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

Secure Use of Open-Source Software – A Systematic Study and Techniques for Java

Andreas Peter Dann

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisors

Prof. Dr. Eric Bodden
Prof. Dr. Ben Hermann

Paderborn, December 8, 2023

Andreas Peter Dann
Secure Use of Open-Source Software – A Systematic Study and Techniques for Java
Dissertation, December 8, 2023
Advisors: Prof. Dr. Eric Bodden and Prof. Dr. Ben Hermann

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

An established practice in modern software development is the use of open-source libraries and frameworks. Even for commercial applications, up to 75% of the application's codebase comes from open-source software. The inclusion of open-source software enables developers to reuse established and well-tested functions in their own applications, increasing development speed and reducing cost. The emergence of popular, public open-source software repositories and dependency-management tools for several programming languages such as Java, JavaScript, Python, and .NET has facilitated this trend.

Using open-source software also involves the risk of including known vulnerabilities. Developers must regularly check if vulnerabilities in used open-source software have been published and take appropriate actions like updating or replacing the known-vulnerable artifact. However, several studies show that developers are often unaware of the fact that their software includes open-source software with known vulnerabilities. Furthermore, the studies show that developers are hesitant to update even known-vulnerable software since they are afraid that an update may introduce unexpected regressions, breaking their application.

In this thesis, we propose methods and tools to cope with the challenges that developers face when including open-source software in their applications. We conducted a case study on 7,024 software projects developed at SAP to investigate problems developers and tools face for detecting known vulnerabilities in included open-source software. Our study shows that developers regularly patch or bundle open-source software to adapt it to their needs or to ease distribution. These modifications pose a challenge for the automatic tool-based detection of vulnerabilities in used open-software software since vulnerability scanners fail to analyze such modified software. To cope with this challenge, we present SootDiff, an approach that enables the detection of open-source software despite applied modifications.

To support developers to update vulnerable or outdated open-source software with minimal risk of breaking their application, we introduce our tool UpCy.
UpCy automatically identifies update(s) that only introduce minimal incompatibil-

ities by inspecting how the software uses the open-source artifact that should be updated. Our evaluation shows that in 70.1% cases where a naïve update leads to incompatibilities, UpCy can effectively suggest updates with no incompatibilities.

Nevertheless, vulnerabilities in the included open-source software are an unfortunate truth. A library that is considered secure today could become exploitable due to a newly discovered vulnerability tomorrow. Thus, it is desirable to design an application to restrict vulnerabilities in one component, like an included open-source library, from affecting others. In this thesis, we study the Java module system and its feature of strong encapsulation, which aims to prevent unintended access to security-sensitive functions or data, regarding its capabilities to enable the isolation of included third-party software. For this purpose, we conduct a case study on a modularized version of the popular open-source web server Apache Tomcat. Our study shows that the module system can partially prevent access to sensitive entities within a module. However, it requires a secure redesign of the application. To support developers with a secure, modular design, we introduce ModGuard, a static analysis to detect unintended data flows that allow the access or manipulation of sensitive entities. Based on our study, we discuss what security measures the module system is missing to achieve a secure integration of open-source libraries and frameworks.
Our studies reveal weak spots in current approaches for detecting and updating known-vulnerable open-source software. The tools SootDiff, UpCy, and ModGuard show that automated approaches can move the secure use of open-source software forward.

Zusammenfassung

In der modernen Softwareentwicklung ist die Einbindung von Open-Source Bibliotheken und Frameworks gängige Praxis. So bestehen selbst kommerzielle Anwendungen mittlerweile bis zu 75% aus Open-Source Software. Die Nutzung von Open-Source Software erlaubt es EntwicklerInnen, etablierte und erprobte Funktionen in den eigenen Anwendungen zu nutzen, so dass sie schnell und kosteneffizient Software programmieren können. Verstärkt hat sich dieser Trend durch das Aufkommen öffentlich zugänglicher Repositorien und Tools zur Verwaltung eingebundener Open-Source Software für viele Programmiersprachen, wie Java, JavaScript, Python und .NET.

Ein Risiko bei der Integration von Open-Source Software ist die Einbindung von Software mit bekannten Sicherheitslücken. EntwicklerInnen müssen regelmäßig überprüfen, ob für die eingesetzte Open-Source Software Sicherheitslücken veröffentlicht wurden und gegebenenfalls betroffene Bibliotheken updaten oder austauschen. Allerdings zeigen verschiedene Studien, dass ihnen häufig nicht bewusst ist, dass sie schwachstellenbehaftete Open-Source Software einsetzen. Weiterhin zeigen die Studien, dass EntwicklerInnen sogar zögern, schwachstellenbehaftete Software zu updaten, da sie befürchten dadurch Fehler in die eigene Anwendung einzubauen.

In dieser Arbeit adressieren wir die Herausforderungen bei der sicheren Verwendung von Open-Source Software. Hierzu führen wir eine Fallstudie an 7.024 von SAP entwickelten Projekten durch, in der wir analysieren, wie EntwicklerInnen Open-Source Software in die eigene Anwendung einbinden und welche Probleme Sicherheitstools bei der Erkennung von bekannten Sicherheitslücken in der eingesetzten Open-Source Software haben. Unsere Fallstudie zeigt, dass EntwicklerInnen bei der Integration von Open-Source Software Modifikationen vornehmen, um sie an ihre eigenen Bedürfnisse anzupassen oder ihre Verteilung zu erleichtern. Unsere Studie zeigt außerdem, dass die derzeit verfügbaren Open-Source Schwachstellenscanner Schwierigkeiten haben, schwachstellenbehaftete Open-Source Software zuverlässig zu erkennen, wenn diese zuvor modifiziert wurden. Um diesem Problem zu begegnen, stellen wir in dieser Arbeit unser Werkzeug SootDiff vor. SootDiff nutzt statische Analysetechniken, um Open-Source Software trotz Modifikationen zuverlässig und automatisiert (wieder-)zuerkennen.

Um das Risiko zu verringern bei einem Update der eingebundenen Open-Source Software Inkompatibilitäten in die eigene Anwendung einzubauen, haben wir das Werkzeug UpCy entworfen. UpCy schlägt automatisch kompatible Updates für veraltete oder schwachstellenbehaftete Open-Source Software vor, indem es analysiert,

wie die schwachstellenbehafteten Bibliotheken genutzt werden. Unsere Evaluation zeigt, dass UpCy in 70.1% der Fälle Updates vorschlagen kann, die keine Inkompatibilitäten aufweisen.

Ein Problem bei der Nutzung von Open-Source Software besteht darin, dass in der Zukunft Sicherheitslücken in Bibliotheken und Frameworks entdeckt werden können, die heute noch als sicher gelten. Daher sollte Software so entworfen werden, dass Sicherheitslücken innerhalb einer Komponente keine Auswirkung auf andere Komponenten haben. Hierzu untersuchen wir, ob das Java Modulsystem, dessen Ziel die Kapselung sicherheitskritischer Daten und Funktionen innerhalb eines Modules ist, die sichere Integration von Open-Source Software ermöglichen kann. Wir untersuchen das Potential des Java Modulsystems in einer Fallstudie an dem Webserver Tomcat. Die Fallstudie zeigt, dass das Modulsystem den Zugriff auf sicherheitsrelevante Klassen und Methoden in einem Modul teilweise verhindern kann. Allerdings erfordert dies Änderungen an der Architektur einer Anwendung. Um EntwicklerInnen beim sicheren Entwurf von Modulen zu unterstützen, haben wir das Werkzeug ModGuard entwickelt. ModGuard detektiert unerwünschte Datenflüsse, die den Zugriff auf oder die Manipulation von sicherheitsrelevanten Daten erlauben. Basierend auf der Tomcat Fallstudie diskutieren wir, welche Mechanismen zusätzlich zur Modularisierung erforderlich sind, um Bibliotheken voneinander und von der restlichen Software zu isolieren.

Unsere Fallstudien decken die Schwächen auf, die derzeitige Werkzeuge bei der zuverlässigen Erkennung schwachstellenbehafteter Open-Source Software haben. Die Werkzeuge SootDiff, UpCy und ModGuard zeigen, dass automatisierte Ansätze einen Beitrag dazu leisten können, die sichere Nutzung von Open-Source Software voranzutreiben.

Acknowledgments

First and foremost, I would like to thank my advisors Eric Bodden and Ben Hermann. I had the great luck of being supervised and receiving feedback from not only one—but two experienced and skilled researchers. Both have always been available and approachable whenever I had questions or doubts concerning my work. This thesis would not have been possible without their guidance, contributions, and continuous support throughout my time at the university and afterward. My research and I greatly benefited from their excellent scientific expertise and constructive feedback. I also would like to thank Dr. Serena Elisa Ponta, Prof. Dr. Yasemin Acar, and Dr. Simon Oberthür for joining the examination committee.

During my Ph.D., I had the opportunity and pleasure to intern at SAP Security Research in Antibes, France. In my internship, I gained an entirely new perspective on my research. Interesting discussions with experts and the Steady team provided me with new insights and triggered new ideas, which helped to advance my research. I want to thank all my colleagues at SAP Security Research who made the internship an exceptionally great experience and opportunity. In particular, I want to thank Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta for guiding my work, the great work atmosphere, and fruitful discussions.

Further on, I want to thank my colleagues from Paderborn University and Fraunhofer IEM for the helpful, constructive, and friendly working environment.

Finally, I want to thank my family and partner for their incredible support and motivation, and the many friends with whom I have had great times within the last few years.

Publications

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

- 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 (Sept. 2022), pp. 3613–3625. DOI: `10.1109/TSE.2021.3101739`
  Parts of Chapter 3 are taken from this paper.
- 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*. SOAP 2019. Phoenix, AZ, USA: ACM, June 2019, pp. 14–19. DOI: `10.1145/3315568.3329966`
  The last sections of Chapter 3 are taken directly or with modifications from the paper.
- Andreas Dann, Ben Hermann, and Eric Bodden. “UPCY: Safely Updating Outdated Dependencies”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. ICSE ’23. Melbourne, Australia: IEEE, May 2023, pp. 233–244. DOI: `10.1109/icse48619.2023.00031`
  Large parts of Chapter 4 are taken from this publication.
- Andreas Dann, Ben Hermann, and Eric Bodden. “ModGuard: Identifying Integrity & Confidentiality Violations in Java Modules”. In: *IEEE Transactions on Software Engineering* 47.8 (Aug. 2021), pp. 1656–1667. DOI: `10.1109/TSE.2019.2931331`
  Parts of Chapter 5 are taken from this paper.

The research prototypes and data sets that we created in the course of these publications and the thesis are publicly available to enable other researchers to validate our results and to build further research upon them. 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 Generality of Contributions 7
1.4 Thesis Structure 8

2 Background **11**
2.1 Terminology & Dependency Management in Java 11
2.2 Dependency Management in Other Programming Languages 14

3 Systematic Study on the Usage of Open-Source Software and Challenges for Their Detection **19**
3.1 Strategies for Detecting Vulnerabilities in Open-Source Software . . . 20
3.2 Study Design 22
3.2.1 Research Questions 22
3.2.2 Study Objects & Methodology 24
3.3 Use of Open-Source Software at SAP 28
3.3.1 RQ1: What Are Practices for Using Open-Source Software at SAP? 28
3.3.2 RQ2: What Vulnerabilities Affect the 20 Most-Used Dependencies? 30
3.3.3 RQ3: How Do Developers Include Open-Source Software? . . 35
3.4 Prevalence & Impact of Modified Open-Source Software 37
3.4.1 RQ4: How Prominent Are the Modifications Outside SAP? . . 37
3.4.2 RQ5: What Is the Impact of the Modifications on Vulnerability Scanners? 40
3.5 Study Summary 46
3.6 Threats to Validity 47
3.7 Achilles: Test Suite for Detecting Modified Open-Source Software . . 49
3.7.1 Diverse Real-World Applications 50
3.7.2 Detecting Vulnerable Open-Source Software 50
3.7.3 Automation and Ease of Use 51
3.7.4 Organization and Distribution 53

3.8 SootDiff: An Approach for Identifying Modified Open-Source Software 54
3.8.1 Dissimilarities Introduced by Java Compilers 55
3.8.2 Jimple: Intermediate Bytecode Representation 58
3.8.3 Compare Modified Bytecode 59
3.8.4 Evaluation 61
3.9 Related Work 63
3.9.1 Case Studies: Use of Vulnerable Open-Source Software 64
3.9.2 Test Suites: Vulnerabilities in Open-Source Software 65
3.9.3 Code Clone Detection 66
3.10 Conclusion 68

4 An Automated Approach for Safely Updating Included Open-Source Software 69
4.1 Safe Backward Compatible Updates 70
4.1.1 Dependency Graph Updates 70
4.1.2 Source and Binary Compatibility 71
4.1.3 Semantic Compatibility 74
4.1.4 Blossom Compatibility 74
4.2 UpCy: Identify Safe Backward Compatible Updates 75
4.2.1 Algorithm 75
4.2.2 Graph Database of the Maven Central Repository 82
4.3 Evaluation 86
4.3.1 Research Questions 86
4.3.2 Study Objects & Methodology 86
4.3.3 Results 88
4.4 Threats to Validity 94
4.4.1 Finding Compatible Updates with UpCy 94
4.4.2 Evaluation 95
4.5 Related Work 96
4.5.1 Studies: How Developers Update (Vulnerable) Dependencies . 96
4.5.2 Update Compatibility Analysis 97
4.5.3 Repository Dependency Graphs 97
4.6 Conclusion 98

5 Securely Integrating Open-Source Software with Java's Module System 101
5.1 Java's Security Architecture & Module System 104
5.1.1 Java 1.2 Security Model 104
5.1.2 The Java Platform Module System 108
5.1.3 Motivating Example of Sensitive Entities Escaping a Module . 111

5.1.4 Excursion: The OSGi Platform . . . 112
5.2 Precisely Defining a Module's Entry Points . . . 113
5.2.1 Explicitly vs. Implicitly Reachable Entry Points . . . 113
5.2.2 Logic-based Specification of the Entry-Point Model . . . 114
5.2.3 Limitations of the Entry-Point Model . . . 119
5.3 ModGuard: Identify Confidentiality or Integrity Violations of Modules 120
5.3.1 Algorithm . . . 121
5.3.2 Limitations . . . 125
5.4 Evaluation . . . 126
5.4.1 Research Questions . . . 126
5.4.2 Study Objects . . . 127
5.4.3 Results . . . 132
5.4.4 Case Study: CVE-2017-5648 in Tomcat (modules) . . . 136
5.5 Limitations of Modules for the Secure Integration of Open-Source Software . . . 138
5.6 Related Work . . . 142
5.6.1 Sandboxes for Native Code . . . 142
5.6.2 Encapsulation and Isolation of OSGi Bundles . . . 143
5.6.3 Escape Analysis . . . 145
5.6.4 Information-Flow Control . . . 146
5.7 Conclusion . . . 147

6 Conclusion and Outlook **149**

Implementations and Data **153**

Bibliography **155**

List of Figures **173**

List of Tables **175**

Listings **177**

1 Introduction

Software has become ubiquitous in our everyday life. From small mobile devices over household appliances to industrial plants - any device that requires electricity is likely to comprise a microchip that runs software. Consequently, software has become a main driver for a company's innovation, and as every company becomes a software company, software development becomes indispensable to nearly every business in the world. The main cost of software development is human resources, and thus developer productivity is tightly coupled with costs. Any measures that reduce the time to develop and improve software quality are of huge economic and social benefit [Lim94; Hei+11; SE13].

To decrease development time and increase quality, the inclusion of Open-Source Software (OSS) is an established practice in software development, even for commercial applications, as much as 75% of the code comes from OSS [Pit16; Pla23; Hei+11; BHD12; Bav+15]. By incorporating functionalities from community-developed, well-tested, established, open-source software, software vendors can reduce development and maintenance costs dramatically.

The use of OSS has been facilitated further by the emergence of build-automation and dependency-management tools that make it easy for developers to include and distribute libraries and frameworks from public software repositories into their own software projects. Nowadays, such build-automation and dependency-management tools as well as public repositories exist for most programming languages [DMG19]. Java has the tools Maven and Gradle and the repository Maven Central, with more than 10.1 million open-source artifacts in 2022, Python has pip, which uses the popular repository PyPI with more than 4.12 million artifacts, and JavaScript has yarn and npm with the NPM registry, etc.

Despite its benefits, the use of OSS comes at a cost. Open-source software does also contain bugs and security vulnerabilities [Kul+18]. An analysis by Synopsys of more than 1000 commercial applications showed that 96% of the applications include OSS, and more than 67% of the applications include vulnerable OSS with, on average, 22 individual vulnerabilities [Pit16]. Crucially, vulnerabilities in widely-used OSS, like Jackson [Cau17], Log4j [NVD21], Apache Commons [NVD22], or Apache Struts [Kre18], already proved to have serious consequences.

An (in)famous example is the Log4Shell vulnerability [NVD21], which was discovered in the popular library Log4j in December 2021. The vulnerability allowed an attacker to execute arbitrary code on a vulnerable system. As a result, widely popular services such as Steam, Apple iCloud, and Minecraft have been impacted [Yah21]. Another prominent example is the Equifax breach [For17; Kre18; NVD17a], caused by vulnerability CVE-2017-5638 in the open-source framework Apache Struts2. The vulnerability also allowed the execution of arbitrary code and caused the breach of sensitive personal information of more than 130 million Americans, e.g., name, address, and social security number [Kre18].

To prevent the exploitation of vulnerabilities in included OSS, developers must—as soon as the vulnerability has been reported—identify the vulnerable OSS and either migrate to a different library or update to a non-vulnerable version. However, neither migration nor updating is trivial, as developers must ensure that they do not introduce regressions into the existing application. Given the large percentage of OSS in modern software, finding known-vulnerable OSS, updating them to a non-vulnerable version, and testing if the update introduces any regressions is costly and manually (almost) infeasible. Consequently, several studies [DH22; HG22; Pra+21; Pas+22; Kul+18; Wan+20; Der+17; MP17; Bog+16; BKH15] show that most developers are often unaware that they use know-vulnerable OSS or are hesitant to update even known-vulnerable OSS, leaving software applications vulnerable for extended periods.

1.1 Research Challenges

Complementary approaches exist to mitigate the impact and exploitation of included, known-vulnerable open-source libraries and frameworks.

One approach to reduce the impact is to find and fix the vulnerable code in the OSS itself. Although developers typically do not have direct control over the OSS's source code, they can relatively easily create their own fork and fix the vulnerable code themselves. While this requires manual effort and security skills, forking can be an option if the maintenance of a particular open-source artifact has been halted and the migration to a different artifact is not possible without major effort.

Another approach to reduce the vulnerabilities' impact is applying software diversification techniques to the OSSs' source or binary code. By using software diversification, the implementations are made more dissimilar to one another while preserving their functionality to reduce the risk of sharing the same vulnerabilities or defects [Har22].

A further approach is migrating the complete application to another OSS with the same functionalities. Depending on the similarity of their functionality and interface, the effort for the migration varies heavily, as parts of the application may require major adaptions.

An established and widely-used approach, which has been well-studied in literature [DJB14; Bog+16; HG22; Bav+15; Der+17; RVV14; Pra+21; Wan+20], is to update the vulnerable OSS to a newer version with the vulnerability fixed. In contrast to migration, the interfaces between updates of the same library or framework are expected to be stable, and thus the effort for adapting the project should be smaller; although studies show that updates regularly introduce breaking changes [DJB14; Bog+16; HG22].

Another approach is to limit the impact of vulnerabilities in OSS by executing them in isolated environments, so-called sandboxes [STM10; Cok+15]. A sandbox limits access to security-critical functions and data, preventing the OSS from executing malicious or dangerous behavior.

In this thesis, we focus on the approaches: (i) finding and updating known-vulnerable or outdated OSS, and (ii) a constructive approach for limiting the impact of known-vulnerable OSS by a secure integration. To do so, developers are faced with the following challenges:

Challenge 1: Identifying Vulnerable Open-Source Software To detect vulnerable OSS in software projects, research and industry have developed several open-source vulnerability scanners, such as the open-source tools OWASP Dependency-Check (DepCheck) [OWA20] and Eclipse Steady [Ecl20], the free tool GitHub Security Alerts [Git20], and commercial tools such as Snyk [Sny23], Synopsys [Syn23], or Mend.io [Men23a]. Since vulnerabilities in OSS pose a high risk, scanners should detect them with high precision and recall. However, developers and vendors frequently fork, patch, re-compile, re-bundle, or re-package existing OSS to add new features, fix bugs, or fix security issues. As a result, developers do not always include the original open-source artifact from the public software repository but include the artifact in modified form.

Modifications decrease the recall of detecting vulnerabilities in used OSS. Open-source vulnerability scanners report publicly announced vulnerabilities—like Common Vulnerabilities and Exposures (CVE)s—using a database like the National Vulnerability Database (NVD). The NVD typically maps a CVE to the unmodified, original OSS artifact only. Thus, CVEs in OSS artifacts with a modified file(-name), version, or code are not always discovered. As undetected vulnerabilities in used (and modified) OSS present a security threat to the software, it is necessary to clarify how widespread such modifications are and their effect is on the precision and recall of vulnerability scanners. Further, tools and techniques are needed to detect vulnerabilities in modified open-source artifacts.

Challenge 2: Updating Open-Source Software without Introducing Incompatibilities
Tools like Greenkeeper [The22], Dependabot [Git22], and Renovate [Men23b] create automated pull requests for updating vulnerable or outdated OSS. However, several studies [DH22; HG22; Pra+21; Pas+22; Kul+18; Wan+20; Der+17; MP17; Bog+16] show that most developers perceive these tools as unreliable and hesitate to integrate the automated update suggestions, as they are afraid of introducing unexpected regressions or unintended side effects [HG22; MP17].

