target file, sliced support files, and the generated stub file are handed to a Java compiler, the target method can be successfully compiled.

3.2.2 Slicing Exceptions

In the following we describe some exceptions to be considered during the slicing step.

Abstract interface / superclass methods: Whenever a class implements an interface or extends an abstract class, every abstract method definition needs to be implemented in concrete subtypes, otherwise the compilation will fail. Because of this, JESS can only slice away methods within classes that are implementing an abstract method if it also slices away the corresponding abstract method definition

from within the interface or superclass. However, if the interface or superclass is originating from an already-compiled class file or the JDK, e.g. the `Collection` interface, JESS cannot slice away any such method definitions. In such cases, JESS only slices away the body of such methods.

Abstract functional interface methods: A functional interface in Java is an interface that defines just one abstract method [Ora21c]. Whenever this is the case, a lambda or method reference expression can be used to create instances of the corresponding interface. In such cases, although the abstract method is typically not referenced explicitly, its definition still needs to be kept such that its defining interface is considered as functional by the Java compiler. Thus, JESS does not slice away the abstract method from within referenced functional interfaces, even when the method is not referenced explicitly. If the interface is not referenced at all, JESS will slice away the whole interface.

There are also exceptions that apply to slicing the bodies and initializers of methods, constructors and fields:

Method return types: Whenever a method signature specifies a return type that is not `void`, the corresponding method body needs to contain a return statement, which returns a value that adheres to the specified type. When this is the case, JESS removes the whole method body and replaces it with a single return statement that returns a dummy-value (e.g. `return 0` or `return "dummy"`). If the specified return type is a reference type (except for String types), JESS inserts a `return null` statement.

Super constructor invocation: Whenever a class extends another class, the Java compiler requires the extending class to (transitively) call a constructor of the super class (via the `super` statement) [Ora21e]. If the super class contains a default constructor (which does not require parameters) this call is automatically generated by the Java compiler. However, when the super class only provides constructors that expect arguments, a super constructor needs to be called explicitly from within a constructor belonging to the extending class. Thus, JESS keeps such super constructor invocations intact and only replaces their arguments with dummy-values.

Final fields: Whenever a field is declared as `final`, it needs to be assigned a value during instance initialization. Thus slicing away the initialization of a final field would result in the Java compiler not being able to compile the respective class. Furthermore, final fields, which are initialized with a literal or an arithmetic expression (e.g. `final int x = y + 1`), are treated as constants by the Java compiler, with the constant being propagated directly to the method referencing it. Therefore, it is not

sufficient to replace the initialization with a dummy value, as this will modify the resulting bytecode. Due to this, if the field is initialized with a literal or arithmetic expression, JESS keeps the original initialization intact.

Type-import-on-demand: Whenever there is a type-import-on-demand declaration (import with asterisk), JESS keeps the import declaration as it might contain important information for the stub generation step. After stub generation, to avoid a compilation fail due to the package path not existing, JESS generates the directory structures described by these import declarations, even when none of these imports is required for compilation.

3.2.3 Stub Generation

In the following we describe the generation of type stubs, characterized by a full signature and no content.

For generating fitting type stubs, JESS applies a greedy algorithm. First, it scans the code parts that remain after slicing for references that could not be resolved within the available code base. For each unresolvable reference, based on the usage context, JESS then applies a set of inference techniques to determine corresponding types. The determined types are required to create type stubs, which will not only satisfy the Java compiler, but result in bytecode that is as close as possible to the original one. Finally, once all unresolvable references have been considered and the appropriate type information has been inferred, JESS generates the respective stub files containing the type stubs. JESS applies the following techniques to infer appropriate stubs for references which are not resolvable within the available code base:

Literal inference: The easiest case for JESS to infer the needed types for, e.g., a method signature, is if the method has been invoked with a literal. The Java language specification clearly describes the types of literals [Ora21b]. For a method invocation, e.g., `foo("abc")`, which is invoked with the literal `"abc"`, the Java specification defines the type of the literal to be `String`.

Definition inference: During definition inference, JESS tries to locate the definition-site of a variable and infers its type from there. This definition-site can either be a variable, field or parameter definition. Here it is important to consider the scopes of variables. At first JESS checks the block in which the variable has been used and then incrementally checks outer blocks, where the scope still applies to the variable usage, until it reaches the method definition. At this point JESS checks the

method parameters. If the definition-site still has not been found, JESS incrementally increases its scanning towards field definitions within the same class, potential outer classes and extended superclasses.

Conditional inference: Whenever an expression is used in a statement that requires a condition, e.g. if-, while- or do-statement, it has to resolve to a boolean type [Ora21a]. This means that if e.g. a method's return type is used as the condition of an if-statement, its return type has to be `boolean` or `Boolean`.

Return statement inference: When an expression is used in a return statement, the type of the expression has to be the same (or more specific) as the return type defined in the method signature.

Static / Non-static inference: To check if a method or field declaration is static, JESS checks if it finds a definition-site for the object/class the method or field is accessed on. If it finds a definition-site, the field/method is declared as non-static, otherwise as static.

FQN inference: To infer the fully qualified name (FQN) of a class, JESS checks all direct import statements within a file to find one where the last part of the import matches the used class. Sometimes classes are imported via asterisk imports. If none of the direct import statements matched and there is only a single asterisk import statement in the file, JESS infers the FQN from the asterisk import.

Superclass / Interface inference: There are some scenarios in which the compilation will fail if a certain class is not extended or a certain interface is not implemented.

- **Exceptions:** When objects are used in throw-statements or caught in try-catch blocks, the Java compiler requires the corresponding classes to extend the respective `Throwable` class. If JESS observes such usages, it adds the corresponding class extension to the generated stubs.

- **Iterable:** If an object is used in a for-each loop, the Java compiler requires the corresponding class to implement the `Iterable` interface. JESS detects such occurrences and adds the appropriate `Iterable` interface implementation.

- **AutoCloseable:** When an object is used as resource within a try-with-resources statement, the Java compiler requires the corresponding class to implement the `AutoCloseable` interface. JESS adds such an interface implementation to the generated stub class if needed.

- **Typecasts:** Whenever an explicit typecast is performed on an object, its corresponding class must inherit or implement the type defined within the typecast. JESS either adds the corresponding interface implementation or the respective extend class statement to the generated class.

Generics inference: Sometimes, when objects are created, generic type arguments are supplied to the created object. However, in order to be able to pass generic type arguments, the respective class needs to specify corresponding generic type parameters. When JESS observes such occurrences, it generates the stub class in a way, that allows for supplying generic type arguments to it.

Type Inference Ambiguities

A precise inference of the corresponding signature is not always possible due to ambiguities that arise when certain information is missing. Dagenais and Hendren [DH08] already described some of the ambiguities that arise in the syntax of partial Java programs. They already describe that precisely inferring the FQN is not possible, when there is no direct import of a class, but multiple asterisk imports are present within the target file. Furthermore, they describe that e.g. it is not always possible to distinguish package names from class names (in case of inner classes). However, the list presented is not exhaustive as we found other ambiguous syntax constructs:

Method invocations without return value usage: When a method is invoked, but its return value is not used, it is unclear what return type has been originally specified in the method declaration. While it is likely that the method does not return any value (i.e. `void` return type), it is still possible that a return type is specified, but the return value is simply dropped after its invocation. Listing 3.4 shows an example of such an occurrence. The return value of the `find` method, which is invoked in line 4, is not used. However, the possible method stubs in line 10 and line 12 are both valid, even though one returns a value of type `String`, while the other does not. In each scenario the Java compiler generates different bytecode.

Method invocations with literal arguments: When a method is invoked with a literal as argument it is often not possible to unambiguously infer the specified parameter type within the method declaration. If a method is invoked with the integer literal `5` as argument, the original method declaration might e.g. specify `int`, `long`, `float` or `double` as parameter type. If a method is invoked with a `null` literal, it can only be inferred that the specified type is not primitive. One example of this

can be again observed in Listing 3.4, as the invoked `find` method can either specify `int` or `long` as parameter type and still be valid.

Static vs. non-static: As static methods or fields can either be accessed on a class itself or object instances of a class, it is not possible to precisely distinguish static class members. While it is likely that members accessed on an object instance are non-static, there is no guarantee. An example of this can be seen in Listing 3.4. It is unclear whether the in line 4 invoked `find` method is declared as static or non-static, since both possibilities are valid.

Generic class methods: In some scenarios, when a method belonging to a generic class is invoked, it is unclear whether the method specifies its parameters explicitly or as a generic type. Listing 3.5 illustrates such an example. In line 3 an object of type `W` is created with `String` supplied as the generic type argument. Then in line 4 the method `add` is invoked on the generic object, with a string literal as argument. Now, if we look at the stub, it is possible that the method `add` specifies a generic type parameter (see line 10) or explicitly specifies `String` as its parameter type (see lines 12 & 13). Each scenario results in different bytecode being generated.

Interface vs. Class: Without having additional context, it is not always possible to distinguish whether a specified type is originating from a class or an interface. When a method is called on an object of such a type, in the bytecode it is either invoked via an `invokevirtual` instruction, in case of originating from a class, or via an `invokeinterface` instruction, in case of originating from an interface.

No explicit return type usage: It may be the case that e.g. a method outside of the available code base is invoked and its return value is assigned to a variable that is declared using the `var` keyword, which does not explicitly assign a type to the variable. In this case it is not possible to precisely determine the return type of the respective method. The same applies to nested method invocations where two methods are outside of the available code base and the return value of one method is directly used as argument for the other method, without being assigned to a variable beforehand (e.g. `foo(bar())`). In this case it is not possible to precisely infer the return type of one method and the specific parameter type of the other method.

Due to the above mentioned syntactic ambiguities there are limits towards how similar the bytecode obtained via JEss can be to the original bytecode. This limit highly depends on how much of the code base is actually available and how much has to be supplemented via stub generation. JEss was designed to generate bytecode that is identical or similar to the original one. However, since this is not always possible, JEss offers the option to mark the locations where ambiguities may have

```java
public class G {
  void process() {
    V v = new V();
    v.find(5);
  }
}

// Stub
public class V {
  public void find(int i) {}
  // or
  public static String find
    (long i) {return "abc";}
}
```

Listing 3.4: Ambiguous type stub 1

```java
public class H {
  void verify() {
    W<String> w = new W<>();
    w.add("value");
  }
}

// Stub
public class W<T> {
  public void add(T t) {}
  // or
  public void add
    (String s) {}
}
```

Listing 3.5: Ambiguous type stub 2

caused imprecision. In case of an unknown FQN, JESS sets the package of the generated stub to a special `unknown` package. By doing so, the `unknown` package within the bytecode can be treated as a wildcard in a potential comparison and therefore still be compared against the original bytecode. Whenever a type (e.g. return or parameter type of a method) cannot be precisely inferred, JESS sets the type within the generated stub to a special `Unknown` type.

3.3 Compilation Heuristic for Compiling Commit Changes

As described in the previous sections, JESS is able to isolate specific areas of interest of Java programs and compile them in isolation. To this end, JESS has been designed to work on any type of Java project. For bytecode-centric dependency scanning we do not want to simply compile parts of provided Java projects, but are specifically interested in compiling the *changes* introduced through fix commits. To further improve success rates beyond what JESS achieves in isolation, we propose a *compilation heuristic* that integrates JESS for commit-level compilation.

Figure 3.2 shows a detailed overview of the compilation heuristic. It starts by cloning the source code repository from which the given commit originates. Figure 3.3 shows an exemplary version history of a project with one commit selected for compilation. At first the heuristic collects all Java source code files changed within the commit. To be able to extract the precise bytecode changes performed in the commit, one needs to compile the changed files before and after the commit is applied. Thus, after

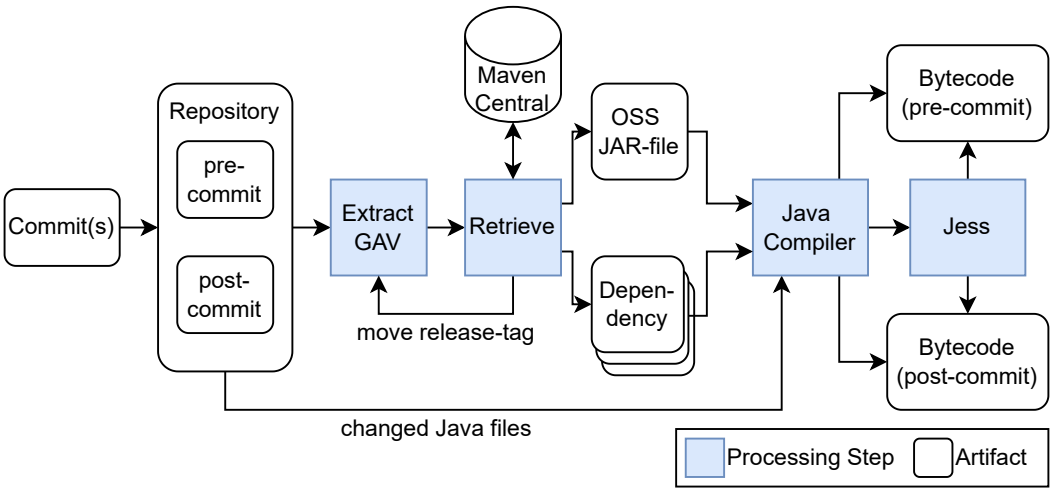


Figure 3.2: Detailed overview of the compilation heuristic for compiling commit changes

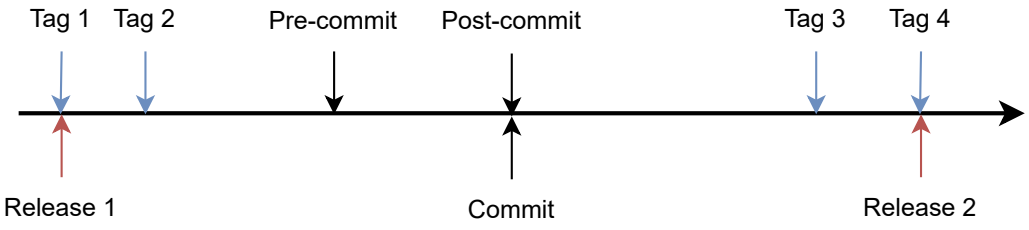


Figure 3.3: Exemplary version history of an OSS project, illustrating a single commit selected for compilation

collecting the changed files, it checks out the revision *before* the selected commit is applied (see pre-commit in Figure 3.3).

To compile the changed files, we need to retrieve all necessary dependencies. This includes a compiled release version of the OSS project, allowing us to avoid compiling the entire project and instead focus only on the changed files. We do not use the compiled release version to extract the changes, as there might have been more changes performed between the release before and after the selected commit.

To retrieve the compiled release version, the heuristic first tries to extract the GAV identifier from the project (see Figure 3.2). This identifier is used to lookup if this specific version of the artifact exists on Maven Central (see Section 2.1) . If it does not find an artifact on Maven Central corresponding to the extracted coordinates, it will checkout the next repository revision tagged with a release tag and repeat the process. Tags are typically used to mark release versions of the project, however sometimes developers tag a specific revision of the project that is not released on Maven Central. Within the example shown in Figure 3.3 one can see that Tag 3 is not pointing towards a release version of the project, but Tag 4 is. Thus, the heuristic needs to move the tag at least two times to find a corresponding release on Maven Central. We perform this step up to ten times, as our empirical testing revealed no cases where additional iterations resulted in successful compilation. If it finds a release version on Maven Central, it retrieves the corresponding JAR file and all available dependencies configured within the `pom.xml` file, which is hosted alongside the JAR file. Extracting the GAV identifier typically works well for Maven projects, as one can simply invoke Maven to obtain this information. For Gradle projects this is often not as trivial. As Gradle build configurations are based on a Groovy/Kotlin-based domain-specific language, Gradle first needs to execute the configuration scripts before the GAV can be extracted. However, this execution often fails for outdated builds. In such cases, we manually inspect the project's configuration scripts to extract the GAV and provide it to the compilation process. Finally, using the retrieved dependencies, as well as the pre-compiled JAR file from Maven Central, the heuristic will forward the collected changed source code files from the repository, to a JDK Java compiler. If the compilation is successful, one will obtain the bytecode of files changed within the given commit (see Figure 3.3). Now to obtain the bytecode of the files *after* the commit changes are applied, the compilation heuristic will repeat the same process analogously for the post-commit version, by checking out the revision that the given commit is pointing to, as depicted in Figure 3.3. If the compilation via the heuristic fails, it will invoke JESS. Although, this is only a secondary option, as JESS might produce bytecode that is different from the original bytecode due to type inference ambiguities presented in Section 3.2.3.

This compilation heuristic is specifically designed to maximize compilation success rates, even for commits that may be years or decades old. Furthermore, it aims at keeping the produced bytecode as similar as possible to the original bytecode.

3.4 Evaluation

In the following we evaluate the effectiveness of JESS's targeted compilation. To do so, we implemented our approach in a tool. We answer the following research questions.

RQ1: How does JESS's compilation perform on popular and current Java projects?

RQ2: How similar is the bytecode obtained via JESS to the original bytecode?

RQ3: To what extent can JESS enable the successful compilation of fix-commit changes?

RQ4: To what extent can the compilation heuristic improve the success rate of compiling fix-commit changes?

The first two research questions focus on evaluating the performance of JESS in general. They aim at showing how effectively it can be used on partial code bases and how similar the generated bytecode is to the one obtained by a successful compilation of the full project. The third and fourth research questions focus on the use case of fix-commit compilation and evaluate JESS and the dedicated compilation heuristic in this scenario.

3.4.1 RQ1: How does JESS's compilation perform on popular and current Java projects?

In this research question we aim at investigating how JESS performs on the latest state of Java projects hosted in source code repositories. Figure 3.4 illustrates our experimental setup for answering RQ1 and RQ2. Based on stars, we selected the 1,000 most popular Java projects on GitHub. From these 1,000 projects we selected the Maven projects [Fou25b], which resulted in 347 total projects considered in our evaluation. We cloned each of the repositories at their most current state and, for each repository, we randomly sampled methods as individual target for compilation via JESS. To have a diverse but manageable dataset, we randomly selected 100 distinct methods or constructors from each repository. We use "methods" to refer

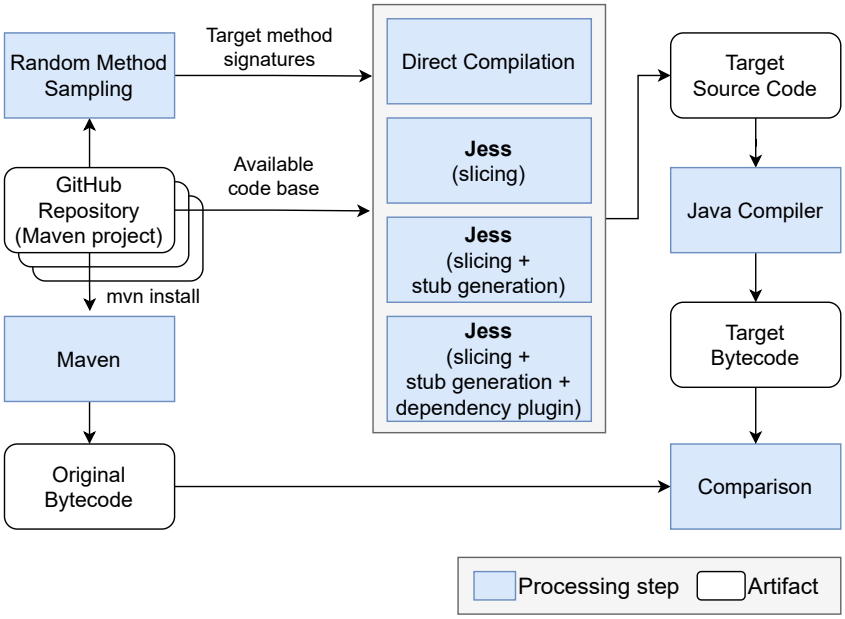


Figure 3.4: Overview of our evaluation setup for RQ1 and RQ2

to both, methods and constructors, from here on. To omit trivial methods like getters and setters, we only considered methods containing at least three lines of code in their body. The code base contained within the cloned repository and the signatures of the sampled target methods are input to JESS. On the same set of target methods, we performed four different compilation procedures. To establish a baseline, we directly provided the target file to the Java compiler for compilation, without processing it with JESS first. We then used JESS to perform three different compilation experiments. In the first experiment, we only used the Java source code within the cloned repository as available code base and solely performed slicing without generating any stub files. In the second experiment, we used the same code base, but complemented the slicing with stub generation. In the final experiment, we added a simple Maven plugin, which (transitively) downloads the available dependencies configured within the project's pom.xml files at the version configured within the project. Note that the custom plugin does not perform any dependency conflict resolution nor does it guarantee that all configured dependencies are available. We use this custom plugin as we cannot use the default Maven Dependency Plugin [Fou24b] for our experiment, as it employs a fail early strategy and stops the dependency resolution process as soon as a single artifact resolution fails. Then, before running JESS, the downloaded JAR files are added to the available code base. In the next step, the target source code generated by JESS (i.e., sliced target file, sliced support files, and optional stub files) is handed to the Java compiler for compilation. To evaluate the similarity of the target bytecode

Table 3.1: Compilation success rates on popular GitHub Java projects using JESS

	Methods	Compilable	Avg. Time (ms)
Direct Compilation	32,970	2,238 (6.79%)	125.7
Jess (slicing)	32,970	12,670 (38.43%)	400.9
Jess (+ stub generation)	32,970	20,539 (62.30%)	464.5
Jess (+ dependency plugin)	32,970	23,744 (72.02%)	831.3

obtained via JESS to the original one (cf. RQ2 in Section 3.4.2), we invoke the default `mvn install` command on each of the cloned projects to compile it via the provided build scripts wherever possible. We then compare the target bytecode to the original bytecode. As previously reported (see Section 3), automatic compilation via invoking the provided build scripts often results in failure, even though the selected projects are at their most current state. Thus, we are only able to perform a comparison for the projects where the default build scripts yielded a successful compilation, which are 134 of 347 projects. We report on the results of the bytecode comparison in Section 3.4.2. Our experiments were executed inside a Docker container running Alpine Linux with Eclipse Temurin JDK 17 and Maven 3.9.3. The container was executed on a Debian 10 machine, configured to use four cores of an Intel Xeon E5-2695 v3 (2.30 GHz) CPU and 32GB of main memory.

Table 3.1 shows the results of our compilation experiments (RQ1). In total we sampled 32,970 distinct methods originating from 21,856 different files. The number of methods listed in Table 3.1 is slightly lower than 100 per project, since a few small projects did not contain enough methods with at least three lines of code. The first row of the table shows our baseline direct compilation. When providing the file containing the target method directly to the Java compiler the success rate was very low at 6.79%, since most files contain references to other files. The compilation success rate increases to 38.43% by slicing the file via JESS first. In fact, references to files outside of the available code base, but not required for compilation, are removed from the respective target and support files. This result shows that a considerable amount of methods within a project can be compiled solely relying on the source code within its repository, without the need to resolve external dependencies at all. However, if the target method references an external dependency, which is not available within the provided code base, compilation fails without stub generation. Applying stub generation after slicing, the success rate of JESS significantly increases to 62.30%. In our final experiment we run our custom dependency download plugin beforehand, to increase the available code base as much as possible. By increasing the code base available to JESS through the downloaded JAR files, the compilation success rate increases to 72.02%, indicating

Table 3.2: Number of stubs generated by JESS for popular GitHub Java projects

	Classes	Fields	Methods	Constructors	Lines
Average	2.19	0.11	1.81	0.18	21.97
Median	2	0	1	0	12
Maximum	19	11	44	8	266

that JESS allows for a targeted compilation of most methods within the investigated projects.

We investigated the cases in which JESS failed (even when using the dependency plugin) and found the main reasons to be: no longer available dependencies, unresolved dependency conflicts, and limitations to the implementation of JESS itself. Our implementation of JESS relies on the symbol-solving capabilities of the popular JavaParser [Jav19] library, thus it inherits JavaParser's limitations, e.g., lack of support for all up-to-date Java features and problems in resolving language constructs like method references (:: operator), lambda expressions, and variable arguments (Varargs). These limitations to JavaParser especially apply in partial code bases.

We additionally measured the time that JESS and the subsequent compilation of the target method require (see Table 3.1). As expected, since no resolution, slicing, or stub generation needs to be performed, a direct compilation of the target file is fast. However, the success rate of a direct compilation is low. Only applying slicing, the average processing time of JESS (including compilation) takes 400ms. As expected the required time increases to 464ms when JESS additionally applies stub generation. Furthermore, as the code base that needs to be considered for resolution increases when using the dependency plugin, the required processing and compilation time also increases to 831ms (not including dependency download time). These results show that, due to its fast compilation times, even when a compilation of the full project is possible, compilation via JESS may be preferable if the area of interest within the code is limited.

Another behavior we investigated was the number of different stubs that JESS needed to generate to make the respective target methods compilable. Table 3.2 shows the results of our investigation. The first row of the table shows the average number of different stub types generated by JESS to enable a successful compilation of target methods, when slicing is not sufficient. The results show the numbers of stubs generated when only the source code within a project's repository is available to JESS. One can see that on average only very few stubs need to be generated to make a method compilable. On average only two classes need to be generated. The most

common required class members are methods, with 1.81 method stubs generated on average. Required stubs for other class members, like fields and constructors, are sparse in comparison to methods. The median numbers of generated stubs, shown in the second row of Table 3.2, are even lower than the averages. However, there are also cases where many stubs need to be generated to enable a compilation. The third row of Table 3.2 shows the maximum number of needed stubs of each type within our dataset. One target method required a total of 19 classes to be additionally generated for compilation. Furthermore, one target method invoked a total of 44 different methods for which JEDD needed to generate stubs. These results show that in most cases only few stubs need to be generated to make a target method compilable without any external dependencies being available.

> JEDD was able to compile up to 72% of randomly sampled methods from popular Java projects, in isolation, only taking 831.3ms per method on average.

3.4.2 RQ2: How similar is the bytecode obtained via JEDD to the original bytecode?

In this research question we investigate the similarity of the bytecode obtained via JEDD to the original bytecode, when the source code is compiled the intended way via provided build scripts. For this investigation we used the same experimental setup as previously described in Section 3.4.1. Wherever we were able to successfully compile a target method via JEDD *and* by invoking the provided Maven build scripts, we compared the bytecode of the target method originating from JEDD and from the default build. We used ASM 9.5 [Con25] to extract the textual representations of the bytecode corresponding to the respective target methods (based on the signature) and performed a textual head-to-head comparison where the method's bytecodes are considered different as soon as there is a single textual difference between the compared bytecodes. Note that, since bytecode is a binary format, the extracted textual representation depends on the ASM framework's interpretation, which is based on the Oracle JVM specification [Ora21d]. To see how similar the resulting bytecodes are, if they are not equal, we computed the normalized Levenshtein Distance (NLD) [YB07] on their textual representations. The NLD is a commonly used metric to calculate the similarity of two text sequences. The Levenshtein Distance represents the number of characters that need to be inserted, deleted or substituted to transform one text sequence into the other. The normalized Levenshtein Distance in addition also considers the lengths of the text sequences to calculate a percentage value of similarity. An NLD of 100% indicates that every single

Table 3.3: Similarity of the bytecode obtained via JESS to the original bytecode

	Methods	Equal	Avg. NLD
Jess (slicing)	7,722	7,162 (92.75%)	0.29%
Jess (+ stub generation)	11,519	8,774 (76.17%)	1.85%
Jess (+ dependency plugin)	13,630	12,192 (89.45%)	0.66%

character needs to be changed, while a value of 0% indicates that both sequences are identical. The lower the NLD is, the more similar are the compared text sequences.

Table 3.3 shows the results of our comparison. The first row of the table shows the comparison results when only slicing is applied by JESS. The number of compared methods is smaller than investigated in Section 3.4.1, as we were only able to compile 134 of the 347 projects using their provided build scripts. Out of the compared 7,722 methods, 92.75% yielded the exact same bytecode as obtained through the intended build. Furthermore, the compared methods were highly similar to each other, as the average NLD is 0.29%, indicating a high degree of similarity between the respective method's bytecodes. The second row of Table 3.3 shows the results when additional stub generation is applied. As expected, when stubs need to be generated to enable a successful compilation, the degree of similarity decreases through the ambiguities that arise. Out of the 11,519 compared methods, 76.17% yielded identical bytecode. The average NLD increased to 1.85%. Though higher than in the case of pure slicing, the bytecode obtained via JESS is still similar to the original bytecode, while increasing the number of compilable methods by more than 62%, indicating the added value of stub generation. Finally, the last row of the table shows the comparison results when the code base available to JESS is augmented with resolvable external dependencies. Out of the 13,630[1] compared methods, 12,192 (89.45%) yielded the exact same bytecode. Furthermore, the average NLD decreased to 0.66%. As expected, due to the availability of external JAR files, the similarity increases compared to the experiment without external dependencies as certain references become resolvable and thus fewer stubs are required for compilation.

We investigated the methods that differ from their original bytecode after purely applying slicing via JESS. While it is clear that, due to ambiguities, the bytecode obtained through compilation supplemented with stubs may contain differences, it is not obvious when only slicing is applied. Our investigation revealed the main causes of the remaining differences to be:

[1] More than 100 methods per project, since one source code method may expand into multiple bytecode methods after compilation, e.g. due to anonymous classes

- Properties that are reflected into the source code right before compilation using Maven [Fou23a].
- Bytecode that is automatically generated by frameworks (e.g. non-null checks), which are configured in the provided build scripts and invoked by Maven, e.g. Project Lombok [Aut25b] or Checker Framework [Aut25a].
- Anonymous classes and synthetic methods generated by the Java compiler are named with an incrementing numeric suffix (e.g. `id$0, id$1, etc.`). This numeric suffix is based on the location within the source code file. Since JESS's slicing removes some of these constructs if not required for compilation, the naming of these constructs may differ. However, this usually only results in a single digit that differs from the original bytecode, making the bytecode still similar to the original one.

Depending on the extent of the available code base, the bytecode obtained via JESS on average only differs by 0.29–1.85% from the original bytecode with 76–93% of all compared methods even yielding identical bytecode.

3.4.3 RQ3: To what extent can JESS enable the successful compilation of fix-commit changes?

In this research question we evaluate JESS's ability to compile changes performed in fix commits. Figure 3.5 shows an overview of our experimental setup. We considered SAP's Project KB [Pon+19] as dataset for our evaluation. Project KB is a manually curated database, which contains vulnerabilities affecting industry-relevant open source projects from the years 2005–2023. In total, it contains 1,297 *vulnerability statements*, which map a Common Vulnerability Enumeration (CVE) [ST25] to the commits within a project's source code repository that fix the corresponding vulnerability. Such commits are referred to as *fix commits*. As described previously, Project KB only contains the snapshot version with the patch applied, but not the corresponding release version. In this experiment we thus assess JESS's performance on directly compiling these snapshots. Each commit in Project KB changes classes and methods that are related to some security vulnerability, making all of them compilation targets for known vulnerability detection tasks. In our evaluation, we only consider statements that affect Java projects and contain corresponding fix commits, which leaves us with 736 unique vulnerability statements. Eight of these produce errors when trying to checkout the referenced commits. We thus exclude these from our experiment. Each vulnerability statement consists of one or more

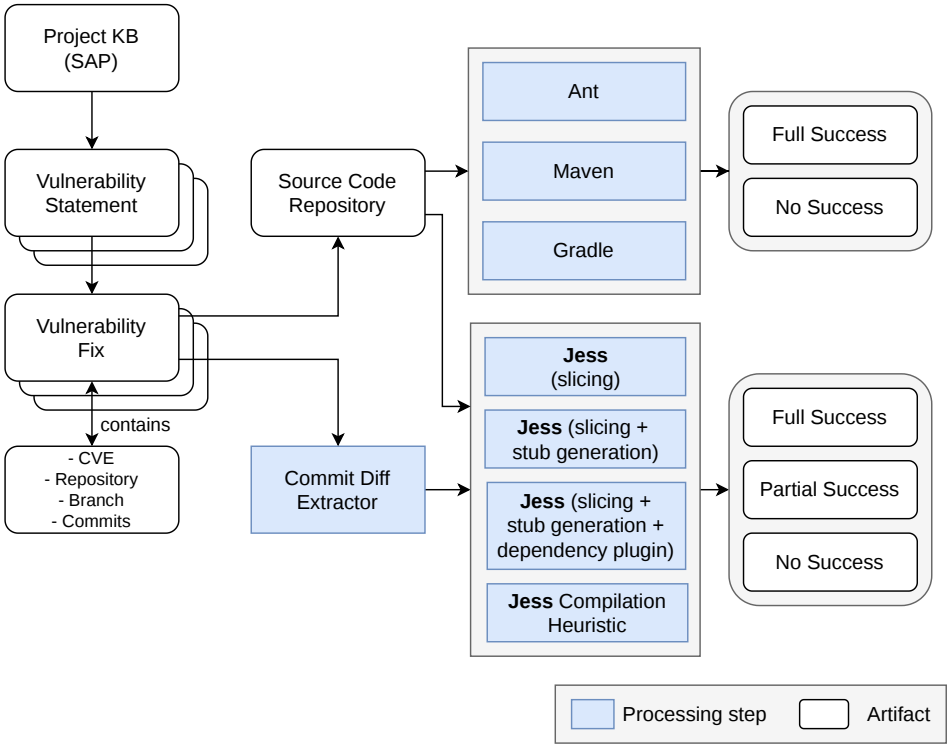


Figure 3.5: Overview of our evaluation setup for RQ3 and RQ4

vulnerability fixes, each containing a set of fix commits that fix a certain CVE. In our case, the 728 considered statements consist of 1,016 vulnerability fixes. We cloned the source code repository belonging to each vulnerability fix, checked out the most recent commit out of the set of fix commits, which we continue to refer to as *snapshot*, and applied one of two subsequent processing steps to the snapshot. At first we are interested in establishing a baseline of how many of these snapshots are able to be compiled by using the build scripts provided within each project. We used the same procedure as Hassan et al. [Has+17] to automatically locate the appropriate build scripts and invoke them in a systematic way. We considered the three popular Java build tools with the respective build commands that provided the highest chance of success as reported by Hassan et al.: Maven [Fou25b] (`mvn compile`), Gradle [Inc25c] (`./gradlew` and `gradle build`) and Ant [Fou24a] (`ant build`). Moreover, since Ant does not define a default build command like Maven and Gradle do, we added the command `ant compile` to our evaluation after manually inspecting the respective build scripts to boost the compilation success rate. In total 1,008 of the 1,016 repository snapshots within our dataset contain build scripts and are thus theoretically compilable via invoking build scripts. Further, we provided the same repository snapshots to JESS. Additionally, we performed a *commit diff extraction* in which we extracted the signatures of classes, methods, constructors and fields

Table 3.4: Compilation success rates when invoking build scripts on Project KB

Build Tool	Compilation Success		Compiled Files	
Ant	2 / 84	(2.4%)	2 / 237	(0.8%)
Maven	109 / 735	(14.8%)	241 / 2,276	(10.6%)
Gradle	6 / 189	(3.2%)	12 / 531	(2.3%)
	117 / 1,008	(11.6%)	255 / 3,044	(8.4%)

that have been modified or added within the respective set of fix commits. We then separately handed each file that has been modified within the fix commits to JESS as target file. Furthermore, we specified all modified class members, as determined by the commit diff extraction, as target members. Like in our previous experiments, we executed JESS with three different configurations (see Section 3.4.1) and provided the repository as available code base. The outcome of invoking a build script is typically binary, either the compilation is successful and one obtains the bytecode of all files within the snapshot or the compilation is not successful and one obtains no bytecode at all. However, when compiling commit changes via JESS the outcome is more fine-grained. When all files that have been modified within the set of fix commits were successfully compiled, we consider it as a full success. If no file could be successfully compiled we consider it as no success. However, in case JESS was able to compile a subset of the targets, we consider it a partial success. The experiments associated with this research question were executed inside a Docker container running Alpine Linux with Eclipse Temurin JDK 11, Maven 3.9.3, Ant 1.10.13 and Gradle 8.0.2. Unlike in RQ1 and RQ2, we used JDK 11 instead of JDK 17, as JDK 11 yields a higher compilation success rate when executing the provided build scripts.

We employed the same experimental setup as in the original study [Sch+24a], but conducted the experiments using a more recent version of Project KB. As a result, the exact outcomes differ from those reported previously.

Table 3.4 shows the results when invoking the provided build scripts for compilation. The results are split up for each individual build tool and show how many snapshots could successfully be compiled by invoking the provided build scripts and how many of the files, modified in the fix commits, could thus be compiled. One can immediately see that the rate of successful compilations is rather low, with projects using Maven as build tool having the highest success rate and projects using Gradle having the lowest. In total, fewer than 12% of the snapshots could be compiled using the provided build scripts, only yielding the bytecode of 8.4% of files changed in the commits, which is a noticeably lower success rate than reported in other

Table 3.5: Compilation success rates when using JEDI on Project KB

	Full Success	Partial Success	Compiled Files
Slicing	219 / 1,008 (21.7%)	501 / 1,008 (49.7%)	1,286 / 3,044 (42.2%)
+ SG	438 / 1,008 (43.5%)	758 / 1,008 (75.2%)	1,987 / 3,044 (65.3%)
+ DP	505 / 1,008 (50.1%)	765 / 1,008 (75.9%)	2,187 / 3,044 (71.8%)

SG = Stub Generation, DP = Dependency Plugin

studies [SP16; Has+17; Tuf+17]. However, Project KB contains entries dating back to 2005, which means that many of the fix commits are old. According to Sulír and Porubän [SP16] the age of the project heavily influences the probability of successful compilation, which could explain the lower success rates.

Table 3.5 shows the compilation results using JEDI. Again, each row represents the different configurations used for JEDI. One can immediately see that JEDI performs significantly better than the provided build scripts. Even when only slicing is applied, JEDI was able to fully compile all commit changes within 21.7% of snapshots. For 49.7% of snapshots it was able to compile it at minimum partially, resulting in at least some successfully compiled files. When comparing the number of files JEDI was able to compile to the number of files compilable via provided build scripts there is an even higher difference. This difference only increases further when JEDI additionally applies stub generation or uses the dependency plugin to download available JAR files. Note that the dependency plugin only works for Maven projects, so for the Ant and Gradle projects no dependencies are downloaded at all. When using JEDI with slicing, stub generation and the custom dependency plugin, it is able to fully compile all commit changes within 50.1% of the snapshots, while being at least able to compile parts of 75.9% of all snapshots. A partial compilation of the fix would allow for a subsequent matching for at least part of the patch. In total JEDI is able to compile 71.8% of all files modified within the fix commits for Java projects listed within Project KB. Note that due to only being able to obtain the bytecode of at most 137 files modified within fix commits (less than 5% of files), using JEDI *and* the provided build scripts, we did not perform another bytecode comparison experiment. The results from the previous experiment (see Section 3.4.2) should also apply here.

In theory, the success rate of JEDI can be increased even further. As long as all constraints imposed by the compiler are satisfied, the compilation will succeed, which can be achieved by e.g. altering the source code designated for compilation. However, as constraints are not unambiguous in partial code bases (see Section 3.2.3), this always comes with a trade-off between success rate and degree of similarity of the

Table 3.6: Compilation success rates when using the custom compilation heuristic on Project KB

Comp. Type	Statements		Fixes		Files	
Java Compiler	534 / 728	(73.4%)	761 / 1,016	(74.9%)	1,835 / 3,053	(60.1%)
Jess Full Success	54 / 728	(7.4%)	65 / 1,016	(6.4%)	118 / 3,053	(3.9%)
Jess Partial Success	67 / 728	(9.2%)	104 / 1,016	(10.2%)	430 / 3,053	(14.1%)
	655 / 728	(90.0%)	930 / 1,016	(91.5%)	2,383 / 3,053	(78.1%)

resulting bytecode to the original one. As JESS was primarily designed for producing bytecode that is suited for a subsequent comparison, we privilege bytecode similarity over success rate.

Relying solely on the provided build scripts enables compilation for only 8.4% of the files affected by fix commits within Project KB. In contrast, using JESS, we were able to retrieve bytecode for 71.8% of all modified files. Moreover, for 75.9% of vulnerability fixes, JESS successfully compiled at least part of the fix.

3.4.4 RQ4: To what extent can the compilation heuristic improve the success rate of compiling fix-commit changes?

In the previous research question, we evaluated JESS's performance in compiling fix-commit changes when applied directly to the snapshots in isolation. We now investigate the extent to which the proposed compilation heuristic (see Section 3.3) can improve the compilation success rate of fix-commit changes. We use the same experimental setup and dataset as for RQ3, but instead of using JESS in isolation, we employ it in the proposed compilation heuristic. We now also consider the eight repository snapshots without provided build scripts.

Table 3.6 presents the compilation success results when applying the compilation heuristic to Project KB. The column "Comp. Type" indicates the success rate at each stage of the heuristic. The "Java Compiler" row reports the number of statements, fixes, and files successfully compiled using only the Java compiler, without the need to invoke JESS. The "Jess Full Success" and "Jess Partial Success" rows report cases where compilation with the Java compiler alone failed, but JESS achieved a fully or partially successful compilation, respectively. One can see that using the compilation heuristic we were able to fully compile 73.9% of statements and 74.9% of fixes within Project KB, without even needing to invoke JESS, by directly providing the changed files to a JDK 17 Java compiler. Note that for 170 of the compiled statements we had to supplement additional information about GAVs. We

forwarded the 194 failing statements to JEss, which successfully compiled 54 in full and partially compiled an additional 67 statements. Using the compilation heuristic, we were able to obtain bytecode for 90% of all relevant statements in Project KB, and even for 91.5% of the corresponding fixes. Furthermore, unlike in RQ3, we compiled not only the repository snapshots after the fix commits had been applied, but also those before the fix commits.

A manual investigation of the failing cases revealed that most failures are due to projects lacking corresponding artifacts on Maven Central, or due to code dependencies on both the pre- and post-commit release versions. Since only one of these artifacts can be provided during compilation, the process fails in such cases. Furthermore, in few cases the compiled release version or the dependencies of the project to be compiled are not stored on Maven Central but on different repositories.

These results demonstrate that the proposed compilation heuristic enables effective generation of bytecode corresponding to fix-commit changes, even when the commits date back several decades and shows significant improvement over invoking provided build scripts or invoking JEss in isolation.

> Using the custom compilation heuristic tailored for compiling commit changes, we were able to compile 90% of fix statements, 91.5% of the corresponding fixes, and 78.1% of all modified files within Project KB.

3.5 Threats to Validity

Our evaluation of JEss's performance on compiling commit changes relies on the vulnerability database Project KB. Project KB contains fix commits affecting industry relevant open source projects. This means that the projects used within our evaluation are typically well maintained. An evaluation on projects that are not industry-relevant may have yielded different results. Furthermore, we only used default build commands (except for Ant) for invoking builds, as Hassan et al. [Has+17] reported them to have the highest success rate for compilation. However, in our specific dataset there may still be different build commands, which could have caused a higher compilation success rate when invoking the provided build scripts. Moreover, we only used Maven projects for our first two evaluation experiments, as we only created a simple dependency download plugin for Maven. While such a plugin can also be created for Gradle and Ant, the performance of dependency resolution may be different for other build tools.

3.6 Related Work

Different approaches have been developed, which aim at increasing the rate of successful compilations of Java projects by fixing the default build process or enabling a compilation of partial code bases. Stubber [SAH21] aims at compiling partial source code bases by providing predefined stub classes. In contrast to JESS, Stubber does not specifically adjust the stub classes to the target file, but modifies the existing source code to fit a previously created stub class, thus changing the resulting bytecode of the target file. Jcoffee [GMP20] leverages the error messages reported by the Java compiler during compilation. Based on the received error, Jcoffee modifies the source code or generates the needed declaration code to supplement the compilation. Stubber and Jcoffee modify the source code of the compilation target within the partial program and therefore alter the resulting bytecode. Dagenais and Hendren [DH08] propose a technique that enables static analysis of partial Java programs by applying a similar type inference approach as JESS does during its stub generation step. Their approach supports up to Java 5-compliant source code, parses the program into a Jimple-based abstract syntax tree and adds missing type information directly to the resulting AST. These approaches use similar strategies as JESS, however, they do not aim at producing bytecode identical to the original compilation. Thus, these approaches are not applicable to the use case JESS has been designed for. Melo et al. [Mel+20] propose a similar concept as Dagenais and Hendren for inferring types of only partially available C programs. SnR [Don+22] also applies type inference strategies to compile incomplete Java code snippets. However, in contrast to other techniques that generate supplemental stub files, SnR leverages a pre-built knowledge base, which contains the type signatures of various libraries. Hassan and Wang [HW17] leverage natural language processing techniques to mine readme files and Wiki pages of software repositories to extract build commands that deviate from default build commands. By doing so they are able to increase the build success rate of projects where all build errors can be solved by running a different set of build commands. HireBuild [HW18] uses a history-driven repair approach to fix failing Gradle build scripts. It uses revisions from a build fix database that have previously been applied to build scripts in order to fix them and automatically creates new build fix templates from the successful fixes. These templates are then automatically applied to the failing build script if a similar failure mode as in the past is causing the build failure. Lou et al. [Lou+19] investigated HireBuild and found multiple shortcomings. Inspired by the findings they propose a new technique called HoBuFF that does not rely on historical information but generates patches based on resources available within the present project. BuildMedic [MMP18] tries to fix

dependency-related build issues in Maven projects by implementing an approach that automatically updates or deletes included dependencies.

Hassan et al. [Has+17] investigate the feasibility and challenges that arise for the automatic building of Java projects in software repositories. Their study shows that automatic building is a challenging task, which often results in failure. Tufano et al. [Tuf+17] specifically investigate the compilability of snapshots within Java software repositories. Their study shows that almost all of the investigated projects contain long periods of snapshots that are not compilable using the provided build scripts. Sulír and Porubän [SP16] study automatic build invocation on more than 7,000 Java projects. Their investigation reveals that projects using Gradle or Ant as build tools have higher build failure rates than projects using Maven. Furthermore, they show that dependency resolution issues are by far the most common issue, which is causing build failures.

3.7 Conclusion

In this chapter we presented our approach JESS, which enables a targeted compilation of Java programs. By resolving all references made from the targeted source code parts, JESS slices away the parts of the code base which are not needed for a successful compilation. If references are not available in the provided code base, JESS can generate stub files which contain type stubs that are inferred from their usage context.

We evaluated JESS on 347 of the most popular Java GitHub projects. Of the 32,970 randomly sampled methods and constructors, JESS was able to compile more than 72% of which the resulting bytecode on average only differed by 0.66% from the bytecode obtained when running the build scripts provided in the code bases. In almost 90% of cases JESS produced equal bytecode. In a real-world experiment using the Project KB vulnerability database, the provided build scripts only achieved successful compilation of 8% of all files modified within the provided fix commits. JESS allowed for the compilation of 70% of all files without the need to invoke any build scripts. Using the dedicated compilation heuristic we were even able to obtain the corresponding bytecode of 90% of all Java entries in Project KB.

These results show that the concept of targeted compilation as provided by JESS effectively enables the compilation of specific program areas of interest, particularly in cases where build scripts fail to provide a successful compilation. They furthermore

demonstrate JESS's capability of compiling changes performed in vulnerability-fixing commits, which is essential for conducting bytecode-centric dependency scanning.

4 Bytecode Normalization

Code similarity analysis techniques have a wide range of applications, including vulnerability detection, code clone detection, plagiarism detection, copyright infringement investigation or program comprehension. To this end, researchers have developed various approaches for code similarity analysis on Java applications [RC08; KKI02; Saj+16; Jia+07; ZH18; PMP+02]. Most available techniques operate on source code. However, source code is not always available, as applications are typically distributed in binary form. This limitation also applies to bytecode-centric dependency scanning, where the source code of included dependencies is often inaccessible.

Only few approaches have been developed for similarity analysis based on bytecode [BM98; KRR14; Yu+19]. This may be due to the increased complexity when trying to compare bytecode instead of source code. As Dann et al. [DHB19] and Kononenko et al. [KZG14] have shown, the comparison of bytecode is more complex than the comparison of source code, since equal source code is compiled into different bytecode, depending on the compiler, version and configuration used. Although the generated bytecode is semantically equivalent, its syntactic structure can differ significantly.

While we can obtain the bytecode corresponding to a source code fix for a vulnerability in an OSS project through targeted compilation (see Chapter 3), the original compilation environment of the dependency is typically unknown. As a result, our approach must account for potential differences in compilation environments.

To overcome this difficulty we investigate the utility of *bytecode normalization* to create a representation that is *independent* of the environment that has been used for compilation. This independent representation can subsequently be used for a bytecode-based code similarity analysis. Our approach to achieve bytecode normalization, which builds upon Dann et al.'s approach [DHB19], is a procedure that

1. translates the bytecode into Jimple, the intermediate representation of the SootUp [Kar+24] Java analysis framework, which reduces the more than 200 available bytecode-instructions to only 15 different Jimple statements,

2. as a baseline first applies common optimizations like constant propagation, dead code removal and unconditional branch folding to further reduce differences, and then specifically, and lastly

3. transforms *compilation differences* induced by different compilation environments.

We uncovered the set of compilation differences by systematically comparing bytecode of popular Java libraries generated by different vendors, versions and configurations of Oracle's Java Development Kit's (JDK) and OpenJDK's compiler (javac). During this initial study, we found a total of 16 classes of compilation differences.