A study by Hejderup et al. [HG22] shows that these concerns are justified. Since these tools rely on the project's test coverage of the used OSS, which is oftentimes low, they can only detect conflicting updates and breaking changes to a small extent [Bog+16; HG22; Kul+18]. Furthermore, these tools do not check if the updated OSS is compatible with other libraries and frameworks in the project. Thus, there is a high risk that incompatibilities in the update remain undiscovered until production. Manually discovering, debugging, and resolving such incompatibilities is cumbersome, and thus the main reason that discourages developers from updating [Bog+16; Kul+18]. To facilitate updating of vulnerable OSS, developers need effective tools that help them to minimize the updating effort and reduce the risk of introducing unwanted regressions and incompatibilities.

Challenge 3: Securely Integrating Open-Source Software Even regularly updating all included OSS to the latest version does not protect from inadvertent vulnerabilities. The Log4Shell [NVD21] and other vulnerabilities showed that 0-day vulnerabilities could occur in any OSS. Thus, it is desirable to design the application and include OSS in such a way that restricts a vulnerability in one component from affecting others.

A mechanism to limit the capabilities of potentially vulnerable code is to run it inside a sandbox, isolated from the rest of the application. However, with the release of Java 17, Java's built-in Security Manager, a cornerstone of Java's sandbox mechanism, has been marked for removal without any replacement or alternative [Ora22; Ora21]. Given Java's module system, which has been introduced with Java 9, and its strong encapsulation for confining security-sensitive entities, it is an open question if modules can be used—at least partly—to isolate components from each other and prevent the leaking of sensitive entities, such as encryption keys, from the application to included OSS [Ora22; Ora21; Ora23b].

Developers need an overview of the module system's capabilities for the constructive, secure design of the application and integration of OSS. Further, developers need tools that help them securely design the application so that security-critical entities are properly confined and encapsulated, preventing access from integrated potentially vulnerable OSS. Tools and concepts are needed to precisely determine which module types can become accessible to the outside to facilitate the isolation of modules in an application and to assess which parts of an application can become accessible to a potentially vulnerable module.

1.2 Thesis Contributions

This thesis contributes novel techniques for mitigating the challenges developers in the Java ecosystem face when maintaining included OSS. We present techniques and prototype implementations for addressing the challenges described above: identifying (modified) OSS, supporting the update of OSS, and evaluating to what extent modules can be used to limit the impact of vulnerable OSS. The contributions made by this thesis are as follows.

Contribution 1: Study on the Use of Vulnerable Open-Source Software In the first contribution, we identify and detect development practices that can decrease the effectiveness of open-source vulnerability scanners to identify included, known-vulnerable OSS (*Challenge 1*). Although several vulnerability scanners appeared in the last decade, no case study exists investigating the impact of typical development practices, like forking, patching, and re-bundling, on their precision and recall. Through an empirical study on 7,024 Java projects developed at SAP, we

study *(i)* types of modifications that developers apply on used open-source artifacts and *(ii)* the impact of these modifications on the effectiveness of vulnerability scanners.

The study identifies four types of modifications: re-compilation, re-bundling, metadata-removal, and re-packaging. To assess the impact of these modifications on the precision of open-source vulnerability scanners, we introduce the novel test suite Achilles for replicating modifications on open-source artifacts and evaluate the precision of six (open-source and commercial) scanners w.r.t. these modifications. The evaluation of the scanners with our test suite shows that all scanners struggle to deal with modified OSS.

To cope with this limitation, we present SootDiff, a tool to identify open-source artifacts even if they have been modified. SootDiff uses static analysis and Soot's intermediate representation Jimple in combination with code clone detection techniques to reduce dissimilarities introduced by re-compilation and re-bundling, and thus can help to identify modifications.

Contribution 2: An Automated Approach for Updating Open-Source Software without Introducing Incompatibilities In the second contribution, we present UpCy. UpCy is a tool to help developers updating an included open-source artifact (*Challenge 2*) by automatically suggesting a list of update steps that introduce minimal or no regressions. Current automated approaches such as Dependabot follow a naïve update approach and propose an update for the outdated or vulnerable open-source artifact only but ignore compatibility with other libraries, frameworks, and the application code. Consequently, several studies [MP17; HG22; Bog+16; BKH15] show that developers mistrust the automatic pull requests these automated approaches create.

To alleviate this situation, UpCy attempts to find a list of update steps with minimal incompatibilities to other artifacts by building a complete dependency graph—a complete representation of all open-source artifacts in the project and their relationships—applying the min-(s,t)-cut algorithm to the dependency graph, and leveraging a graph database of Maven Central to identify updates that are compatible with each other.

We evaluate UpCy on 29,698 updates in 380 well-tested, open-source Java projects and compare it with the naïve updates applied by state-of-the-art approaches. Our evaluation shows that UpCy can effectively find updates with fewer incompatibilities, and even 70.1% of the generated updates have zero incompatibilities.

Contribution 3: Analysis of the Security Implications of the Java Module System and a Static-Analysis for Modules In the third contribution, we analyze the security implications of Java's module system [Ora15b; Ora23b] w.r.t. its capabilities to confine internal types and data, enabling a secure integration of OSS (*Challenge 3*). To do so, we present ModGuard. ModGuard is a static code analysis to check the isolation of a module and what types and data of the module can eventually become reachable from the outside. ModGuard is based on our formal definition of a module's entry points: all methods and types that can become reachable outside their declaring module.

We evaluate ModGuard in a case study of the Apache Tomcat web server, and discuss what developers need to do to benefit from the module system's encapsulation guarantees. Further, we discuss the shortcomings and limitations of the module system for securely integrating OSS. Our discussion points out that further work is needed to isolate included OSS and that modules can only partially limit inadvertent vulnerabilities.

1.3 Generality of Contributions

In this thesis, we use the programming language Java and the build-automation and dependency-management tool Maven as examples in our study and for evaluating our tools. However, the study's results and the developed concepts can be adapted to other programming languages and build-automation tools.

For example, the results of our study (*Contribution 1*), that known-vulnerable OSS is commonly used in commercial and open-source software projects, are equally reported for other programming languages and dependency-management tools by several studies [Pra+21; HVG18; PVM20; Wan+20; HG22; MNT20]. Further, the identified challenges for detecting known-vulnerable OSS also apply to other build-automation and dependency-management tools like npm or pip. In contrast to Java dependencies, which are compiled to platform-independent bytecode (cf. Section 5.1), npm and pip artifacts are distributed as source code (cf. Chapter 2). Thus, the identification of vulnerable code has to be done on source code but not on compiled bytecode. A study by Wyss et al. [WDD22] and a study by Latendresse et al. [Lat+23] show that also npm packages contain modified source-code clones of their (transitive) dependencies without any indication to the original package. Such clones have been subject to shrinking, syntax translation, e.g., transpilation of CommonJS to ES6, and dead code elimination using tree-shaking.

The detection of such modified clones is incompatible with source-code clone detection techniques, which focus on the deletion and insertion of source-code statements, and thus requires new mechanisms [WDD22]. For languages that are compiled to platform-dependent binaries, like C and C++, the detection of code clones also has to cope with incomparable instruction sets of different architectures and alternative compiling configurations, as discussed in the approaches by Hu et al. [Hu+17] and Tang et al. [Tan+23].

The concepts and algorithms behind UpCy for suggesting updates with minimal incompatibilities (*Contribution 2*) can be directly applied to programming languages and tools that use a global dependency graph and apply similar conflict resolution mechanisms as Maven, e.g., pip permits only a single, non-conflicting version of a dependency in a project, too. UpCy's algorithms depend on creating a dependency graph and call graph for a software project only; both can be constructed for most programming languages. Even if the call graph is unsound or cannot be constructed, the algorithms can solely work on the dependency graph, though it decreases its precision. For dependency-management tools like npm, which maintain a complete dependency graph per dependency and permit conflicting dependency versions, UpCy's algorithms must be adapted.

Our evaluation of a programming language's module concept (*Contribution 3*) and its use for limiting the impact of vulnerable OSS also applies to other module systems. The formalized entry-point model and the module escape analysis can be transferred to other programming languages that implement a module concept.

1.4 Thesis Structure

The remainder of this thesis is structured as follows: In Chapter 2, we introduce the terminology and concepts necessary to understand the thesis, including the build-automation and dependency-management tools Maven for the programming language Java, pip for Python, npm for Node.js/JavaScript, and Conan for C/C++.

In Chapter 3, we present an in-depth case study regarding the use of open-source software in commercial and open-source projects (*Contribution 1*). From this study, we derive challenges that state-of-the-art and future tools for detecting known-vulnerable OSS in applications must consider. Later, we discuss the shortcomings of current tools and present SootDiff, a first approach to cope with the challenges.

In Chapter 4, we present UpCy, an approach to automatically find updates of vulnerable or outdated OSS with minimal incompatibilities (*Contribution 2*). UpCy aims to find updates that reduce the risk of introducing unexpected regression into an application.

In Chapter 5, we present ModGuard, an analysis to detect unintended data flows over which security-sensitive entities become accessible outside their declaring module (*Contribution 3*). The analysis supports the development of modules to confine internal types and data effectively. The analysis is based on a formal definition of a module's entry point that we developed. Further, we discuss the limitations of the module system for the secure integration of OSS.

In Chapter 6, we conclude the thesis with a summary of our contributions and present an outlook for the secure usage of OSS.

2 Background

In this chapter, we discuss the necessary background and terminology to understand this thesis. Throughout this thesis, we rely on the established terminology used by the build-automation tool Maven.
We start with introducing the build-automation and dependency-management tool Maven and the public open-source software repository Maven Central [Son22], in Section 2.1.
We follow with a detailed explanation of Maven's dependency resolution mechanism for resolving and choosing which artifacts will be included in the project. Finally, we compare Maven and its dependency resolution mechanisms with popular build-automation tools for other languages, pip for Python, npm for Node.js/JavaScript, and Conan for C/C++ in Section 2.2, to clarify to what extent our concepts generalize.

2.1 Terminology & Dependency Management in Java

We use the term *software artifact* for a software library or framework that is a separately distributed software component that consists of a logically grouped set of classes, functions, and resources. A *dependency* is a software artifact that is used by another artifact (the dependent artifact) or project. In Java, a software artifact is commonly distributed as a JAR file. JAR files assemble packages, compiled bytecode classes, native code, and other resources.

Build-automation and dependency-management tools like Maven [Apa23b], Gradle [Gra23], and Ant [Apa22] combine two tasks: They provide tooling to ease the building (compilation, testing, packaging, etc.) of a project and the management (distribution and inclusion) of Open-Source Software (OSS) artifacts as dependencies from private or public artifact repositories. Maven Central Java's the most popular public repository, with over 10.1 million open-source artifacts and millions of downloads per week in 2022 [Son20; Apa23b]. Java's tools use a similar syntax for declaring dependencies and pulling JARs from repositories; for the remainder, we use Maven's syntax as an example.

Listing 2.1 shows an example of a project's configuration file `pom.xml` declaring a dependency on a vulnerable version of Apache Struts2, which caused the Equifax breach. To declare a dependency, developers specify the artifact's unique identifier as the triple: *groupId* identifying the vendor, *artifactId* identifying the product, and *version*. This identifier triple is called *GAV* (groupId, artifactId, version).

```
1  <dependency>
2    <groupId>org.apache.struts</groupId>
3    <artifactId>struts2-core</artifactId>
4    <version>2.3.5</version>
5    <scope>compile</scope>
6  </dependency>
```

Listing 2.1: Example declaration of an artifact as a dependency in a project's `pom.xml` for the build-automation and dependency-management tool Maven.

By declaring a dependency's *scope*, developers can also specify when a dependency is included: during development or during runtime. Accordingly, we distinguish between *release dependencies*, shipped with the application and included during runtime, and *development-only dependencies*, only used during development, e.g., for testing or code generation. Release dependencies have the scopes: *compile* (the artifact is required for compilation and execution), *runtime* (the artifact is required during runtime only), *provided* (the artifact is expected to be provided by the execution environment, e.g., the Java runtime or a web server running the application) or *system* (the artifact is required for compilation and during runtime, but is not managed by Maven). Meanwhile development-only dependencies, e.g., the JUnit testing framework, have the scope *test* (the artifact is only available during the execution of test cases). Since only release dependencies are shipped with an application, vulnerabilities in development-only dependencies are not exploitable in production.

Based on the dependencies declared in the configuration file, the build-automation and dependency-management tool automatically downloads, includes, and configures the artifacts as project dependencies. In addition to the declared artifacts, the build-automation tool transitively includes artifacts that the declared artifacts themselves depend on (and also the artifacts on which those newly introduced dependencies depend on and so on ...). To resolve all dependencies, the build-automation tool constructs the project's dependency graph, which specifies all dependencies and their relations, and includes the dependencies for compilation and execution.

Figure 2.1 shows an example dependency graph. In a dependency graph, nodes represent artifacts—the root node is the project itself, and the directed edges connect to dependent artifacts. A dependency is called a *direct* dependency of an artifact

node n if the dependency and n are connected through a path of length one. A dependency connected through a longer path is called a *transitive* dependency of the artifact node n.

Although often visualized as a tree, strictly mathematically speaking, a project's dependency relations form a graph: a single dependency node can have multiple predecessors since multiple dependencies may be dependent on the same artifact, so-called duplicates, or on conflicting versions of the same artifact, so-called conflicts. Duplicate dependencies occur when the project or dependencies depend on the same artifact; for instance, in Figure 2.1, w is a duplicate dependency because it is included by t and z. Conflicting dependencies occur when the project or dependencies depend on the same artifact but in different versions.

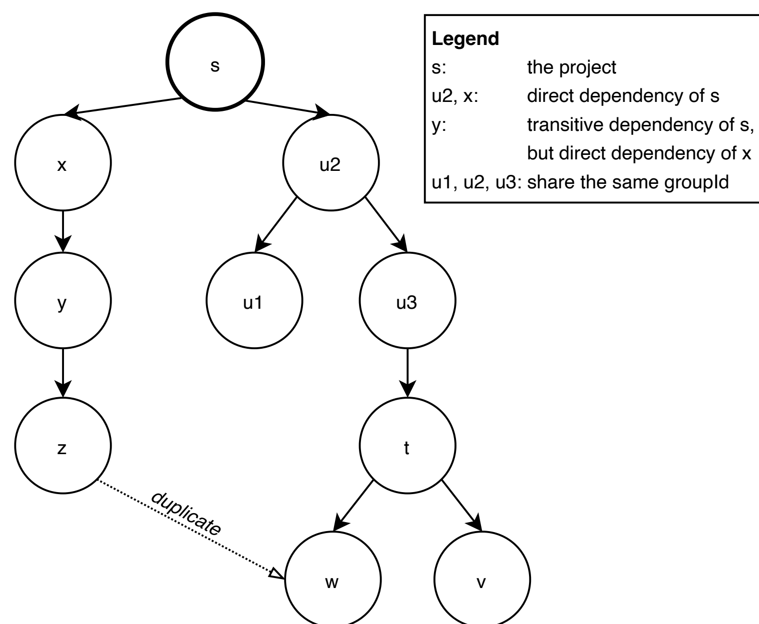


Figure 2.1: Example of a Maven dependency graph containing a duplicate dependency.

When running a Java application, the Java Virtual Machine (JVM) locates the application's classes and JAR files using the classpath parameter. A Java application using the default class loader only supports a flat, linear classpath. When loading a class, Java's default class loader linearly traverses the class files and JAR files on the classpath and loads the first found class with a matching fully-qualified name, shadowing all other instances of the class [LB98; GED03]. Consequently, an application using the default Java class loader cannot load duplicate or conflicting versions of the same class simultaneously. Since the default Java class loader picks the first found class on the classpath, the choice of the loaded classes and libraries depends on the ordering of the JAR files on the classpath. In the worst case, this behavior can

lead to inconsistent behavior, e.g., the project was compiled with an older version of a library placed before the new version on the classpath, but on the runtime classpath, the newer version is placed before the older version. Note that custom implementations of Java class loaders can support the loading of conflicting classes. However, implementing custom class loaders is complex and considered a bad practice if not done carefully [OSG11]. An example framework implementing custom class loaders is the OSGi platform [OSG23], which we describe in combination with Java's class loading mechanism in Section 5.1.

To avoid inconsistent compile-time and runtime behavior caused by a different ordering of JARs on the classpath, the build-automation and dependency-management tool resolves all conflicting and duplicate dependencies in the dependency graph. To do so, the tool transforms the dependency graph into a directed, rooted dependency tree. To create the dependency tree and resolve ambiguous relations like circular dependencies, duplicates, or conflicting versions, Maven automatically picks the dependency closest to the project (the shortest path starting from the root node in the dependency graph). All other instances of that dependency [Apa23a] are shadowed and not included in the project's classpath, e.g., in Figure 2.1, the edge marked as *duplicate* is removed in the dependency tree. The project's complete set of direct and transitive dependencies in the dependency tree is called Software Bill of Materials (SBOM).

2.2 Dependency Management in Other Programming Languages

Build-automation and dependency-management tools exist for most languages, e.g., pip for Python, npm for JavaScript, NuGet for C#, etc. In the following, we introduce pip, which works similarly to Maven, and npm, which follows a complementary approach. We further introduce Conan for C/C++ as an example of a compiled language.

Python and pip Pip enables developers to download and install Python artifacts, so-called packages, from the popular open-source repository Python Package Index (PyPI). In Python, software artifacts are typically distributed as wheel files. A wheel file is a ZIP-format archive with a specially formatted file name. It bundles the Python source code into a single archive. To include a package, developers specify the artifact's identifier: the artifact's unique name and, optionally, its version in a

project's `Pipfile`. In pip, a dependency usually does not declare an exact version but a version range or a minimal version the project requires. Maven version 2 also allows developers to use version ranges in `pom.xml` files following the Semantic Versioning Schema v.1.0 [Pre21]. The syntax is still valid in Maven version 3 [Apa23b], but less common compared to Python.

Listing 2.2 gives an example of dependency declarations in a Pipfile. In pip, for instance, developers can specify a package's exact version using ==, specify a minimal version using >=, or specify that any version is valid using *. In the latter cases, pip determines the exact version during installation and conflict resolution.

```
1 [packages]
2   beanie = "==1.11.7"
3   matplotlib = "*"
4   flask = ">=2.3.2"
```

Listing 2.2: Example declaration of artifacts with version constraints as dependencies in a Python project's `Pipfile` for the tool pip.

When an artifact is installed, pip resolves the artifact's direct and transitive dependencies. To do so, pip also builds a dependency graph where nodes can depend on the same artifact but in different versions, allowing conflicts. In the example Listing 2.2, the project depends on the artifacts `beanie` and `flask`. `Beanie` again depends on the artifact `click` in a version greater than or equal to 7, and `flask` depends on the library `click` with a version greater than or equal to 8.1.3. Analog to Maven, pip cannot install multiple versions of a dependency. To resolve conflicts, pip builds the complete dependency graph and installs the version that conforms to all constraints in the dependency graph. In the given example, pip installs the dependency `click` in version 8.1.3. If pip cannot find a version that conforms to all constraints, it stops the dependency resolution and asks the developers to resolve the conflict manually [Pip23].

Node.js and npm Npm is the build-automation and dependency-management tool for the JavaScript runtime environment Node.js. It enables developers to install and include artifacts (npm packages) from the public npm registry. In Node.js, packages are typically distributed as tar files, which bundle the (minified) source code into a single archive. Dependencies are declared similar to pip, as shown in Listing 2.3. In addition to pip's version specifier, npm allows the specifier ^, which enables the inclusion of all artifacts that do not increment the first non-zero portion of the version, following the Semantic Versioning Schema [Pre21]. A package may also include transitive dependencies.

```
1  "dependencies": {
2    "archiver": "^5.3.0",
3    "aws-sdk": ">=2.1067.0",
4    "axios": "^0.21.4", }
```

Listing 2.3: Example declaration of artifacts with version constraints as dependencies in a `package.json` for the tool npm.

In contrast to Maven and pip, Node.js allows multiple versions of the same package to co-exist in the dependency graph. The Node.js runtime maintains a complete dependency graph for each dependency. Thus, a Node.js project can contain multiple conflicting and duplicate dependencies.

C/C++ and Conan In contrast to languages like Java, Python, and JavaScript, dependency-management tools only have a low adaption rate in C/C++ projects [MP18; Tan+23]. In fact, no unified language-specific dependency-management tool for C/C++ exists [Tan+23]. Although build-automation tools like CMake [Kit23] ease the building of projects, CMake does not manage a project's dependencies. Instead, the dependencies for building and execution are retrieved from the default installed libraries on the host system, or the tool requires the developers to manually distribute and include the dependencies.

The most used C/C++ dependency-management tool is Conan [JFr23] with the public repository ConanCenter. Conan packages are typically distributed as precompiled binaries, but Conan also supports compiling dependencies from source. To do so, Conan packages also declare a `conanfile.py` that specifies a package's dependencies and build instructions. Developers specify their project's dependencies within a `conanfile.txt` or a `conanfile.py`. Listing 2.4 shows a dependency declaration to the library `zlib` in version 1.2.11, and further packages in different versions. Like npm, Conan supports version ranges following the Semantic Versioning Schema [Pre21].

```
1  [requires]
2    zlib/1.2.11
3    fmt/[>9.1.0 <10.1.1]
4    openssl/[~3.1.4]
5    bzip2/[^1.0.6]
```

Listing 2.4: Example declaration of artifacts as dependencies in a `conanfile.txt` for the tool Conan.

Like Maven, Conan does not enable conflicting dependencies to co-exist in the dependency graph. When a dependency conflict occurs, Conan asks the developer to resolve the conflict manually.

Comparison to Java Maven and pip use the same structure to include dependencies in a project. Both use a global dependency tree with duplicate and conflicting dependencies resolved. Although Maven, pip, and Conan apply different conflict resolution mechanisms, both include exactly one version of each dependency in the global dependency tree only. In contrast, npm maintains a whole dependency graph per dependency, allowing multiple conflicting versions to co-exist in the project. Consequently, concepts that rely on Maven's or pip's dependency graph can be directly applied to each other but must be adapted for npm.