We implemented bytecode normalization in a tool called JNORM, and evaluated it on more than 300 of the most popular Java projects on GitHub by compiling the same source code within various compilation environments with different compiler vendors, versions and target levels. The evaluation shows that even a single increase of the compiler version may result in up to 46% of all generated bytecode files containing compilation differences. By applying JNORM's bytecode normalization one can reduce these differences by as much as 99%. Thus, bytecode normalization can function as an important enabler for bytecode similarity analysis in all cases in which source code is not available.

The remainder of this chapter is structured as follows: In Section 4.1, we present a study regarding the use of different Java compilers and target levels in real-world projects that we used to define JNORM's scope. Afterwards, Section 4.2 presents the concept of Java bytecode normalization implemented in JNORM and an overview of the uncovered compilation difference classes. Section 4.3 presents an evaluation of JNORM on a a set of real-world Java projects. Possible threats to the validity of our experiments are presented in Section 4.4. We present related work in Section 4.5 and conclude in Section 4.6.

4.1 Study: Usage of different Compilers and Target Levels in Java Projects

Before designing our approach to bytecode normalization and developing JNORM, we investigated which Java compilers and target levels are commonly used in real-world Java projects, in order to determine which ones need to be considered for normalization. To do so, we used the SEART GitHub Search dataset [DAB21], which was created to enable large scale studies on GitHub projects without being limited

by the GitHub API usage limitations and contains metadata (but no file contents) for a large amount of repositories hosted on GitHub. We queried the SEART GitHub Search dataset to search for all repositories with Java as main language. This query resulted in 91,113 GitHub repositories containing Java projects. We further limited the dataset for our experiment by only including projects that use Maven [Fou25b] as build tool, since it is not as straightforward to determine the used compiler with other build tools. While a Maven build configuration is specified in a static XML file, a build configuration for Gradle [Inc25c], another popular build tool for Java, is dynamically specified in a set of build scripts, which need to be executed to determine the configured compiler. However, executing a Gradle build is a time-consuming task and error-prone task [Has+17]. After this filtering step we were left with 29,514 Maven projects, of which we cloned the default branch as of March 2023.

The default way of compiling Maven projects is to invoke the Maven Compiler Plugin [Fou25e], which handles all steps related to the compilation. By default the Maven Compiler Plugin uses the compiler of the JDK or OpenJDK installed on the system. However, the Java compiler to be used can be changed by setting the `compilerId` property in the plugin's configuration [Fou25h].

To investigate the usage of different compilers in our dataset, we scanned the project's build configuration for this specific property. Due to the SEART GitHub Search dataset only containing repository metadata and the now applying GitHub API limitations, we only scanned the project's root build configuration file. Though it is possible to configure different compilers for certain modules of a project, typically the whole project is compiled with the same compiler that is defined in the project's root build configuration file.

Table 4.1a shows the results of this experiment. In 99.5% of the projects, the default compiler provided by the JDK installed on the system (whether from Oracle or another OpenJDK vendor) is used. There are few instances where the javac-with-errorprone compiler, is used. Javac provides an additional command line interface to the default JDK compiler. The other two notable compilers use Eclipse's JDT compiler for compilation. Note that many of the Eclipse compiler usages listed in Table 4.1a are only part of a debug profile in the build configuration and are therefore only used for debug purposes, as the JDT core compiler provides a large set of debug utilities [Fou25a]. The projects are still compiled with the default JDK compiler for distribution.

Table 4.1: Compiler and target level configuration statistics in real-world Java projects

(a) Compiler

Compiler	Number
Default JDK	29,355
javac-with-errorprone	63
Eclipse/JDT	48
groovy-eclipse-compiler	35
Others	13

(b) Target level

Target Level	Number
8	12,260
6	3,550
7	3,287
11	2,063
17	871
5	728
Others	598

These results show that the vast majority of real-world projects use the JDK's compiler for compilation, there are only few instances in real-world projects in which other compilers are used.

To investigate the usage of Java target levels we used the same process, but scanned the root build configuration file for the properties defining the Java version for compilation, e.g. `source`, `target` or `release` properties [Fou25g; Fou25f]. In 79% of all scanned Java projects the target level was explicitly defined. In cases where it is not defined, the default level of the used compiler is utilized. Table 4.1b shows the usage amount of levels in the investigated projects where it was explicitly defined. More than 50% of projects are configured to use Java 8 as target. Target levels prior to Java 5 or levels in-between Long-Term-Support (LTS) versions are only used sparsely.

Based on these results, we limit our normalization approach to the compilers provided by the JDK and OpenJDK, and to Java target levels 5–8, 11, and 17.

4.2 Java Bytecode Normalization with JNorm

Java bytecode normalization allows for the removal of differences in Java bytecode that are solely introduced by the usage of different compilation environments. In the following we describe how we detected the compilation differences in the first place, as well as the details of our bytecode normalization approach and its implementation in JNorm.

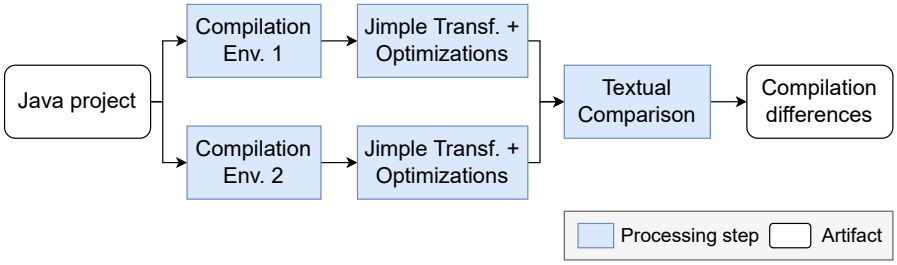


Figure 4.1: Setup to determine compilation differences

4.2.1 Investigation of Compilation Differences

Before the development of our bytecode normalization approach JNorm, we performed a study to investigate the differences induced by different compilation environments. Figure 4.1 shows the setup we used to determine compilation differences. For each comparison, we supplied the source code of various versions of the popular Java libraries Apache commons-io, Apache commons-lang, Jackson-databind, SLF4J and Google Guava, to two different environments for compilation. Afterwards, to reduce dissimilarities, we transformed the resulting bytecodes to Jimple and applied code optimizations, provided by the SootUp analysis framework. Finally, we conducted a textual comparison of the optimized Jimple representations to identify any remaining compilation differences. Two files were deemed different and subsequently manually inspected, if even a single character mismatch was detected.

Our compilation environments included the javac compilers shipped with JDKs 5–8, 11, and 17. Moreover, this version range covers all Long-Term-Support (LTS) versions of the Java ecosystem until August 2023. A usage study of Java compilers and target levels in Java projects, which revealed these to be the by far most relevant compilers and versions, is available within an electronic appendix in our provided artifact.

We consider three types of parameters, JDK vendor, JDK version, and Java target level. We used Oracle's JDK, as well as OpenJDKs distributed by Amazon Corretto and Eclipse Adoptium.

In total, our setup revealed 16 compilation difference classes, present in the investigated projects, which are listed in Tables 4.2 and 4.3. Table 4.2 shows the difference classes produced by changing the JDK version, while Table 4.3 shows the difference classes produced when adjusting the Java target level. We did not find any vendor-related difference classes in our initial experiments.

Table 4.2: Difference classes on JDK version change

ID	JDK Version	Compilation Difference Class
N1	5 → 6 & 7 → 8	Synthetically generated methods
N2	5 → 6	Arithmetic
N3	6 → 7	CharSequence toString invocation
N4	7 → 8	Empty try-catch-finally block
N5	7 → 8	String constant concatenation
N6	8 → 11	Method reference operator
N7	8 → 11	Buffer method invocation
N8	8 → 11	Try-with-resources
N9	8 → 11	Duplicate checkcasts
N10	11 → 17	Enums

Table 4.3: Difference classes on target level change

ID	Target Level	Compilation Difference Class
N11	6 → 7	Outer class object creation
N12	8 → 11	Dynamic string concatenation
N13	8 → 11	Nest-based access control
N14	8 → 11	Invocation of private methods
N15	8 → 11	Inner class instantiation
N16	multiple[1]	Insertion or removal of typechecks

Furthermore, we inspected the official JDK release notes [Cor25a] related to newly released compiler versions. However, this inspection did not reveal any so far uncovered difference classes. Our evaluation performed on more than 300 of the most popular Java projects (see Sections 4.3.3 and 4.3.4) also revealed no additional difference classes.

In the following we describe JNORM's approach to bytecode normalization and how it transforms the identified compilation difference classes into a representation that is common across all investigated compilation environments.

4.2.2 Overview of JNORM

Figure 4.2 depicts an overview of JNORM. First, JNORM transforms a Java bytecode file (.class file) into Jimple format, which is specifically designed for efficient optimizations and transformations. Note that JNORM also has the capability to process

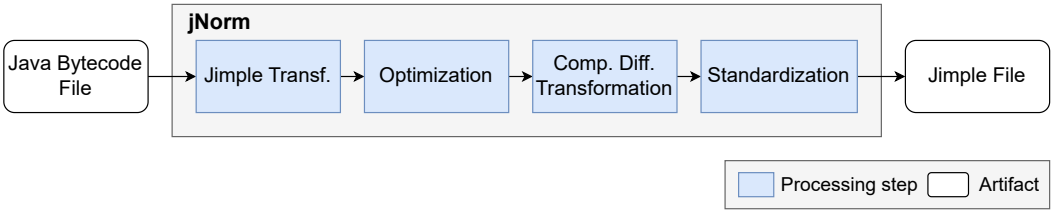


Figure 4.2: Overview of JNORM

multiple bytecode files at once, and therefore full Java projects, but because each file is normalized independently of others we will explain bytecode normalization of single files. To reduce the initial set of differences for the following steps of the normalization process, JNORM applies different types of common optimizations to the Jimple representation of the input bytecode file. Afterwards, in the Compilation Difference Transformation step (see Figure 4.2), JNORM handles the remaining set of compilation differences by performing certain transformations on the optimized Jimple representation. These transformations are targeted towards specific constructs that we found to be compiled differently based on the used compilation environment. JNORM detects these constructs within the target program and transforms them into a normalized representation. The applied transformations interfere with the naming scheme of local variables inside the target programs, which cause the introduction of new dissimilarities. JNORM handles these dissimilarities by standardizing (see Figure 4.2) the order and naming scheme of variables. After the normalization process is finished, JNORM outputs the normalized Jimple representation.

4.2.3 Jimple Transformation and Optimization

The first step of Java bytecode normalization consists of transforming the targeted bytecode file into a Jimple representation. This allows for a convenient application of common program optimizations provided by SootUp. We apply the following optimizations to each method [Uni25]:

- **Nop Elimination:** Removes `nop` instructions.
- **Empty Switch Elimination:** Removes empty `switch` statements that only contain the `default` case.
- **Cast and Return Inlining:** Inlines and simplifies return statements by eliminating unnecessary type casts of local variables performed immediately before return.

- **Local Splitting:** Splits local variables that are redefined multiple times into distinct variables, each assigned only once.
- **Local Aggregation:** Identifies local variables with a single use and safely aggregates them to eliminate unnecessary copies.
- **Type Assigning:** Assigns type information to local variables.
- **Constant Propagation and Folding:** Replaces expressions that entirely consist of compile-time constants (e.g. 2 * 3) by the constant result.
- **Dead Assignment Elimination:** Removes assignment statements to local variables, whose value is not subsequently used.
- **Unreachable Code Elimination:** Removes unreachable code.

These optimizations already contribute to a decrease of dissimilarities introduced during compilation [DHB19]. However, after applying the optimizations, many important compilation differences still remain, which are targeted in the next step.

4.2.4 Compilation Difference Transformation

Through our investigation (see Section 4.2.1) we identified 16 compilation difference classes summarized in Tables 4.2 and 4.3. In the following we describe the identified classes and the transformations applied by JNORM in detail.

The transformations that JNORM performs are not arbitrarily chosen. Each transformation produces a version that is generated by at least one compiler within our dataset. Furthermore, the decision whether to transform a compilation difference class to the older or the newer version is also not arbitrary. Typically one of the two versions contains more information than the other (e.g. a more specific return type in the newer version or the amount of string concatenation calls before their combination into a single call). As we cannot simply add information that is unavailable when only having access to the bytecode, we have to transform the difference class to the version that contains less information, therefore stripping some information from the generated bytecode. However, this information cannot be used for similarity analysis, since, based on the used compilation environment, it is not guaranteed to be present in the bytecode.

Note that we do not aim at generating an executable version of the bytecode with all semantics preserved, but at preserving information that is possibly important for a similarity analysis. Similarity analysis approaches that additionally require an

executable version of the analyzed application, can use the original bytecode that has not been normalized, in addition to the normalized version.

N1: Synthetically generated methods

In many cases, the JDK compiler generates synthetic methods and inserts them into classes. Often this is used to generate bridge methods that enable access to private members. Such synthetically generated methods are marked by the compiler with a specific `synthetic` flag [Cor25b]. Depending on the JDK version, these methods are not always generated in certain cases. For example, starting with JDK 6, when a method in a class uses a `Comparator` to define a specific ordering of objects, the compiler may automatically generate a corresponding `sort` method within the class. Thus, such synthetic methods introduce differences and cannot be reliably used for a code comparison.

Transformation: JNORM removes synthetic methods from the Jimple representation, as these methods are compiler-generated and cannot be directly modified in the source code.

N2: Arithmetic

In certain cases, the bytecode produced by JDK 6 and later replaces integer subtractions with additions involving negative operands. A statement like `i1 = i1 - 5`, generated by JDK5, is replaced by a conversion of the positive number to a negative one (`i1 = (int)-5`) and a subsequent addition with the negative operand like `i2 = i2 + i1`, by JDK6 and higher.

Transformation: Whenever JNORM identifies an addition involving negative integers, it converts it into a subtraction.

N3: CharSequence toString invocation

The JDK 7 compiler introduced a change in how the `toString` method is represented in the bytecode when invoked on an object of type `CharSequence`. As illustrated in Listing 4.1 (with subtle differences highlighted), the invocation type `interfaceinvoke` replaces `virtualinvoke`, and the method's return type is refined from `java.lang.Object` to the more specific `java.lang.CharSequence`.

```
1  java.lang.CharSequence r1;
2  java.lang.String r2;
3
4  // JDK6:
5  r2 = virtualinvoke r1.<java.lang.Object: java.lang.String toString()>();
6
7  // JDK7:
8  r2 = interfaceinvoke r1.<java.lang.CharSequence:
9    java.lang.String toString()>();
```

Listing 4.1: `toString` method invocation (Jimple)

Transformation: Whenever JNORM identifies a call to a `toString` method with a `java.lang.CharSequence` return type, it converts the method call to its previous, more generic, version.

N4: Empty try-catch-finally block

In most cases, a try-catch-finally block includes one or more catch clauses that handle specific types of thrown exceptions. However, catch blocks can be empty or even missing completely. A try-catch-finally block with empty (or even missing) catch blocks is a syntactically valid Java construct, used to execute some instructions, no matter what happens in the try block. Prior to JDK8, the JDK compiler produces a redundant exception catching block[2] in the bytecode, if a catch block is empty or missing.

Transformation: If JNORM identifies such redundant exception catching blocks, it removes them from the Jimple representation of the bytecode.

N5: String constant concatenation

Starting with JDK 8, the compiler introduces optimizations for string concatenation. Whenever multiple string constants are concatenated, compilers prior to JDK8 would use multiple calls to the `StringBuilder.append` method. A simple concatenation like

```
String helloWorld = "Hello "+ "World!";
```

[2] In bytecode and Jimple there exists no notion of catch blocks. We use this terminology in synonym with exception traps.

```
1  org.apache.commons.io.IOFileFilter r0;
2
3  // JDK8:
4  virtualinvoke r0.<java.lang.Object: java.lang.Class getClass()>();
5
6  // JDK11:
7  staticinvoke <java.util.Objects:
8    java.lang.Object requireNonNull(java.lang.Object)>(r0);
```

Listing 4.2: Method reference operator usage (Jimple)

would result in two calls to the `StringBuilder.append` method, one receiving "Hello " and the other receiving "World!" as argument. However, as of JDK8, the compiler concatenates these two strings at compile time and produces a single call to `StringBuilder.append`. This holds true only for subsequent string constants: whenever a substring assigned to a *variable* is involved in the concatenation, multiple `StringBuilder.append` calls are used still.

Transformation: JNORM identifies consecutive `StringBuilder.append` calls with string constants that are not assigned to variables and merges them into a single method invocation.

N6: Method reference operator

With the release of JDK8, the method reference operator (::) was introduced to the Java programming language. It allows one to refer to a method with the help of its declaring class or object name and is especially useful in combination with streams. Listing 4.2 shows how the operator usage is handled during compilation. Before performing the actual method call, if the operator is referring to a method of an object, a null check is performed at runtime. This is done to ensure that the object, the referred method belongs to, actually exists and is not `null`. The usual way to perform null checks in JDK8 and earlier is to call the method `getClass` on the object to check. This mechanism was replaced in newer JDKs by invoking the static `requireNonNull` method.

Transformation: For normalization, JNORM transforms all occurrences back to the old null-checking mechanism.

```
1 java.nio.ByteBuffer r0;
2
3 // JDK8:
4 virtualinvoke r0.<java.nio.ByteBuffer: java.nio.Buffer flip()>();
5
6 // JDK11:
7 virtualinvoke r0.<java.nio.ByteBuffer: java.nio.ByteBuffer flip()>();
```

Listing 4.3: Buffer method invocation (Jimple)

N7: Buffer method invocation

Starting with JDK11, the return types of methods in all subclasses of `java.nio.Buffer` were further refined to return the specific subclass type instead of the generic `java.nio.Buffer` type. Listing 4.3 shows that methods of class `java.nio.ByteBuffer`, compiled with JDK11, return `ByteBuffer` instead of `Buffer`. This holds true for every subclass of `java.nio.Buffer` and any method returning a `Buffer` object.

Transformation: When JNORM detects the invocation of a method from a subclass of `java.nio.Buffer` that returns a `Buffer`, it updates the return type to the more specific subclass type.

N8: Try-with-resources

The try-with-resources statement allows the declaration of resources to be used within the statement, ensuring they are automatically closed at the end, regardless of whether an exception is thrown. Whenever a try-with-resources statement is used in the source code, the JDK compiler generates multiple nested exception handlers, since the bytecode does not provide a separate instruction for such a statement. In some cases, prior to JDK11, these wrapped exception handlers are redundant, since they do not cover any application code but only automatically generated exception handling code. These redundant exception handlers are not created as of JDK11.

Transformation: Whenever JNORM identifies an exception handler that only covers automatically generated exception handling code, it removes the exception handler and its corresponding code from the declaring method.

N9: Duplicate checkcasts

Due to a bug [Cor05] fixed as of JDK11, earlier compilers may insert the same `checkcast` instruction twice, one after the other.

Transformation: JNORM removes redundant typechecks for normalization, if it identifies such duplicates.

N10: Enums

Enums in Java are special types that can only take on certain predefined values. When an enum is created, the JDK compiler creates a separate class for each enum and defines the possible values inside the `clinit` method, which acts as a static initializer. In contrast to a constructor, which is called when an object of a class is initialized, the `clinit` method is called when the class itself is initialized. Prior to JDK17, (cf. Listings 4.4 and 4.5) the initialization of the possible enum values is performed directly inside the `clinit` method, while in JDK17 the definition is moved to its own method, which is called from `clinit`.

Transformation: If JNORM detects that the enum values are initialized within the `clinit` method, it moves the initializations into its separate method and calls this method from `clinit`.

N11: Outer class object creation

Changing the Java target level from 6 to 7 affects the generated bytecode when an inner class instantiates a sibling inner class within the same outer class, as demonstrated in the following listing:

```
SiblingInnerClass sic = getOuterClass().new SiblingInnerClass();
```

In this case the method `getOuterClass` returns a reference to the outer class shared by both inner classes, the one that contains the above statement and the one that is created by the statement. Whenever this is the case, the compiler inserts a check to verify, that the method `getOuterClass` does not return `null`. This is done in the same way, as described for difference class N6, where the previous way of performing

```
1  static void <clinit>(){
2    Vehicle $r0, $r1, $r2, $r4, $r5,
         $r6;
3    Vehicle[] $r3;
4
5    $r3 = newarray (Vehicle)[3];
6    $r4 = <Vehicle: Vehicle CAR>;
7    $r3[0] = $r4;
8    $r5 = <Vehicle: Vehicle BIKE>;
9    $r3[1] = $r5;
10   $r6 = <Vehicle: Vehicle PLANE>;
11   $r3[2] = $r6;
12   <Vehicle: Vehicle[] $VALUES> =
         $r3;
13   return;
14 }
```

Listing 4.4: JDK11 enum definition

```
1  static void <clinit>() {
2    Vehicle[] $r3;
3
4    $r3 = staticinvoke <Vehicle:
         Vehicle[] $values()>();
5    <Vehicle: Vehicle[] $VALUES> =
         $r3;
6    return;
7  }
8
9  private static Vehicle[] $values(){
10   Vehicle $r1, $r2, $r3;
11   Vehicle[] $r0;
12
13   $r0 = newarray (Vehicle)[3];
14   $r1 = <Vehicle: Vehicle CAR>;
15   $r0[0] = $r1;
16   $r2 = <Vehicle: Vehicle BIKE>;
17   $r0[1] = $r2;
18   $r3 = <Vehicle: Vehicle PLANE>;
19   $r0[2] = $r3;
20   return $r0;
21 }
```

Listing 4.5: JDK17 enum definition

a null-check via the `getClass` method is replaced by a call to the `requireNonNull` method.

Transformation: JNORM transforms all occurrences back to the old null-checking mechanism, as it does for difference class N6.

N12: Dynamic string concatenation

In Java 11 and later the old string concatenation approach of repeatedly calling the `StringBuilder.append` method (see N5), is replaced by a single `invokedynamic` instruction, which defers the resolution of a method call to runtime. This change was introduced to optimize the performance of string concatenations [Shi15]. Listing 4.6 showcases the differences of string concatenation compiled for target levels 8 and 11. Previously, for each part of the string concatenation, one call of the `StringBuilder.append` method was required. However, in the new version, a dynamic approach that looks similar to template-based string building is generated. A single dynamic call of the `makeConcatWithConstants` method is performed, where string literals are concatenated into a single literal, while dynamic values are expressed by placeholders (see `\u0001` in line 23 in Listing 4.6) that are replaced by the resolved value during runtime.

```
int i0;
java.lang.StringBuilder $r0, $r1, $r2, $r3;
java.lang.String[] r5;

// Target Level 8:
$r0 = new java.lang.StringBuilder;
specialinvoke $r0.<StringBuilder: void <init>()>();
$r1 = virtualinvoke $r0.<StringBuilder:
 StringBuilder append(java.lang.String)>("Amount: ");
$r2 = virtualinvoke $r1.<StringBuilder:
  StringBuilder append(int)>(i0);
$r3 = virtualinvoke $r2.<StringBuilder:
  StringBuilder append(java.lang.String)>(" Pieces");
virtualinvoke $r3.<StringBuilder: java.lang.String toString()>();

// Target Level 11:
dynamicinvoke "makeConcatWithConstants" <java.lang.String (int)>(i0)
  <java.lang.invoke.StringConcatFactory:
    java.lang.invoke.CallSite makeConcatWithConstants(
      java.lang.invoke.MethodHandles$Lookup,
      java.lang.String, java.lang.invoke.MethodType,
      java.lang.String, java.lang.Object[]
  )>("Amount: \u0001 Pieces");
```

Listing 4.6: String concatenation (Jimple)

Transformation: JNorm transforms the old string concatenation procedure into a template-based concatenation using `invokedynamic`.

N13: Nest-based access control

With the release of Java 11, a new concept for accessing members of inner classes, called nest-based access control [Ros13], was introduced to the language specification. When inner classes are defined within a class, the JDK compiler compiles each inner class into its own separate file. The JVM treats each class as a separate entity and therefore disallows access to private members from methods outside of the class. However, the Java language specification *does* allow such access to private members of inner classes, if they are originating *from the outer class*. Prior to the release of Java 11, such access was handled by the compiler generating public bridge methods in the inner class for each private member. These bridge methods can then be used by the outer class to circumvent calling a private method. Starting from Java version 11, this indirect access via generated bridge methods is not necessary anymore. A new property has been introduced that marks inner classes as *nestmates* of their outer class, which tells the JVM that access to private members is explicitly

allowed between the marked classes. The JVM then automatically puts appropriate access-control checks into place. This change was introduced due to transparency, simplicity and security reasons.

Transformation: If JNORM finds classes that use bridge methods to access private members of their respective inner classes, it transforms them to the nest-based access pattern created when specifying target level 11.

N14: Invocation of private methods

On top of adding nest-based access control, Java 11 introduced a new way to invoke private methods, even within the same class. Prior to Java 11, all private methods were invoked via the `invokespecial` instruction. As of Java 11, to be consistent with the rules of the nest-based access control specification, certain private-method invocations were changed to use `invokevirtual` instructions [Ros13].

Transformation: JNORM transforms `invokevirtual` private-method invocations to `invokespecial` invocations.

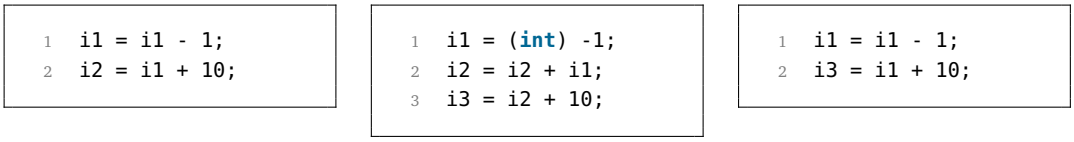
N15: Inner class instantiation

When transitioning from Java 8 to Java 11, the bytecode for instantiating inner classes was modified. In some cases, in Java 8 and earlier, when an inner class is instantiated within the outer class, the JDK compiler generates a spurious empty class. This behavior serves no apparent purpose and was removed as of Java 11.

Transformation: If JNORM finds spurious empty classes, it removes them.

N16: Insertion or removal of typechecks (aggressive transformation)

The bytecode instruction `checkcast` is used to verify the type of an object at runtime. It is commonly generated when a developer performs an explicit type cast, allowing the JVM to check whether the object is compatible with the specified type. However, when changing the JDK version or target level, the compiler's behavior regarding typechecks changes. In contrast to the other compilation difference classes, this difference class cannot be isolated to a single version change, as it happens to different extents at various JDK version or target level changes. In some cases, the compiler inserts `checkcast` instructions even when the developer did not explicitly write a type cast, and conversely, it may omit `checkcast` instructions for casts that



(a) JDK5 (not normalized)

(b) JDK6 (not normalized)

(c) JDK6 (normalized)

Figure 4.3: Application of standardization (Jimple)

are present in the source code. Whether the compiler places a `checkcast` instruction or not often depends on the used compilation environment.

Transformation: By default JNORM does not transform such typechecks, as we were not able to detect a pattern that indicates whether a typecheck should be removed or inserted, by just having access to the bytecode. Still, JNORM offers an *aggressive normalization mode* where it removes all `checkcast` instructions from the normalized Jimple representation of the bytecode. Such transformation removes information that can be used for similarity analysis and possibly changes the application's semantics rather than just adopting a format produced in a different compilation environment. In some cases, e.g. when the change between two bytecode fragments only consists of typecheck insertions or removals, this loss of information makes the normalized fragments indistinguishable.

4.2.5 Standardization

Since the names of local variables are removed by default after compiling Java source code into bytecode (bytecode uses an operand stack instead of local variables), all local variables within the Jimple representation are named by concatenating their inferred type with an ascending integer number. After applying transformations that create, remove, or reorder local variables, such as the Arithmetic or Try-with-resources transformations, the ordering of local variable definitions and their naming scheme might become inconsistent. Because of this, we remove unused local variables and reorder definitions of used local variables based on their usage order, which stays consistent during all optimizations and transformations. Afterwards, we rename the local variables based on their usage order. This ensures a standardized naming scheme across all methods, even after applying transformations.

Figure 4.3 shows why standardization is necessary in some cases. Listing 4.3a shows subtraction generated by the JDK5 compiler, while Listing 4.3b shows subtraction

output by the JDK6 compiler. After applying normalization to the code fragment in Listing 4.3b (see N2: Arithmetic), we obtain the code shown in Listing 4.3c. Since we removed the intermediate variable `i2`, the logical naming following the variable deletion does not match the version that did not require any normalization. Therefore, to ensure consistency between the code fragments, we must standardize and rename all subsequent variable usages.

4.3 Evaluation

In the following we evaluate JNORM's normalization performance. We answer the following research questions.

RQ5: Does the vendor of the JDK compiler influence bytecode generation?

RQ6: How effective is JNORM in normalizing differences across JDK versions?

RQ7: How effective is JNORM in normalizing differences across Java target levels?

RQ8: How prevalent are the individual compilation difference transformations of JNORM?

The first three research questions focus on JNORM's normalization effectiveness within different compilation environments. The final research question gives an overview about the most common compilation difference classes. We used similar experimental setups for each of the research questions.

4.3.1 Experimental Setup

To evaluate JNORM's normalization performance, we use the approach depicted in Figure 4.4.

We selected real-world Java projects based on the following process: Initially, we used the GitHub Search API to identify the 1,000 most-starred projects with Java as their primary language, as of August 2023. We excluded two projects that, alongside Java files, also contained other JVM-based programming languages like Groovy or Clojure, as the compiled classes would interfere with further evaluation steps. Then we filtered out projects not using Maven as build tool, as Maven's static configuration files in XML format allow for an automated change of the compilation setup without knowing the project's build structure in detail. After this step we were left with 322

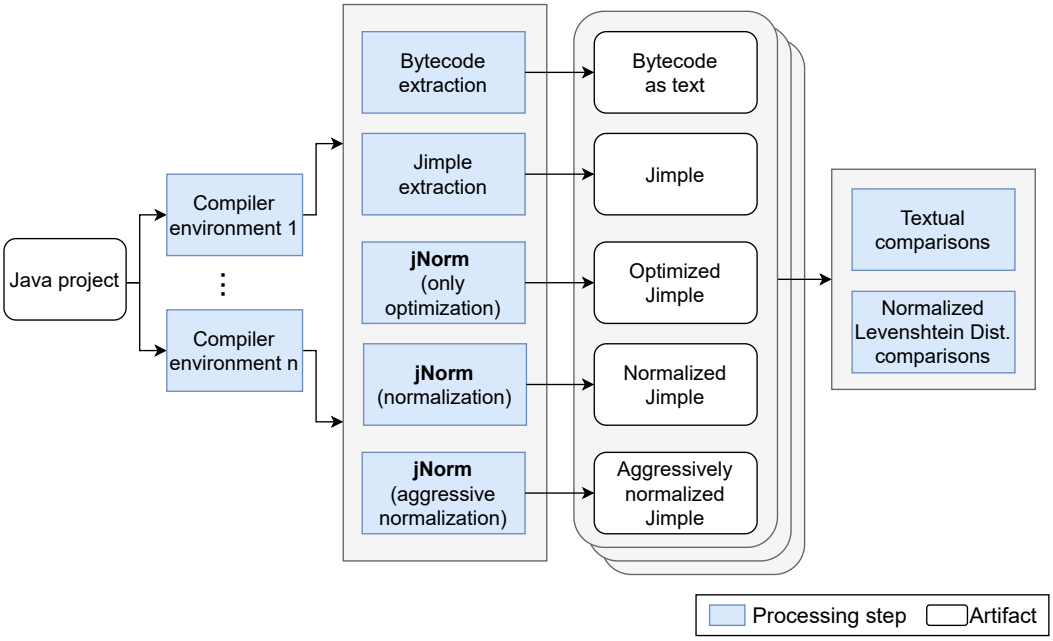


Figure 4.4: Overview of our experimental setup for bytecode normalization

Maven projects. Finally, we excluded nine projects that, when compiled twice within the same compilation environment, would produce different bytecode, because of code generation at compilation time. This is typically due to files being generated for testing purposes or due to parser code being generated from a grammar, which in some cases produces random identifiers. This left us with a set of 313 Java projects, including tutorial projects, popular libraries, frameworks and real-world applications.

We cloned each project's Git repository. As automatic compilation is a known problem for Java projects [Has+17; SP16; Tuf+17], we increased the likelihood of successful compilation in the next step by switching to the latest release tag, if available. As depicted in Figure 4.4, we compiled each of the projects within different compilation environments. We chose the compilation environments based on the setting we were interested in for the respective research questions.

To evaluate the normalization effectiveness of JNORM, we applied different procedures to the compiled projects, as shown in Figure 4.4. The "Bytecode extraction" component in the figure uses ASM 9.3 [Con25] to extract the textual representation of the bytecode from the compiled class files (omitting all debug information). Furthermore, we extracted the plain Jimple representation of the compiled classes in textual form, without applying any optimizations or transformations. The plain bytecode and Jimple representations can be used as a baseline to establish the number of differences induced by different compilation environments. To assess the individual

contributions of JNorm's normalization steps, we let JNorm run in different modes. "JNorm (only optimization)" only applies the Jimple parsing and optimizations described in Section 4.2.3. "JNorm (normalization)" applies all the steps described in Section 4.2.4 with transformations N1–N15, but keeps all typechecks in place (default normalization mode). "JNorm (aggressive normalization)" differs from the previous as it also removes all typechecks from the resulting Jimple representation. Applying all procedures, we obtain five sets of files per compilation environment and project:

- Bytecode as text
- Jimple
- Optimized Jimple
- Normalized Jimple
- Aggressively normalized Jimple

We apply different comparisons to each of the resulting file sets generated within different compilation environments, resulting in multiple comparisons per project. We begin with a textual head-to-head comparison at both the file and method levels. To this end, we compare files with the same fully qualified name and methods with the same signature that were produced in different compilation environments. As soon as there is a single textual difference between the compared files or methods, they are classified as being *different*. If a method is present in one file, but not the other, it is classified as *disjunct*.

In addition to textual head-to-head comparisons, which only allows for a binary classification of equality, we compute the normalized Levenshtein Distance (NLD) [YB07] between the compared files and methods. The NLD is a measure used to calculate the similarity of two text sequences. The Levenshtein Distance counts the number of required character insertions, deletions or substitutions to transform one text sequence into the other. The *normalized* Levenshtein Distance additionally takes the length of the text sequences into account and produces a similarity value between 0% and 100%, with 100% indicating that every single character needs to be changed and 0% indicating that both text sequences are identical.

Table 4.4 shows the JDKs considered in our evaluation. According to a 2024 survey on the state of the Java ecosystem [Rel24], projects using JDK versions 7, 8, 11, and 17 account for over 97% of all Java projects. Furthermore, we considered the three most popular JDK vendors according to the survey, which include Oracle's JDK, Amazon Corretto's (AC) OpenJDK and Eclipse Adoptium's (EA) OpenJDK, in

Table 4.4: JDKs considered in our evaluation of bytecode normalization

JDK Version	Oracle JDK	AC OpenJDK	EA OpenJDK
7	1.7.0_80	-	-
8	1.8.0_333	8.342.07.4	8u352-b08
11	11.0.16	11.0.16.9.1	11.0.17+8
17	17.0.4.1	17.0.5.8.1	17.0.5+8

our evaluation. Note that AC and EA do not distribute OpenJDK versions prior to version 8. Moreover, only a single project within our dataset can be compiled using Oracle's JDK5 and JDK6, thus we do not consider these two no longer supported JDKs in our evaluation [Cor24a].

We executed the compilations on a Debian 10 system, configured to use four cores of an Intel Xeon E5-2695 v3 (2.30 GHz) CPU and 32GB of main memory. We used Maven 3.8.6 for the invocation of builds. Moreover, we executed the normalization via JNORM insides a Docker container running Debian 12 with Amazon Corretto JDK 17 and Maven 3.9.9. Finally, we employed the same experimental setup as in the original study [Sch+24b], but conducted the experiments using a JNORM version ported from Soot to its successor framework SootUp (Snapshot version released as of 2025-07-31). As a result, the exact outcomes differ from those reported previously.

4.3.2 RQ5: Does the vendor of the JDK compiler influence bytecode generation?

Before evaluating the differences introduced by varying JDK or Java versions, we first assess whether different JDK *vendors* produce differences in the generated bytecode. Even though most vendors build upon the same OpenJDK source code, there are still some adjustments regarding e.g. security fixes or performance improvements [Ser25]. This research question aims at determining whether these changes may affect the generated bytecode. To this end, we compiled our full dataset of Java projects using the compilers provided by the JDKs listed in Table 4.4. We then compared all bytecode files generated by the different vendors' compilers—using the same Java and JDK version—to assess any differences.

Table 4.5: Normalization results for different JDK versions. Percentage in brackets indicates the share of files and methods with compilation differences.

	Files	Diffs	Methods	Diffs	NLD	Disj.
JDK7 - JDK8 (28)						
Bytecode	7,793	958 (12.29%)	46,916	97 (0.20%)	4.16%	4
Jimple	7,722	1,917 (24.82%)	43,044	3,065 (7.12%)	1.75%	4
Optimized	7,722	1,924 (24.91%)	43,044	3,084 (7.16%)	1.75%	4
Normalized	7,722	57 (0.73%)	40,370	61 (0.15%)	2.52%	0
Aggressive	7,722	26 (0.33%)	40,370	27 (0.06%)	4.63%	0
JDK8 - JDK11 (90)						
Bytecode	37,991	6,136 (16.15%)	326,398	7,796 (2.38%)	7.81%	1,704
Jimple	37,818	4,587 (12.12%)	303,358	6,190 (2.04%)	4.50%	1,702
Optimized	37,818	4,508 (11.92%)	303,358	6,103 (2.01%)	4.49%	1,702
Normalized	37,818	1,268 (3.35%)	269,050	1,898 (0.70%)	6.18%	8
Aggressive	37,818	321 (0.84%)	269,050	594 (0.22%)	11.29%	8
JDK11 - JDK17 (80)						
Bytecode	40,772	9,088 (22.28%)	356,973	1,901 (0.53%)	20.24%	1,768
Jimple	40,622	3,560 (8.76%)	330,458	4,948 (1.49%)	7.51%	1,760
Optimized	40,622	3,520 (8.66%)	330,458	4,923 (1.48%)	7.58%	1,760
Normalized	40,622	394 (0.09%)	288,829	552 (0.19%)	2.55%	0
Aggressive	40,622	71 (0.17%)	288,829	86 (0.02%)	3.96%	0

We found no differences in the generated bytecode that could be attributed to the JDK vendor. This indicates that the JDK vendor does not influence the output of the JDK's compiler when using the same Java version.

Based on this result, we consider a single JDK vendor (Oracle) in the remaining research questions.

4.3.3 RQ6: How effective is JNorm in normalizing differences across JDK versions?

To investigate JNorm's normalization effectiveness on different JDK versions, we kept all compilation settings at the project's configured default values and only varied the used JDK version within our experimental setup (see Section 4.3.1). For this experiment we considered versions 7, 8, 11, and 17 of Oracle's JDK.

Table 4.5 shows the results of our comparisons. The first column indicates the pairs of JDK versions used to generate the artifacts being compared (see Section 4.3.1). For instance, the first row compares artifacts generated with JDK 7 and JDK 8. The

number inside the parentheses indicates the amount of projects we were able to compile with the respective JDKs. As we were not able to compile every project with all JDKs in our experimental setup, the number of compared projects varies based on the successful builds for each JDK. Note that only a few projects could be compiled using JDK 7, primarily due to the widespread adoption of Java 8 features such as default interface methods, streams, and lambda expressions in modern Java projects. To isolate differences introduced by incremental version increases, we compare a JDK version with the next higher version in our experimental setup. To confirm that we do not miss differences by only comparing incremental version increases, we initially performed a comparison of projects compiled with JDK7 and JDK17 and compared the resulting set to the union of all incremental comparisons. In total we were able to compile ten projects using each version of Oracle's JDK in our experimental setup, comprising 4,621 bytecode files. This analysis showed that the set of differences obtained when comparing JDK7 to JDK17 is equal to the union of the sets of differences obtained when comparing each incremental version increase, i.e., $D_{7\to8} \cup D_{8\to11} \cup D_{11\to17} = D_{7\to17}$ with $D_{i\to j}$ representing the set of files containing compilation differences when comparing bytecode files yielded by the JDK i and JDK j compilers. This comparison holds true for all five processed sets of artifacts (bytecode, plain, optimized, normalized and aggressively normalized Jimple), showing that a comparison of incremental version increases does not miss any compilation differences.

Columns two and three present the aggregated results of the textual comparison at the file level, while the remaining columns display the corresponding results at the method level. Columns "Files" and "Methods" show the total number of files and methods we managed to compile and compare with the respective JDKs. The "Diffs" columns show the total number of files or methods identified as different by the textual head-to-head comparison and their respective shares. The "NLD" column shows the average NLD of methods with detected differences, reflecting the degree of dissimilarity induced by the compilation into individual methods. Note that only methods identified as unequal by the textual comparison are included in the calculation of the NLD. Thus, a higher value does not necessarily indicate a higher degree of difference. The "Disj." (disjunct) column represents the number of methods present within the file generated by one JDK, but not within the file generated by the other JDK, e.g. synthetically generated bridge-methods. Therefore a direct comparison of such methods is not possible. Note that the number of files generated by SootUp is slightly lower than the number of bytecode files. This is due to SootUp combining some information into a single file during the Jimple transformation.

One immediately noticeable observation is the high number of differences in bytecode files at the file level, whereas the share of differences is significantly lower at the method level. A detailed investigation into the differing bytecode files revealed this to be due to the presence of nested class information, which in bytecode is contained inside the class, but outside of methods. Depending on the used JDK, different modifiers are used or the order of these definitions varies. One can also observe that by simply converting the bytecode to Jimple, the number of file-level differences considerably decreases for JDK8–JDK17. Nested class information, in contrast to the bytecode representation, is stored implicitly in the Jimple representation, which causes the disappearance of many dissimilarities. At the method level, the number of differences between bytecode and Jimple remains similar. This indicates that besides removing some information at class level, the plain conversion to Jimple itself does not significantly contribute to the normalization of method-level bytecode. For the comparison of JDK7 to JDK8 the Jimple transformation even *introduces* differences, which are removed later on during the normalization stage. Additionally, it can be seen that the optimization step does not significantly contribute to the normalization by itself either. On the contrary, when looking at the results for the normalized file set, the number of dissimilarities and disjunct methods heavily decreases. The remaining dissimilarities decrease even further when applying an aggressive normalization. Compilation differences at the file level decrease by up to 99%—and by up to 95% at the method level—when comparing plain bytecode with aggressively normalized Jimple, depending on the JDK versions compared.

We investigated the remaining differences in more detail to determine whether other compilation difference classes may have been overlooked. We found that the remaining differences are mostly due to more complex cases of compilation difference classes N4 and N8, which target try-catch blocks. Sometimes when such try-catch blocks are nested in specific ways, JNORM fails to apply the corresponding transformation correctly. Other differences are due to incorrect optimizations applied by the SootUp framework.

Since our initial identification of compilation difference classes was based on a small set of Java projects, and the subsequent evaluation across a large dataset of real-world projects revealed only a few edge cases of already known difference classes not yet handled by JNORM, we believe that our normalization effectively addresses the most common difference classes observed in projects compiled with the investigated JDK versions.

JNORM removes up to 99% of file-level and 95% of method-level differences caused by compiling identical source code with different JDK versions.

4.3.4 RQ7: How effective is JNorm in normalizing differences across Java target levels?

When a Java class is compiled, the target level of the JDK is typically recorded in the class file as a major version identifier [Ora21d]. While in some cases this information can be used to compile the source code to the version specified within the bytecode files, this is not possible when one wants to directly compare two already compiled bytecode files. Thus it is also important to assess JNorm's effectiveness on normalizing differing target levels.

To do so, we fixed the JDK version and adjusted the target level in each project's build configuration, within our experimental setup (see Section 4.3.1). All other build settings have been kept at each project's provided configuration. We consistently used Oracle's JDK11 compiler in our experiment, as it is the recent JDK, which offers backwards compatibility down to target level 6. To adjust the project's target level we scanned each project of our dataset for build files (pom.xml). Inside each of the detected build files, we adjusted the *target*, *release*, or *java.version* properties, which are used to declare the desired Java target level [Fou25g; Fou25f], to compile the project to the desired target levels. To validate that all projects were compiled with the intended target level, we verified the major version indicator [Ora21d] within the resulting bytecode files and removed it subsequently to not interfere with the textual comparison.

Table 4.6 shows the results of our experiment. The structure of the table is identical to that of Table 4.5, except that the first column now represents Java target levels rather than JDK versions. As in the previous experiment, we compare each target level to the next higher one within our experimental setup, to best isolate compilation differences. To ensure that no differences are overlooked by limiting our comparisons to incremental version increases, we also compare target levels with the maximum possible version distance. Again, the following equation holds for the twelve projects (1,245 bytecode files) that can be compiled to each target level within our experimental setup, when using JDK11 $D_{11.6\to11.7} \cup D_{11.7\to11.8} \cup D_{11.8\to11.11} = D_{11.6\to11.11}$ with $D_{11.i\to11.j}$ representing the set of files containing differences when comparing bytecode files yielded by JDK11 set to target levels i and j.

The number within the parentheses inside the first column reflects the number of projects we were able to compile using JDK11 configured with the respective Java target levels. The total number of successful builds is lower than in our previous experiment. This decrease is due to the modified build configuration required for this experiment, which often prevents projects from compiling successfully.

Table 4.6: Normalization results for different target levels of the JDK11 compiler. Percentage in brackets indicates the share of files and methods with compilation differences.

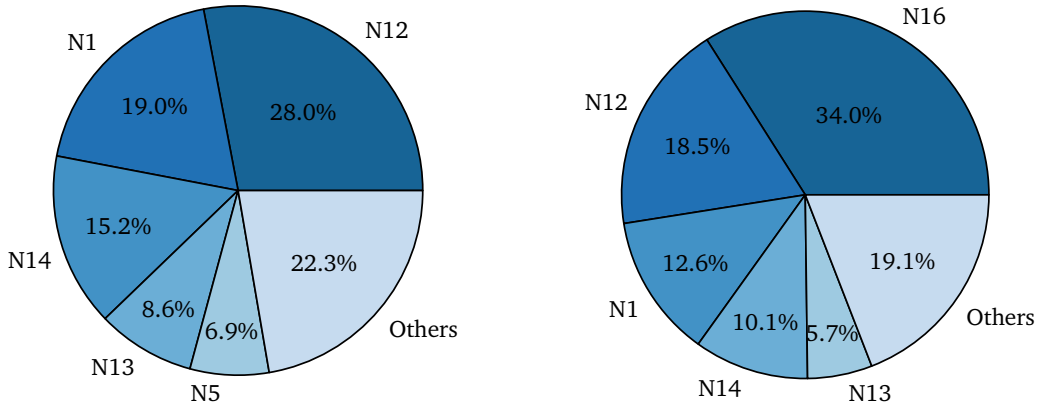
	Files	Diffs	Methods	Diffs	NLD	Disj.
T6 - T7 (23)						
Bytecode	6,051	23 (0.38%)	40,235	47 (0.11%)	2.36%	0
Jimple	5,907	224 (3.79%)	36,427	362 (0.99%)	6.28%	0
Optimized	5,907	228 (3.85%)	36,427	357 (0.98%)	6.02%	0
Normalized	5,907	32 (0.54%)	33,964	43 (0.12%)	1.40%	0
Aggressive	5,907	9 (0.15%)	33,964	19 (0.05%)	0.94%	0
T7 - T8 (30)						
Bytecode	3,833	66 (1.72%)	31,851	85 (0.27%)	1.68%	2
Jimple	3,833	194 (5.06%)	29,752	370 (1.24%)	0.75%	2
Optimized	3,833	209 (5.45%)	29,752	382 (1.28%)	0.78%	2
Normalized	3,833	82 (2.13%)	28,097	128 (0.45%)	1.47%	0
Aggressive	3,833	8 (0.20%)	28,097	8 (0.02%)	2.60%	0
T8 - T11 (75)						
Bytecode	26,117	12,053 (46.15%)	239,722	24,251 (10.11%)	18.30%	3,530
Jimple	26,093	9,557 (36.62%)	222,636	25,569 (11.48%)	15.59%	3,528
Optimized	26,093	9,558 (36.63%)	222,636	25,580 (11.48%)	15.59%	3,528
Normalized	26,093	333 (1.27%)	199,887	476 (0.23%)	2.30%	0
Aggressive	26,093	141 (0.54%)	199,887	163 (0.08%)	1.65%	0

Target levels 6, 7, and 8 exhibit notably few compilation differences, as shown in Table 4.6. Many of these are removed by the (aggressive) normalization. We investigated the remaining differing files and discovered that all remaining differences are due to wrong type assignment performed by SootUp. The situation changes when moving from target level 8 to 11, as visible in the third row of Table 4.6. Almost half of the compared files show differences in a textual head-to-head comparison. At the method level, this value drops to approximately 10%. Furthermore, the NLD is very high, indicating that the compiled methods are significantly dissimilar. This large number of differences is due to multiple highly used features being affected by the target level increase. From Java target level 8 to 11, the handling of string concatenation, private method calls, and inner classes has changed. These are features that are frequently used within Java projects. The conversion to Jimple does not significantly impact the number of differences, in most cases even adding to the number of differences. The subsequent optimization also does not remove any differences in the textual comparison. The normalization, however, significantly decreases the number of dissimilarities. After aggressive normalization, the number of dissimilarities decreases by more than 98.8% when comparing plain bytecode to aggressively normalized Jimple. At the method level, the dissimilarity decreases even further, by 99.3%.