3 Systematic Study on the Usage of Open-Source Software and Challenges for Their Detection

The use of vulnerable Open-Source Software (OSS) is a known problem in today's software development. Although several tools appeared in the last decade to detect known-vulnerable dependencies, no systematic study exists investigating the impact of typical development practices, e.g., forking, patching, and re-bundling, on the tools' precision and recall.

In this chapter, we explore the use of (vulnerable) OSS in commercial and open-source projects as well as challenges for their detection. Through an empirical study on 7,024 Java projects developed at SAP in Section 3.2, we identify that developers use open-source artifacts from Maven Central directly or apply four different types of modifications: re-compilation, re-bundling, metadata-removal, and re-packaging. We study these modifications on Maven Central to validate our findings in an open-source context in Section 3.3. Our study shows that these modifications are nonexclusive to projects developed at SAP but also occur in open-source projects; we found that for a selected set of vulnerable Java classes, more than 87% are re-bundled and 56% are re-packaged on Maven Central.

In Section 3.4, we assess the impact of these modifications on six tools for detecting known-vulnerable open-source dependencies: the open-source scanners OWASP Dependency-Check (DepCheck) and Eclipse Steady, the free scanner GitHub Security Alerts, and three commercial scanners. The results show that the scanners struggle with the identified modifications, presenting a challenge for detecting known vulnerabilities in (modified) OSS. We discuss threats to the validity of our study and shortcomings in Section 3.6.
To facilitate reproducibility and drive further development of open-source vulnerability scanners, we present Achilles, a novel test suite with 2,558 test cases that allow replicating the modifications on open-source artifacts in Section 3.7.

To identify (modified) OSS, even if the contained Java classes have been re-compiled or re-bundled, we present our tool SootDiff in Section 3.8. SootDiff uses the static analysis framework Soot and its intermediate representation Jimple, in combination with code clone detection techniques, to identify code clones and to reduce dissimilarities introduced due to re-compilation with different Java compilers and Java versions. Our experiments show that SootDiff successfully identifies clones in 102 of 144 cases, whereas bytecode comparison succeeds in 58 cases only. The results show that SootDiff can help to mitigate the found challenge of identifying known vulnerable classes in re-compiled, re-bundled, or re-packaged OSS.

We give an overview of related work in Section 3.9, and complete with a conclusion of our study and SootDiff in Section 3.10.

3.1 Strategies for Detecting Vulnerabilities in Open-Source Software

Most open-source vulnerability scanners check different sources and vulnerability databases to check if a given open-source artifact is affected by a known vulnerability [Pas+18]. The main and most complete public source of vulnerabilities is the National Vulnerability Database (NVD) [NIS20].[1] The NVD links a vulnerability (Common Vulnerabilities and Exposures (CVE)) to a set of operating systems, hardware, or software using the Common Platform Enumeration (CPE) standard to list the affected artifacts. CPE is a naming schema that defines standardized methods for assigning names to IT product classes [NIS20]. A CPE in version 2.3 has the form `cpe:2.3:<part>:<vendor>:<product>:<version>:<update>:<edition>:<language>:<sw_edition>:<target_sw>:<target_hw>:<other>`, where `part` specifies an application (`a`), operating system (`o`) or hardware (`h`), the other elements identify the affected product(s), wildcards are allowed. Although the NVD is the most complete public source of vulnerabilities, CPEs do not cover all artifacts of an OSS with full accuracy. In addition, CPEs often use a different granularity and schema than dependency-management tools, making the mapping from the dependency's identifier used by the build-automation tool like its groupId:artifactId:version (GAV) to CPEs complex, for instance, the CPE product element may not correspond to the artifactId in the GAV.

[1]Further sources of vulnerabilities are vendor-, product-, or software-specific advisory boards and bug trackers used to report security issues.

For detecting known-vulnerable dependencies, vulnerability scanners build a project's Software Bill of Materials (SBOM) and query a vulnerability database if known vulnerabilities for the found artifacts exist. Usually, the tools do not only query a single vulnerability database but a combination of public and commercial vulnerability databases, including the NVD. Different strategies exist for matching an artifact against the entries (and CPEs) in a vulnerability database. The most popular strategies are name-based and code-based matching [PPS18; PPS20; NDM16].

Tools like DepCheck and GitHub Security Alerts apply name-based matching to identify known-vulnerable dependencies. First, these tools extract from the declared GAVs and the artifacts' JAR files the vendor, product, and version of each artifact. Second, the tools try to match the found information to CPEs in the NVD or the software identifiers in Mend.io's vulnerability database [Git20] using fuzzy-matching. Finally, the tools report vulnerabilities with a CPE matching the identifiers that they crafted from the GAVs and JAR files. Name-based matching approaches suffer from both false positive and false negatives [NM13; DBM19]. False negatives occur when the NVD is incomplete, and the set of CPEs does not contain all affected artifacts. False positives occur when the CPEs over-approximate the affected versions or specify a complete application instead of the affected artifact [NM13]. For instance, CVE-2018-1271 declares the complete Spring framework as vulnerable using the CPEs `cpe:2.3:a:vmware:spring_framework:4.3.0:*:*` up to `cpe:2.3:a:vmware:spring_framework:4.3.14:*:*`, whereas only the single library `web-mvc` with the GAVs `org.springframework:spring-webmvc:4.3.0.RELEASE` up to `4.3.14.RELEASE` is actually vulnerable [Piv20]. In this case, a name-based matching strategy will likely report every artifact of the Spring framework as vulnerable. The effectiveness of a name-based matching strategy entirely relies on the completeness and correctness of the identifiers and CPEs in the database and the correctness of the dependency's meta-information: vendor, product, and version.

Tools like Eclipse Steady apply code-based matching [PPS20; PPS18]. Instead of mapping the dependency's vendor, product, and version to identifiers in a vulnerability database, Eclipse Steady maps the bytecode of the classes in the dependency's JAR against a custom database containing the bytecode of known-vulnerable artifacts. To do so, Eclipse Steady computes the digest of each JAR and the fully-qualified name (FQN) of all classes and methods the JAR contains. Eclipse Steady then uses its custom database to check if vulnerabilities have been reported for the found FQNs.

Code-based matching approaches require the creation of a separate database containing the vulnerable software constructs of each vulnerability, e.g., classes, constructors, methods, and their FQN, since public databases like the NVD typically do not contain this information. To derive the vulnerable constructs, the commits that fix the vulnerability must be identified for each disclosed vulnerability. The effectiveness of code-based approaches depends on the similarity between the identified vulnerable bytecode stored in the database and the bytecode contained in the included dependency, which may differ due to forking, re-compilation, re-bundling, or re-naming of classes.

The matching approaches of commercial vulnerability scanners' and databases are not publicly available. They may use a combination of name-based and code-based matching or rely on other data, e.g., file digests, timestamps, file names, or additional attributes.

3.2 Study Design

As described in Section 3.1, research and industry have developed several open-source vulnerability scanners, e.g., the open-source tools DepCheck and Eclipse Steady, the free tool GitHub Security Alerts, and commercial tools such as Snyk [Sny23], Synopsys [Syn23], or Mend.io [Men23a], which apply name-based matching, code-based matching, or a combination of several attributes for identifying known-vulnerable dependencies in a project.

Although previous studies [Hei+11; BHD12; Kul+18; Pit16; Pas+22] investigate to which extent open-source or commercial applications include (vulnerable) OSS, no study exists that examines what development practices [Bav+15; Kul+18; Hei+11; BHD12], like forking, patching, re-compiling, or re-packaging, developers apply to included open-source artifacts. Further, no study exists that investigates how precisely open-source vulnerability scanners can identify known-vulnerabilities in such modified artifacts.

3.2.1 Research Questions

To fill this gap, we investigate in a two-folded case study (i) how developers include (vulnerable) OSS, and (ii) the prevalence and impact of modifications, such as forking and patching, on the precision and recall of vulnerability scanners.

In the first part, we clarify how developers use OSS in a commercial context, particularly at SAP. To do so, we investigated **RQ1: What are developer practices for using OSS at SAP?** We analyzed the metrics of the SBOMs of 7,024 Java projects developed at SAP. We checked how many dependencies a vulnerability scanner has to analyze by computing the average number of (direct and transitive) dependencies a project includes. Vulnerability scanners are especially beneficial for detecting vulnerabilities in transitive dependencies, as developers only include direct dependencies explicitly and are often unaware of all transitive dependencies their projects include. Thus, developers likely miss vulnerable transitive dependencies in manual reviews [Pas+18; Kul+18]. Further, we computed the ratio of direct to transitive dependencies. To check how many (vulnerable) dependencies could actually be exploited in a project, we reviewed the ratio of release dependencies to development-only dependencies. Since only release dependencies are available during production, only vulnerabilities in those are exploitable. To evaluate how critical vulnerability scanners are for detecting known-vulnerable OSS in a commercial software project, we checked how extensively OSS is used. To do so, we computed the ratio of open-source to proprietary dependencies.

We applied the following procedure to investigate the prevalence of vulnerable OSS. We first selected a representative set of the 20 most-used dependencies at SAP and then checked **RQ2: What vulnerabilities affect the 20 most-used dependencies?** Therefore, we checked for all observed versions of the 20 most-used dependencies, in total, 723 different open-source artifacts, which vulnerabilities affect them. This semi-manual classification also serves as the basis for our test suite Achilles.

In our case study, we found that developers include not only unmodified artifacts directly from public open-source repositories but also artifacts in modified form. As already stated by Ponta et al. [PPS20; PPS18], developers fork, patch, re-compile, re-bundle, and re-package existing open-source artifacts resulting in modified metadata (name, vendor, version, timestamp, etc.) and bytecode [Bav+15; Kul+18; Hei+11; BHD12]. To elaborate on this observation, we investigated **RQ3: How do developers include OSS?** and classified the observed modifications into four different types: re-compilation, re-packaging, metadata-removal, e.g., removal of `MANIFEST.MF` files, re-bundling multiple OSS into a single file (so-called Uber-JAR), or combinations of those.

In the second part, we investigated the prevalence of the found modifications and their impact on the precision and recall of vulnerability scanners. To check their prevalence, we studied **RQ4: How prominent are the modifications outside SAP?** by evaluating how often they occur on the open-source repository Maven Central.

To evaluate **RQ5: What is the impact of the modifications on vulnerability scanners**, we compared the precision and recall of six vulnerability scanners w.r.t. the identified modifications. To do so, we chose a representative set of 16 vulnerabilities from *RQ1* and *RQ2*, applied the identified modifications, and investigated the precision and recall of three commercial vulnerability scanners,[2] GitHub Security Alerts, and the open-source scanners DepCheck and Eclipse Steady by computing precision, recall, and F1-score in these test scenarios.

3.2.2 Study Objects & Methodology

Case studies are suitable means to gain an in-depth understanding of real-world situations and processes [RH09], like development practices regarding the use of OSS. Thus, we answer our research questions with a software industry case study at SAP.

Studied Projects & Project Metric Extraction In our study, we investigated 7,024 different Java projects, covering a wide range of enterprise applications, platforms, inhouse tools, micro-services, and monoliths. Specifically, we investigated the SBOMs of each project created by the vulnerability scanner Eclipse Steady, which uses the build-automation tool Maven to generate the SBOMs and resolve dependencies [PPS18]. The generated SBOMs contain the project's unique identifier (GAV) and the attributes of the used dependencies. In particular, the SBOM states for each dependency: the GAV, the scope, if the dependency is direct or transitive, the JAR's filename, and the JAR's SHA1.

The initial data set consisted of 49,752 different SBOMs generated by Eclipse Steady. Since the SBOMs were generated during the build process, the data set included a separate SBOM for each project version. We only included each project's latest SBOM to balance our data set. As a result, the filtered data set consisted of the SBOMs of 7,024 distinct projects (projects with distinct groupId:artifactId (GA)).

To study the prevalence of the studied open-source artifacts and the found modifications in the open-source community, we investigated the artifacts hosted on Maven Central utilizing its search index and the statistics provided by Maven Central and MvnRepository [Mvn20].

[2] Due to license restrictions, we cannot disclose their names.

Dependency Selection To answer the research questions *RQ2*, *RQ4*, and *RQ5*, we investigated in semi-manual reviews what vulnerabilities affect the 20 most-used dependencies and what Java classes contain the vulnerable code. As a result, we investigated 723 different artifacts—all found versions of the 20 most-used dependencies.

To ensure software industry relevance, we selected the sample set from the project data set as follows: We only considered release dependencies that are available during the build, that are dependencies with the scopes *compile* or *runtime*.

Table 3.1: The selected sample set of the 20 most-used artifacts in the 7,024 projects developed at SAP, grouped by groupId:artifactId (GA). The table shows how many different projects use that artifact and how popular—based on its usage—the artifact is on Maven Central.

Used by #projects at SAP	Artifact GA (groupId, artifactId)	Popularity on Maven Central [Mvn20]
3,211	commons-codec:commons-codec	18
3,026	org.slf4j:slf4j-api	1
2,899	com.fasterxml.jackson.core:jackson-annotations	23
2,854	com.fasterxml.jackson.core:jackson-core	21
2,851	com.fasterxml.jackson.core:jackson-databind	6
2,831	org.apache.httpcomponents:httpcore	57
2,781	commons-logging:commons-logging	17
2,774	org.apache.httpcomponents:httpclient	13
2,662	com.google.code.gson:gson	10
2,617	org.springframework:spring-core	28
2,574	org.springframework:spring-beans	36
2,533	org.springframework:spring-context	16
2,518	org.springframework:spring-aop	-
2,503	org.springframework:spring-expression	-
2,495	commons-io:commons-io	3
2,371	org.apache.commons:commons-lang3	5
2,133	org.springframework:spring-web	32
2,105	com.google.guava:guava	2
2,046	javax.validation:validation-api	44
1,895	org.springframework:spring-webmvc	48

To validate the relevance of our sample set within the open-source community, we checked if the selected 20 most-used artifacts are equally popular on the public repository Maven Central. Table 3.1 shows the ranking of the selected 20 most-used artifacts at SAP among the 100 most-used artifacts at Maven Central, based on the statistics provided by MvnRepository [Mvn20] in column *Popularity*. In contrast to our study, the MvnRepository popularity ranking includes development-only dependencies, which are typically included with the scopes *test, provided*, or *system*, and dependencies for non-Java languages, like Kotlin, Scala, and Closure. We excluded test, mock, and non-Java dependencies from the ranking to achieve a fair comparison and re-computed the popularity. In total, we excluded 38 dependencies: `junit`, `mockito-*`, `easymock`, `scala-*`, `clojure`, `org.renjin`, `kotlin-*`, `android-*`.

The table shows that 18 out of the 20 most-used artifacts at SAP are within the 100 most-used artifacts on Maven Central. Only two artifacts, `spring-expression` and `spring-aop`, are outside the 100 most-used artifacts on Maven Central. In summary, the table shows that the artifacts' popularity slightly differs within SAP and Maven Central.

To study how often the sample set is used within open-source projects, we computed the number of usages for all 723 artifacts (GAVs) on Maven Central, resulting in the log-normal distribution in Figure 3.1. The figure shows that open-source projects hosted on Maven Central also regularly use the artifacts in the sample set.

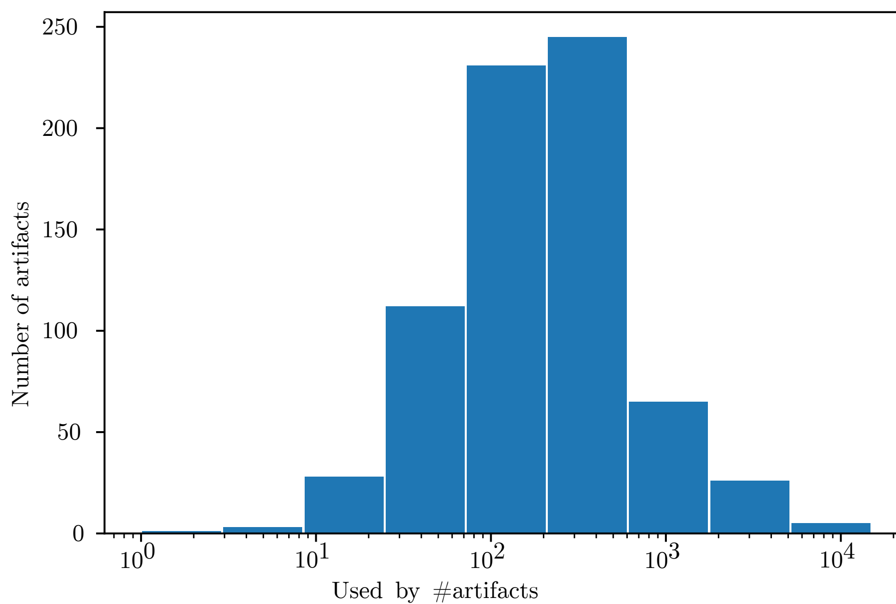


Figure 3.1: To understand how the open-source community also uses the selected sample, this graph reports the #usages of the 723 artifacts (GAVs) as reported by `mvnrepository.com`, showing a log-normal distribution (X-axis has logarithmic scale).

Vulnerable Dependency Identification As described in Section 3.1, all current approaches for matching vulnerabilities to OSS artifacts are prone to false positives and false negatives. Moreover, the databases are incomplete w.r.t. the reported vulnerabilities and the set of CPEs, resulting in false negatives.

We used three vulnerability scanners to identify vulnerable dependencies to reduce the likelihood of false positives and false negatives: the open-source scanners Eclipse Steady and DepCheck, and the commercial scanner C3. The scanners apply different matching strategies: Eclipse Steady applies code-based matching, whereas DepCheck uses name-based matching. For C3 there is no public information available describing the applied matching strategy. By choosing these scanners, our results rely on three different vulnerability databases: Eclipse Steady uses its open-source database [Pon+19; SAP20], DepCheck uses the NVD [NIS20], and C3 uses a commercial database. Hereby, we aim to improve the validity of our results by balancing out the shortcomings of one particular database.

To achieve soundness, we classified the scanners' reports in semi-manual reviews (cf. Section 3.3.2) into true and false positives. In total, we classified 2,558 scanner reports for the 723 distinct artifacts (GAVs).

Identification of Modifications on Maven Central To assess the prevalence of modifications outside SAP in the open-source community, we checked how often the found modifications occur on Maven Central. To do so, we first identified for each vulnerability (identified in *RQ2*) what class files are vulnerable in the artifact's JAR file. To identify the vulnerable class files, we manually checked the source-code commits that fixed the identified vulnerability and investigated which class files the commits changed. Second, we computed how often the identified vulnerable classes occur in modified form on Maven Central w.r.t. the identified modifications.

To check if the bytecode of a vulnerable class matches the found—and potentially modified—classes on Maven Central, we compared their bytecode using the tool SootDiff (cf. Section 3.8). SootDiff's comparison was specifically designed to be resilient to changes induced by various compilation schemes, and thus allows us to check for bytecode equivalence even if a modification has been applied to one of the classes.

3.3 Use of Open-Source Software at SAP

In this section, we answer the research questions *RQ1*, *RQ2*, and *RQ3* by investigating the development practices regarding the use of OSS in a commercial context within our empirical study on 7,024 Java projects developed at SAP, one of the world's largest software development companies.

3.3.1 RQ1: What Are Practices for Using Open-Source Software at SAP?

To answer *RQ1*, we first computed the average *number of dependencies per project*. Therefore, we calculated the number of distinct dependencies' GAVs, the arithmetic mean, and the standard deviation [Saj+14]. We found that, on average, a project includes 94.78 direct and transitive dependencies with distinct GAVs, with a standard deviation of 124.61. The high standard deviation shows that the number of included dependencies—which is also the size of the SBOM—heavily varies among projects.

Estimating the number of dependencies of a project by counting the distinct GAVs seems simple, however, it may suffer from several issues as described by Pashchenko et al. [Pas+18]. For instance, if a developer declares a dependency on `spring-context`, its transitive dependencies with the same groupId are counted separately. This overweights dependencies that are released as multiple JARs, e.g., frameworks like Spring or Struts. To overcome this issue, we count the number of distinct groupIds per project as proposed by Pashchenko [Pas+18]. This resulted in 36.34 ($sd = 35.84$) direct and transitive dependencies with distinct groupId per project, with a median of 25. The number of dependencies highly varies per project, ranging from 0 to 228 dependencies. Unless stated otherwise, we use this grouping for the remainder of the study.

Second, we computed the *ratio of direct to transitive dependencies*, shown in Figure 3.2. The figure shows that only 21% are direct dependencies, whereas 79% are transitive.

Our investigation showed that more than 50% of the studied projects incorporate at least one of the frameworks: Spring or Struts. Since the Spring and Struts frameworks include many transitive dependencies with different groupIds, the amount

of observed transitive dependencies is high in our study—although we only count distinct GAs. For instance, the framework `spring-boot-starter:2.17` introduces 17 dependencies with different groupIds, e.g., `javax.*`, `ch.qos.logback`, `org.apache.*`.

Third, we investigated how many dependencies are deployed with a release. Therefore, we computed the *percentages of used scopes*, shown in Figure 3.2. The figure shows that most dependencies (58.1%) have the default scope *compile*, and thus are present during compile time, testing, and runtime. 4.9% have the scope *runtime*, and thus are present on the runtime classpath. 20.2% have the scope *provided* or *system*, and thus are present at runtime only but not during compile time. 16.8% have the scope *test*, and thus are development-only dependencies, not deployed in production. The high number of dependencies with the scopes *runtime*, *provided*, or *system* shows that 25.1% of the dependencies are not shipped with the application but are pre-installed on the host system on which the application is executed, which may differ from the build system. However, vulnerability scanners are executed on the build system, e.g., a Jenkis, GitLab, or GitHub build server, and thus cannot identify vulnerabilities in artifacts provided later—during runtime.