Again we performed a manual inspection of the remaining differing files and methods to uncover possibly overlooked compilation difference classes. The inspection showed that the remaining differences can mostly be attributed to incorrect optimizations applied by SootUp and incorrect transformations of dynamic string concatenation and nest-based access control applied by JNORM in specific scenarios (e.g. boolean variables being handled as integer values in the string concatenation).

As stated in the previous research questions, the lack of newly uncovered compilation difference classes suggests that we have identified the most common compilation difference classes for projects compiled to the investigated Java target levels. Furthermore, we investigated compilation environments involving changes to both the JDK version *and* the target level, however, we did not uncover any additional difference classes.

JNORM removes up to 98.8% of file-level and 99.3% of method-level differences caused by compiling identical source code with different configured Java target levels.



(a) Normalization **(b)** Aggressive Normalization

Figure 4.5: Average prevalence of the individual compilation difference classes

4.3.5 RQ8: How prevalent are the individual compilation difference transformations of JNorm?

To investigate the prevalence of the compilation difference classes and their individual contribution to the normalization, we tracked each applied transformation within the normalization process of our dataset used throughout RQ6 and RQ7. We counted each transformation once per methods it was applied to.

Figure 4.5 shows the average number of transformations across each JDK version and target level setting. JNorm applies an average of 407 transformations per project during the normalization process. It can be observed that only few compilation difference classes account for the majority of the transformations. Transformation N12, which normalizes string concatenations, represents 28% of all transformations applied in plain normalization mode. This is due to the large number of string concatenations used in Java projects. Transformation N12 is followed by transformations N1, which removes synthetic methods, and transformation N14, which handles the invocation of private methods. Both of these transformations are frequently applied during normalization. The remaining transformations are needed infrequently.

In aggressive normalization, transformation N16, which removes all typechecks, accounts for the largest share of transformations, at 34%. The reason is that the compiler frequently inserts typechecks within the bytecode. Our aggressive approach of eliminating *all* typecheck occurrences further enhances this effect. Since JNorm applies transformation N16 only after all other transformations, the distribution of the other compilation difference classes remains the same as in plain

normalization. On average, JNORM applies 616 transformation per project during aggressive normalization.

> Transformations N16 (insertion or removal of typechecks), N12 (dynamic string concatenation), N1 (synthetically generated methods), and N14 (invocation of private methods) account for 75.2% of all applied transformations, making them the most prevalent during normalization.

4.4 Threats to Validity

For our evaluation of JNORM we exclusively relied on projects that use Maven as build tool. While other Java build tools such as Gradle [Inc25c] exist, Maven is the most popular [Jet23]. Furthermore, we limited our experiments to Java versions 5–8, 11 and 17. Although this version range includes all LTS versions up to August 2023 and represents the most widely used Java versions in projects (see Section 4.1), other versions may produce different evaluation results and introduce unidentified compilation differences. While our large-scale evaluation included a substantial number of Java projects, there may still be rare cases of compilation-induced differences that were not uncovered within our dataset. Moreover, we tried to isolate the detected compilation difference classes to the specific configuration change they are caused by. There may also be other configuration changes that cause the same differences. However, since JNORM is unaware of the specific configuration used, it will normalize the differences regardless. Even though JNORM, in its default mode, only transforms constructs from one compilation environment output to that of another environment, the applied transformations may still introduce semantic changes. Furthermore, in few cases there are incorrect analyses, transformations and optimizations applied by SootUp before the application of JNORM's transformations. Finally, our analysis of the most commonly used compilers and target levels in real-world Java projects is limited to projects that specify their configuration through the Maven Compiler Plugin. Other methods of declaring compiler usage in Maven projects, such as creating a custom compiler plugin that extends the Maven Compiler Plugin, were not considered.

4.5 Related Work

A wide range of similarity analysis techniques have been developed for source code, bytecode, and binary code, each employing its own normalization strategies.

Bytecode level: Only few approaches have been developed for bytecode similarity analysis. However, there are various scenarios in which Java source code is not available. Whenever this is the case, the comparison has to be performed on the bytecode. SeByte [KRR14] is a similarity detector targeting Java bytecode. It divides the bytecode into tokens and separates them based on their types to employ the Jaccard similarity measure for matching. Baker and Manber [BM98] leverage a combination of the similarity comparison tools Diff, Siff and Dup to determine the degree of similarity of Java bytecode files. Yu et al. [Yu+19] use the Smith-Waterman algorithm to determine the similarity of two bytecode snippets. They extract instruction and method-call sequences from the bytecode and apply the Smith-Waterman algorithm to align the extracted sequences. Ji et al. [JWC08] propose an approach to perform a plagiarism detection on bytecode. They divide the bytecode into sequences and utilize the adaptive local alignment to find potential plagiarisms. Davis and Godfrey [DG10] propose an approach to find clones that works on Assembler and bytecode. Their approach implements a greedy matching of instruction types and arguments by using an internal weight measure. Chen et al. [CLZ14] present an approach that aims at detecting application clones on Android markets. They utilize control flow graphs to compare apps to each other and find clones in the Dalvik bytecode.

These approaches do not explicitly clarify how they handle bytecode differences introduced by different compilation environments.

Source code level: For similarity analysis on source code level many approaches have been developed. NiCad [RC08] is a textual based code clone detector that targets a variety of programming languages. It uses different means of normalization and is designed to be easily extensible. CCFinder [KKI02] transforms the input source code into a set of tokens and performs the comparisons on this set of tokens. SourcererCC [Saj+16] uses a similar token-based detection approach. However, SourcererCC specifically aims at high scalability and is optimized towards large software repositories. JPlag [PMP+02] divides the source code into token strings and applies a greedy string tiling algorithm to find plagiarisms within sets of applications. DECKARD [Jia+07] leverages the Abstract Syntax Tree representation of an application's source code to perform the similarity analysis. StoneDetector [SAH20] utilizes dominator trees, a compiler-level representation, to identify structural clones

that differ in syntax but share identical control flow structures. DeepSim [ZH18] uses a deep learning model to find semantic similarities within code snippets that are syntactically different. Oreo [Sai+18] is another code clone detection tool that leverages deep learning. It uses a pre-trained model that utilizes several code metrics to decide whether two code snippets are clones of each other, even if their syntactical similarity is below 70%.

Source code based similarity analysis approaches have become much more permissive to syntactic differences over the years. This allows some tools to perform a similarity analysis across intermediate representations that are syntactically similar to the targeted source code. Selim et al. [SFZ10] investigated the potential benefits of supplementing Java source code with the Jimple intermediate representation for code clone detection. To evaluate this approach, they applied the clone detection tools CCFinder and Simian to Jimple code, which is syntactically similar to Java source code.

Ragkhitwetsagul et al. [RKC18] evaluate and compare 30 different code similarity detection techniques, including code clone detectors, plagiarism detectors and compression tools, within different similarity analysis scenarios.

Binary level: Binary similarity analysis is a complex task due to the low-level nature of machine code and the significant variability in compilation environments and optimization levels. David et al. [DPY16; DPY17; DPY18] propose multiple approaches that decompose the assembly code of the binary into strands, which encode specific semantic behaviors in small units. Similar to JNORM and SootDiff, their approach first transforms units into LLVM-IR to obtain a more uniform representation, followed by LLVM-IR-specific optimizations and transformations before comparison. Luo et al. [Luo+14] model the semantics of binaries with a set of symbolic formulas that represent input-output relations and use a theorem solver to determine their similarity. Hemel et al. [Hem+11] created the Binary Analysis Tool which uses different comparison strategies, like string matching, compression and a binary delta check to find software license violations within binaries. Many approaches like SAFE [Mas+19] and Xu et al.'s approach [Xu+17] use machine learning to determine the similarity of binaries. Marcelli et al. [Mar+22] investigate and compare various machine learning based approaches that try to classify the similarity of binaries. Haq and Caballero [HC21] present a comprehensive survey of binary code similarity analysis, in which they systematically categorize and evaluate 70 different approaches developed since 1999.

While most binary similarity analysis techniques are not directly applicable to bytecode similarity analysis, they can, in theory, be adapted for that purpose.

Compiler influence: Some studies have explored the relationship between compilers and similarity analysis. Kononenko et al. [KZG14] investigate a compilation's degree of influence on code clone detection. By comparing clones detected in Java source code and its compiled bytecode, they observe differing sets of results. Ragkhitwetsagul and Krinke [RK17] investigate how compilation and decompilation influences the clone detection performance on Java code bases. They suggest that decompilation can aid as a complementary measure to source code based clone detection, but is not sufficient on its own. Dann et al. [DHB19] investigate the impact of different compilation environments on the resulting bytecode. They propose the bytecode comparison tool SootDiff, which adopts an approach similar to JNORM. However, it supports only one of the transformations defined in our work (string constant concatenation) and is limited to Java versions 5 through 8. Xiong et al [Xio+22] investigate sources of non-determinism in the Java build process that hinder builds from being reproducible. They uncover 14 patterns that may introduce non-equivalences in the build and present corresponding mitigation strategies. In the context of JNORM many of these sources of non-determinism are addressed by the conversion of bytecode to Jimple.

4.6 Conclusion

In this chapter we presented the concept of bytecode normalization for code similarity analysis. Bytecode normalization addresses the problem of comparing the bytecode of Java applications compiled in different compilation environments. This is particularly important for bytecode-centric dependency analysis, as the source code of included dependencies is often inaccessible, necessitating similarity analysis directly at the bytecode level. By converting bytecode into the intermediate representation Jimple, applying common optimizations, transforming remaining compilation differences and applying naming standardization, we create a representation that is always identical, no matter the JDK and Java (LTS) version used to compile the source code. To this end, we identified and presented 16 compilation difference classes that are introduced in the bytecode as a result of using different JDK and Java versions.

Based on the concept of bytecode normalization we implemented JNORM. Our evaluation on a large set of popular real-world Java projects revealed that compiling identical Java source code in different compilation environments can result in up to 46% of the generated bytecode files differing, even for single incremental version

changes. Using JNORM, we were able to reduce compilation-induced differences by as much as 99% across most of the investigated compilation environments.

JNORM produces a code representation that is independent of the compilation environment, thereby reducing the complexity required for subsequent similarity analysis. This abstraction is essential for enabling accurate and efficient dependency analysis directly at the bytecode level.

Since its inception, JNORM has been used in various contexts related to *alternative and reproducible builds*, which aim to determine whether two independently produced binaries truly originate from the same source code. This is an increasingly important concern in software supply chain security. These applications further underscore the versatility of JNORM and its suitability for domains beyond bytecode-centric dependency scanning.

5 Bytecode-centric Dependency Scanning

In recent years numerous open-source [OWA24; Inc24c; Rer24; PPS20] and commercial dependency scanners [Lim24; Inc24a; Men24; Lab24] have been developed to reduce the risks associated with OSS usage by identifying known-to-be-vulnerable dependencies. These dependency scanners employ different approaches and leverage various types of information, such as metadata associated with dependency inclusions, to detect known-to-be-vulnerable dependencies in software projects. As previously discussed in Chapter 1, various types of modifications are commonly applied to Java dependency inclusions. In these scenarios, state-of-the-art dependency scanners, no matter the underlying approach, are impeded in their ability to identify known-to-be-vulnerable dependencies in Java projects.

To combat this issue, we propose JARALYZER, a *bytecode-centric* dependency scanning approach. In contrast to other dependency scanning approaches, bytecode-centric dependency scanning relies entirely on bytecode, the only consistently available information regardless of modifications, enabling it to detect the inclusion of known-to-be-vulnerable dependencies, even when they have been altered. This is made possible by meeting the requirements specified in Section 1.1. As an extension of the code-centric approach proposed by Ponta et al. [PPS20], bytecode-centric dependency scanning bases its analysis on fix commits available in OSS projects' source code repositories. However, unlike the code-centric approach, it does not require the fix commits and the scanned dependencies to exist within the same code representation. This is enabled by first compiling the fix commits by employing our targeted compilation approach presented in Chapter 3, followed by applying our bytecode normalization technique presented in Chapter 4 to both the compiled commits and the scanned dependencies, which facilitates a subsequent comparison. Moreover, JARALYZER employs a code property graph–based comparison technique that goes beyond syntax by also analyzing control and data flow. This enables comparatively precise identification of whether known-to-be-vulnerable dependencies are present in the analyzed software project and whether corresponding fixes have been applied, even when the changes are minimal. Importantly, the CPG-based

approach does not rely on fully qualified construct names, thereby enabling the detection of re-packaged (type 4) known-to-be-vulnerable dependencies.

Our evaluation of JARALYZER shows that, when it comes to detecting known-to-be-vulnerable dependencies included in modified form, it outperforms five other popular dependency scanners, four OSS and one commercial. With a miss rate of at most 6% for vulnerabilities in re-packaged dependencies compared to unmodified ones, it is the only scanner capable of handling all types of modifications identified by Dann et al [Dan+21]. But even when applied to unmodified dependencies, when directly compared to the state-of-the-art code-centric dependency scanner Eclipse Steady, JARALYZER reported 28 more true vulnerabilities and 29 fewer false warnings.

The remainder of this chapter is structured as follows: In Section 5.1 we describe the concept and details behind bytecode-centric dependency scanning via JARALYZER. Then, in Section 5.2 we evaluate bytecode-centric dependency scanning by comparing JARALYZER to five state-of-the-art dependency scanners on modified and unmodified dependency inclusions. We give an overview of potential threats to the validity of our experiments in Section 5.3. Afterwards we present related work in Section 5.4 before we conclude in Section 5.5.

5.1 Bytecode-centric Dependency Scanning with JARALYZER

Figure 5.1 shows an overview of JARALYZER's architecture. The approach consists of two major stages: *(1)* the knowledge-base creation and the *(2)* dependency scanning. The constructed knowledge base serves as the foundation for the dependency scanning. It contains all vulnerabilities that JARALYZER can detect.

As an extension of the code-centric approach, JARALYZER uses fix commits as input, which are commits fixing vulnerabilities in the source code repositories of OSS. In the first step of the knowledge-base creation stage, JARALYZER compiles the fix commits to bytecode using two techniques: First, it applies the custom compilation heuristic we developed for targeted compilation of commit changes (see Section 3.3) to compile the code modifications introduced in the fix commits and obtain their corresponding bytecode representation. After compilation of the fix commits, JARALYZER normalizes the resulting bytecode using JNORM (see Chapter 4) to enable a subsequent comparison. This step is necessary to remove differences that exist solely

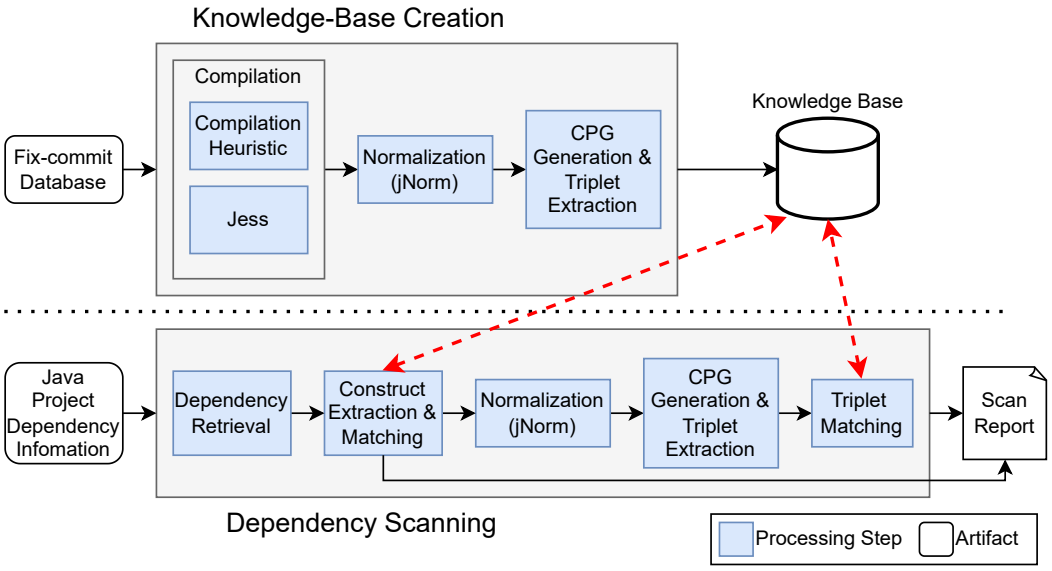


Figure 5.1: Overview of JARALYZER

due to the use of different compilers. If not removed, those differences would interfere with matching the applied fix in the dependency scanning stage. Afterwards, for each changed method in a given fix commit, JARALYZER generates a code property graph (CPG) [Yam+14], a representation that combines syntax, control flow, and data dependency information of a method (see Section 2.7). Finally, JARALYZER extracts comparable string triplets [BH20], which encode edge information, from the generated CPGs and stores them in the knowledge base. JARALYZER uses these triplets to identify the presence of the fix in the next stage.

To scan a software project for known-to-be-vulnerable dependencies, JARALYZER performs the steps described in the dependency scanning stage (see Figure 5.1). First, JARALYZER uses the underlying build-automation tool of the project being scanned to extract all its dependencies. JARALYZER then scans each of the extracted JAR files individually for potential vulnerabilities. To do so, JARALYZER matches all *constructs* (class, interface or method) within the JAR file to its knowledge base. If JARALYZER identifies a matching construct, it indicates that an included dependency in the software project may be vulnerable. However, at this point, it remains unclear whether the version in use has already applied the fix. To determine the presence of the fix, all constructs added, removed or changed in the fix are considered. In particular, if the scanned construct is a method *changed* in the fix—which is the most common scenario—JARALYZER normalizes it, generates a CPG for it and transforms it into comparable string triplets. Finally, JARALYZER compares the triplets yielded from the scanned method's CPG to the triplets stored within its knowledge base. Based on the degree of matched triplets, JARALYZER reports the method as fixed or

vulnerable. This procedure is performed for each construct in each JAR file, resulting in a final report that contains all identified vulnerable dependencies.

In the following we will describe each processing step in detail.

5.1.1 Knowledge-Base Creation

The knowledge-base creation stage is essential to bytecode-centric dependency scanning. In contrast to scanners relying on metadata, JARALYZER requires the bytecode corresponding to vulnerability fixing commits. This stage only needs to be executed once to create JARALYZER's knowledge base, which is then available for every dependency scan. After construction, for every CVE entry, the knowledge base contains the *fully qualified name* (FQN) of each construct modified in the fix commits related to that CVE. Each CVE entry contains at least one modified construct. The FQN uniquely identifies a construct by specifying its full package and class name. Additionally, the knowledge base contains each construct's modification type (added, removed, or changed), and the precise bytecode additions, removals and changes, extracted from the CPG as string triplets.

Fix-commit Database

As an extension of the code-centric approach, JARALYZER relies on a fix-commit database as input to create its knowledge base. A fix-commit database maps CVE entries to the commits in an OSS project's source code repository that fix the respective CVE entries. Possible fix-commit database options include SAP's Project KB [Pon+19], CVEFixes [BNM21] or MoreFixes [Akh+24].

Compilation

As dependencies are included in bytecode format but fix commits are in source code, to enable a comparison JARALYZER needs to generate bytecode for each construct modified by the fix commits of each CVE. However, simply triggering a standard compilation with the configured build-automation tool is insufficient, as it frequently leads to failure [Tuf+17; Has+17; SP16; Zha+19]. As shown in RQ3 (see Section 3.4.3), performing a standard compilation following the approach of Tufano et al. [Tuf+17] successfully compiles only 14.9% of CVE entries in the Project KB fix-commit database. However, for bytecode-centric dependency scanning

one does not need to compile the entire code base, but only the source code files modified within the respective fix commit. Leveraging this fact through the custom compilation heuristic presented in Section 3.3 significantly improves the compilation success rate to 90% of CVE entries in Project KB.

Normalization

Obtaining the bytecode of fix commits alone is not sufficient for detecting the presence of vulnerable bytecode in included dependencies. Since we do not know what compilation environment the original dependencies have been compiled in, and different Java compilation environments produce different bytecode for equal source code, as we have shown in RQ6 and RQ7 (see Sections 4.3.3 an 4.3.4), the bytecode produced by JARALYZER may differ from the one included in the scanned dependencies. Since vulnerability fixes are often small, involving only a few lines of code, compilation differences may well mask the original fix and make a comparison impossible. To make the bytecode comparable, we apply JNORM (see Chapter 4) right after compilation. JNORM normalizes the bytecode and removes nearly all differences introduced by different compilation environments. After applying JNORM, compilation differences are mostly removed and the bytecode can be compared.

CPG Generation & Triplet Extraction

After normalization, JARALYZER uses the SOOTUP [Kar+24] static analysis framework to generate code property graphs (CPG) (see Section 2.7) for each method changed in the fix commits. To extract the precise code instructions that comprise the fix, one must compute the differences between the CPG of the method before the fix was applied and the CPG of the method after the fix was applied. This requires determining subgraph isomorphisms—a problem that is NP-complete—which makes a direct comparison of CPGs computationally expensive [BH20]. We thus resort to an approximation proposed by Bowman and Huang [BH20] and extract graph *triplets*. For every edge e within a given CPG g, a triplet t is defined as (n_s, e_l, n_t) where e_l denotes the label of edge e, n_s denotes the source node label and n_t the target node label of edge e within graph g. The set T is the set of all triplets for a given CPG g. Once we have created a CPG g_{vul} and a CPG g_{fix} for a method before and after the fix has been applied we extract the respective triplet sets T_{vul} and

T_{fix}. We then define the *context triplets* CT, *positive triplets* PT, and *negative triplets* NT:

$$CT = T_{vul} \cap T_{fix} \qquad PT = T_{fix} \setminus T_{vul} \qquad NT = T_{vul} \setminus T_{fix}$$

CT contains all triplets that have not been changed by the fix, PT contains triplets that have been added by the fix and NT contains triplets that have been removed by the fix. Thus, PT and NT contain the fine-grained code changes performed within the fix. Finally, we store the generated triplet sets for each method in JARALYZER's knowledge base, alongside the information about the FQN of the method and the affecting CVE entry.

5.1.2 Dependency Scanning

In the dependency scanning stage, JARALYZER is applied to a Java software project containing dependency information and tries to identify whether known-to-be-vulnerable dependencies are included.

Dependency Retrieval

JARALYZER starts with retrieving all dependencies included in the scanned project. Depending on the used build-automation tool, it invokes a command of the tool (e.g. the `copy-dependencies` goal of the maven-dependency-plugin [Fou24b]) to retrieve the JAR files corresponding to each dependency as defined in the project's dependency information (e.g. `pom.xml` for Maven projects).

Construct Extraction & Matching

In this step, JARALYZER individually processes each previously retrieved dependency JAR file, first extracting the FQN of every construct within it. Then, JARALYZER queries the knowledge base with each extracted FQN to verify if any of these FQNs is associated with a vulnerability. This query results in a set of CVE entries potentially affecting the JAR file. JARALYZER proceeds to scan the JAR file for each potential CVE entry individually. Here it distinguishes between constructs removed, added or changed by the fix: *(1)* If, within the scanned JAR file, a construct is detected that is marked as *removed* by the fix of the respective CVE entry, JARALYZER classifies the corresponding construct as vulnerable, otherwise as fixed. *(2)* If a method is marked

as *added* by the fix but is missing from the scanned JAR file while its declaring class is present, JARALYZER classifies the construct as vulnerable, otherwise as fixed. *(3)* If JARALYZER detects a method in the scanned JAR file that has been *changed* by the fix, it extracts the method body and proceeds with normalization (see Section 5.1.1) and CPG generation (see Section 5.1.1), following the same process used in the knowledge-base creation stage.

Triplet Matching

After generating a CPG g_m and extracting the set of triplets T_m for the matched and normalized method (see Section 5.1.1), JARALYZER performs a triplet matching to determine whether the method is in a vulnerable or fixed state. To do so, it compares the triplet set T_m to the set of positive triplets PT and negative triplets NT of the matched method within the knowledge base. If the equation $|NT \cap T_m| \geq |PT \cap T_m|$ holds, it indicates that the matched method is more or equally similar to the method *before* the fix has been applied. JARALYZER thus classifies the matched method as vulnerable. If the fix did not remove or change any code, but just added code within a method, the set of negative triplets NT is always empty. In such cases, JARALYZER verifies, whether $|PT \cap T_m| \geq \theta_{PT}$, for a configurable threshold θ_{PT}.

In the end, JARALYZER counts these matched constructs that are associated with a specific vulnerability and have been classified as vulnerable. If the number of constructs classified as vulnerable is greater than or equal to those classified as fixed, JARALYZER reports the included JAR file as vulnerable to this specific CVE entry. Especially when the initial fix is changed over time, it might be the case that some constructs are reverted to their pre-fix state and JARALYZER thus classifies some constructs as fixed and others as vulnerable. In cases involving discrepancies, tools such as Eclipse Steady require human intervention to reach a decision. In contrast, because JARALYZER is designed to minimize human involvement, we decided on this majority criterion, which yielded the best performance in empirical testing. Finally, JARALYZER continues to process the next CVE entry potentially affecting the JAR file.

After scanning all provided JAR files, JARALYZER generates a final scan report that lists all discovered vulnerable dependencies along with their associated CVE entries.

5.1.3 Re-Packaging Detection

JARALYZER has been designed to not require any metadata about included dependencies within its dependency scanning stage. This means that, by construction, it does not impact JARALYZER's detection capabilities whether the included dependencies are re-compiled, re-bundled or their associated metadata is removed (type 1–3 modifications). As long as the bytecode is available, JARALYZER will be able to scan them for known-to-be-vulnerable OSS. However, to uniquely identify constructs and to initially match them to its knowledge base, JARALYZER does rely on their FQNs. The FQNs of constructs, however, change when the dependency is re-packaged (type 4 modification, see Section 2.2). Listings 5.1 and 5.2 show an example of such a re-packaging[1], which is typically applied to bytecode using tools like the Maven Shade Plugin [Die+24]. Listing 5.1 shows the original class, while Listing 5.2 shows the same class, but re-packaged. The comments in lines 4 and 7, above class `C` and method `foo`, show their respective FQNs. Differences between both listings are highlighted. Due to re-packaging, which prepends the string "r" in front of each package name within the dependency, the FQNs of all constructs change and will never match with the knowledge base in case a vulnerability is associated with method `foo` in Listing 5.1. To cover type 4 modifications, JARALYZER uses a *re-packaging detection* mode that does not rely on FQNs for matching. The re-packaging detection follows the same process shown in Figure 5.1, yet instead of relying on FQNs during the construct-matching step, it uses the *unqualified* names of constructs (class names *without* package names). E.g., instead of querying the knowledge base for the fully qualified method signature `r.a.C: foo(r.a.b.X)` (see Listing 5.2), JARALYZER will query for all methods in the knowledge base having the *unqualified* signature `C: foo(X)`. As the unqualified signature does no longer uniquely identify methods, JARALYZER additionally checks for the *class context* to avoid spurious matches: it checks whether all sibling methods and fields of the scanned method are also present in the method matched in the knowledge base. The information about the class context has also been collected during the knowledge-base creation stage. As an example, if the method name `foo` from Listing 5.2 matches the knowledge base, the class context will consider the sibling field `baz` (line 6 in Listing 5.2). If the class context exceeds a configurable threshold θ_{CC}, JARALYZER considers the method as a correct match.

Finally, the triplet matching step also slightly differs from JARALYZER's default process. As in Java bytecode type names are always fully resolved, JARALYZER has to unqualify all triplets. E.g., consider the call of method `bar` in line 9 of Listing 5.2. Within

[1] Re-packagings are applied directly to bytecode. We show it on source code for illustration purposes.

```java
package a;
import a.b.X;

// a.C
class C {
  int baz;
  // a.C: foo(a.b.X)
  void foo(X x) {
    int i = x.bar();
  }
}
```

Listing 5.1: Original class

```java
package r.a;
import r.a.b.X;

// r.a.C
class C {
  int baz;
  // r.a.C: foo(r.a.b.X)
  void foo(X x) {
    int i = x.bar();
  }
}
```

Listing 5.2: Re-packaged class

the bytecode, and thus within its corresponding triplet, the method call will be expressed with the fully qualified name of `X` (i.e., `r.a.b.X`). Thus, before matching, we unqualify all triplets within the scanned JAR and the knowledge base. To avoid false-positive matches, before comparing the triplet set of the scanned method T_m to the set of positive and negative triplets, we verify that the equation $|CT \cap T_m| > \theta_{CT}$ holds for a configurable threshold θ_{CT}. Doing this, we verify whether the part of the method that has *not* been changed by the fix, expressed by the set of context triplets CT, is sufficiently similar to the scanned method.

These adjustments however, come with the cost of potentially missing vulnerable dependencies or erroneously reporting a dependency as vulnerable, often depending on the choice of the configurable thresholds. Thus, depending on the use case, the default mode and the re-packaging detection mode can be run in tandem or independent of each other. When only running the default mode, one will not identify any re-packaged known-to-be-vulnerable dependencies, while only running the re-packaging detection mode, one might receive more false-positive vulnerability reports.

Although re-packaging detection could be added as a step within JARALYZER's main pipeline, we added it as a parallel mode to avoid repeated, costly executions of the full pipeline, including normalization and CPG-generation, caused by ambiguous method signatures. By providing two independent modes, developers can either run both or choose the one that best suits their specific needs.

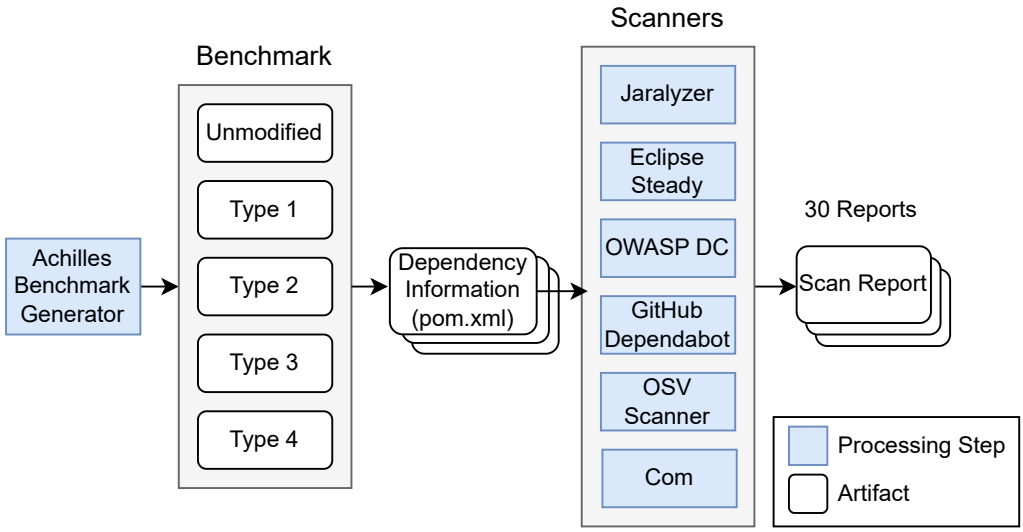


Figure 5.2: Overview of our experimental setup for bytecode-centric dependency scanning

5.2 Evaluation

In the following we evaluate the effectiveness of JARALYZER. To do so we answer the following research questions.

RQ9: How does JARALYZER compare to state-of-the-art dependency scanners in identifying *modified* known-to-be-vulnerable dependencies?

RQ10: How does JARALYZER compare to the state-of-the-art code-centric dependency scanner in identifying *unmodified* known-to-be-vulnerable dependencies?

RQ11: How runtime-efficient is JARALYZER?

The first research question focuses on the challenge that current dependency scanners face, when dependencies are included in modified form. The second research question evaluates JARALYZER's detection performance in the general case, when dependencies are not modified. To do so, we compare JARALYZER against the state-of-the-art code-centric dependency scanner Eclipse Steady. The third research question evaluates JARALYZER's runtime efficiency.

5.2.1 Experimental Setup

Figure 5.2 shows an overview of our experimental setup. The evaluation is based on the Achilles benchmark generator created by Dann et al. [Dan+21]. Achilles comes with a dataset of 534 distinct OSS artifacts and allows for an automatic creation of Maven projects having dependencies on a selected subset of these artifacts, either in unmodified or modified form (type 1–4). Most of the provided artifacts are different versions of the same OSS. We only included the latest provided version[2] of each distinct OSS (unique groupId and artifactId), as including multiple versions of the same OSS within a single project would lead to version conflicts. Using the Achilles benchmark generator we created five distinct Maven projects, each having 56 artifacts as dependencies, including popular OSS such as Guava, Spring and Jackson-Databind. According to Endor Labs [PM24], these are among the Java OSS accounting for most vulnerability findings. We kept the 56 artifacts together in a single project, as splitting them should not impact tool behavior and would require introducing arbitrary splitting criteria. Based on the definition presented in Section 2.1, we create five distinct projects $P = (C, D)$, with C being empty as no application code is provided and D containing the 56 artifacts from Achilles as dependencies. While one of the created projects contained the dependencies in unmodified form, the other four projects contained *modified* dependencies according to types 1–4 (see Section 2.2). We then provided the dependency information (`pom.xml` files) belonging to each of the generated projects to six different dependency scanners: JARALYZER, Eclipse Steady [PPS20], OWASP DependencyCheck [OWA24], GitHub Dependabot [Inc24b], OSV Scanner [Inc24c] and a commercial dependency scanner (Com). Applying each tool to the five different projects within our benchmark, we obtained a total of 30 scan reports containing the CVE entries that each tool detected for the individual projects.

As JARALYZER's knowledge base, we use SAP's Project KB [Pon+19], as it is also the knowledge base utilized by Eclipse Steady. It contains a total of 1,297 manually-curated CVEs from 2005–2023, affecting popular and industry-relevant Java and Python OSS. We filtered out all entries that do not affect Java OSS or include no changes to Java source code files from Project KB. This left us with 728 relevant entries. As reported in RQ4 (see Section 3.4.4), using the compilation heuristic presented in Section 3.3, we were able to compile 655/728 (90%) entries of Project KB for Java.

[2]For OSS comprised of multiple modules, e.g. Spring and Jetty, we did not include the latest version provided in Achilles but the version with the most compatible modules

All dependency scanners were executed within a Docker container (v26.1.4) running Alpine Linux v3.19 with Eclipse Temurin JDK 17. The system running the container, was configured to use four cores of an Intel Xeon E5-2695 v3 (2.3 GHz) CPU and 32GB of RAM. Furthermore, we used OWASP DependencyCheck v11.1.0, OSV Scanner v1.4.3 and Eclipse Steady v3.2.5. GitHub Dependabot was executed in December 2024. We did not execute Eclipse Steady's reachability analysis, as our evaluation focuses on comparing detection performance. Eclipse Steady classifies certain reported vulnerabilities as "unknown", requiring manual analysis (57 in the unmodified benchmark, all of them in the modified benchmarks). In fact, for 4 out of the 56 artifacts in the unmodified benchmark, the source code is not available on Maven Central and for the modified benchmarks, no source code is available. We count these cases as reported vulnerabilities. It is worth noting that, even when source code is available, there is no guarantee that it is accurate: Maven Central does not verify if the source code matches the actual bytecode and even allows uploading placeholder files that contain no source code at all [Son25]. Another interesting observation we made is that Eclipse Steady performs significantly worse when Achilles packages the artifacts on Windows compared to Linux, so we let Achilles package all artifacts on a Linux machine. Finally, we executed JARALYZER in default and re-packaging detection mode and configured the (default) threshold values:

$$\theta_{PT} = 0.5 \qquad \theta_{CC} = 0.3 \qquad \theta_{CT} = 0.3$$

Through empirical testing, we found these values to provide the best trade off between detecting re-packaged known-to-be-vulnerable dependencies and minimizing wrong alerts.

5.2.2 RQ9: How does JARALYZER compare to state-of-the-art dependency scanners in identifying *modified* known-to-be-vulnerable dependencies?

In this research question we evaluate JARALYZER's performance on *modified* dependencies, which pose a major challenge for current dependency scanners [Dan+21; Die+24]. To measure the effectiveness of dependency scanners on modified dependencies, Dann et al. used a small benchmark of seven distinct artifacts created with the Achilles benchmark generator, applying the same modifications as we did in our experimental setup (see Section 5.2.1). To determine whether each tool

Table 5.1: Detected CVE entries by each scanner in comparison to JARALYZER on modified dependency inclusions. Bold numbers indicate CVE entries missed by JARALYZER.

	Agreed	Type 1	Type 2	Type 3	Type 4
Jaralyzer	61	61	**58**	**58**	**54**
Steady		59	61	61	-
Jaralyzer	78	78	78	78	**74**
OWASP DC		74	56	-	56
Jaralyzer	57	57	57	57	**56**
Dependabot		56	-	-	-
Jaralyzer	64	64	64	64	**62**
OSV Scanner		-	-	-	-
Jaralyzer	72	72	72	72	**71**
Com		72	-	-	-

reported a true or false positive, they rely on a ground truth, which they have semi-automatically uncovered within their [Dan+21] study and added to the Achilles benchmark generator. We manually inspected the ground truth used in their experiment and uncovered that it is not fully accurate. Even in their small benchmark containing only 7 distinct artifacts and 15 distinct CVE entries, two of the test cases are incorrect. They report httpclient 4.1.3 as being affected by CVE-2015-5262, whereas it only affects httpclient as of version 4.3[3]. Furthermore, they erroneously do not consider CVE-2018-11040 as affecting spring-webmvc 5.0.0.RELEASE[4]. Because of these inaccuracies, we decided not to rely on the ground truth provided by Achilles. Instead, we compare JARALYZER head-to-head with each of the five other tools within our setup. We assume that if a tool is unaffected by a specific type of modification, it should report the same CVE entries for both the unmodified and modified benchmarks. First, we filter all the CVE entries where both tools *agree* upon the respective CVE entry affecting the unmodified benchmark. These agreed upon CVE entries are the only ones we consider for the modified benchmarks. Then, we investigate how many of the same CVE entries reported for the unmodified benchmark by both tools are also reported for the type 1–4 benchmarks.

Table 5.1 presents the results of this experiment. It shows the head-to-head comparison of JARALYZER with each of the tools and the number of agreed upon CVE entries for the unmodified benchmark (column "Agreed"). Columns "Type 1" to "Type 4" show how many of agreed upon CVE entries are reported by each tool for the type 1–4 benchmarks. None of the tools JARALYZER is compared against is able to handle

[3] `https://issues.apache.org/jira/browse/HTTPCLIENT-1478`
[4] `https://security.snyk.io/vuln/SNYK-JAVA-ORGSPRINGFRAMEWORK-467268`

all types of modifications. While some tools perform better on certain types of modifications, e.g., Eclipse Steady performing well on types 1–3, GitHub Dependabot, OSV Scanner and the commercial tool completely fail for type 2–4 modifications. Throughout type 1–3 modifications, JARALYZER does not miss any CVE entry, except for three in its comparison to Eclipse Steady. A detailed investigation of these entries reveals this to be an interesting case, where the three missed CVE entries are actually *false positives* for the unmodified benchmark reported by both JARALYZER and Eclipse Steady. These false positives are caused by the code-centric nature of the tools. The corresponding CVE entries (CVE-2013-6429, CVE-2014-0225, CVE-2015-2080) are affecting the multi-module projects Spring and Jetty. If the fix commits associated with a CVE entry modify code across multiple modules of a project, even though the vulnerability only affects a single module, this may interfere with detection in certain cases. For example, CVE-2015-2080 affects the jetty-http module of the Jetty project, however, its fix commits modify code in both the jetty-http *and* jetty-util modules. Now, if the JAR files corresponding to jetty-http and jetty-util are scanned individually, both tools erroneously report jetty-util as being vulnerable, since the majority of the fix code, which is applied to jetty-http, cannot be identified within jetty-util. However, in type 2–4 modifications, all dependencies are re-bundled into a single JAR-file. This then allows the tools to match all code changes from the respective fix commits at once. In contrast to Eclipse Steady, JARALYZER is able to take advantage of this, and correctly determines that the re-bundled JAR file is unaffected by these CVE entries, reducing the signaled entries from 61 to 58 (54 for type 4). An important point to consider is that, due to the unavailability of source code, all CVE entries flagged by Eclipse Steady in the modified benchmarks, including those not agreed to by JARALYZER, are classified as "unknown", meaning that Eclipse Steady requires manual analysis for each case.

None of the compared tools is able to handle type 4 modifications, except for OWASP DependencyCheck, which still misses 22 CVE entries. For type 4 modifications, JARALYZER, even though it does not find all CVE entries it detected in the other benchmarks, considerably outperforms the other tools. The missed CVE entries are caused by not being able to rely on the FQNs and thus JARALYZER having to rely on its re-packaging detection mode with the configured thresholds. To further assess whether modifications affect JARALYZER's *precision*, we examined whether it reported CVE entries for type 1–4 modifications that were not reported for the unmodified benchmark. This did not occur.

While our results show that no tool within our experimental setup, except JARALYZER, is able to effectively handle modified dependencies, we are not fully able to confirm the results of Dann et al. Regarding the ability to handle different types

of modifications, our results align with those of Dann et al., with e.g. OWASP DependencyCheck being the only tool to partially support type 4 modifications. However, unlike their findings, we do not observe the same significant performance degradation for modifications that the tools can still partially handle (e.g., types 1–3 for Eclipse Steady), even when accounting for the inaccurate ground truth.

Finally, we evaluated JARALYZER on the small benchmark (with adjusted ground truth) used within Dann et al.'s study. JARALYZER achieved 100% recall and precision across all modification types.

> Among the evaluated tools, JARALYZER is the only one capable of handling all types of modified dependency inclusions, missing at most 6% of vulnerabilities in type 4 modifications.

5.2.3 RQ10: How does JARALYZER compare to the state-of-the-art code-centric dependency scanner in identifying *unmodified* known-to-be-vulnerable dependencies?

This research question considers the scenario where *no* modifications are applied to the dependency inclusions. To evaluate JARALYZER's detection performance, we directly compare it to the state-of-the-art code-centric dependency scanner Eclipse Steady. We perform a study similar to the one conducted by Ponta et al. [PPS20], where they compare the findings of Eclipse Steady to the findings of OWASP DependencyCheck and manually review the cases where the tools disagree. In their conducted study they determine that Eclipse Steady outperforms OWASP DependencyCheck and detects considerably fewer false positives while also finding more true positives. Thus, we compare JARALYZER to Eclipse Steady.

As reported in Section 5.2.2, there are 61 CVE entries that both tools report and agree upon for the unmodified benchmark. In this research question we primarily focus on the cases where the tools *disagree*. We only considered the 655 CVE entries shared across the respective knowledge bases of JARALYZER and Eclipse Steady.

Tables 5.2a and 5.2b present the CVE entries only reported by one of the tools. There are 43 CVE entries, which JARALYZER reports but Eclipse Steady does not (Table 5.2a). Conversely, Eclipse Steady reports 44 CVE entries that JARALYZER does not (Table 5.2b). As previously noted, we consider CVE entries that are part of both knowledge bases.

Table 5.2: Disjoint CVE reports by JARALYZER and Eclipse Steady (unmodified benchmark)
✓ = true positive; ✗ = false positive

(a) JARALYZER reports

CVE	KB	GA	NVD	SV
commons-fileupload-1.3.2				
CVE-2016-6793[a]	-	✗	✗	✗
dom4j-1.6.1				
CVE-2020-10683	-	✓	✓	✓
itextpdf-5.5.0				
CVE-2017-9096	-	✓	✓	✓
jackson-databind-2.9.7				
CVE-2019-12086	✓	✓	✓	✓
CVE-2019-12384	✓	✓	✓	✓
CVE-2019-12814	✓	✓	✓	✓
CVE-2019-14379	✓	✓	✓	✓
CVE-2019-14439	✓	✓	✓	✓
CVE-2019-14892	✓	✓	✓	✓
CVE-2019-14893	✓	✓	✓	✓
CVE-2019-16942	✓	✓	✓	✓
CVE-2019-16943	✓	✓	✓	✓
CVE-2019-17267	✓	✓	✓	✓
CVE-2019-17531	✓	✓	✓	✓
CVE-2019-20330	✓	✓	✓	✓
CVE-2020-8840	✓	✓	✓	✓
CVE-2020-9546	✓	✓	✓	✓
CVE-2020-9547	✓	✓	✓	✓
CVE-2020-9548	✓	✓	✓	✓
CVE-2020-10650	✓	✓	✓	✓
CVE-2020-10672	✓	✓	✓	✓
CVE-2020-10968	✓	✓	✓	✓
CVE-2020-10969	✓	✓	✓	✓
CVE-2020-11112	✓	✓	✓	✓
CVE-2020-11113	✓	✓	✓	✓
CVE-2020-11619	✓	✓	✓	✓
CVE-2020-11620	✓	✓	✓	✓
CVE-2020-14060	-	✓	✓	✓
CVE-2020-14061	-	✓	✓	✓
CVE-2020-14062	-	✓	✓	✓
CVE-2020-14195	-	✓	✓	✓
CVE-2020-24616	✓	✓	✓	✓
CVE-2020-24750	✓	✓	✓	✓
jetty-server-9.4.10.v20180503				
CVE-2019-10241	✓	✓	✓	✗
CVE-2019-10247	-	✓	✓	✓
CVE-2019-17632	-	✗	✗	✗
jetty-servlet-9.4.10.v20180503				
CVE-2019-10241	✓	✗	✗	✗
jetty-util-9.4.10.v20180503				
CVE-2019-10241	-	✗	-[b]	✗
CVE-2019-10246	-	✗	-[b]	✗
CVE-2021-44832	-	✗	✗	✗
netty-all-4.0.36.Final				
CVE-2021-21290	-	✓	✓	✓
poi-ooxml-3.14				
CVE-2019-12415	✓	✓	✓	✓
undertow-core-1.4.23.Final				
CVE-2017-2670	-	✗	✗	✓

[a] CVE actually affects Apache Wicket, however, affected file is copied from commons-fileupload

[b] The NVD entry did not contain any version indications

(b) STEADY reports

CVE	KB	GA	NVD	SV
bcprov-jdk15on-1.58				
CVE-2015-6644	-	✗	✗	-
CVE-2016-1000338	-	✗	✗	✗
CVE-2016-1000339	-	✗	✗	✗
CVE-2016-1000340	-	✗	✗	✗
CVE-2016-1000341	-	✗	✗	✗
CVE-2016-1000342	-	✗	✗	✗
CVE-2016-1000343	-	✗	✗	✗
CVE-2016-1000344	-	✗	✗	✗
CVE-2016-1000345	-	✗	✗	✗
CVE-2016-1000346	-	✗	✗	✗
CVE-2016-1000352	✗	✗	✗	✗
commons-compress-1.9				
CVE-2019-12402	-	✗	✗	✗
groovy-all-2.4.7				
CVE-2015-3253	✗	✗	✗	✗
hibernate-validator-5.4.1.Final				
CVE-2014-3558	✗	✗	✗	✗
httpclient-4.5.2				
CVE-2013-4366	✗	✗	✗	✗
jackson-databind-2.9.7				
CVE-2017-17485	-	✗	✗	✗
CVE-2018-5968	✗	✗	✗	✗
CVE-2018-11307	✗	✗	✗	✗
CVE-2018-12022	✗	✗	✗	✗
CVE-2018-12023	✗	✗	✗	✗
jackson-dataformat-xml-2.9.3				
CVE-2016-3720	-	✗	✗	✗
jetty-http-9.4.10.v20180503				
CVE-2015-2080	✗	✗	✗	✗
CVE-2017-7657	✓	✗	✓	✗
jetty-server-9.4.10.v20180503				
CVE-2016-4800	-	✗	✗	✗
CVE-2017-7656	✓	✓	✓	✓
CVE-2017-7657	✓	✗	✓	✗
CVE-2017-7658	✓	✓	✓	✓
CVE-2018-12538	-	✓	✗	✗
jetty-util-9.4.10.v20180503				
CVE-2016-4800	-	✗	✗	✗
CVE-2017-9735	-	✗	✗	✗
CVE-2018-12536	-	✗	✗	✓
okhttp-2.7.0				
CVE-2016-2402	✓	✓	✓	✓
spring-core-5.0.4.RELEASE				
CVE-2015-0201	✗	✗	✗	✗
spring-oxm-5.0.4.RELEASE				
CVE-2014-0054	✗	✗	✗	✗
CVE-2014-0225	-	✗	✗	✗
CVE-2014-3578	-	✗	✗	✗
spring-web-5.0.4.RELEASE				
CVE-2013-6429	✗	✗	✗	✗
CVE-2013-6430	-	✗	✗	✗
CVE-2014-0054	-	✗	✗	✗
CVE-2014-3578	-	✗	✗	✗
tomcat-embed-core-8.5.33				
CVE-2020-13934	-	✓	✓	✓
undertow-core-1.4.23.Final				
CVE-2018-14642	-	✓	-[b]	✓
CVE-2019-3888	-	✓	✓	✓
CVE-2020-10705	-	✓	✓	✓

For each of the reported CVE entries we performed a manual investigation to determine whether it is a true or false report. To do so we considered four different OSS vulnerability advisories. In particular we considered the GitHub Advisory Database (GA) [Inc25a], the National Vulnerability Database (NVD) [ST25], the Snyk Vulnerability Database (SV) [Lim25], and Project KB (KB) [Pon+19]. Although we included the OSV Scanner in our experiments, we do not consider the OSV database in this experiment, as it primarily aggregates existing vulnerability data (e.g. NVD and GA) and does not provide independent classification of new vulnerabilities [Inc25b]. In fact, Project KB contains not only fix commits that we used as described in Section 5.2.1, but also manual assessments indicating whether an artifact is affected by a specific CVE entry. Eclipse Steady uses these manual assessments to enhance its detection performance and cope with its inability to compare source code with bytecode. To not interfere with our experiments, we removed these manual assessments from Eclipse Steady's and JARALYZER's respective knowledge bases. However we use them for our manual investigation. We consider a CVE report to be a true positive (false positive, resp.) for an artifact, if all advisories that contain the CVE entry agree on the artifact being affected (not affected, resp.) by the given CVE entry. Two researchers independently verified whether the advisories listed the respective CVE entry to be affecting the specific artifact and version as reported by either tool. Furthermore, to analyze the causes of disagreement between the tools, we reviewed their individual scan reports and manually inspected the corresponding fix commits to identify potential sources of misclassification. Unlike JARALYZER, Eclipse Steady produces scan reports with limited details, which makes it difficult to pinpoint the exact reasons behind erroneous results. The reports only list the CVE entries assigned to the scanned artifacts and classify them as either "vulnerable" or "unknown". In 37 of 44 cases, Eclipse Steady marked the entries as "unknown" and requested manual analysis.