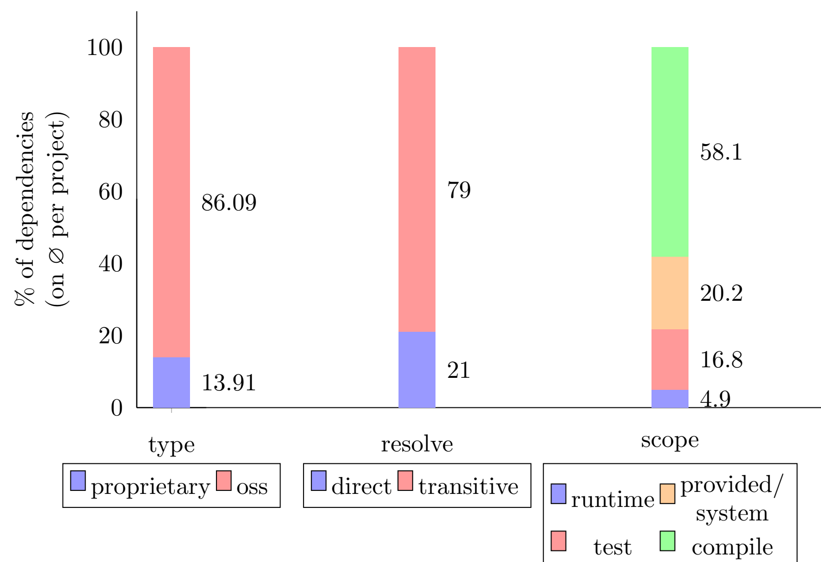


Figure 3.2: To understand to what extent and how developers include open-source artifacts, this graph reports: the average ratio of OSS to proprietary dependencies, the ratio of direct to transitive dependencies, and the scopes developers use; grouped by groupId:artifactId (GA).

Fourth, we computed the ratio of open-source to proprietary dependencies to check to *what extent OSS* is used in a commercial context. To distinguish between open-source and proprietary dependencies, we classified each artifact hosted with the

same GA on Maven Central as open-source and any other as proprietary. Figure 3.2 shows the results. As other repositories exist for Java, e.g., Sonatype, JCenter, or Redhat JBoss, this ratio is only a lower bound.

On average, 86.09% of the dependencies are open-source, whereas only 13.91% are proprietary. Moreover, open-source dependencies are ubiquitous as 95.43% of the projects include at least one open-source artifact.

Findings from RQ1: On average, a Java project includes 36 direct and transitive dependencies, 86% of them being open-source. Most dependencies are transitive (79%), and are release dependencies (58.1%).

3.3.2 RQ2: What Vulnerabilities Affect the 20 Most-Used Dependencies?

We investigated how many vulnerabilities affect the 20 most-used dependencies, which we selected as described in Section 3.2.2. As vulnerabilities typically only affect specific version range(s), we checked all versions of the artifacts we observed in our data set. In total, we observed 723 different artifacts (GAVs) for the 20 most-used dependencies.

We used the vulnerability scanners Eclipse Steady, DepCheck, and C3 to identify known vulnerabilities in these artifacts. By using the scanners, which rely on different vulnerability databases, we aim to improve the validity of the results, as described in Section 3.2.2. As input for those scanners, we created a separate Maven project for each artifact with a direct dependency on that artifact, including its optional dependencies.

Table 3.2 shows the number of findings separated per scanner for the GAs of the artifacts, and the number of distinct reported vulnerabilities. The column *Scanned Artifacts* shows the scanned artifact, which we declared as a direct dependency in the created Maven project. The number in brackets shows how many versions of the artifact were scanned. The column *Vulnerability reported in Artifacts* shows for which artifact—the artifact itself or one of its (transitive) dependencies—the scanners reported a vulnerability. Again, the number in brackets shows in how many versions of the artifact the scanners reported vulnerabilities. The cell is highlighted if a vulnerability in the artifact itself was reported.

The column *#Dist. Vulnerabilities* gives the total number of distinct reported vulnerabilities, and the column *#Findings* gives the number of findings the scanners reported. For instance, consider the first row in Table 3.2. We scanned the artifact guava in 35 versions; the scanners reported vulnerabilities in 25 versions of guava. The scanners found 1 unique vulnerability (column *Dist Vulnerabilities*) across all versions: Steady reported it in 12 different versions, DepCheck in 25 versions, and C3 in 25 versions.

The column *Findings* in Table 3.2 includes both true-positive and false-positive findings. We classified all findings later in semi-manual reviews. Note that Eclipse Steady and C3 reported vulnerabilities that are not officially listed in the NVD, and thus do not have a CVE number, e.g., bugs or vulnerabilities disclosed in bug trackers. Eclipse Steady and C3 report those non-official vulnerabilities since both scanners use custom databases, whereas DepCheck only uses the NVD. Though we considered these vulnerabilities in Table 3.2, we ignored them for creating our test suite Achilles and for comparing the scanners in *RQ5*, as different scanners may name the same issue differently or may not consider them as vulnerabilities, thereby hindering a fair comparison. In total, the scanners reported 251 distinct vulnerabilities for 534 of the 723 different artifacts. Since a single CVE usually affects multiple versions, e.g., CVE-2016-3720 affects the artifacts `jackson-dataformat-xml` version 2.0.0–2.7.4, the scanners generated 2,558 distinct findings (GAV, vulnerability).

Table 3.2: To identify known vulnerabilities in our sample set of the 20 most-used dependencies (723 artifacts), we used the scanners Steady, OWASP DependencyCheck (DepCheck), and C3. The table gives an overview of the findings that the scanners reported. Highlighted are cases in which the artifact itself is reported as vulnerable.

Scanned Artifact (#versions)	Vulnerability reported in Artifact/Dependency (#versions)	#Dist. Vulnerabilities	#Findings		
			Steady	DepCheck	C3
guava (35)	guava (25)	1	12	25	25
httpclient (22)	httpclient (18)	7	4	9	40
jackson-databind (54)	groovy (1)	2	4	0	0
	jackson-databind (53)	16	227	203	218
spring-aop (61)	spring-core (52)	25	17	374	29
spring-beans (61)	groovy-all (8)	2	30	18	24
	spring-core (52)	25	23	29	375

Continued on next page

Scanned Artifact (#versions)	Vulnerability reported in Artifact/Dependency (#versions)	#Dist. Vulner-abilities	#Findings		
			Steady	DepCheck	C3
spring-context (60)	bsh (2)	1	60	60	55
	hibernate.validator (2)	1	0	4	0
	groovy-all (10)	2	45	28	34
	hibernate-validator (5)	2	7	8	7
	jruby (4)	6	1	18	2
	jsoup (1)	1	0	2	0
	spring-core (51)	25	28	368	26
	spring-expression (46)	2	92	0	0
spring-core (62)	commons-collections (1)	4	5	10	8
	spring-core (53)	24	20	390	31
spring-expression (57)	spring-core (48)	25	4	29	11
	spring-expression (10)	2	20	0	0
spring-web (46)	axis (1)	3	0	9	0
	axis-saaj (1)	3	0	9	0
	commons-fileupload (3)	6	50	49	9
	commons-httpclient (1)	1	3	0 0	
	groovy-all (8)	2	18	12	17
	guava (1)	1	9	9	9
	httpasyncclient (1)	1	0	1	0
	httpclient (7)	5	26	5	24
	jackson-databind (28)	13	396	134	134
	jackson-dataformat-xml (20)	5	23	90	14
	jetty-http (21)	9	116	192	74
	jetty-security (15)	1	33	0	0
	jetty-server (20)	6	148	0	109
	jetty-servlet (19)	1	38	0	0
	jetty-util (20)	4	75	0	68
	netty-all (9)	2	10	15	4
	okhttp (2)	1	3	3	3
	org.apache.axis (1)	1	3	0	0
	protobuf-java (1)	1	0	31	31
	spring-core (43)	25	22	305	20
	spring-expression (37)	2	74	0	0
	spring-oxm (14)	4	24	0	8

Continued on next page

Scanned Artifact (#versions)	Vulnerability reported in Artifact/Dependency (#versions)	#Dist. Vulner-abilities	#Findings		
			Steady	DepCheck	C3
	spring-web (44)	12	42	0	105
	taglibs (1)	1	0	1	1
	tomcat-embed-core (7)	8	30	15	32
	undertow-core (4)	6	12	0	44
spring-webmvc (45)	bcprov-jdk14 (2)	14	180	481	37
	bcprov-jdk15on (1)	3	16	24	16
	castor (1)	1	0	5	0
	commons-beanutils (3)	2	88	45	43
	commons-collections (4)	4	18	46	21
	commons-compress (1)	1	33	0	33
	commons-fileupload (1)	5	5	0	0
	dom4j (1)	1	1	0	0
	groovy-all (8)	2	17	12	15
	guava (1)	1	24	25	25
	itextpdf (1)	2	2	0	3
	jackson-databind (28)	13	407	139	139
	jackson-dataformat-xml (19)	5	21	85	13
	jasperreports (8)	6	0	68	0
	lucene-queryparser (1)	1	29	0	0
	ognl (1)	1	43	44	44
	poi (8)	7	35	107	72
	poi-ooxml (3)	24	24	0	24
	spring-core (38)	23	12	20	251
	spring-expression (33)	2	66	0	0
	spring-oxm (13)	5	12	0	19
	spring-tx (1)	1	1	0	1
	spring-web (40)	13	75	0	100
	spring-webmvc (38)	8	28	28	99
validation-api (5)	bsh (1)	1	4	0	0

The table shows that scanners reported mostly vulnerabilities for transitive dependencies (the non-highlighted cells). The vulnerability scanners did not report anything for the artifacts `commons-codec`, `commons-io`, `commons-logging`, `commons-lang3`, `gson`, `httpcore`, `jackson-annotations`, `jackson-core`, and `slf4j-api`, which we therefore omitted. The complete set of reported vulnerabilities and our classification are published along with the test suite (cf. Chapter 6).

Findings from RQ2: Eclipse Steady, DepCheck, and C3 generated 2,558 findings for 534 of 723 different artifacts with 251 CVEs. The majority of vulnerabilities affect transitive dependencies.

To investigate the ratio of true-positive and false-positive reported vulnerabilities, we semi-manually classified the 2,558 findings using the following procedure:

1. We checked if the NVD [NIS20] or Eclipse Steady's database references a source-code commit, issue, or pull request fixing the reported vulnerability and identified the changed vulnerable classes.

In total, we found source-code commits for 96 vulnerabilities and identified 254 vulnerable classes.

2. If we found a commit, we checked if the reported JAR file contained the bytecode of the identified vulnerable classes.

   2.1. To do so, we first determined the vulnerable methods, classes, and static initializers using Eclipse Steady and its database [PPS18].

   2.2. Second, we compared the bytecode of the vulnerable classes, methods, and static initializer with the bytecode contained in the reported JAR using SootDiff (cf. Section 3.8).

   2.3. If the bytecode of at least one class, method, or static initializer matched the vulnerable code, we classified the finding as true positive.

In these steps, we classified 427 reports as true positive, 793 as false positive, and left 1338 for further investigation.

3. If we could not find a source-code commit for the vulnerability or SootDiff's comparison failed, we manually searched the NVD and Eclipse Steady's database for links to issue boards and bug trackers.

4. If we found a link to an issue or bug tracker, we checked if the description stated the vulnerable artifacts and the affected versions.

   4.1. If a description existed and exactly matched the reported artifact, we classified the finding as true positive.

   4.2. If a description existed but did not match the reported artifact or the NVD entry referred to a different artifact, we classified the finding as false positive.

In these steps, we classified 351 reports as true positive, 829 as false positive, and 158 as ambiguous.

5. If we could not find a link or the description in the NVD or Steady's database, or the artifact identifier was ambiguous, we fell back to the CPEs in the NVD [NIS20].

   5.1. If the CPE matched the reported artifact, we classified the finding as true positive.

   5.2. If the CPE did not match, we classified the finding as false positive.

For the remaining reports, we classified 15 as true positive, 141 as false positive, and left 2 as ambiguous.

Using this procedure, we could not classify two findings: one finding for CVE-2013-5855 in `javax.faces-api:2.2` and one for CVE-2014-7810 in `el-api:1.0`. We manually checked the two reported JARs for these findings and found that they only contain the vulnerable artifacts' API (abstract classes and interfaces) but no implementation. Thus, we classified them as false positive.
In total, we classified 903 (35%) as true positive in 349 artifacts and 1,655 as false positive of the 2,558 reports.

> **Findings from RQ2:** We classified 903 of the 2,558 findings in reviews as true positive, and the rest as false positive. Further, we identified in 349 (65%) of the 534 reported artifacts 100 unique CVEs, that were classified as true positive.

3.3.3 RQ3: How Do Developers Include Open-Source Software?

In our study, we observed *four types* of modification that developers applied to the included open-source artifacts:

Unmodified JAR: Most commonly, we observed that developers include an artifact directly from Maven Central using its plain, original GAV.

Patched JAR (type 1): We noticed that developers include OSS with a slightly modified GAV, e.g., `com.google:guava:23.0_fix3`. We investigated that these type 1 artifacts occur if developers or distributors fork the source code of an OSS and patch it. They indicate these changes by appending a suffix string like `fix` to the version.

The JAR file does not contain the original bytecode but the (modified) re-compiled bytecode. Due to the re-compilation, the classes' bytecode, digests, timestamps, and the JAR's digest changes [PPS20; PPS18].

In our study, we observed the modifications: re-bundling, metadata-removal, and re-packaging along with so-called Uber-JARs—sometimes also called fat-JARs. Uber-JARs merge multiple open-source artifacts into a single JAR to ease deployment and distribution. In the following, we elaborate on Uber-JARs and how they relate to the observed modifications re-bundling, metadata-removal, and re-packaging.

Uber-JAR (type 2): We found projects that include dependencies with GAVs that do not indicate which original open-source artifact from Maven Central the JAR file contains, e.g., `com.my:servicebundle:1.0`. Such JAR files (re-)bundled multiple open-source artifacts (and their transitive dependencies) into a single JAR file, a so-called *Uber-JAR*. Examples are, for instance, the `jar-with-dependencies` files, which can be found on Maven Central and can be easily generated with Maven's assembly or shade plugin. In contrast to patch modifications (type 1), the plugins preserve the original bytecode, digests, and timestamps.

For *Uber-JARs*, we found two further sub-types:

Bare Uber-JAR (type 3): In rare cases, multiple artifacts are merged into an *Uber-JAR*, but the original artifacts' `pom.xml` files, the original folders `META-INF`, and the original file timestamps are removed from the Uber-JAR. Since Maven's shade and assembly plugins preserve the `pom.xml` by default, this case is supposedly relevant for legacy Uber-JARs built before the advent of these plugins, e.g., with Ant.

Re-packaged Uber-JAR (type 4): Re-packaged Uber-JARs are similar to normal Uber-JARs (type 2), but contain re-packaged artifacts. When re-packing is applied to an artifact's JAR file, a fixed string is typically prepended to the original FQN classes in the JAR, e.g., `org.shaded`. To continue to link, the references to classes, methods, and fields are also adapted in the classes' bytecode to the new FQNs. As a result, the re-packaged classes' bytecode, digests, and file timestamps are changed. Such re-packaging can be configured with the Maven shade plugin and is usually applied to avoid name clashes.

> **Findings from RQ3:** We identified four types of modification that developers apply to included OSS: patched JAR, Uber-JAR, bare Uber-JARs, and re-packaged Uber-JARs.

3.4 Prevalence & Impact of Modified Open-Source Software

In this section, we answer the research questions *RQ4* and *RQ5* by investigating the prevalence of the identified modifications on the public open-source repository Maven Central and the modifications' impact on the precision and recall of vulnerability scanners.

3.4.1 RQ4: How Prominent Are the Modifications Outside SAP?

To check that the observed modifications are not only specific to our data set, which is composed of Java projects developed at SAP, we computed the prevalence of the found modifications on Maven Central. As a basis for detecting the modifications, we used the vulnerable open-source artifacts that we identified in *RQ2*. In particular, we investigated how often the vulnerable classes of the found vulnerable artifacts are subject to modification types 1–4. To do so, we first identified which classes are changed in the source-code commits that fix the 251 CVEs that we identified in *RQ2* (cf. Section 3.3.2). Table 3.3 shows the prevalence of type 1–4 modifications on Maven Central. In the following, we describe our procedure for identifying the modifications.

Table 3.3: To understand how prevalent the modifications are on Maven Central, this table reports how often the identified 254 vulnerable classes were subject to the modifications and in how many different artifacts they occurred.

	Patched JAR (type 1)	Uber-JAR (type 2)	Bare Uber-JAR (type 3)	Re-pack. Uber-JAR (type 4)
# vuln. classes modified	143	222	222	17
found in #artifacts with different GAV*	5,919	36,609	24,500	168
found in #artifacts with different GA†	360	6,723	3,882	89

* GAV - groupId, artifactId, version
† GA - groupId, artifactId

Detection of Patched JARs (type 1) Developers usually change code in a subset of the classes only when forking or patching an OSS. However, releasing the patched artifact requires the re-compilation of all classes. Consequently, in type 1 modifications, developers typically only change the code of a few classes. However, all classes are re-compiled when releasing the JAR file, changing (potentially) the classes' bytecode, digests, and timestamps.

To measure how prevalent type 1 modifications are, we checked how often the vulnerable classes (a subset of the classes in the JAR) have been re-compiled. Suppose we find a re-compiled version of a vulnerable class in an artifact that is not the original artifact. In that case, we assume the artifact was created during forking and re-compilation. Since re-compilation may change a class' bytecode and digest, but not the FQN, we used the following procedure to discover re-compiled classes:

1. We queried Maven Central for classes that have identical FQNs as the identified 254 vulnerable classes.
2. For the found classes, we determined how many classes have equivalent bytecode as the vulnerable classes but a different SHA1, using SootDiff. These classes' bytecode changed due to using a different Java compiler or compiling for a different Java version (cf. Section 3.8).

We found 50,702 artifacts on Maven Central that contain at least one class with an identical FQN. For 143 (56%) of the 254 classes, we found re-compiled versions with a different SHA1 but equivalent bytecode on Maven Central. We found such re-compiled classes in 5,919 (11.6% of the 50,702) artifacts with 360 distinct GAs.

Our procedure fails to identify re-compilation if the source code of the vulnerable classes has been changed significantly. In such cases, SootDiff fails to detect that the vulnerable class and the changed class originate from the same source code. Thus, our results are only a lower bound.

Detection of Uber-JARs (type 2) Bundling multiple open-source artifacts into an Uber-JAR does neither change the bytecode nor the metadata, e.g., file names, FQNs, timestamps, SHA1; the files are just copied into a new JAR file. We applied the following steps to check the prevalence of *Uber-JARs* on Maven Central:

1. We queried Maven Central for classes that have identical FQN as the 254 vulnerable classes.
2. For the found classes, we checked how many classes have the same SHA1 as the vulnerable classes.

We found 36,609 artifacts that contained copies of the 254 vulnerable classes on Maven Central. We identified re-bundling for 222 (87%) out of the 254 classes across 6,723 artifacts with different GA. Further, we found that commonly classes are re-bundled in two or three artifacts with different GAs (with quartiles Q1: 2, Q2: 2 Q3: 3); thus, re-bundling often occurs within the same groupId and artifactId. However, we also found at max that the class `org.bouncycastle.math.ec.custom.sec.SecP256R1Curve` was re-bundled in 27 artifacts with distinct GAs.

Detection of bare Uber-JAR (type 3) Further analysis showed that 3,882 (57%) out of those 6,723 Uber-JARs do not contain a `pom.xml` in the `META-INF` folder, and thus are *bare*.

Detection of re-packaged Uber-JAR (type 4) Uber-JARs may also re-package classes by prepending a fixed string to the classes' FQN to avoid name clashes between classes bundled together from different artifacts. Since FQNs are embedded within a class' bytecode, the bytecode itself, timestamp, and the SHA1 change, but the file name remains unchanged. To discover re-packaged Uber-JARs on Maven Central, we executed the following steps:

1. We queried Maven Central for classes with the same file name (**not** FQN) as the 254 vulnerable classes.
2. For the found classes, we checked how many classes have equal bytecode, in terms of local-sensitive hash distances [OCC13], and are contained in artifacts with different GAs.

We found 16,665 artifacts containing a class with the same file name as a vulnerable class but a different FQN; for 174 of the 254 classes. To check if the bytecode is similar to one of the vulnerable classes, we created a unified bytecode representation using SootDiff (cf. Section 3.8) and computed its local-sensitive hash (TLSH) [OCC13]. If the TLSH distance was lower than 20, we considered it re-packaging. Otherwise, we assumed the class originated from another artifact. In total, we found re-packaging for 17 classes in 89 distinct GAs.

Findings from RQ4: Re-compilation, re-bundling, and re-packaging are common in commercial and open-source projects. We found that more than 87% of the checked classes are re-bundled, and more than 56% are re-compiled on Maven Central. Thus, vulnerabilities reported for one OSS actually affect many other artifacts as well.

3.4.2 RQ5: What Is the Impact of the Modifications on Vulnerability Scanners?

To assess the impact of the identified modifications on vulnerability scanners, we evaluate the precision and recall of the open-source vulnerability scanners DepCheck and Eclipse Steady, the free scanner GitHub Security Alerts, and the commercial scanners[2] C1, C2, and C3 w.r.t. these modifications.

Table 3.4: Test cases for evaluating the precision and recall of vulnerability scanners. The column on the right shows which scanners reported the vulnerability in *RQ2*.