Table 5.2a shows the CVE entries only reported by JARALYZER, with corresponding artifacts in gray boxes. According to our manual investigation, 35 out of 43 reports are true positives for JARALYZER and false negatives for Eclipse Steady. Note that 30 of them affect jackson-databind. Eclipse Steady misses those, even though source code is available, because it cannot account for code transformations outside of methods, such as field initializations or initializer blocks, due to its inability to directly compare bytecode with source code. These jackson-databind vulnerability fixes consist of either changing a regular expression stored in a field or adding entries to serialization blocklists in static initializers. Since JARALYZER operates on bytecode rather than source code, and because the Java compiler inlines field initializations

and static initializer code into methods, in contrast to Eclipse Steady, it natively supports such cases.

However, we also uncovered five cases where JARALYZER reported false positives, according to the advisories. All can be attributed to erroneously reporting a CVE for multiple modules of the same project, due to spurious fix commits that contain changes unrelated to the actual fix as described in Section 5.2.2, or because of incorrect re-packaging matches due to the threshold configurations. For CVE-2016-6793, which JARALYZER reported as affecting commons-fileupload-1.3.2, we made an interesting observation. While this CVE entry actually affects Apache Wicket[5], JARALYZER reports it because it detected a re-packaging of the class affected by the vulnerability. Further investigation revealed that the developers of Apache Wicket copied vulnerable code from commons-fileupload into the Wicket project. The vulnerability within commons-fileupload received its own CVE entry[6], even though they describe the same vulnerability. The CVE entry for commons-fileupload is not part of Project KB and thus not reported by either tool. Therefore, we do not consider this as a clear false positive, but rather as an unclear case. We identified three other unclear cases, where the four advisories reported conflicting information.

To summarize, of the 43 CVE entries reported only by JARALYZER, we classified 35 as true positives, 4 as false positives and 4 as unclear.

Table 5.2b contains the CVE entries only reported by Eclipse Steady. According to our investigation, 33 out of 44 are false positives. Only in 7 cases Eclipse Steady reported a true positive that JARALYZER missed. Furthermore, 4 reports are unclear, since the advisories reported conflicting information. According to Eclipse Steady's scan report, it was unable to automatically classify the false positive cases and instead deferred their evaluation to manual analysis. This might be due to an inability to retrieve the corresponding source code, inaccuracies in the retrieved code or due to ambiguous fix commit matches [SE21]. Due to the limited details in the scan reports the exact cause cannot be determined. A similar issue appears for the five JARALYZER reports classified as true positives, which Eclipse Steady missed, excluding the ones related to jackson-databind. The corresponding fix commits involve changes to method bodies and should be thus detectable by Eclipse Steady. However, the scan report does not provide enough details to determine the exact reason for the missed detections.

[5] https://security.snyk.io/vuln/SNYK-JAVA-ORGAPACHEWICKET-31022
[6] https://security.snyk.io/vuln/SNYK-JAVA-COMMONSFILEUPLOAD-30401

Table 5.3: Required scanning times for the unmodified benchmark containing 56 OSS dependencies

Tool	Jaralyzer	Steady	OWASP DC	OSV Scanner	Com
Time	131s (63s)	1,387s	9s	5s	30s

Using the same methodology, we also validated the 61 CVE reports where both tools agree. According to the advisories, both tools reported 28 true positives and 16 false positives. The 17 remaining cases are unclear due to advisory disagreements.

As noted in Section 5.2.2, the Achilles benchmark includes a ground truth that we initially excluded from our experiments due to identified inaccuracies. Based on the findings from this research question, we conducted an additional validation of the provided ground truth. Overall, it contains classifications only for 35 of the 148 CVE reports (23%) generated by either JARALYZER or Eclipse Steady, covering 9 of the 87 cases (9%) where the tools disagreed and 26 of the 61 cases (42%) where they agreed. We further examined the 10 out of 35 ground truth classifications that contradicted the findings of JARALYZER. This involved a manual review of the corresponding reports, including an analysis of advisory data, inspection of code changes in the relevant fix commits, and investigation of related issue tracker entries. Our analysis indicated that 9 out of the 10 disagreeing classifications were likely incorrect within the ground truth. We reported these 9 discrepancies, as well as the 52 cases absent from Achilles, where JARALYZER and all considered advisories agreed, to the authors of Achilles, who subsequently incorporated our feedback into the benchmark[7].

JARALYZER outperforms Eclipse Steady on unmodified dependencies. It identifies 35 unique true positives while reporting only 4 false positives. In contrast, Eclipse Steady detects only 7 unique true positives but produces 33 false positives.

5.2.4 RQ11: How runtime-efficient is JARALYZER?

We investigated how much time each of the scanners in our experimental setup (see Section 5.2.1) required to perform a vulnerability scan on the unmodified benchmark. On average, an industry-grade Java project contains 36 dependencies [Dan+21]. In comparison, our benchmark includes 56 dependencies, moderately more than a typical industry-grade project.

[7] `https://github.com/secure-software-engineering/achilles-benchmark-depscanners/pull/44 & 45`

Table 5.3 shows the time required by each scanner. Code-centric scanners, in contrast to metadata-based scanners, typically come with significant overhead, which naturally increases the scan time. JARALYZER, e.g., needs to extract methods from the scanned JAR file, normalize and generate code property graphs for them. These are all complex tasks, which take a non-trivial amount of time. In total, JARALYZER takes 131 seconds to scan the 56 OSS dependencies in our benchmark. As stated previously, we execute JARALYZER in its default mode *and* re-packaging detection mode. Executing JARALYZER's re-packaging mode is only required to detect type-4 modified dependencies. Running only its default mode, JARALYZER does not miss any vulnerabilities in the unmodified benchmark and only takes 63 seconds. Meanwhile, Eclipse Steady takes 1,387 seconds, requiring more than ten times the scanning time for the benchmark. Due to the lightweight approach behind metadata-based scanners, they require lower runtimes. Specifically, OWASP DependencyCheck required 9 seconds, OSV Scanner required 5 seconds, and the commercial scanner required 30 seconds. GitHub Dependabot performs the analysis automatically, when a commit containing dependency metadata is pushed to GitHub, where analysis results can then be obtained via a request to the GitHub API. In contrast to the other scanners, this analysis does not run locally, but on the GitHub servers. We thus were unable to measure its precise runtime and do not include it here.

For projects of above-average size, JARALYZER takes between 63 and 131 seconds to complete the scan. Overall, JARALYZER is more than ten times faster than Eclipse Steady.

5.3 Threats to Validity

A few potential threats might impact the validity of our experiments with JARALYZER. We used the Achilles benchmark generator to create the individual benchmarks serving as a baseline for our evaluation. Achilles only provides versions of OSS artifacts released by 2021, thus for many artifacts we did not use the latest releases. Furthermore, while we did not use Achilles inaccurate ground truth, we still relied on Achilles correctly modifying the dependency inclusions. Although, our manual investigation revealed these to be most likely correct. The number of CVE entries detected by JARALYZER within type 4 modifications depends on the configured threshold parameters. Lowering these thresholds may enable JARALYZER to detect more CVE entries in the type 4 benchmark but could also reduce its precision in the unmodified benchmark. We selected the default values based on empirical testing

with our benchmark. However, this approach may lead to overfitting, meaning other datasets might achieve better performance with different values. JARALYZER was unable to compile 61 CVE entries from Project KB, which we therefore excluded from RQ10 (see Section 5.2.3). However, in cases where compilation fails or vulnerability fixes lack source code changes, the detection can be augmented by using metadata, as demonstrated by Eclipse Steady. Finally, the effectiveness of code-centric dependency scanning depends on the quality of fix commits. When fix commits include spurious changes unrelated to the actual fix, the dependency scanning performance may be affected.

5.4 Related Work

Current legislation like the EU's Cyber Resilience Act and the US's Cybersecurity Executive Order require software development companies to focus on the secure usage of OSS components. Therefore many commercial and open-source tools have been developed to detect known-to-be-vulnerable OSS dependencies. Within our study we used the popular metadata-based tools OWASP DependencyCheck [OWA24], OSV Scanner [Inc24c] and GitHub Dependabot [Inc24b], as well as the code-centric tool Eclipse Steady [PPS20; PPS18; PPS15]. Over the years, the term *software composition analysis* (SCA) has emerged to describe dependency scanners, particularly in commercial contexts. Popular commercial SCA tools include Endor Labs SCA [Lab24], Snyk Open Source SCA [Lim24], Mend SCA [Men24] and Black Duck SCA [Inc24a].

Some approaches under the name of patch presence testing attempt to decide whether a patch has been applied to a specific binary. FIBER [ZQ18] generates signatures from the security patch's C/C++ source code and searches for its presence in the target binary. Osprey [Sun+21] lifts the C/C++ binary into a platform-agnostic intermediate representation (IR) and performs the patch presence test based on this IR. PDiff [Jia+20] performs patch presence testing within downstream kernel images by generating semantic summaries of the patch and checking whether the target image is closer to the image before or after the patch is applied. PS^3 [Zha+24] uses semantics-level symbolic execution to extract signatures that are stable under different compiler options. It then uses those signatures to test the presence of the patch within the target C/C++ binary. BScout [Dai+20] uses feature-extraction and leverages line number information to match lines of Java source code, belonging to the patch, to lines of bytecode within the target file. Doing so, Dai et al. can determine whether the patch has been applied or not. PPT4J [Pan+24] employs a

similar feature-extraction based approach, which extracts features that are stable through the compilation process and uses these features to determine whether the patch is applied or not. FIBER, PDiff, and PS^3 target C/C++ binaries, BScout and PPT4J address Java applications. In the future the above mentioned approaches could complimentary be integrated into JARALYZER's vulnerability scanning stage to further aid with the CPG-based patch presence testing. Since they do not perform the initial library matching nor address modifications, we did not include them into our evaluation, since a comparison with dependency scanners aiming for similar goals is more relevant and adequate.

Some approaches explicitly target third-party library (TPL) identification for Android to find licensing issues or vulnerable components usages. OSSPolice [Dua+17] finds license violations and security risks within Android apps by maintaining a database of unique app features and comparing the target app to this database to determine the exact TPL used. ATVHunter [Zha+21] applies control-flow graph based feature-extraction to more reliably identify the correct TPL and version. PHunter [Xie+23] relies on special obfuscation-resilient features to determine whether patches are applied to an Android TPL, even with code obfuscations applied.

In recent years, researchers have published different approaches to identify security-relevant commits and resulting fix-commit databases. Ponta et al. [Pon+19] created a manually-curated set of fix commits, called Project KB, focusing on industry-relevant OSS projects. Bhandari et al. [BNM21] propose an approach that automatically extracts fix commits from CVE entries within the NVD. They create the CVEFIXES database, containing fix commits for 5,365 CVE entries affecting OSS projects of various programming languages. Sabetta et al. [Sab+24] propose the rule-based tool PROSPECTOR, which tries to automatically find and rank fix commits for specified vulnerability identifiers. Akhoundali et al. [Akh+24] propose an approach that uses PROSPECTOR and create the MOREFIXES database, containing fix commits for 26,617 CVEs affecting a diverse set of OSS projects.

5.5 Conclusion

In this chapter we presented JARALYZER, a novel *bytecode-centric* dependency scanner for Java. JARALYZER utilizes targeted compilation and bytecode normalization to transform source code fix commits and their dependencies into a unified representation, enabling effective comparison. It further employs a CPG-based approach to verify the presence of fixes, even when those fixes have evolved over time. These

enhancements over the code-centric approach allow JARALYZER to effectively address the challenges posed by modified dependencies.

An evaluation performed on 56 popular OSS artifacts, including five state-of-the-art dependency scanners, showcased that, while all other tools suffer performance deterioration when dependencies are modified, JARALYZER's performance remains stable. In few cases, it was even able to exploit the modifications and report fewer false positives. But even when dependencies are unmodified, JARALYZER outperforms the state-of-the-art code-centric dependency scanner Eclipse Steady, by detecting more vulnerabilities and producing fewer false alerts. These results position JARALYZER to be the code-centric dependency scanner of choice for Java projects, especially when it comes to identifying modified dependencies.

6 Conclusion and Outlook

Open-source software (OSS) dependencies are integral to modern software code bases, often accounting for the majority of their total content. By leveraging proven and well-tested OSS components, developers can reduce development time and cost while enhancing overall quality. However, this dependence also introduces significant security risks. Developers can easily include vulnerable dependencies into their software projects. If such vulnerable dependencies are not identified and addressed correctly, they can cause severe damage, as demonstrated by the infamous Equifax Breach and Log4Shell security incidents. It is thus crucial to be able to reliably identify such dependencies as quickly as possible. To this end, numerous open-source and commercial dependency scanners have been developed to detect the inclusion of known-to-be-vulnerable dependencies. However, multiple independent studies have demonstrated that even state-of-the-art dependency scanners fail to reliably identify known-to-be-vulnerable dependencies in Java projects when the dependencies are included in a *modified* form. Such modifications, including re-compilations, re-bundlings, re-packagings and removal of metadata, are common in the Java ecosystem, can be applied transparently, and even be found in the Java standard library.

In this thesis, we presented a *bytecode-centric* dependency scanning approach capable of reliably identifying inclusions of known-to-be-vulnerable dependencies, even when they have been modified. In contrast to other dependency scanning approaches, it does not rely on any metadata and instead analyzes only the bytecode of the included dependencies, the only information consistently available when modifications have been applied. To facilitate this, we have developed novel techniques that are crucial to enabling bytecode-centric dependency scanning.

In our first contribution we presented JESS, our approach towards *targeted compilation* of Java programs. Fix commits, which contain the actual code changes addressing known vulnerabilities, serve as the basis for bytecode-centric dependency scanning. However, these fix commits are applied to the Java source code of the dependencies. Therefore, to compare them with their bytecode counterparts, the source code must first be compiled. As demonstrated by our study and several others, relying solely on the project's provided build scripts is insufficient to reliably compile

repository snapshots. We therefore propose JESS, an approach that uses slicing and stubbing to isolate parts of the code base for compilation, thereby avoiding the need to compile the entire repository snapshot. Our evaluation of JESS demonstrated that it, leveraged within our proposed heuristic, successfully enabled compilation of 90% of entries in the popular Project KB fix-commit database, making it a crucial first step toward comparing Java source code fix commits with bytecode artifacts.

As demonstrated in our second contribution, being able to compile fix commits is still insufficient to enable a reliable comparison of Java source code and bytecode, due to differences induced by different compilation environments. Compiling identical source code with two different versions of the JDK compiler can result in up to 46% of the generated bytecode files containing differences. Since vulnerability fixes often consist of only small code changes, these compilation-induced differences can easily mask the actual fixes. To address this problem we presented JNORM, our approach towards *bytecode normalization*. JNORM converts Java bytecode into the intermediate representation Jimple, applies common code optimizations, and transforms 16 classes of differences into a form that is independent of the compilation environment. These 16 difference classes were initially identified through our extensive empirical evaluation of various Java compilers, vendors, versions, and settings. Using JNORM, we can remove 99% of all differences introduced by varying compilation environments, making it a critical second step in comparing Java source code to bytecode.

Finally, in our third contribution we presented JARALYZER, our *bytecode-centric* dependency scanner. JARALYZER builds upon our first two contributions, which serve as the foundation for converting vulnerability-fixing commits and included dependencies into a comparable representation. To accurately determine the presence or absence of a fix within scanned dependencies JARALYZER employs a code property graph-based comparison technique that captures not only syntactic information but also control and data flow. The combination of these techniques enables JARALYZER to identify known-to-be-vulnerable dependencies even when modifications have been applied, making it the only dependency scanner in our evaluation capable of reliably handling modified dependencies. Moreover, even when applied to unmodified dependencies, JARALYZER identified more known-to-be-vulnerable dependencies while reporting fewer false positives than the state-of-the-art scanner Eclipse Steady.

While the bytecode-centric approach overcomes several limitations of current dependency scanners, it also presents challenges compared to metadata-based methods. Primarily, it is significantly more complex, leading to increased computational time and resource requirements. This complexity results in more involved implementa-

tions and a higher potential for errors. Another important limitation of code-centric, and by extension bytecode-centric, approaches is their reliance on fix commits. Although using fix commits offers advantages, such as reducing delays between fix availability and security advisory updates, it also introduces drawbacks. These approaches depend heavily on best practices being followed during software development. Fix commits containing spurious changes unrelated to the vulnerability fix or extensive refactorings of the initial fix can pose challenges for code-centric methods. Additionally, compiling fix commits raises another issue. Although our targeted compilation technique achieved a 90% success rate across Project KB, it required manual effort and still failed to compile certain fix commits, rendering the corresponding vulnerabilities undetectable by the bytecode-centric approach.

Looking ahead, we envision a hybrid approach that combines the lightweight efficiency of metadata-based approaches with the precision and reliable detection capabilities of code-centric, particularly bytecode-centric, approaches. Rather than compiling fix commits, resources could be invested in precomputing and indexing all artifacts hosted on popular artifact repositories such as Maven Central. This would enable the identification of re-compilations, re-bundlings, and re-packagings prior to scan time, allowing for the *augmentation* of dependency manifest files. The augmented manifest files could then be scanned using existing metadata-based dependency scanners.

Another promising direction for future research is reachability analysis, applied after dependency scanning to determine whether the vulnerable parts of a dependency are actually utilized by the downstream software that includes it. Because code-centric approaches already analyze the concrete code changes that fix vulnerabilities, incorporating reachability analysis is a natural next step for bytecode-centric dependency scanning. Initial applications in both research and commercial tools have demonstrated that reachability analysis can significantly reduce false vulnerability alerts. However, modern Java applications make extensive use of dynamic language features, such as reflection, native calls, and dynamic proxies, which pose significant challenges for existing reachability techniques. Addressing these limitations remains an important goal for future work.

Future research may even extend beyond reachability analysis to the automated synthesis of proof-of-vulnerability (POV) tests. POV tests aim to construct concrete execution scenarios in which a vulnerability originating in an included dependency is actually triggered within the context of the including project, thereby demonstrating the vulnerability's real, exploitable presence. Whereas reachability analysis can only indicate that a vulnerable method is invoked by the including project, a POV test can

provide stronger evidence by confirming that the vulnerability is actually triggered under realistic project conditions. Thus, POV tests help developers determine whether a reported vulnerability is real rather than a false alarm, enabling them to more accurately prioritize and remediate issues in vulnerable dependencies.

Although challenges remain to be addressed, bytecode-centric dependency scanning advances the state-of-the-art in detecting known-to-be-vulnerable dependencies in Java projects. By using JARALYZER as their dependency scanner of choice, developers can be more confident that no known vulnerabilities originate from third-party components, allowing them to focus resources on securing their own code.

Implementations and Data

Throughout the research presented in this thesis, we created several datasets and implementations. These resources have been made publicly available to enable other researchers to reproduce and build upon our work.

Targeted Compilation

JESS is publicly available as an open-source project on GitHub:

`https://github.com/stschott/jess`

All experiments, results, datasets and implementations associated with JESS and targeted compilation are available on Zenodo:

`https://doi.org/10.5281/zenodo.12804723`

Bytecode Normalization

JNORM is publicly available as an open-source project on GitHub:

`https://github.com/stschott/jnorm-tool`

All experiments, results, datasets and implementations associated with JNORM and bytecode normalization are available on Zenodo:

`https://doi.org/10.5281/zenodo.12625104`

Bytecode-centric Dependency Scanning

JARALYZER is publicly available as an open-source project on GitHub:

`https://github.com/stschott/jaralyzer`

All experiments, results, datasets and implementations associated with JARALYZER and bytecode-centric dependency scanning are available on Zenodo:

`https://doi.org/10.5281/zenodo.17829100`

Bibliography

[Akh+24] Jafar Akhoundali, Sajad Rahim Nouri, Kristian Rietveld, and Olga Gadyatskaya. "MoreFixes: A large-scale dataset of CVE fix commits mined through enhanced repository discovery". In: *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2024, pp. 42–51 (cit. on pp. 88, 106).

[BM98] Brenda S Baker and Udi Manber. "Deducing Similarities in Java Sources from Bytecodes." In: *USENIX Annual Technical Conference*. 1998, pp. 179–190 (cit. on pp. 51, 80).

[BNM21] Guru Bhandari, Amara Naseer, and Leon Moonen. "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software". In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2021, pp. 30–39 (cit. on pp. 88, 106).

[BH20] Benjamin Bowman and H Howie Huang. "VGRAPH: A robust vulnerable code clone detection system using code property triplets". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2020, pp. 53–69 (cit. on pp. 87, 89).

[CLZ14] Kai Chen, Peng Liu, and Yingjun Zhang. "Achieving accuracy and scalability simultaneously in detecting application clones on android markets". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 175–186 (cit. on p. 80).

[Chi+20] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, et al. "Code-based vulnerability detection in node. js applications: How far are we?" In: *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 2020, pp. 1199–1203 (cit. on p. 14).

[DAB21] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. "Sampling projects in github for MSR studies". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 560–564 (cit. on p. 52).

[DH08] Barthélémy Dagenais and Laurie Hendren. "Enabling static analysis for partial java programs". In: *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. 2008, pp. 313–328 (cit. on pp. 19, 30, 47).

[Dai+20] Jiarun Dai, Yuan Zhang, Zheyue Jiang, et al. "{BScout}: Direct whole patch presence test for java executables". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1147–1164 (cit. on pp. 22, 105).

[DHB19] Andreas Dann, Ben Hermann, and Eric Bodden. “Sootdiff: Bytecode comparison across different java compilers”. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. 2019, pp. 14–19 (cit. on pp. 3, 4, 21, 51, 58, 82).

[Dan+21] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. “Identifying challenges for oss vulnerability scanners-a study & test suite”. In: *IEEE Transactions on Software Engineering* 48.9 (2021), pp. 3613–3625 (cit. on pp. 2, 7, 9, 10, 86, 95–97, 103).

[DPY16] Yaniv David, Nimrod Partush, and Eran Yahav. “Statistical similarity of binaries”. In: *Acm Sigplan Notices* 51.6 (2016), pp. 266–280 (cit. on p. 81).

[DPY17] Yaniv David, Nimrod Partush, and Eran Yahav. “Similarity of binaries through re-optimization”. In: *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 2017, pp. 79–94 (cit. on p. 81).

[DPY18] Yaniv David, Nimrod Partush, and Eran Yahav. “Firmup: Precise static detection of common vulnerabilities in firmware”. In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 392–404 (cit. on p. 81).

[DG10] Ian J Davis and Michael W Godfrey. “From whence it came: Detecting source code clones by analyzing assembler”. In: *2010 17th Working Conference on Reverse Engineering*. IEEE. 2010, pp. 242–246 (cit. on p. 80).

[Die+24] Jens Dietrich, Shawn Rasheed, Alexander Jordan, and Tim White. “On the security blind spots of software composition analysis”. In: *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. 2024, pp. 77–87 (cit. on pp. 2, 92, 96).

[Don+22] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. “SnR: constraint-based type inference for incomplete Java code snippets”. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1982–1993 (cit. on p. 47).

[Dua+17] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. “Identifying open-source license violation and 1-day security risk at large scale”. In: *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2017, pp. 2169–2185 (cit. on p. 106).

[ECZ22] Douglas Everson, Long Cheng, and Zhenkai Zhang. “Log4shell: Redefining the web attack surface”. In: *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*. 2022, pp. 1–8 (cit. on p. 1).

[GMP20] Piyush Gupta, Nikita Mehrotra, and Rahul Purandare. “Jcoffee: Using compiler feedback to make partial code snippets compilable”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 810–813 (cit. on pp. 19, 47).

[HD09] James Hamilton and Sebastian Danicic. “An evaluation of current java bytecode decompilers”. In: *2009 Ninth IEEE international working conference on source code analysis and manipulation*. IEEE. 2009, pp. 129–136 (cit. on p. 21).

[HC21] Irfan Ul Haq and Juan Caballero. “A survey of binary code similarity”. In: *Acm computing surveys (csur)* 54.3 (2021), pp. 1–38 (cit. on p. 81).