Artifact	Vulnerability	Manual Classification: True Positive	Reported By
httpclient 4.1.3	CVE-2012-6153	yes	Steady, C3
	CVE-2014-3577	yes	Steady, DepCheck, C3
	CVE-2015-5262	no	Steady, DepCheck, C3
jackson-databind 2.9.7	CVE-2018-19362	yes	Steady, DepCheck, C3
	CVE-2018-19361	yes	Steady, DepCheck, C3
	CVE-2018-19360	yes	Steady, DepCheck, C3
spring-webmvc[3] 5.0.0.RELEASE	CVE-2018-1271	yes	Steady, DepCheck, C3
spring-core 5.0.5.RELEASE	CVE-2018-1258	no	DepCheck, C3
	CVE-2018-11039	no	DepCheck
	CVE-2018-1257	no	DepCheck
	CVE-2018-11040	no	DepCheck
spring-expression 5.0.4.RELEASE	CVE-2018-1270	yes	Steady, DepCheck
	CVE-2018-1275	yes	Steady, DepCheck
spring-web 5.0.5.RELEASE	CVE-2018-15756	yes	Steady, DepCheck, C3
	CVE-2018-11039	yes	Steady, DepCheck, C3
guava 23.0	CVE-2018-10237	yes	Steady, DepCheck, C3

As artifacts, we selected from the 20 most-used open-source artifacts the seven artifacts that were themselves vulnerable in the latest version. As vulnerabilities, we choose for each artifact the most recent reported vulnerabilities. Note that we created the test cases based on the results of DepCheck, Eclipse Steady, and C3, as described in Section 3.3.2.

[3] spring-webmvc is also affected by CVE-2018-11039. However, at time of the benchmark creation the vulnerability was not reported by any scanner. We left it out for comparison but added it to our test suite Achilles.

Table 3.4 shows the test cases we used as input. Column *Reported By* shows which scanners reported the vulnerability, and column *Manual Classification* shows if we classified them as true or false positives in our manual review. For example, consider the reported vulnerabilities for `spring-core`. We classified all vulnerabilities as false positives. The scanners C3 and DepCheck reported `spring-core` as vulnerable. However, our manual inspection showed that the vulnerabilities do not affect the artifact spring-core but affect the artifact `spring-web`.

We evaluated the vulnerability scanners' precision and recall for the four modification types that we discovered in our study (cf. Section 3.3.3). For all types, we used the same test cases (cf. Table 3.4) but created the following scenarios using our test generator Achilles (cf. Section 3.7).

Unmodified JAR

- GAV: all artifacts keep their original GAV
- Metadata: the original `pom.xml` and MANIFEST file are kept
- JARs: the artifacts are kept as separate JAR files
- Classes: the bytecode is not modified, providing an anchor for comparing the modifications' impact

Patched JAR (type 1)

- GAV: all artifacts get a slightly modified GAV (appending the string `fix` or `patch`)
- Metadata: the `pom.xml` and MANIFEST are kept
- JARs: the artifacts are kept as separate JAR files
- Classes: classes are re-compiled

Uber-JAR (type 2)

- GAV: single Uber-JAR with a random GAV
- Metadata: the `pom.xml` and MANIFEST of the original artifacts are kept
- JARs: all artifacts are merged into a single Uber-JAR
- Classes: the original bytecode and the timestamps of the original files are untouched

Bare Uber-JAR (type 3)

- GAV: single Uber-JAR with a random GAV
- Metadata: the `pom.xml` and MANIFEST file are removed
- JARs: all artifacts are merged into a single Uber-JAR
- Classes: the original classes are kept, but the timestamps of the files are updated

Re-packaged Uber-JAR (type 4)

- GAV: single Uber-JAR with a random GAV
- Metadata: the `pom.xml` and MANIFEST file are kept
- JARs: all artifacts are merged into a single Uber-JAR
- Classes: the original classes are re-packaged, changing the FQNs, bytecode, and timestamps

For each scenario, we applied the identified modifications (type 1–4) to the artifacts to generate modified JAR file(s) and a Maven project (`pom.xml`). The created Maven project has the (modified) GAV(s) as dependencies, and serves as an input for the scanners. Next, we executed the vulnerability scanners on each project and computed their precision, recall, and F1-score.
Table 3.5 shows the results. The table shows that the scanners' precision and recall heavily differ even for unmodified JARs. The commercial scanner C2 does not find any vulnerabilities in types 1–4, and thus seems unable to deal with modifications at all. GitHub Security Alerts does not find any vulnerabilities in types 2–4, and detects the same vulnerabilities for unmodified and type 1 JARs. C3 performs similarly to GitHub Security Alerts but with higher precision and recall. Based on Table 3.5, Security Alerts, C2, and C3 appear to rely heavily on metadata (file name, GAV, pom.xml) to detect vulnerable artifacts, as they do not detect vulnerabilities in types 2–4.

The results further show that almost all scanners fail to detect vulnerabilities in type 4; DepCheck and C1 are the only scanners reporting vulnerabilities in type 4. Eclipse Steady performs best for unmodified JARs.
The table shows that Eclipse Steady performs better for types 2 and 3 than for type 1. Thus, re-compilation and patching decrease the precision more than lost or modified metadata.

Note that our test cases are based on the results obtained from DepCheck, Eclipse Steady, and C3. The fact that Eclipse Steady achieves a precision of 0.92 and a recall of 1.0 for unmodified JARs means that all findings, except CVE-2015-5262 for `httpclient:4.1.3`, that we use in the case study were marked as true positives in the manual review, and there were no false negatives w.r.t. the findings of DepCheck, and C3. However, the results show that the precision of all scanners heavily decreases with the modifications.

We further investigated the scanners' accuracy to detect the included artifacts in the test cases. Therefore, we studied the generated findings of each scanner and checked which artifacts the scanners identified.

Table 3.5: To understand how the modifications impact the vulnerability scanners' effectiveness, this table reports the vulnerability scanners' precision, recall and F1-score for type 1–4 modifications on the test cases. The scanners marked with * were used in the construction of the test cases; in bold the highest score.

	Unmodified JAR			Patched JAR (type 1)			Uber-JAR (type 2)			Bare Uber-JAR (type 3)			Re-pack. Uber-JAR (type 4)		
	precision	recall	F1	precision	recall	F1	precision	recall	F1	precision	recall	F1	precision	recall	F1
DepCheck*	0.31	0.91	0.47	0.32	**0.91**	0.48	0.17	0.17	0.17		–		0.17	0.18	0.17
Eclipse Steady*	**0.92**	**1.00**	**0.96**	0.33	0.73	0.46	**0.41**	0.75	**0.53**	**0.36**	**0.73**	**0.48**		–	
Security Alerts	0.50	0.45	0.48	0.50	0.45	0.48		–			–			–	
C1	0.64	0.64	0.64	0.32	0.64	0.42	0.34	**0.96**	0.51		–		**0.78**	**0.64**	**0.70**
C2	0.75	0.82	0.78		–			–			–			–	
C3*	0.64	0.82	0.72	**0.64**	0.82	**0.72**		–			–			–	

Table 3.6: The table shows the artifacts that the different scanners failed detect in the scenarios in the form `artifact:version`. In the unmodified scenario the scanners detect the artifacts and the versions from the test fixtures correctly (cf. Table 3.4). The scanners marked with * were used in the construction of the test cases.

	Unmodified JAR	Patched JAR (type 1)	Uber-JAR (type 2)	Bare Uber-JAR (type 3)	Re-pack. Uber-JAR (type 4)
DepCheck*	✓	✓	• jackson-databind:N/A • *SPF*†:N/A	-	• jackson-databind:N/A • *SPF*†:N/A
Eclipse Steady	✓	• spring-*:3.0.0–4.2.0,< 5.0.5	• jackson-databind:2.6–2.9 • spring-*:3.0.0–4.2.0, < 5.0.5	• jackson-databind:2.6–2.9 • spring-*:3.0.0–4.2.0, < 5.0.5	-
Security Alerts*	✓	✓	-	-	-
C1	✓	• spring-core as *SPF*†:5.05	• *SPF*†:N/A	-	• *SPF*†:5.0.0–5.0.5
C2	✓	-	-	-	-
C3*	✓	✓	-	-	-

✓ all artifacts and versions correctly detected
SPF† generalized, unspecific artifact *spring-framework* - the scanner failed to detect the different spring artifacts but only reported spring-framework as a fallback
N/A not assessed version - the scanner could not detect the artifact's version
\- the scanner reported no findings and no artifacts

Table 3.6 shows the artifacts that the scanners reported in the different scenarios. The table describes the cases in which the scanners reported a different than the original artifact or failed to identify their version. The scanners correctly identified `httpclient` and `guava`, including their versions, in all cases in which they reported any findings. The accuracy of the remaining artifacts differs for type 1–4 modifications. For the unmodified artifacts, all scanners detected the original artifacts correctly, including their complete identifier and version. With re-compilation (type 1) applied, the accuracy of Steady and C1 dropped: Steady widened the version range for the spring artifacts, and C1 fell back to the more general `spring-framework`. Similarly, for type 2–4, the accuracy for detecting the `jackson-databind` versions and the different spring artifacts decreases for DepCheck, Steady, and C1.

Table 3.5 and Table 3.6 show that the modifications not only reduce the scanners' accuracy to identify if an artifact is affected by a CVE but also reduce the accuracy to detect the included artifacts.

Findings from RQ5: All scanners struggle to identify vulnerable dependencies if the artifact's JAR files are modified (type 1–4).

3.5 Study Summary

Use of OSS at SAP Our study emphasizes that using OSS is an established practice, even in enterprise software applications. Our observations concerning the use of OSS (*RQ1*) and how many vulnerabilities affect the used OSS (*RQ2*) align with previous studies [Pit16; WD14; Pas+18; Pon+19; Kul+18], which studied the use of (vulnerable) OSS in commercial and open-source applications. Further, the results for *RQ1* show that the number of dependencies heavily differs per project, ranging from 0 to 228. *RQ1* shows that vulnerability scanners must not only check dependencies with the scope *compile* but must check all release dependencies as they constitute a relevant share. Previous studies [Pas+18; Kul+18; Pit16], which point out the importance of transitive dependencies, support this observation.

Notably, we identified a lower number of vulnerabilities (251 CVEs for 534 different GAVs) than reported in the studies [Pit16; Kul+18], which state that each application contains 22.5 different vulnerabilities on average and 81.5% of the applications use outdated dependencies.

The classification of reported CVEs in true positives and false positives in *RQ2* shows that vulnerability scanners tend to produce many false positives; only 903 of 2,558 findings are true positives. While the false-positive warnings are relatively unproblematic in an early development phase as updating a dependency can be done easily, updating dependencies during the release or the operational lifetime can become costly as it impacts the delivery schedule, may cause downtimes, or introduce unexpected regressions during production [PPS18; Bav+15]. Consequently, vulnerability scanners must keep the number of false positives low.
Note that during the course of writing the thesis, we updated our manual classification of the findings and scanner results. Since the NVD description of the vulnerabilities has been updated with further versions of the artifacts, the absolute numbers differ from the ones reported in the publication [Dan+22] (859 true positive vs 903). This emphasizes the need to regularly check the included OSS and the need for scanners to detect known-vulnerable artifacts with high precision, as false negatives leave an application vulnerable.

A significant finding of our research question on how developers include OSS (*RQ3*) is the fact that developers of commercial applications include OSS in re-compiled, re-packaged, re-bundled, or Uber-JARs form with (partially) metadata removed or modified, e.g., different file names, or changed timestamps.

Prevalence & Impact of Modified Open-Source Software Our investigation of modifications on the public open-source repository Maven Central (*RQ4*) shows that not only commercial applications include modified OSS but also open-source projects.

We found that such modifications heavily decrease the precision and recall of vulnerability scanners (*RQ5*)—all scanners struggle to identify vulnerabilities in modified OSS. Only the scanners C2 and Eclipse Steady provide reasonable results in the presence of re-compilation and lost metadata. However, bytecode modifications resulting from re-compilation or re-packaging in combination with the loss of metadata (bare and re-packaged Uber-JARS), e.g., file names and timestamps, are challenging for all scanners. Nevertheless, the results of *RQ3* and *RQ4* show that the modifications are prevalent at SAP and Maven Central, and thus must be addressed by vulnerability scanners. Remarkably, our case study shows that—even the vulnerability scanners used in constructing the test cases—fail to deal with *modified JAR files*.

3.6 Threats to Validity

The results of our study may be affected by errors in the data collection process and the accuracy (immediacy) of the information in the NVD and the scanners' vulnerability database, which are continuously updated. Thus, the exact results may change after their creation.[3] In the following, we discuss the threats to validity in detail.

Use of Open-Source Software at SAP Since we conducted our study on projects developed at SAP, the applicability to commercial Java projects, in general, is limited as the impact of development practices, tools, and guidelines must be considered. Further, the studied projects already apply Eclipse Steady, which may bias our results, as the development teams may update dependencies more regularly compared to teams who do not apply any scanners. Nevertheless, SAP is one of the world's leading software development companies, with a diverse product portfolio, and our study aligns with previous open-source and software industry case studies [Pit16; WD14; Pas+18; Pas+22; Pon+19; Kul+18]. Additionally, the 20 most-used arti-

[3]The initial results were created in 2019 and updated at the end of 2022 while writing this thesis. Since the found vulnerability descriptions in the NVD have been updated, we also updated the manual classification, leading to a different number of true positive CVE compared to the publication [Dan+22].

facts are also popular within the open-source community (cf. Section 3.2.2), and the modifications also occur on Maven Central, indicating that our results are also applicable to other—particularly to open-source—projects.

Our decision to check vulnerabilities for the 20 most-used OSS and the selection of scanners influences the concrete number of reported findings, false positives, and false negatives; other scanners may use more or less precise vulnerability databases or apply other matching techniques. However, to achieve soundness in determining if a vulnerability affects a given OSS, without relying on a single (and potentially erroneous) vulnerability database, we semi-manually classified the 2,558 findings.

To limit the likelihood that we miss a vulnerability for a given OSS (false negatives), we applied the scanners Steady, DepCheck, and C3 to generate an initial set of findings. The scanners apply different matching strategies: name-based vs. code-based, and rely on different databases: Steady's Database [Ecl20], NVD [NIS20], and a commercial database. Nonetheless, all scanners and databases are continuously updated and cannot guarantee the absence of false negatives or false positives. Thus, the absolute number of false positives differs with the chosen scanners and databases. Crucially, our results regarding the use of vulnerable OSS and the ratio of false positives align with the results reported by other studies [Pit16; Kul+18; Pas+18; Pon+19] or are even less.

Prevalence & Impact of Modified Open-Source Software To check the prevalence of the modifications in open-source projects, we checked how often the vulnerable classes that we identified in *RQ2* occur on Maven Central in modified form. This selection may be inaccurate because we do not know how often these classes are re-compiled or re-bundled compared to other classes, e.g., if the found classes are more often or rarely re-bundled. Hence, our results may under- or over-approximate. However, the results still show that such modifications are also popular on Maven Central and must be addressed by vulnerability scanners.

A further pitfall in evaluating the scanners' precision and recall is the fact that the evaluated vulnerability scanners Steady, DepCheck, and C3 were also used in the creation of our test suite Achilles. Although we semi-manually classified the 2,558 findings to ensure soundness, we cannot ensure completeness, as false negatives (findings missed by all scanners) may occur.

A bias towards Eclipse Steady may only occur in the case of unmodified JAR, as the tests used to create the benchmark were confirmed as correct during the semi-manual review. For modified OSS, instead, Achilles generates new JAR files with new metadata, e.g., digest, GAV. As the newly created modified JAR files are unknown to all scanners, they allow fair comparison.

Further, the conclusion that vulnerability scanners struggle with modifications is independent of any particular scanner, as even the scanners used in the construction fail—to different degrees—to deal with modified JARs. Particularly, the good result of C1 for type 4 shows that the study does not favor the scanners that have been used in Achilles' creation.

3.7 Achilles: Test Suite for Detecting Modified Open-Source Software

Our study shows that vulnerability scanners struggle to identify known vulnerabilities in (modified) OSS and that the scanners' precision and recall heavily differ (cf. Section 3.4). Crucially, our study shows that modified OSS, compromising re-compiled, re-packaged, or re-bundled classes from other open-source projects, commonly occur on Maven Central. Thus, further research and development are necessary to alleviate this situation and improve scanners' detection techniques for modified OSS.

As, to the best of our knowledge, no test suite exists to facilitate a reproducible and comparative assessment of vulnerability scanners w.r.t. these modifications, we developed Achilles. Achilles provides the options to include (vulnerable) OSS dependencies, remove vulnerable classes, and apply the modifications: re-compilation, re-packaging, Uber-JAR creation, and removal of metadata, e.g., file names, timestamps, or pom.xml files. As an input, Achilles requires a set of GAVs and a ground truth, listing the vulnerabilities that affect the given GAVs. With Achilles, we also provide 2,558 test cases and ground truth, which we created in our manual classification in our case study (cf. Section 3.3).

Since benchmarks play a strategic role in computer science research and development by providing a ground truth for evaluating algorithms and tools, we constructed Achilles based on the following criteria introduced by the widespread DaCapo benchmark [Bla+06].

Diverse real-world applications: The test cases should not consist of artificially created programs. Instead, the benchmark should contain OSS and vulnerabilities collected from real-world projects to provide a compelling focus for evaluating real-world usage.

Detecting vulnerable OSS (precision and recall): The test cases and ground truth should enable measuring if a vulnerability scanner successfully detects included OSS with published vulnerabilities (recall) and to what extent a scanner raises false warnings (precision).

Automation and ease of use: The test cases should be in a format consumable by vulnerability scanners and enable the measurement of their accuracy.

In the following, we explain how Achilles implements these criteria, its organization, and its use.

3.7.1 Diverse Real-World Applications

To closely resemble real-world applications, we directly create the test cases from the results of our study. As test data, we choose the 2,558 findings from *RQ2* for the 534 Maven artifacts and 251 distinct vulnerabilities together with our manual classification into true and false positives (cf. Section 3.3). In result, each test case contains the identifier (GAV) of an artifact and the list of (true positive) vulnerabilities. As we identified that the same artifacts are also used in the open-source community, the test cases not only replicate the settings at SAP but also in open-source projects. Further, Achilles enables the application of the identified modifications to the artifacts before serving them as input to a vulnerability scanner.

3.7.2 Detecting Vulnerable Open-Source Software

For evaluating a vulnerability scanner's precision and recall, each test fixture is specified as a human-readable JSON file, stating the published vulnerability, the (affected) GAV, a short description, the ground truth, and a timestamp specifying when the test fixture was created. Listing 3.1 shows a test fixture for the vulnerability *CVE-2016-3720*.

```
1  { "comment": "https://github.com/jeremylong/DependencyCheck/issues/517 -- XML
       Injection",
2  "gav": { "version": "2.4.3",
3    "artifactId": "jackson-dataformat-xml",
4    "groupId": "com.fasterxml.jackson.dataformat" },
5  "cve": "CVE-2016-3720",
6  "details": [
7    {"contained": true,
8     "commit": "f0f19a4c924d9db9a1e2830434061c8640092cc0",
9     "affectedFile": "/com/fasterxml/jackson/dataformat/xml/XmlFactory.class",
10    "qname": "com.fasterxml.jackson.dataformat.xml.XmlFactory",
11    "diff": "..." },
12   {"contained": true,
13    "commit": "f0f19a4c924d9db9a1e2830434061c8640092cc0",
14    "affectedFile": "/com/fasterxml/jackson/dataformat/xml/XmlFactory.class",
15    "qname": "com.fasterxml.jackson.dataformat.xml.XmlFactory(ObjectCodec,int,int
          ,XMLInputFactory,XMLOutputFactory,String)",
16    "diff": "" }],
17 "vulnerable": true }
```

Listing 3.1: Example test fixture of Achilles for the artifact jackson-dataformat-xml:2.4.3 and the vulnerability CVE-2016-3720. The test fixture contains the result of the manual classification (*vulnerable*), and—if available—information about the vulnerable classes and the commit fixing the vulnerability.

Optionally, if we could identify the source-code commit fixing the vulnerability (cf. Section 3.3.2), the test fixture also contains the commit-hash, the FQNs of the vulnerable constructs, SootDiff's result, and a boolean flag whether the artifact contains the vulnerable bytecode.

3.7.3 Automation and Ease of Use

For composing a Maven project with the (un-)modified open-source artifact and a ground truth, Achilles provides a graphical user interface. Figure 3.3 shows the workflow for using Achilles to create test cases and for evaluating a scanner's precision and recall. First, one must choose which test fixtures—that are the (vulnerable) artifacts—the test Maven project should include. Next, one can choose to apply modifications to the artifacts. Based on the selection, Achilles generates the (modified) JAR files and ground truth. Second, the generated output is used as input for a scanner. Third, the findings reported by the scanner are compared with the ground truth to evaluate the scanner's precision and recall.

For generating modified JAR files, Achilles can apply different modifications, based on the four identified modifications in our study (cf. *RQ3*), shown in the form of a feature diagram in Figure 3.4:

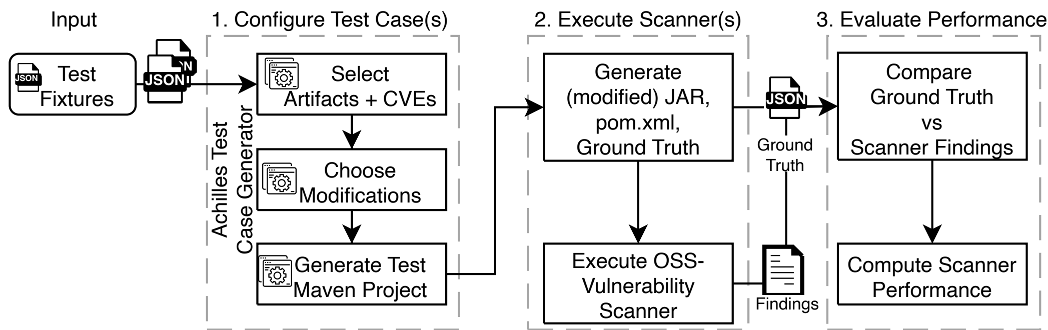


Figure 3.3: Achilles' process steps for evaluating OSS-vulnerability scanners.

GAV To evaluate to what extent vulnerability scanners are resilient to simple changes in the GAV or to an unknown GAV: the original GAV can be kept, can be modified by appending a version suffix (e.g., the string `-fix-01`), which developers use to indicate own forks (*type 1*), or can be replaced by a random GAV, which is usually the case for Uber-JARs (*type 3–4*).

Metadata To evaluate to what extent scanners are resilient to modified metadata: the metadata, in particular, the `pom.xml` file in the `META-INF` folder, can be removed, which happens if developers use legacy build-automation tools to create Uber-JARs (*type 3*).

JAR To encompass the effect of *Uber-JARs*: all artifacts can be kept as separate JAR files (*type 1–2*), or can be bundled into a single *Uber-JAR* (*type 2–4*). Optionally, the original timestamps of the files in the JAR can be kept.

Classes To encompass re-compilation: the original class files can be copied (*type 2–3*), the source code can be re-compiled using the original FQN (*type 1*), or the classes can be re-packaged by prepending the string `com.repackage` to each FQN (*type 4*).

Vulnerable Code Optionally, Achilles can remove the vulnerable classes from a JAR file. This can be used to evaluate if vulnerability scanners apply code-based matching successfully detect if the vulnerable code is contained in a given JAR.

If one does not apply any modifications, Achilles uses the bold-marked settings in Figure 3.4, producing unmodified artifact JAR files.

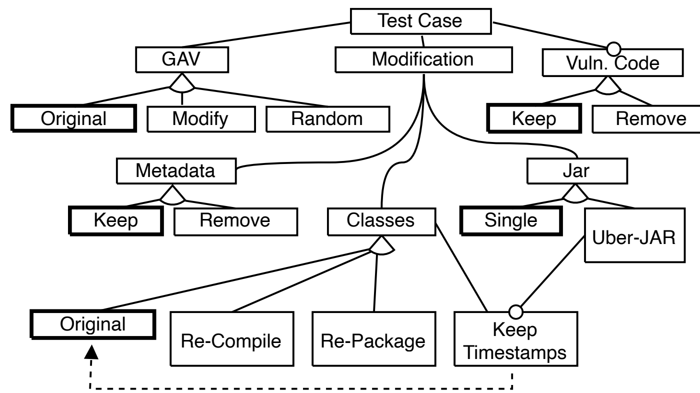


Figure 3.4: Feature diagram of the options that the Achilles generator provides for re-creating the modifications on JAR files. By default, the features in bold are selected, generating unmodified JAR files.

As an output, Achilles generates a Maven project (`pom.xml`) with the (modified, vulnerable) JAR files as dependencies and a ground truth. Finally, one can use the generated project directly as input for a vulnerability scanner. To compute precision and recall, the vulnerabilities affecting the included OSS are specified in the ground truth based on the selected test fixtures.

3.7.4 Organization and Distribution

To allow the community to extend Achilles and provide further test cases, we distribute Achilles publicly on GitHub (cf. Chapter 6). Due to license issues, Achilles itself does not contain the artifacts' source- or bytecode; instead, the JAR files are downloaded and modified on demand from the Maven Central repository.

Figure 3.5 shows the organization of the test suite. The folder `generator` contains Achilles' Java source code, and at the top-level, the repository provides an executable JAR of the test case generator. The folder `case-study` contains the scenarios that we used in Section 3.4.

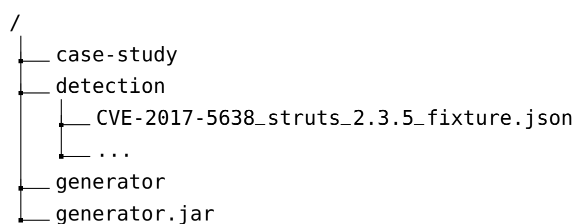


Figure 3.5: Layout of Achilles

3.8 SootDiff: An Approach for Identifying Modified Open-Source Software

Our case study shows that modifications heavily decrease the effectiveness of vulnerability scanners. The modifications have in common that file attributes, such as timestamps, file names, and artifact names, are removed or changed, and that the bytecode of the classes is modified: If an artifact is forked and re-compiled, the bytecode changes when a different Java compiler or different target version is used. If an artifact is (re-)bundled into an Uber-JAR, the timestamps of the files may change. If an artifact is re-packaged, the classes' FQN and the references in the bytecode change.
To detect vulnerabilities in modified dependencies, vulnerability scanners require techniques that are resilient to bytecode modifications to either identify the original artifact from which the bytecode originates and then check if known vulnerabilities for the original artifact exist, or to check if the bytecode corresponds to any known-vulnerable bytecode.

We develop the approach SootDiff to cope with bytecode modifications. SootDiff uses the intermediate representation Jimple [VH98] of the static analysis framework Soot [Lam+11] to create a unified bytecode representation churning out bytecode dissimilarities, without relying on the availability of source code or unmodified bytecode.

SootDiff integrates and enriches Soot by implementing Myers' Diff algorithm tailored for Jimple and additional optimization steps to reduce dissimilarities originating from different Java compilers or versions. In contrast to Java bytecode, which uses more than 200 different instructions, Jimple uses only 15 distinct instructions. As a result, many different but functionally equivalent code constructs are likely to coincide on the Jimple level. In contrast to techniques that identify artifacts using features or metadata that are orthogonal to bytecode, e.g., method signatures or files' timestamps, SootDiff checks a class' bytecode including its method bodies, which is required to identify the included artifact's exact version given the artifact's bytecode—which is in turn required to check if a known vulnerability affects the given artifact.

To validate the feasibility of SootDiff for comparing bytecode generated by different compilers and versions, we compare it to a naïve approach directly comparing the bytecode. Therefore, we compiled 16 Java source-code files with different Java compilers and for different Java versions, yielding 144 different bytecode classes.

For the generated bytecode classes, we checked how often SootDiff and a bytecode comparison correctly identify if two bytecode classes originate from the same source code. Our comparison shows that SootDiff successfully detects that the bytecode classes originate from the same source code for 102 test cases. In contrast, a comparison based on bytecode succeeds for 58 test cases only. Thus, SootDiff can serve as a basis for designing resilient code-based matching techniques. In the following, we introduce SootDiff's design and evaluate its performance.

3.8.1 Dissimilarities Introduced by Java Compilers

Several compilers exist for generating Java Virtual Machine (JVM)-compatible bytecode from Java source code. The most widespread compilers are Oracle's Javac (the Java Development Kit (JDK)'s default compiler), the Eclipse Compiler for Java (ECJ)[4], IBM Jikes[5], or the GNU Compiler[6] for the Java programming language (GCJ). Since the Java language specification [Ora23d] does not state if and how a compiler should optimize certain Java language features, optimizations are a design decision of the compiler vendor. Consequently, the bytecode created by different compilers is subject to different optimizations.

As an example, consider the Java source class `Point2d`[7] in Listing 3.2. The Java class declares a method `dprint(String s)` with a string parameter. If the class' private boolean field `debug` is set to true, the method prepends the argument with the string `Debug:` and prints it on the command line.

```
1  class Point2d {
2  private boolean debug;
3  public void dprint(String s){
4    if (debug)
5      System.out.println("Debug:"+s);
6    }
7  }
```

Listing 3.2: Example source class `Point2d.java`.

[4] https://www.eclipse.org/jdt/core/
[5] https://sourceforge.net/projects/jikes/
[6] https://web.archive.org/web/20070509055923/http://gcc.gnu.org/java/
[7] Sample Java Classes University Illinois https://www.cs.uic.edu/~sloan/CLASSES/java/

If the class is compiled with Javac and ECJ for the target version 1.8, both compilers apply different modifications, shown in Listing 3.3 and Listing 3.4. Listing 3.3 shows the (decompiled) bytecode generated by the ECJ and Javac compiler for target version 1.8. Listing 3.4 shows the corresponding Jimple code. We will introduce Jimple in the next section in detail.

```
1   class Point2d {
2   private boolean debug;
3
4   public void dprint(String);
5   Code:
6    0: aload_0
7    1: getfield      #20
          // Field debug:Z
8    4: ifeq          29
9    7: getstatic     #35
          // Field java/lang/System.out:
          Ljava/io/PrintStream;
10  10: new           #41
        // class java/lang/
        StringBuilder
11  13: dup
12  14: ldc           #43
        // String Debug:
13  16: invokespecial #45
        // Method java/lang/
        StringBuilder."<init>":(Ljava/
        lang/String;)V
14  19: aload_1
15  20: invokevirtual #47
        // Method java/lang/
        StringBuilder.append:(Ljava/
        lang/String;)Ljava/lang/
        StringBuilder;
16  23: invokevirtual #51
        // Method java/lang/
        StringBuilder.toString:()Ljava/
        lang/String;
17  26: invokevirtual #55
        // Method java/io/PrintStream.
        println:(Ljava/lang/String;)V
18  29: return
```

(a) Decompiled Bytecode from ECJ and target version 1.8

```
1   class Point2d {
2   private boolean debug;
3
4   public void dprint(String);
5   Code:
6    0: aload_0
7    1: getfield      #4
          // Field debug:Z
8    4: ifeq          32
9    7: getstatic     #8
          // Field java/lang/System.out:
          Ljava/io/PrintStream;
10  10: new           #9
        // class java/lang/
        StringBuilder
11  13: dup
12  14: invokespecial #10
        // Method java/lang/
        StringBuilder."<init>":()V
13  17: ldc           #11
        // String Debug:
14  19: invokevirtual #12
        // Method java/lang/
        StringBuilder.append:(Ljava/
        lang/String;)Ljava/lang/
        StringBuilder;
15  22: aload_1
16  23: invokevirtual #12
        // Method java/lang/
        StringBuilder.append:(Ljava/
        lang/String;)Ljava/lang/
        StringBuilder;
17  26: invokevirtual #13
        // Method java/lang/
        StringBuilder.toString:()Ljava/
        lang/String;
18  29: invokevirtual #14
        // Method java/io/PrintStream.
        println:(Ljava/lang/String;)V
19  32: return
```

(b) (Decompiled) bytecode from Javac target version 1.8

Listing 3.3: Comparison of the (decompiled) bytecode generated with ECJ compiler and Javac, both with target version 1.8, from the source class `Point2d.java`. References to the constant pool are resolved and typeset as comments in green.

```
1   class Point2d extends Object{
2   private boolean debug;
3   public void dprint(String){
4    Point2d r0;
5    java.lang.String r1, $r6;
6    boolean $z0;
7    java.lang.StringBuilder $r2, $r4;
8    java.io.PrintStream $r3;
9
10   r0 := @this: Point2d;
11   r1 := @parameter0: java.lang.
         String;
12   $z0 = r0.<Point2d: boolean debug>;
13
14   if $z0 == 0 goto label1;
15   $r3 = <java.lang.System: java.io.
         PrintStream out>;
16   $r2 = new java.lang.StringBuilder;
17   specialinvoke $r2.<java.lang.
         StringBuilder: void
         <init>()>("Debug:") ;
18   $r4 = virtualinvoke $r2.<java.lang
         .StringBuilder: java.lang.
         StringBuilder append(java.lang
         .String)>(r1);
19   $r5 = virtualinvoke $r5.<java.lang
         .StringBuilder: java.lang.
         String toString()>();
20   virtualinvoke $r3.<java.io.
         PrintStream: void println(java
         .lang.String)>($r5);
21  label1:
22   return;} }
```

(a) Jimple parsed from bytecode generated with the Compiler ECJ target version 1.8

```
1   class Point2d extends Object{
2   private boolean debug;
3   public void dprint(String)
4   { Point2d r0;
5    java.lang.String r1, $r6;
6    boolean $z0;
7    java.lang.StringBuilder $r2, $r4,
         $r5 ;
8    java.io.PrintStream $r3;
9
10   r0 := @this: Point2d;
11   r1 := @parameter0: java.lang.
         String;
12   $z0 = r0.<Point2d: boolean debug>;
13
14   if $z0 == 0 goto label1;
15   $r3 = <java.lang.System: java.io.
         PrintStream out>;
16   $r2 = new java.lang.StringBuilder;
17   specialinvoke $r2.<java.lang.
         StringBuilder: void
         <init>()>() ;
18   $r4 = virtualinvoke $r2.<java.lang
         .StringBuilder: java.lang.
         StringBuilder append(java.lang
         .String)>("Debug:");
19   $r5 = virtualinvoke $r4.<java.lang
         .StringBuilder: java.lang.
         StringBuilder append(java.lang
         .String)>(r1);
20   $r6 = virtualinvoke $r5.<java.lang
         .StringBuilder: java.lang.
         String toString()>();
21   virtualinvoke $r3.<java.io.
         PrintStream: void println(java
         .lang.String)>($r6);
22  label1:
23   return;} }
```

(b) Jimple parsed from bytecode generated with Javac target version 1.8

Listing 3.4: Comparison of the Jimple code parsed from the bytecode generated with ECJ compiler and Javac, both with target version 1.8, from the source class `Point2d.java`.

Oracle's Javac compiler produces slightly different bytecode than the ECJ, shown in Listing 3.4b with the differences highlighted. The ECJ compiler inlines the constant string `"Debug:"` directly into the `StringBuilder` constructor in Line 17 in Listing 3.4a, and thus generates one variable and one invoke instruction less (`$r5`). In contrast, the bytecode generated by Javac first creates an empty `StringBuilder` and then appends the string `"Debug:"` to the `StringBuilder` in line 18 in Listing 3.4b. Thus, the bytecode generated by Javac and the bytecode generated by ECJ for the source class `Point2d.java` differs, and the Javac bytecode contains an addition `invokespecial` instruction for appending the string `Debug:` to the empty initialized `StringBuilder` in Listing 3.3b.

The example shows that different Java compilers apply different optimization strategies, resulting in different bytecode. However, bytecode dissimilarities occur not only between different Java compilers, but a single compiler may also produce different bytecodes for the same source class. Java compilers support the option to generate bytecode for different Java language versions. As different versions of the Java language and JVM support different bytecode instructions, a compiler may generate different bytecode for different versions, e.g., Java 1.8 or Java 1.5.

Thus, simply comparing the bytecode for detecting re-compiled classes is insufficient as different Java compilers produce slightly different bytecode; even one compiler produces different bytecode for different Java versions. Furthermore, the comparison of Listing 3.4b and Listing 3.4a shows that the plain usage of Jimple also is insufficient since even the (unoptimized) Jimple code parsed from the bytecode generated by Javac and ECJ differs.

3.8.2 Jimple: Intermediate Bytecode Representation

The static analysis framework Soot [Lam+11] uses an intermediate representation named Jimple [VH98] to represent Java source and bytecode for static code analysis. Jimple serves as an abstraction layer by drastically reducing the number of instructions needed to represent bytecode. The Jimple intermediate representation maps the 200 different bytecode instructions to 15 Jimple instructions. The reduced number of instructions mitigates the dissimilarities introduced by different Java versions and compilers. SootDiff relies on Jimple to compare the bytecode generated by different compilers and to ease the detection of code clones or for determining if two bytecode classes originate from the same source code.

Listing 3.3 and Listing 3.4 show the bytecode and the Jimple representation of the Java class `Point2d.java` from Listing 3.2. Like bytecode, Jimple transforms Java's control structures, e.g., if-else, loops, to goto instructions and program labels. For instance, the if-condition in the source code, shown in Listing 3.2 in Line 4, is translated to the branch instruction `ifeq` in bytecode, shown in Listing 3.3b in Line 8, and transformed to `if $z0 == 0` in Jimple, shown in Listing 3.4b in Line 14. The majority of lines in the bytecode, namely the Line 10-18, constructs the string `"Debug:"+s` using `java.lang.StringBuilder` referring to methods and strings (#9–#14) stored in the class' constant pool.

In contrast to bytecode, the Jimple code declares all variables and types explicitly, shown in Listing 3.4b. The bytecode instructions for constructing the string and invoking the `java.lang.StringBuilder` are fully resolved in Lines 17-20. Moreover, the condition on the boolean variable `debug` (`$z0`) is made explicit in Line 14.

3.8.3 Compare Modified Bytecode

To check if two bytecode classes originate from the same source code, SootDiff uses the Jimple representation and reduces dissimilarities, e.g., string concatenation as in the example above. This enables SootDiff to match classes with equivalent behavior even when they are produced with different Java compilers or for different Java versions, e.g., when an artifact has been re-compiled.

Figure 3.6 gives an overview of SootDiff's comparison.

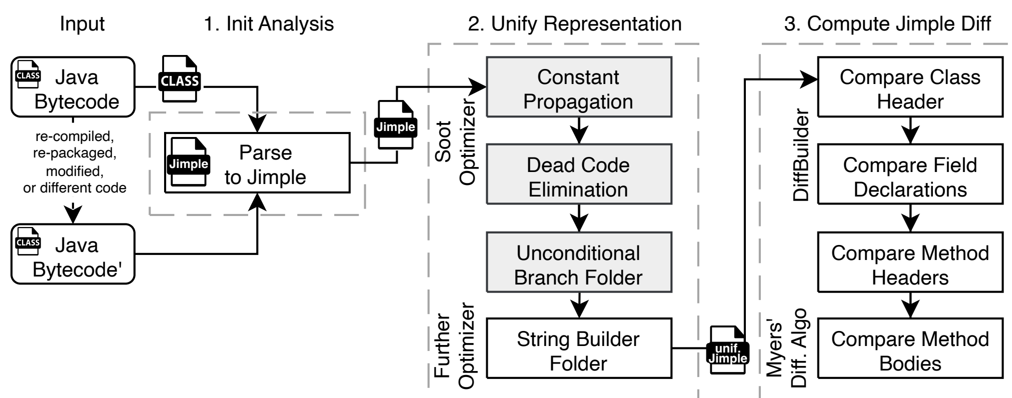


Figure 3.6: SootDiff's process steps for comparing the bytecode of two classes using Jimple and optimizers to be resilient against bytecode modifications. Soot's optimizers are in gray.

Input As an input, SootDiff uses the bytecode of two or multiple classes and outputs a set of diff chunks utilizing clone detection algorithms.

1. Producing Jimple SootDiff uses the Soot framework [Lam+11] to produce Jimple from the bytecode classes. To do so, SootDiff passes the bytecode files to Soot to transform them to Jimple.

2. Unify Representation Next, SootDiff optimizes the Jimple representation using Soot's internal Jimple optimizer [VH98]. In the optimization step, SootDiff applies constant-propagation, dead-code-elimination, and unconditional-branch folder to the Jimple code [VH98; Lam+11]. These optimization steps reduce dissimilarities.

To reduce further dissimilarities, SootDiff runs additional optimizers on the Jimple representation. At present, SootDiff contains an optimization step to reduce dissimilarities when constructing strings in Java using the `java.lang.StringBuilder` API, as shown in the Listing 3.4b in Section 3.8.1. Further steps for optimizing and evaluating simple arithmetic expressions can be added.
For instance, an optimization step may transform an instruction of the form `int val = 7 + 5 * 3 + x` to `int val = 22 + x` by evaluating the arithmetic expression, which is, partially done by the ECJ. Finally, SootDiff compares the optimized Jimple representations using Myers' Diff algorithm [Mye86] and reports the differences in the form of diff chunks using `apache.commons.DiffBuilder`.

3. Computation of Jimple Diff Finally, SootDiff computes the diff chunks between the compared classes, using Myers' algorithm [Mye86] to compare method bodies using `java-diff-utils`.[8] Myer's diff algorithm is a greedy algorithm for calculating the differences between two strings and a sequence of edits to convert one string to the other. The algorithm recursively finds the longest common subsequence with the smallest edit sequence for two sequences, and is, for instance, used in GNU DiffUtils. Once this is done, the algorithm recursively compares two subsequences preceding and following the matched sequence until no more sequences are left for comparison.
Note that SootDiff's method body comparison is open to be used with different code clone detection techniques.

[8] `https://github.com/java-diff-utils/java-diff-utils`

For comparing the signatures of classes, methods, and fields semantically, SootDiff implements `apache.commons.DiffBuilder`[9] directly in Soot. Thus, SootDiff's comparison is independent of the ordering of methods or fields in bytecode, e.g., different compilers may output class member declarations and method parameters in different sequences, which would yield a difference in Myer's algorithm.

3.8.4 Evaluation

In the following, we evaluate the effectiveness of SootDiff to detect if two bytecode classes originate from the same source code. To do so, we compare SootDiff with a simple bytecode comparison approach using GNU DiffUtils. In our evaluation, we apply SootDiff and DiffUtils on the bytecode generated by different Java compilers and for different Java versions. As Java source-code test classes, we use the Java sample programs provided by the University of Illinois [Pro04], which cover most features of the Java library classes.
To generate the test cases shown in Table 3.7, we compile the Java source code using the compilers Javac, ECJ, and GCJ for the language versions 1.5 to 1.8. As reference bytecode classes, we generate bytecode for Java 1.8 using Javac.

To evaluate SootDiff's effectiveness, we compare the diff-results gained from comparing the different bytecode with DiffUtil directly against the diff-results produced by SootDiff. To compare the "plain" bytecode of two classes with DiffUtils, we first parse the generated bytecode and generate a textual representation using ASM's `org.objectweb.asm.util.TraceClassVisitor`. Afterward, we compare the textual representations generated by the `TraceClassVisitor` using the established diff library `com.github.difflib.DiffUtils`. Thereby, we exclude the Java language version information contained in the bytecode. To back up our results, we (re-)validate them by running the Unix tool `diff` on the binary files. To generate SootDiff's results, we apply SootDiff on the generated bytecode classes and create diff chunks using Myers' diff algorithm. We report two classes as equal if SootDiff does not report any diff chunks.