[Har+19] Nicolas Harrand, César Soto-Valero, Martin Monperrus, and Benoit Baudry. “The strengths and behavioral quirks of Java bytecode decompilers”. In: *2019 19th International working conference on source code analysis and manipulation (SCAM)*. IEEE. 2019, pp. 92–102 (cit. on p. 21).

[Has+17] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. “Automatic building of java projects in software repositories: A study on feasibility and challenges”. In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2017, pp. 38–47 (cit. on pp. 3, 6, 19, 21, 42, 44, 46, 48, 53, 69, 88).

[HW17] Foyzul Hassan and Xiaoyin Wang. “Mining readme files to support automatic building of java projects in software repositories”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 277–279 (cit. on p. 47).

[HW18] Foyzul Hassan and Xiaoyin Wang. “Hirebuild: An automatic approach to history-driven repair of build scripts”. In: *Proceedings of the 40th international conference on software engineering*. 2018, pp. 1078–1089 (cit. on p. 47).

[Hem+11] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. “Finding software license violations through binary code clone detection”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. 2011, pp. 63–72 (cit. on p. 81).

[JWC08] Jeong-Hoon Ji, Gyun Woo, and Hwan-Gue Cho. “A plagiarism detection technique for Java program using bytecode analysis”. In: *2008 third international conference on convergence and hybrid information technology*. Vol. 1. IEEE. 2008, pp. 1092–1098 (cit. on p. 80).

[Jia+07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. “Deckard: Scalable and accurate tree-based detection of code clones”. In: *29th International Conference on Software Engineering (ICSE’07)*. IEEE. 2007, pp. 96–105 (cit. on pp. 51, 80).

[Jia+20] Zheyue Jiang, Yuan Zhang, Jun Xu, et al. “Pdiff: Semantic-based patch presence testing for downstream kernels”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 1149–1163 (cit. on p. 105).

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. “CCFinder: A multilinguistic token-based code clone detection system for large scale source code”. In: *IEEE transactions on software engineering* 28.7 (2002), pp. 654–670 (cit. on pp. 51, 80).

[Kar+24] Kadiray Karakaya, Stefan Schott, Jonas Klauke, et al. “Sootup: A redesign of the soot static analysis framework”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2024, pp. 229–247 (cit. on pp. 16, 51, 89).

[KRR14] Iman Keivanloo, Chanchal K Roy, and Juergen Rilling. “SeByte: Scalable clone and similarity search for bytecode”. In: *Science of Computer Programming* 95 (2014), pp. 426–444 (cit. on pp. 51, 80).

[KZG14] Oleksii Kononenko, Cheng Zhang, and Michael W Godfrey. “Compiling clones: What happens?” In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 481–485 (cit. on pp. 51, 82).

[Lou+19] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. “History-driven build failure fixing: how far are we?” In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 43–54 (cit. on p. 47).

[Luo+14] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 389–400 (cit. on p. 81).

[Lus18] Jeff Luszcz. “Apache struts 2: how technical and development gaps caused the equifax breach”. In: *Network Security* 2018.1 (2018), pp. 5–8 (cit. on p. 1).

[MMP18] Christian Macho, Shane McIntosh, and Martin Pinzger. “Automatically repairing dependency-related build breakage”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 106–117 (cit. on p. 47).

[Mar+22] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, et al. “How machine learning is solving the binary function similarity problem”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 2099–2116 (cit. on p. 81).

[Mas+19] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. “Safe: Self-attentive function embeddings for binary similarity”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2019, pp. 309–329 (cit. on p. 81).

[McC60] John McCarthy. “Recursive functions of symbolic expressions and their computation by machine, part I”. In: *Communications of the ACM* 3.4 (1960), pp. 184–195 (cit. on p. 23).

[Mel+20] Leandro TC Melo, Rodrigo G Ribeiro, Breno CF Guimarães, and Fernando Magno Quintão Pereira. “Type inference for C: Applications to the static analysis of incomplete programs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42.3 (2020), pp. 1–71 (cit. on p. 47).

[Pan+24] Zhiyuan Pan, Xing Hu, Xin Xia, et al. “PPT4J: Patch Presence Test for Java Binaries”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–12 (cit. on pp. 22, 105).

[PPS15] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. “Impact assessment for vulnerabilities in open-source software libraries”. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, pp. 411–420 (cit. on pp. 14, 105).

[PPS18] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. “Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 449–460 (cit. on pp. 4, 13, 14, 105).

[PPS20] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. “Detection, assessment and mitigation of vulnerabilities in open source dependencies”. In: *Empirical Software Engineering* 25.5 (2020), pp. 3175–3215 (cit. on pp. 1, 2, 4, 13, 14, 85, 95, 99, 105).

[Pon+19] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. “A manually-curated dataset of fixes to vulnerabilities of open-source software”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 383–387 (cit. on pp. 6, 20, 41, 88, 95, 101, 106).

[PMP+02] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. “Finding plagiarisms among a set of programs with JPlag.” In: *J. Univers. Comput. Sci.* 8.11 (2002), p. 1016 (cit. on pp. 51, 80).

[RK17] Chaiyong Ragkhitwetsagul and Jens Krinke. “Using compilation/decompilation to enhance clone detection”. In: *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. IEEE. 2017, pp. 1–7 (cit. on p. 82).

[RKC18] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. “A comparison of code similarity analysers”. In: *Empirical Software Engineering* 23 (2018), pp. 2464–2519 (cit. on p. 81).

[RC08] Chanchal K Roy and James R Cordy. “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization”. In: *2008 16th iEEE international conference on program comprehension*. IEEE. 2008, pp. 172–181 (cit. on pp. 51, 80).

[Sab+24] Antonino Sabetta, Serena Elisa Ponta, Rocio Cabrera Lozoya, et al. “Known Vulnerabilities of Open Source Projects: Where Are the Fixes?” In: *IEEE Security & Privacy* (2024) (cit. on p. 106).

[Sai+18] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. “Oreo: Detection of clones in the twilight zone”. In: *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2018, pp. 354–365 (cit. on p. 81).

[Saj+16] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. “Sourcerercc: Scaling code clone detection to big-code”. In: *Proceedings of the 38th international conference on software engineering*. 2016, pp. 1157–1168 (cit. on pp. 51, 80).

[SAH20] André Schäfer, Wolfram Amme, and Thomas S Heinze. “Detection of similar functions through the use of dominator information”. In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE. 2020, pp. 206–211 (cit. on p. 80).

[SAH21] André Schäfer, Wolfram Amme, and Thomas S Heinze. “Stubber: Compiling Source Code into Bytecode without Dependencies for Java Code Clone Detection”. In: *2021 IEEE 15th International Workshop on Software Clones (IWSC)*. IEEE. 2021, pp. 29–35 (cit. on pp. 19, 47).

[Sch+24a] Stefan Schott, Wolfram Fischer, Serena Elisa Ponta, Jonas Klauke, and Eric Bodden. “Compilation of Commit Changes Within Java Source Code Repositories”. In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*. IEEE, 2024, pp. 325–336 (cit. on pp. ix, 43).

[Sch+24b] Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. “Java Bytecode Normalization for Code Similarity Analysis”. In: *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 37:1–37:29 (cit. on pp. ix, 71).

[Sch+26] Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. “Bytecode-centric Detection of Known-to-be-vulnerable Dependencies in Java Projects”. In: *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*. 2026 (cit. on p. ix).

[SFZ10] Gehan MK Selim, King Chun Foo, and Ying Zou. “Enhancing source-based clone detection using intermediate representation”. In: *2010 17th working conference on reverse engineering*. IEEE. 2010, pp. 227–236 (cit. on p. 81).

[SP16] Matúš Sulír and Jaroslav Porubän. “A quantitative study of java software buildability”. In: *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. 2016, pp. 17–25 (cit. on pp. 3, 6, 19, 21, 44, 48, 69, 88).

[Sun+21] Peiyuan Sun, Qiben Yan, Haoyi Zhou, and Jianxin Li. “Osprey: A fast and accurate patch presence test framework for binaries”. In: *Computer Communications* 173 (2021), pp. 95–106 (cit. on p. 105).

[Tuf+17] Michele Tufano, Fabio Palomba, Gabriele Bavota, et al. “There and back again: Can you compile that snapshot?” In: *Journal of Software: Evolution and Process* 29.4 (2017), e1838 (cit. on pp. 3, 6, 19, 21, 22, 44, 48, 69, 88).

[Xie+23] Zifan Xie, Ming Wen, Haoxiang Jia, et al. “Precise and efficient patch presence test for android applications against code obfuscation”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 347–359 (cit. on p. 106).

[Xio+22] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R Cogo, and Zhen Ming Jiang. “Towards build verifiability for java-based systems”. In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 2022, pp. 297–306 (cit. on p. 82).

[Xu+17] Xiaojun Xu, Chang Liu, Qian Feng, et al. “Neural network-based graph embedding for cross-platform binary code similarity detection”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 363–376 (cit. on p. 81).

[Yam+14] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. “Modeling and discovering vulnerabilities with code property graphs”. In: *2014 IEEE symposium on security and privacy*. IEEE. 2014, pp. 590–604 (cit. on pp. 17, 18, 87).

[Yu+19] Dongjin Yu, Jiazha Yang, Xin Chen, and Jie Chen. “Detecting java code clones based on bytecode sequence alignment”. In: *IEEE Access* 7 (2019), pp. 22421–22433 (cit. on pp. 51, 80).

[YB07] Li Yujian and Liu Bo. “A normalized Levenshtein distance metric”. In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1091–1095 (cit. on pp. 39, 70).

[Zha+24] Qi Zhan, Xing Hu, Zhiyang Li, et al. “PS3: Precise Patch Presence Test based on Semantic Symbolic Signature”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–12 (cit. on p. 105).

[Zha+21] Xian Zhan, Lingling Fan, Sen Chen, et al. “Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1695–1707 (cit. on p. 106).

[Zha+19] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. “A large-scale empirical study of compiler errors in continuous integration”. In: *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2019, pp. 176–187 (cit. on p. 88).

[ZQ18] Hang Zhang and Zhiyun Qian. “Precise and accurate patch presence test for binaries”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 887–902 (cit. on p. 105).

[ZH18] Gang Zhao and Jeff Huang. “Deepsim: deep learning code functional similarity”. In: *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2018, pp. 141–151 (cit. on pp. 51, 81).

Webpages

[Aut25a] Checker Framework Authors. *The Checker Framework*. Accessed 2025-04-14. 2025. URL: `https://checkerframework.org` (cit. on p. 41).

[Aut25b] Project Lombok Authors. *Project Lombok*. Accessed 2025-04-14. 2025. URL: `https://projectlombok.org` (cit. on p. 41).

[Con25] OW2 Consortium. *ASM: Java bytecode manipulation and analysis framework*. Accessed 2025-14-04. 2025. URL: https://asm.ow2.io (cit. on pp. 39, 69).

[Cor05] Oracle Corporation. *JDK-6246854 : Unnecessary checkcast in generated code*. Accessed 2025-05-08. 2005. URL: https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6246854 (cit. on p. 63).

[Cor24a] Oracle Corporation. *Oracle Java SE 6 and JRockit End of Support*. Accessed 2025-05-22. 2024. URL: https://support.oracle.com/knowledge/Middleware/2244851_1.html (cit. on p. 71).

[Cor25a] Oracle Corporation. *JDK Release Notes*. Accessed 2025-05-08. 2025. URL: https://www.oracle.com/java/technologies/javase/jdk-relnotes-index.html (cit. on p. 56).

[Cor25b] Oracle Corporation. *Oracle JVM Specification - Chapter 4. The class File Format*. Accessed 2025-05-08. 2025. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.8 (cit. on p. 59).

[Cor24b] The MITRE Corporation. *About the CVE Program*. Accessed 2024-12-13. 2024. URL: https://www.cve.org/About/Overview (cit. on p. 12).

[Fou25a] The Eclipse Foundation. *Eclipse JDT (Java development tools)*. Accessed 2025-03-28. 2025. URL: https://projects.eclipse.org/projects/eclipse.jdt (cit. on pp. 4, 15, 53).

[Fou24a] Apache Software Foundation. *Apache Ant Project*. Accessed 2025-04-14. 2024. URL: https://ant.apache.org (cit. on p. 42).

[Fou24b] Apache Software Foundation. *Apache Maven Dependency Plugin*. Accessed 2025-04-14. 2024. URL: https://maven.apache.org/plugins/maven-dependency-plugin (cit. on pp. 36, 90).

[Fou25b] Apache Software Foundation. *Apache Maven Project*. Accessed 2025-04-14. 2025. URL: https://maven.apache.org (cit. on pp. 35, 42, 53).

[Fou21] OWASP Foundation. *OWASP Top Ten*. Accessed 2024-12-13. 2021. URL: https://owasp.org/www-project-top-ten/ (cit. on p. 1).

[Fou25c] OWASP Foundation. *CycloneDX*. Accessed 2025-11-07. 2025. URL: https://cyclonedx.org/ (cit. on p. 13).

[Fou23a] The Apache Software Foundation. *Apache Maven Resourcees Plugin - Filtering*. Accessed 2025-04-14. 2023. URL: https://maven.apache.org/plugins/maven-resources-plugin/examples/filter.html (cit. on p. 41).

[Fou25d] The Apache Software Foundation. *Apache Maven Assembly Plugin*. Accessed 2025-04-04. 2025. URL: https://maven.apache.org/plugins/maven-assembly-plugin/ (cit. on p. 10).

[Fou25e] The Apache Software Foundation. *Apache Maven Compiler Plugin*. Accessed 2025-04-28. 2025. URL: https://maven.apache.org/plugins/maven-compiler-plugin/ (cit. on p. 53).

[Fou25f] The Apache Software Foundation. *Apache Maven Compiler Plugin - Setting the –release of the Java Compiler*. Accessed 2025-04-28. 2025. URL: https://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-release.html (cit. on pp. 54, 75).

[Fou25g] The Apache Software Foundation. *Apache Maven Compiler Plugin - Setting the -source and -target of the Java Compiler*. Accessed 2025-04-28. 2025. URL: https://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html (cit. on pp. 54, 75).

[Fou25h] The Apache Software Foundation. *Apache Maven Compiler Plugin: Using Non-Javac Compilers*. Accessed 2025-04-28. 2025. URL: https://maven.apache.org/plugins/maven-compiler-plugin/non-javac-compilers.html (cit. on p. 53).

[Fou25i] The Apache Software Foundation. *Apache Maven Shade Plugin*. Accessed 2025-04-04. 2025. URL: https://maven.apache.org/plugins/maven-shade-plugin/ (cit. on pp. 10, 11).

[Fou23b] The Linux Foundation. *SPDX*. Accessed 2025-11-07. 2023. URL: https://spdx.dev/ (cit. on p. 13).

[Pro17] The GNU Project. *GNU Compiler for Java (GCJ)*. Accessed 2025-03-28. 2017. URL: https://gcc.gnu.org/wiki/GCJ (cit. on pp. 4, 15).

[Goo25a] Google. *Black Duck Documentation 2025.1 (Match Type*. Accessed 2025-04-15. 2025. URL: https://documentation.blackduck.com/bundle/bd-hub/page/InternalProjectVersions/dataTable.html#data_table__MatchType (cit. on p. 13).

[Goo25b] Google. *OSV Advisory*. Accessed 2025-04-15. 2025. URL: https://osv.dev/ (cit. on p. 11).

[Cor25c] Oracle Corporation. *The Java HotSpot Performance Engine Architecture*. Accessed 2025-04-04. 2025. URL: https://www.oracle.com/java/technologies/whitepaper.html (cit. on p. 15).

[Inc24a] Black Duck Software Inc. *Black Duck SCA*. Accessed 2024-12-13. 2024. URL: https://www.blackduck.com/software-composition-analysis-tools/black-duck-sca.html (cit. on pp. 1, 13, 85, 105).

[Inc24b] GitHub Inc. *GitHub Dependabot Alerts*. Accessed 2024-12-13. 2024. URL: https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts (cit. on pp. 13, 95, 105).

[Inc25a] GitHub Inc. *GitHub Advisory Database*. Accessed 2025-04-15. 2025. URL: https://github.com/advisories (cit. on pp. 11, 101).

[Inc24c] Google Inc. *OSV Scanner*. Accessed 2024-12-13. 2024. URL: https://google.github.io/osv-scanner/ (cit. on pp. 1, 13, 85, 95, 105).

[Inc25b] Google Inc. *OSV - Frequently Asked Questions*. Accessed 2025-06-23. 2025. URL: https://google.github.io/osv.dev/faq/#ive-found-something-wrong-with-the-data (cit. on p. 101).

[Inc25c] Gradle Inc. *Gradle Build Tool*. Accessed 2025-04-14. 2025. URL: https://gradle.org (cit. on pp. 42, 53, 79).

[Jav19] JavaParser.org. *About JavaParser*. Accessed 2025-04-14. 2019. URL: https://javaparser.org/about.html (cit. on p. 38).

[Jet23] JetBrains. *The State of Developer Ecosystem 2023*. Accessed 2025-05-23. 2023. URL: https://www.jetbrains.com/lp/devecosystem-2023/java/ (cit. on p. 79).

[IBM14] IBM. *IBM Jikes Compiler for the Java Language*. Accessed 2025-03-28. 2014. URL: https://sourceforge.net/projects/jikes/ (cit. on pp. 4, 15).

[Lab24] Endor Labs. *Endor Labs SCA*. Accessed 2024-12-13. 2024. URL: https://www.endorlabs.com/use-cases/reachability-based-sca (cit. on pp. 1, 13, 85, 105).

[Lim24] Snyk Limited. *Snyk Open Source SCA*. Accessed 2024-12-13. 2024. URL: https://snyk.io/product/open-source-security-management/ (cit. on pp. 1, 13, 85, 105).

[Lim25] Snyk Limited. *Snyk Vulnerability Database*. Accessed 2025-01-10. 2025. URL: https://security.snyk.io/ (cit. on p. 101).

[Men24] Mend.io. *Mend SCA*. Accessed 2024-12-13. 2024. URL: https://www.mend.io/sca/ (cit. on pp. 1, 13, 85, 105).

[Cor25d] Oracle Corporation. *The Java programming language Compiler Group*. Accessed 2025-03-28. 2025. URL: https://openjdk.org/groups/compiler/ (cit. on pp. 4, 15).

[Ora21a] Oracle. *Java Language Specification: Chapter 14. Blocks, Statements, and Patterns*. Accessed 2025-04-14. 2021. URL: https://docs.oracle.com/javase/specs/jls/se17/html/jls-14.html (cit. on p. 29).

[Ora21b] Oracle. *Java Language Specification: Chapter 15. Expressions*. Accessed 2025-04-14. 2021. URL: https://docs.oracle.com/javase/specs/jls/se17/html/jls-15.html (cit. on p. 28).

[Ora21c] Oracle. *Java Language Specification: Chapter 9. Interfaces*. Accessed 2025-04-14. 2021. URL: https://docs.oracle.com/javase/specs/jls/se17/html/jls-9.html (cit. on p. 27).

[Ora21d] Oracle. *JVM Specification: Chapter 4. The class File Format*. Accessed 2025-14-04. 2021. URL: https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-4.html (cit. on pp. 39, 75).

[Ora21e] Oracle. *Using the Keyword super*. Accessed 2025-04-14. 2021. URL: https://docs.oracle.com/javase/tutorial/java/IandI/super.html (cit. on p. 27).

[Ora25] Oracle. *JDK Releases*. Accessed 2025-03-28. 2025. URL: https://www.java.com/releases/ (cit. on p. 4).

[Cor25e] Oracle Corporation. *The Java Language Environment - Chapter 4: Architecture Neutral, Portable, and Robust*. Accessed 2025-04-04. 2025. URL: https://www.oracle.com/java/technologies/architecture-neutral-portable-robust.html (cit. on p. 15).

[OWA24] OWASP. *OWASP Dependency-Check*. Accessed 2024-12-13. 2024. URL: https://owasp.org/www-project-dependency-check/ (cit. on pp. 1, 13, 85, 95, 105).

[Pla23] Henrik Plate. *State of Dependency Management 2023*. Accessed 2024-12-13. 2023. URL: https://www.endorlabs.com/learn/state-of-dependency-management-2023 (cit. on p. 1).

[PM24] Henrik Plate and Darren Meyer. *2024 Dependency Management Report*. Accessed 2024-12-13. 2024. URL: https://www.endorlabs.com/lp/2024-dependency-management-report (cit. on pp. 1, 12, 13, 95).

[Rel24] New Relix. *2024 State of the Java Ecosystem Report*. Accessed 2025-05-22. 2024. URL: https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem (cit. on p. 70).

[Rer24] RertireJS. *Retire.js*. Accessed 2024-12-16. 2024. URL: https://retirejs.github.io/retire.js/ (cit. on pp. 1, 85).

[Ros13] John Rose. *JEP 181: Nest-Based Access Control*. Accessed 2025-05-08. 2013. URL: https://openjdk.org/jeps/181 (cit. on pp. 65, 66).

[SE21] SAP SE. *Eclipse Steady - Library Assessment*. Accessed 2025-06-25. 2021. URL: https://eclipse-steady.github.io/steady/user/manuals/library_assessment/ (cit. on pp. 14, 102).

[Ser25] Amazon Web Services. *Amazon Corretto 8*. Accessed 2025-05-22. 2025. URL: https://docs.aws.amazon.com/corretto/latest/corretto-8-ug/what-is-corretto-8.html (cit. on p. 71).

[Shi15] Aleksey Shipilev. *JEP 280: Indify String Concatenation*. Accessed 2025-05-08. 2015. URL: https://openjdk.org/jeps/280 (cit. on p. 64).

[Son25] Inc. Sonatype. *Maven Central Repository - Requirements*. Accessed 2025-07-01. 2025. URL: https://central.sonatype.org/publish/requirements/#supply-javadoc-and-sources (cit. on p. 96).

[ST25] National Institute of Standards and Technology. *National Vulnerability Database*. Accessed 2025-04-14. 2025. URL: https://nvd.nist.gov (cit. on pp. 11, 41, 101).

[Uni25] Secure Software Engineering Group - Paderborn University. *SootUp Documentation - Body Interceptors*. Accessed 2025-08-01. 2025. URL: https://soot-oss.github.io/SootUp/latest/bodyinterceptors/ (cit. on p. 57).

List of Figures

1.1 High-level overview of thesis contributions 5

2.1 Exemplary project timeline of a vulnerability affecting an OSS library . 11
2.2 Code property graph for the method shown in Listing 2.3 [Yam+14] . 18

3.1 Overview of JESS 22
3.2 Detailed overview of the compilation heuristic for compiling commit changes 33
3.3 Exemplary version history of an OSS project, illustrating a single commit selected for compilation 33
3.4 Overview of our evaluation setup for RQ1 and RQ2 36
3.5 Overview of our evaluation setup for RQ3 and RQ4 42

4.1 Setup to determine compilation differences 55
4.2 Overview of JNORM 57
4.3 Application of standardization (Jimple) 67
4.4 Overview of our experimental setup for bytecode normalization 69
4.5 Average prevalence of the individual compilation difference classes . . 78

5.1 Overview of JARALYZER 87
5.2 Overview of our experimental setup for bytecode-centric dependency scanning 94

List of Tables

3.1 Compilation success rates on popular GitHub Java projects using JESS . 37
3.2 Number of stubs generated by JESS for popular GitHub Java projects . 38
3.3 Similarity of the bytecode obtained via JESS to the original bytecode . 40
3.4 Compilation success rates when invoking build scripts on Project KB . . 43
3.5 Compilation success rates when using JESS on Project KB 44
3.6 Compilation success rates when using the custom compilation heuristic on Project KB 45

4.1 Compiler and target level configuration statistics in real-world Java projects 54
4.2 Difference classes on JDK version change 56
4.3 Difference classes on target level change 56
4.4 JDKs considered in our evaluation of bytecode normalization 71
4.5 Normalization results for different JDK versions. Percentage in brackets indicates the share of files and methods with compilation differences. . 72
4.6 Normalization results for different target levels of the JDK11 compiler. Percentage in brackets indicates the share of files and methods with compilation differences. 76

5.1 Detected CVE entries by each scanner in comparison to JARALYZER on modified dependency inclusions. Bold numbers indicate CVE entries missed by JARALYZER. 97
5.2 Disjoint CVE reports by JARALYZER and Eclipse Steady (unmodified benchmark) 100
5.3 Required scanning times for the unmodified benchmark containing 56 OSS dependencies 103

Listings

2.1 Java method example 16
2.2 Jimple method example 16
2.3 Example of a Java method for code property graph creation 17

3.1 Compilation target file 24
3.2 Compilation support file 24
3.3 Generated stub file 24
3.4 Ambiguous type stub 1 32
3.5 Ambiguous type stub 2 32

4.1 `toString` method invocation (Jimple) 60
4.2 Method reference operator usage (Jimple) 61
4.3 Buffer method invocation (Jimple) 62
4.4 JDK11 enum definition 64
4.5 JDK17 enum definition 64
4.6 String concatenation (Jimple) 65

5.1 Original class 93
5.2 Re-packaged class 93