Table 3.7 shows the comparison's results. The table highlights in green the test cases for which the bytecode comparison fails but SootDiff produces correct results. The table shows that the bytecode comparison succeeds for 58 out of 144 test cases only, although we ignore the Java version information in the generated `.class` files. Consequently, the generated bytecode contains more differences than the Java

[9] https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/builder/DiffBuilder.html

version, which validates our initial assumption that different compilers produce different bytecode for the same source code. In contrast, the table also shows that for 102 out of 144 test cases the comparison of Jimple code succeeds; even without any Jimple optimization steps, 66 test cases succeed.

Table 3.7: Evaluation of the efficiency of bytecode and SootDiff's comparison on bytecode generated by different Java compilers.

	javac			ecj				gcj	
	1.5	1.6	1.7	1.5	1.6	1.7	1.8	1.5	1.6
ArrayDemo.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
DivBy0.java	✓/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
FunctionCall.java	✓/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
HelloData.java	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●
HelloWorld.java	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●
HelloWorldException.java	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●
KeyboardIntegerReader.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
KeyboardReaderError.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
KeyboardReader.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
MyFileReader.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
MyFileWriter.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
Point2d.java	✗/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
Point3d.java	✓/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
PointerTester.java	✓/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
PointerTester$Point2d.java	✗/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
PointerTester$Point3d.java	✓/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●

[1] results are in the form `bytecode/Jimple`
[2] Comparison of bytecode: fail ✗/ success ✓
[3] Comparison of Jimple: fail ❍/ success ●
[4] The bytecode comparison ignores the different bytecode version number in the generated `.class` files.
[5] Highlighted in green are the cases in which SootDiff correctly detects that the bytecode originate from the same source class, although the Bytecode differs.
[6] The Sample Java Source Classes are from a benchmark from Prof. Robert H. Sloan, University Illinois [Pro04]

The test case `DivBy0` fails because Javac and ECJ optimize unused locals differently, as shown in Listing 3.5a-3.5c. The bytecode generated by ECJ optimizes the assignment to the variable `i` in Listing 3.5a in Line 6, whereas the Javac leaves the assignment unchanged in Line 6 in Listing 3.5b. Since the divide instruction in the Javac bytecode may throw an `ArithmeticException` Soot does not remove the assignment statement. Consequently, the optimized Jimple code differs, and thus SootDiff's comparison fails. As both bytecode versions would result in different runtime behavior, this behavior is acceptable for checking if two artifacts originate from the same source code—or more importantly, when checking for vulnerable artifacts—share the same vulnerabilities.

```java
1  void funct2(){
2  println("");
3  int i, j, k;
4  i = 10;
5  j = 0;
6  k = i/j;}
```

(a) Java Source Code DivZero.java

```
1  void funct2(){
2  int i2;
3  PrintStream $r0;
4  $r0=PrintStream.out;
5  invoke $r0.<println>;
6  i2 = 10 / 0;
7  return;}
```

(b) Jimple from bytecode generated with Javac target ver. 1.8

```
1  void funct2(){
2  PrintStream $r0;
3  $r0=PrintStream.out;
4  invoke $r0.<println>;
5  return;}
```

(c) Jimple from bytecode generated with ECJ target ver. 1.8

Listing 3.5: Comparison of compiler optimizations of Javac and ECJ from the source class DivZero.java. Javac and ECJ optimize unused locals differently, resulting in bytecode dissimilarities.

Further 36 test cases fail because ECJ and Javac optimize control structures differently, leading to a different organization of basic blocks and branch instructions. For instance, the test case KeyboardReader contains a while loop that depends on a condition of the form `while z != 0`. Javac transforms this condition to a conditional jump of the form `if z == 0 goto end of loop`, whereas the ECJ generates a conditional jump of the form `if z!=0 goto loop`. Currently, SootDiff does not provide an optimization step for these differences.

Findings: SootDiff can effectively detect if two bytecode classes originate from the same source code in 102 out of 144 cases, whereas bytecode comparison succeeds in 58 cases only.

3.9 Related Work

In this section, we present related studies investigating the use of (vulnerable) open-source dependencies and related test suites for detecting vulnerabilities in OSS. Further, we discuss related work for SootDiff that aims to detect different types of code clones (modifications).

3.9.1 Case Studies: Use of Vulnerable Open-Source Software

The use of (vulnerable) OSS and its security implications have been well-studied in literature in commercial projects [PPS18; Pas+22; WD14; Pit16; Mar+18; SE13] and open-source projects [Kul+18; HG22; Bav+15; Bog+16; Hei+11]. Most related is the empirical study on the use of vulnerable open-source dependencies in commercial Java applications developed at SAP by Pashchenko et al. [Pas+18; Pas+22]. The authors applied the code-based vulnerability matching approach of Eclipse Steady to assess the risk and the required effort for mitigating vulnerabilities in included transitive OSS from the developers' perspective. The authors found that although most vulnerabilities are located in transitive dependencies, developers can fix—in fact—80% of the vulnerable release dependencies either by fixing a bug in a single OSS project or by updating a direct dependency [Pas+22]. The authors conclude that although developers do not actively include transitive dependencies, they can easily recognize and update those since most transitive dependencies originate from a framework dependency that the developers actively include and are actually well aware of. To account for this observation, the authors provide a new methodology (Vuln4Real) for assessing the impact of vulnerable dependencies on a project by grouping (transitive) vulnerabilities that developers can "easily" fix into groups based on their groupId, as we did in our study.

Further studies in the commercial context by Synopsys [Syn23], and Williams and Dabirsiaghi [WD14] report that more than 67% of the investigated applications include vulnerable dependencies with on average 22.5 vulnerabilities per dependency [Syn23].

In the open-source context, Kula et al. [Kul+18] and Bavota et al. [Bav+15] investigate the use of (vulnerable) OSS and how developers update known-vulnerable dependencies. Kula et al. [Kul+18] found that the majority (81.5%) of studied projects include outdated or vulnerable OSS. Critically, both studies conclude that most developers are reluctant to update even vulnerable dependencies since they are either unaware that the dependencies are vulnerable, are unaware that new versions exist, or are afraid of breaking the application, leaving the applications vulnerable.

Lopez et al. [Lop+17] also investigate how developers include OSS. Analogous to our observation of modified artifacts' JARs, the authors found that source-code clones and modified source code occur in 80% of all studied open-source projects.

While these studies investigate the use of known-vulnerable OSS in commercial or open-source contexts, the studies do not examine the prevalence of modified OSS nor their impact on the precision and recall of vulnerability scanners. Our case study complements existing work and shows that—on top of source-code clones—vulnerable code clones are also introduced during the build process of downstream projects by forking, re-bundling, and re-packaging.

3.9.2 Test Suites: Vulnerabilities in Open-Source Software

Most related to Achilles is the SourceClear benchmark [Sou20]. The benchmark provides test cases invoking the vulnerable code of open-source artifacts for Java, Scala, Ruby, Python, C, C#, JavaScript, PHP, and Go. However, the benchmark does not replicate modifications, such as re-packed class files, deleted or lost metadata. Further, the benchmark's ground truth does *not* specify the vulnerable artifact nor the published vulnerabilities. Instead, the benchmark's ground truth specifies the number of vulnerable, direct, and transitive dependencies, and the call chain invoking the artifact's vulnerable code.

Vulnerability collections and benchmarks for assessing the precision and recall of static and dynamic application security testing tools are BugBox [Nil+13] for PHP, the SAMATE reference data set [NIS18], SecuriBench [Liv12], and the Juliet Test-Suite [NIS17]. The benchmarks provide test cases with known security flaws for evaluating application security testing tools but do not provide test cases for detecting known-vulnerable OSS.

Widespread benchmarks like the DaCapo [Bla+06] benchmark suite, the Qualitas Corpus [Tem+10], or the XCorpus [Die+17] provide test corpora for evaluating static analyses but do not provide test cases for detecting known vulnerabilities in OSS.

Vulnerability databases like the NVD [NIS20] and ExploitDB [Off20] miss information or are too coarse-grained to enable a comparison of vulnerability scanners: they do not specify which classes and methods are vulnerable nor provide ground truth, and the identifiers, e.g., CPEs, often over- or under-approximate. In contrast, Achilles' test cases contain a curated set of vulnerable OSS and manually crafted ground truth.

3.9.3 Code Clone Detection

Finding equal or similar parts within source code is a well-known problem called *code clone detection*. Koschke [Kos07] distinguishes between three different types of code clones:

Type 1 clones are an exact copy of the original code without modifications, except for whitespaces and comments.

Type 2 clones are syntactically identical copies with only slight renaming, e.g., renaming variables or function identifiers.

Type 3 clones are copies with further modifications, e.g., the addition or deletion of statements.

Bytecode Clone Detection Baker and Manber [BM98] present a clone detection approach based on disassembled bytecode using the tools Siff, Dup, and Diff to detect type 2–3 clones. The approach reports the clone detection results based on untyped stack-based bytecode. In contrast, SootDiff uses Jimple to detect equal classes and methods, which is a typed three-address code, and thus closer to source code than bytecode. Consequently, SootDiffs results are easier to understand and interpret from a developer's perspective. Moreover, the comparison based on Jimple and the application of normalization steps allows SootDiff to reduce dissimilarities introduced by different Java compilers and Java versions.

Keivanloo et al. [KRR14] introduce the code similarity and clone detector SeByte, which operates on Java bytecode, to detect type 1–3 code clones and even, in some cases, semantic similarities. SeByte splits the bytecode into tokens into three dimensions: instructions, method calls, and types and computes for each dimension similarity measures, such as the Jaccard similarity. SeByte combines the different similarity measurements for each dimension to detect code clones and to enable code clone search.

Yu et al. [Yu+19] present an approach for detecting type 1–3 clones and a subset of semantic code clones. The proposed approach extracts two features from a given bytecode body of a method: the instruction sequence and the method call sequence. The approach applies the Smith-Waterman algorithm, an algorithm for detecting similar regions of genes, on both sequences to align and match the extracted sequences. If the similarity of both sequences corresponds to a predefined threshold, the approach reports a code clone.

Source-Code Clone Detection Several approaches for detecting clones in different programming languages in source code exist [J H94; MLM96; Bax+98; KKI02]. The approaches apply different clone detection techniques directly on source code to discover type 1–3 clones, e.g., string-based, token-based, abstract syntax tree (AST-)based, metric-based, etc. [Kos07; RKC18].

While the source code of classes can be recovered successfully using decompilers, which makes existing source-code clone detection techniques applicable to discover modified open-source artifacts, using decompiled source code has the disadvantage that it does not allow further normalization that the intermediate representations Jimple offers, e.g., reduced instruction set, and thus we would generate more coarse results.

Intermediate Representation Code Clone Detection Most related to SootDiff is the approach jNorm created by Schott et al. [Sch+23]. JNorm builds on top of SootDiff's approach. JNorm parses classes' bytecode to Jimple and applies further normalization steps to reduce dissimilarities, as we did with our String optimizer.

Selim et al. [SFZ10] present an approach similar to SootDiff from a technical point of view. Like SootDiff, they use Soot's intermediate representation Jimple to detect type 3 source-code clones. As an input, the approach solely uses Java source-code files to produce Jimple code. They apply Java source clone detection tools on the Jimple representation. To report detected clones to users, they map the results from Jimple back to Java.

While the presented approach is similar to SootDiff, significant differences exist. First, we assume that no Java sources are available, and thus we solely work on bytecode. Second, we do not run existing code clone detection tools for source code but tailor the comparison to Jimple. Third, SootDiff does not aim to detect type 3 clones but to detect if two bytecode classes, which may have been generated by different Java compilers—and are only slightly modified—originate from the same source code. Consequently, we cannot trust existing source-code clone techniques to detect modified open-source artifacts reliably.

3.10 Conclusion

In this chapter, we presented a case study on 7,024 Java projects developed at SAP investigating the use of (modified) OSS and their impact on the precision and recall of vulnerability scanners. The results of our case study show that, even in commercial applications, the majority (86%) of the dependencies are OSS and that most dependencies are included transitively (79%). Thus, known-vulnerable OSS are an important issue for Java applications.

Remarkably, our case study shows that developers also include modified OSS, compromising re-compiled, re-packaged, or re-bundled classes from other open-source projects, and that such modifications commonly occur on Maven Central.

Crucially, our case study on the open-source scanners: Dependency-Check, Eclipse Steady, GitHub Security Alerts, and three commercial vulnerability scanners shows that the observed modifications heavily decrease the scanners' precision and recall. In fact, our results show that all vulnerability scanners struggle to cope with modified OSS. Consequently, known vulnerabilities in modified OSS may remain undetected, posing a threat to the application. To replicate our results and to foster research and development by enabling evaluation and comparison of vulnerability scanners, we present the test suite Achilles—including 2,558 test cases and ground truth. We conclude that matching techniques that are resilient to the loss of metadata and bytecode modifications are needed.

As a first approach for the design of matching techniques that are resilient to bytecode modifications, we present SootDiff. SootDiff enables the comparison of (re-compiled) bytecode based on Soot's intermediate representation Jimple. To compare the generated Jimple representation, SootDiff leverages the established Myers' diff algorithm. Although SootDiff currently uses only a String optimization step in combination with Soot's default optimizers, e.g., dead code eliminators, the evaluation shows that SootDiff produces promising results. SootDiff's evaluation also shows that additional optimization steps to reduce dissimilarities, e.g., the organization of basic blocks and control structures, will further improve the detection of code clones. Promising results in this direction have been generated by Schott et al. [Sch+23].

4 An Automated Approach for Safely Updating Included Open-Source Software

In the previous chapter, we showed that known-vulnerable Open-Source Software (OSS) is a problem in both commercial and open-source software projects. Although vulnerabilities in OSS are a severe threat to an application and must be closed immediately (cf. Log4Shell [NVD21]), recent research [Kul+18; Bav+15; MP17] shows that developers hesitate to update included OSS and mistrust automated update tools such as Dependabot, since they are afraid of introducing incompatibilities that break their project. This mistrust is not unjustified as the tools do not ensure compatibility with other included OSS. In result, applications remain vulnerable, despite a fixed version of the vulnerable OSS being available.

To alleviate this situation, we present UpCy, an automated approach that supports developers in updating known-vulnerable or outdated OSS. We start by comprehensively defining the conditions a safe backward compatible update has to satisfy in Section 4.1. We separate the conditions into programming language constraints (source & binary compatibility), which are required to compile and execute the program successfully, and constraints to other OSS in the project (dependency graph compatibility).

To support developers in finding an update for vulnerable or outdated OSS with a minimal number of incompatibilities automatically, we introduce UpCy in Section 4.2. UpCy encodes the constraints a backward-compatible update has to fulfill in the form of a unified dependency graph. On this unified dependency graph, UpCy applies the min-(s,t)-cut algorithm and leverages a self-created dependency graph of the public open-source repository Maven Central to find compatible artifacts.

We evaluate UpCy on 1,823 open-source Java projects sampled from GitHub in Section 4.3. In our evaluation, we compare the effectiveness of UpCy with the naïve updates that state-of-the-art tools, like Dependabot, produce on 29,698 updates. Our results show that UpCy successfully finds updates with fewer incompatibilities in 35.1% of the cases where state-of-the-art approaches fail; importantly, 99% of these

updates have zero incompatibilities. Further, our results show that UpCy produces update steps comprising the update of two dependencies on average, indicating that compatibility can be maintained with reasonable effort.

We give an overview of related work in Section 4.5, and we complete the chapter by discussing open challenges and required adaption to other dependency-management tools in Section 4.6.

4.1 Safe Backward Compatible Updates

When developers aim to update a dependency, they must consider any update carefully since their project may unexpectedly suffer from regression-inducing changes, such as bugs or semantic changes, that break the Application Programming Interface (API) contract between the application and the (updated) open-source artifact [HG22].

To achieve a safe backward compatible update or at least minimize the effort for adapting the application's code, developers need to consider several types of incompatibilities between the different versions of the artifact, and how the project uses the artifact's API. In the following, we discuss the considerations developers must make when updating.

4.1.1 Dependency Graph Updates

For a global dependency graph, like the one used by the build-automation tools Maven and pip (cf. Chapter 2), developers have the following options for updating a specific artifact [Hua+20], shown as t in Figure 4.1:

i If the artifact is a direct dependency of the project, developers can directly add the new version of the artifact to the build-automation tool's configuration file, e.g., Maven's `pom.xml`.
In this case, the dependency is replaced directly with the new version.

ii If the artifact is a transitive dependency, developers have two options for updating:

ii.i They can specify the updated version t' of the artifact in the build-automation tool's configuration file directly, transforming the new version to a direct dependency. Due to Maven's shortest-path resolution mechanisms, the direct dependency then shadows all other (transitive) instances of this dependency, shown in Figure 4.1b.

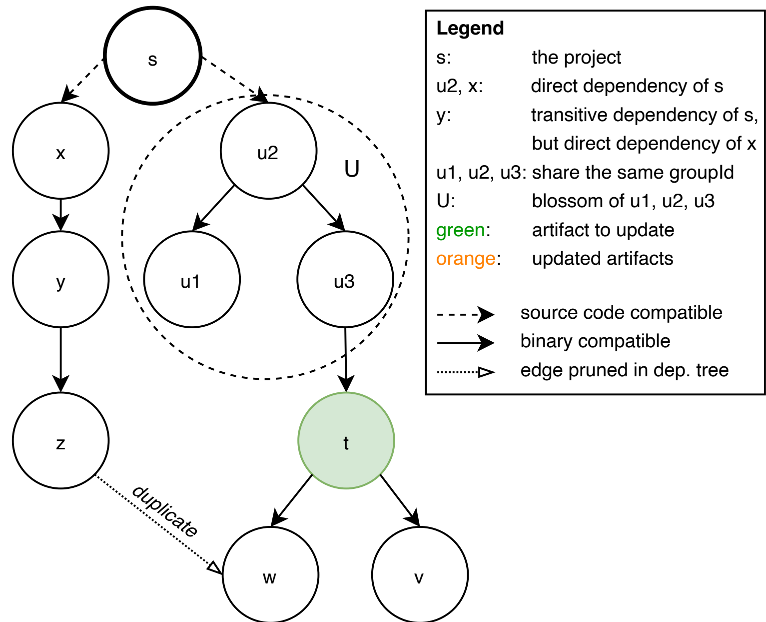
ii.ii They can check if any predecessor u in the dependency graph, lying on the path from the project node to the artifact, can be updated to a newer version that itself depends on a newer version of the artifact, shown in Figure 4.1c. If such a predecessor u exists, developers can transform the update u' to a direct dependency.

In all cases, developers must assess the update's effect on the dependency graph since every update can introduce new (direct and transitive) dependencies that may lead to conflicts or shadowing of existing dependencies. For instance, consider the duplicate dependency w in Figure 4.1. Assume a developer updates the artifact t to a new version t' by declaring t' directly in the build-automation configuration file, making it a direct dependency, shown in Figure 4.1b. If t' itself depends on a new version w', then w' will shadow the dependency w of z—whether desired or not. As a result, such an update may introduce unintended regressions. For instance, artifact z may fail to run successfully with the new artifact w'.

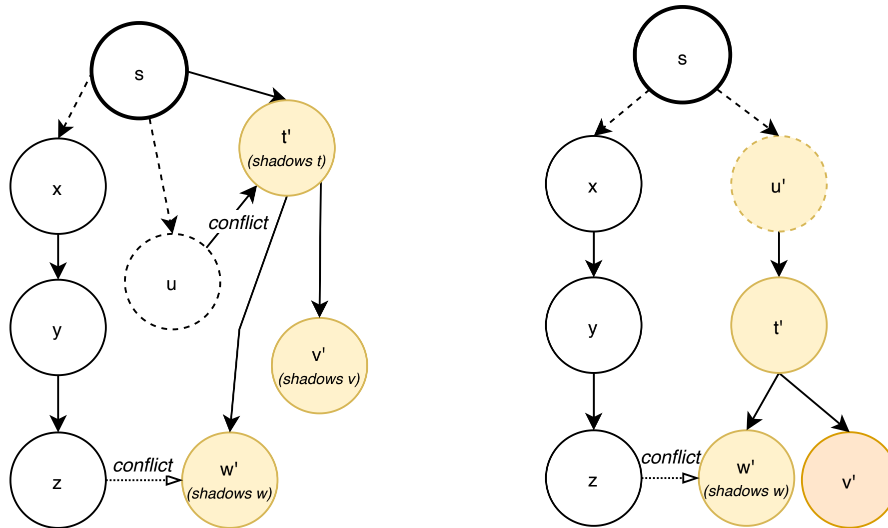
Since shadowing and the inclusion of transitive dependencies are automatically managed by the dependency-management tool Maven without notifying developers, introduced regressions caused by incompatible dependencies are tedious to detect and resolve manually. With an increasing number of dependencies, duplicates, and conflicts, the complexity of maintaining and reasoning about the compatibility between artifacts increases. However, to avoid introducing unexpected regressions, developers must ensure that all nodes in the dependency graph that use the updated artifact continue to compile, link, and execute successfully [DJB14]. A study by Wang et al. [Wan+22] showed that this is difficult to diagnose in practice, leading to runtime exceptions and unexpected program behavior.

4.1.2 Source and Binary Compatibility

To ensure an update is *safe*, the updated artifact's API must be *backward compatible* with the currently used version of the artifact [DJB14]. In particular, the artifact's API types, methods, and fields must not be subject to changes that prevent compilation, linking, or execution of formerly valid client code.



(a) Example of a Maven dependency graph. Dependencies whose API is directly invoked by the project's code must be source compatible (dashed lines), whereas dependencies whose API is invoked by another dependency must be binary compatible.



(b) Updating the transitive dependency t by transforming t' to a direct dependency.

(c) Updating the predecessor u of the transitive dependency t.

Figure 4.1: The options for updating the transitive dependency t and resulting changes in the dependency graph.

In Java, but also other compiled languages, it makes sense to distinguish between *source* and *binary* compatibility issues [Ecl07; DJB14]. Source compatibility requires that the source code of existing clients continue to compile without errors against the revised API. Source-code incompatibilities result in compilation errors when the project is re-compiled against the updated artifact [Ecl07]. Source-code *in*compatible changes are, for instance, the removal of public methods, types, or the addition of checked exceptions to an API method, as these exceptions must be handled by client code. To re-compile the project successfully, source-code compatibility must be fulfilled for all dependencies (transitive and direct) whose APIs are *directly* invoked in the project's source code. Typically, source-code incompatibilities in the project code can be easily corrected as the source code is directly available and changeable by developers.

Application Binary Interface (ABI) compatibility requires that preexisting binaries continue to link without errors against the revised API. In Java, binary incompatibilities lead to failures during linking or invocation [Ora15a]. Examples of ABI *in*compatible changes are the removal of API types, methods, fields, or changes in a method's signature. Binary compatibility must be fulfilled if an artifact is used by other dependencies that cannot be re-compiled, which is typically the case for transitive dependencies. Thus, all dependencies in the dependency tree must continue to link successfully to the bytecode classes in the updated artifact. In contrast to source-code incompatibilities, binary incompatibilities in dependencies cannot be easily fixed as the application developers usually do not have control over the artifact's source code—except if they create their own fork, as we have seen in Chapter 3.
Note that binary compatibility does not imply source compatibility nor vice versa, although some compatibility issues affect both, e.g., the removal of API types [DJB14].

Within a dependency tree, an artifact must fulfill exactly one of the two compatibility types for each incoming edge, depending on the nature of the dependency that the edge represents, shown in Figure 4.1. A study from Dietrich et al. [DJB14] shows that source and binary compatibility are regularly violated between different versions of an artifact. Although tools like SigTest [Tul22] check if the API of an artifact's new version is source-code and binary backward compatible, existing tools do not support reasoning about the compatibility between several connected artifacts in the dependency tree. Even more, approaches to indicate compatibility between different versions of an open-source artifact, like *Semantic Versioning* [Pre21], fail, as maintainers release new changes based on their self-interpretation of backward compatibility, which often leads to source or binary incompatibility, shown in a study by Bogart et al. [Bog+16].

4.1.3 Semantic Compatibility

Checking artifacts for source and binary incompatibilities is a necessary precondition but is insufficient; *semantic* compatibility is also required.

Semantic compatibility requires that the semantics of the API do not change. Artifacts with semantic compatibility issues are source and binary compatible, yet introduce *in*compatible runtime behavior that invalidates formerly valid assumptions made in client code. Examples of semantic incompatibilities are changes in a method's pre- and postcondition [Ecl07]: "Method preconditions are things that a caller must arrange to be true before calling the method, and that an implementor of the method [implementing an abstract method or overriding it], may presume to be true" [Ecl07]. As an example of an incompatible precondition change, assume that a method formerly accepted null values as arguments, but the updated version throws a `NullpointerException`. Consequently, client code that passes null values to the method becomes invalid.

"Method postconditions are things that an implementor must arrange to be true before returning, and that a caller presumes to be true after the method returns" [Ecl07]. As an example of an incompatible postcondition change, assume that a method returns an empty list if it does not find a specific element in a given list, but the updated method returns null to indicate that no element could be found. In effect, formerly valid client code checking if the list is empty will fail with a `NullpointerException`.

A sound and precise detection of semantic incompatibility is undecidable, however, several approaches exist that use or synthesize test cases to detect semantic compatibility issues w.r.t. the way the project uses an artifact [HG22; Wan+22].

4.1.4 Blossom Compatibility

Our study in Chapter 3 and the study by Pashchenko et al. [Pas+18] show that developers regularly include frameworks, and those frameworks typically consist of a large number of (transitive) dependencies that share the same vendor (groupId in Maven). Since these dependencies are highly interconnected and usually require the same version to run correctly, all framework artifacts must be updated consistently to a new version if a single artifact of the framework is updated. We refer to artifacts with the same groupId in the dependency tree as *dependency blossoms*: they can be merged into a single node like regular blossoms in graphs, shown in Figure 4.1.

To achieve safe backward compatible updates, developers must consider that all dependencies in a blossom node have the same (or a compatible) version. Typically, only those are guaranteed to work together by the framework authors [Pas+18].

API Compatibility Restriction Note that a safe backward compatible update does not require source-code, binary, and semantic compatibility for the complete API. Instead, only those API types and methods that are actually used during compile- and runtime must be compatible.

4.2 UpCy: Identify Safe Backward Compatible Updates

As described in Section 4.1, the options for updating an artifact depend on its position in the dependency tree: (i) a *direct dependency* can only be explicitly updated, (ii) a *transitive dependency* can either be transformed into a direct dependency, or any of its predecessors can be transformed into a direct dependency and updated. To reduce the risks of introducing regressions and to keep the effort for adapting the application's code low, one has to choose an update option that leads to the least amount of source, binary, semantic, dependency tree, and blossom incompatibilities, which is manually almost impossible given the size of a typical dependency tree with more than 94 artifacts (cf. Section 3.3).

To find compatible updates automatically, we created UpCy. Given a Maven project, an artifact, and its target version, UpCy determines the necessary steps to update the artifact with minimum source, binary, and blossom incompatibilities. To do so, UpCy analyzes how the project uses the artifact by merging the project's dependency graph and call graph into a unified dependency graph. On this graph, UpCy leverages the min-(s,t)-cut algorithm to identify which update option leads to the least amount, or even no, incompatibilities. As an output, UpCy computes a list of artifacts that developers should add as direct dependencies to their application to update the given artifact to the target version, and the number of incompatibilities the update introduces.

4.2.1 Algorithm

Figure 4.2 gives a high-level overview of UpCy, and Algorithm 1 depicts the algorithm in pseudocode.

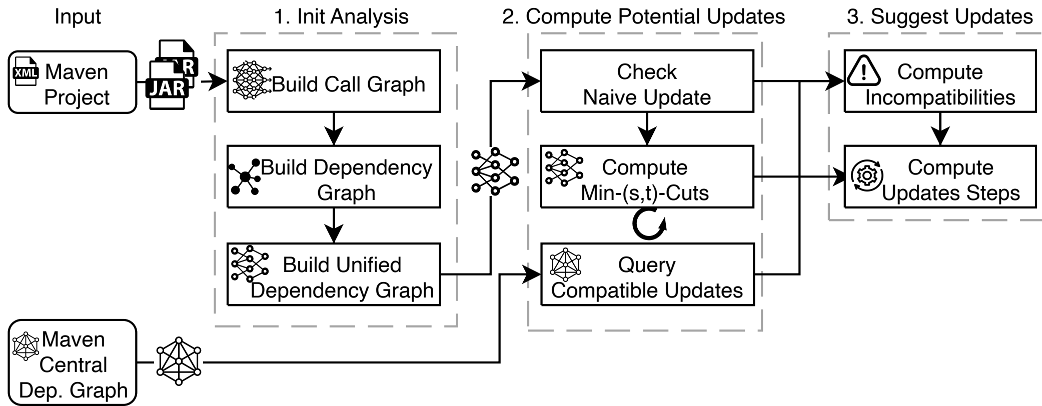


Figure 4.2: UpCy's process steps for computing compatible updates using min-(s,t)-cut algorithm and a dependency graph of the Maven Central repository.

Algorithm 1 UpCy's algorithm for identifying safe backward compatible updates.

Input: mavenCentralGraph, project, libToUpdate, targetVersion
Output: listUpdateSteps

/* *build unified dep. graph* */
1: depGraph ← buildDependencyGraph(project)
2: callGraph ← buildCallGraphSoot(project)
3: unifiedDepGraph ← unify(callGraph, depGraph)
/* *naive Update* */
4: updateGraph ← queryRepoGraphForUpdates(mavenCentralGraph, libToUpdate, targetVersion)
5: incmpts ← computeIncompatibilities(unifiedDepGraph, updateGraph)
6: **if** incmpts = ∅ **then**
/* *the root node of the updateGraph is the updated library* */
7: **return** (getRootNodes(updateGraph), incmpts)
8: **end if**
/* *update based on min-(s,t)-cut* */
9: minCuts ← computeMinCuts(unifiedDepGraph, libToUpdate)
10: **for** minCut ∈ minCuts **do**
11: query ← mapToCypher(minCut, unifiedDepGraph)
12: updateGraph ← queryRepoGraphForUpdates(mavenCentralGraph, query)
13: incmpts ← computeIncompatibilities(unifiedDepGraph, updateGraph)
/* *root nodes of the updateGraph are the nodes to add as direct deps* */
14: **if** incmpts = ∅ **then**
15: **return** (getRootNodes(updateGraph), incmpts)
16: **else**
17: listUpdateSteps ∪ (getRootNodes(updateGraph), incmpts)
18: **end if**
19: **end for**
20: **return** listUpdateSteps

Input To find compatible updates and explore the options for updating the given artifact to a specific version, all versions of that artifact and all versions of its predecessors in the dependency graph must be known to UpCy. Furthermore, to check which existing artifacts the update will shadow, the complete list of direct and transitive dependencies the update introduces (*updateGraph*) must be known. Since Maven Central does not provide information about an artifact's dependencies, we created a complete dependency graph of Maven Central ourselves, using the graph database Neo4j. We describe our dependency graph of Maven Central in Section 4.2.2. Further, UpCy requires the Maven project, the artifact *libToUpdate*, and the target version *targetVersion* the artifact should be updated to as an input.

1. Building the Unified Dependency Graph In the first step, UpCy checks which API calls the project and the other dependencies invoke. To do so, UpCy constructs the *unified dependency graph* (*unifiedDepGraph*). The unified dependency graph maps the sources and targets of method calls to the dependencies in the dependency graph. To construct the unified dependency graph, UpCy first builds the project's dependency graph using the Maven dependency graph plugin [Fer22] (line 1 in Algorithm 1). The plugin allows the construction of the complete dependency graph, including duplicate, conflicting, and transitive dependencies, using the Maven dependency resolution mechanisms. In contrast to the "flat" dependency tree, the computed dependency graph contains all information about shadowed, conflicting, and duplicate dependencies, which Maven prunes in the tree. Figure 4.3 shows an example project's dependency graph with the project node s, blossom U, and the artifact to update t.

Next, UpCy constructs a graph containing the API usage between the artifacts and the project. We refer to the usage of an artifact as using an artifact's API types, methods, and fields. Thereby, we do not distinguish between the use of direct or transitive dependencies but consider both equally. In particular, UpCy constructs a call graph specifying the use of API methods as they are among the most common forms of artifact usage. UpCy statically constructs the call graph representing the call paths between the project and its dependencies, and the call paths between dependencies, excluding calls to the project or the Java Class Library (JCL). To construct the call graph, UpCy uses the class hierarchy analysis of the static analysis framework Soot [Val+10] with all project classes as entry points, similar to the approach presented by Pashchenko et al. [Pas+18] (line 2).

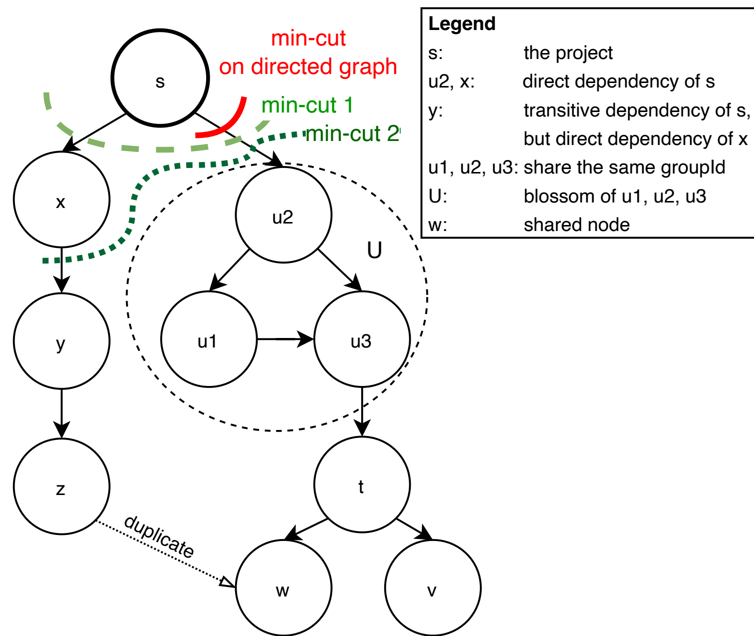


Figure 4.3: Example min-(s,t)-cuts computed by UpCy on the unified dependency graph for updating t with minimal incompatibilities. Each edge represents a source or a binary compatibility relation between the connected artifacts.

Finally, UpCy merges the constructed dependency and the constructed call graph to create the unified dependency graph. For each found call edge in the call graph, UpCy maps its source and target to the artifacts in the dependency graph. Edges within an artifact (same source and target) are pruned as they do not represent any API invocation between artifacts. Further, all call edges between two nodes are merged into a single edge containing all API calls in the form of a set. Thus, each edge in the unified dependency graph represents a set of API calls that must be *binary* or *source*, and *semantically* compatible.

2. Naïve Update Before computing updates that involve the update of multiple artifacts, UpCy evaluates if a naïve update—simply declaring the updated artifact t' as a direct dependency, which is the approach state-of-the-art tools like Dependabot [Git22] apply, yields incompatibilities (lines 4–8) [HG22]. In contrast to existing approaches, UpCy automatically computes changes in the dependency tree and resulting incompatibilities, and thus helps developers to spot regression by showing which API calls suffer from source, binary, or potentially semantic incompatibility. To compute which artifacts the update will shadow, UpCy queries our graph database of Maven Central to get all direct and transitive dependencies (*updateGraph*) of the update t', and computes incompatibilities as described in detail in Section 4.2.1.

3. Minimizing Incompatibilities using Min-(s,t)-Cuts If the naïve update suffers from incompatibilities, UpCy tries to identify alternative updates with fewer incompatibilities (lines 9–19). If applied to the project, an incompatible update requires high maintenance by developers. Developers are forced to identify which (transitive) dependencies are affected by the incompatible update and must adapt their project's code accordingly. If the incompatibility cascades to an API call between dependencies, they also have to replace those since they usually do not have control over the code and `pom.xml` of these transitive, third-party dependencies; except if they fork and adapt the OSS dependency themselves, as we studied in Chapter 3. If they do not have the resources to modify the third-party dependency's code—which is typically the case—those artifacts must be updated as well, e.g., in Figure 4.3, if the API calls that U invokes on t are binary incompatible with the updated version t', U also needs to be updated.

To identify compatible updates, UpCy computes a min-(s,t)-cut on the project's unified dependency graph. Each edge in the unified dependency graph represents a dependency relation, originating from the dependency graph. Further, each edge defines a *use relation*, originating from the call graph, the edge represents the API calls between the connected artifacts. If we could not find API calls between the artifacts, we still assume that a dependency relation also declares a use relation since it is recommended practice to declare every used dependency as a direct dependency [Apa23a]. Thus, the unified dependency graph is an over-approximation and may contain edges to superfluous, unused dependencies—that are dependencies that are included but whose API is not used. Consequently, each edge encodes a compatibility requirement; the API used in the target artifact must be binary, source, or blossom, and semantically compatible depending on the position in the dependency graph. For the example graph in Figure 4.3, the edge from node $u3$ to t specifies that $u3$ depends on t, and $u3$ invokes t's API. Thus, the update t' must be binary compatible w.r.t. the API that $u3$ invokes.

Based on the unified dependency graph, we can derive the following conditions a compatible update of an artifact t has to fulfill:

i The artifact t or a predecessor u of t must be updated by adding it as a direct dependency to the project.

ii All artifacts that are used by another artifact n and that are shadowed when updating t may also need to be updated to avoid incompatibilities—in other words, all artifacts that use a (conflicting or duplicate) dependency that will

change in the update, but are not (transitive) dependencies of t may also need to be updated, e.g., the artifact z also uses the artifact w, which will be shadowed in the update.

iii The dependency graph can be separated into two disjoint sets of nodes: artifacts that are updated, and artifacts that do not need to be updated.

iv The edges connecting artifacts that are in disjoint sets represent compatibilities that are potentially violated when updating t.

To identify update steps with minimal incompatibilities between the artifacts in the dependency tree, UpCy separates the unified dependency graph into two partitions: one containing the project s, and the other containing the artifact t, with a minimal number of edges crossing the two partitions. Finding these partitions can be optimally solved using the min-(s,t)-cut algorithm [FF56]. The min-(s,t)-cut algorithm computes for a graph and a pair of nodes (s, t), a cut of the graph into two separate partitions S and T with given nodes $s \in S$ and $t \in T$ that is minimal w.r.t. the weight of the edges crossing the partitions. In the resulting cut, only nodes that are connected by the edges that lie on the cut must be ensured to be compatible with each other for an update of the artifacts in the sink partition T. All artifacts in the partition T are compatible with each other if the root nodes are updated: an updated artifact in T is either a root node, and then updated directly; or the artifact has been automatically updated as a declared transitive dependency of a root node by the tool Maven. Since UpCy aims to minimize the incompatibilities between nodes and each edge in the dependency graph represents one compatibility requirement (binary or source), we assign all edges equal weight. The root nodes of the sink partition, which are the target of the edges on the cut, are the artifacts that developers have to update by adding them as direct dependencies to the build-automation tool's configuration file.

However, simply computing the min-(s,t)-cut on the *directed* unified dependency graph does not minimize incompatibilities as the cut ignores blossom, duplicate, and conflicting dependencies. For a directed graph, like the unified dependency graph, a min-(s,t)-cut is equivalent to the maximum flow according to the max-flow min-cut theorem [FF56]. Thus, edges to duplicate or conflicting dependencies located behind the updated artifact t, e.g., the duplicate w, are ignored as the directed edge from z to w is not part of a flow from s to t, and thus not part of the min-cut.
For instance, computing the min-(s,t)-cut on the directed unified dependency graph produces min-cuts of weight 1, as shown in Figure 4.3. The min-cut cuts the edge between s and $u2$. Crucially, the min-cut computed on the directed unified depen-

dency graph ignores compatibilities: the min-cut ignores the blossom compatibility for $u1$, $u2$, $u3$, and more importantly, the cut ignores the compatibility edge between z and w.

To overcome this limitation, UpCy computes the min-(s,t)-cut on an adapted representation of the unified dependency graph: nodes that belong to the same framework are merged into blossom nodes, e.g., $u1, u2, u3$ merged to blossom U, and the edges are undirected. Using this representation of the unified dependency graph, UpCy computes min-cuts of weight 2 for the graph in Figure 4.3, e.g., *min-cut 1* and *min-cut 2*.

Given the computed min-cuts, UpCy checks if new versions of the nodes in the sink partition T exist on Maven Central to update the artifact t to the requested target version by querying our dependency graph of Maven Central for an *updateGraph*, as explained in detail in the next section. If our graph database returns a solution (a non-empty *updateGraph*), UpCy computes all incompatibilities w.r.t. the (updated) artifacts in the *updateGraph* and API calls on the edges of the unified dependency graph. For instance, for *min-cut 1*, UpCy checks if the API calls between $s \rightarrow u2$ and $s \rightarrow x$ suffer from incompatibilities if $u2$ and x are updated, respectively, for *min-cut 2*, the calls $s \rightarrow u2$ and $x \rightarrow y$ if $u2$ and y are updated. UpCy stops when it finds a min-(s,t)-cut with zero incompatibilities, or when no further min-cuts can be found.

Finally, UpCy outputs the list of the found update steps (*listUpdateSteps*) and incompatibilities (line 20 in Algorithm 1). Developers can then add the dependencies in the computed update steps as direct dependencies.

4. Compute Incompatibilities When querying the Maven Central dependency graph with Neo4's query language Cypher, UpCy receives an *updateGraph*. The *updateGraph* contains the new versions of the artifacts (the new versions for the root nodes in T) and all their (transitive) dependencies. For instance, for the min-cut 1 in Figure 4.3, the update graph contains the dependency tree for the artifacts $u2$, $u1$, $u3$, and x, merged into a single graph. Given the *updateGraph* and *unifiedDepGraph*, UpCy computes which API incompatibilities occur and which existing dependencies will be shadowed if the new artifacts are added as direct dependencies to the project (lines 5, 13).

First, UpCy computes which artifacts the final, updated dependency tree will contain and which will be shadowed. To do so, UpCy checks for each dependency in the *updateGraph* if the dependency will have the shortest path in the final dependency

tree, shadowing all other instances of that artifact. Second, if the dependency will shadow an existing dependency in the final, updated dependency tree, UpCy checks all edges in the unified dependency graph that target the former instance of that dependency in the current existing dependency tree for incompatible API calls. To do so, UpCy queries SigTest [Tul22] and SootDiff (cf. Section 3.8), for a list of API types and methods that are source or binary incompatible and whose method bodies changed—indicating potential semantic incompatibilities, and intersects them with the API calls in the unified dependency graph.

For example, assume the naïve update t' of t will introduce a new version w', as shown in Figure 4.1b. UpCy checks if the API calls that u invokes on t are binary compatible with t', and if the API calls from z to w' are compatible. To that end, UpCy creates the list of incompatible API's for t, t' and w, w' using SigTest, and checks if the reported incompatible API types also occur as API calls in the unified dependency graph. If the dependency will not become part of the project's dependency tree because an older instance shadows it, UpCy reports a forward compatibility issue.

4.2.2 Graph Database of the Maven Central Repository

Creation and Structure UpCy aims to identify updates without regressions by finding newer versions of the open-source artifacts that are compatible with each other. To do so, UpCy searches for artifacts with certain properties, e.g., with a certain version, or an artifact that (transitively) depends on another artifact with a certain version. Since the Maven Central repository's API does not provide information about open-source artifacts and their (transitive) dependencies, we created a graph database of Maven Central with Neo4j as our storage backend, allowing us to query Maven Central using the query language Cypher [Neo23]. Our graph database is essentially a large dependency graph containing the open-source artifacts as nodes, with file name, timestamp, groupId:artifactId:version (GAV), and the dependency relations as edges, with scope and classifier. In our database, all dependencies of an artifact, including parent and import dependencies, are transitively resolved, specifying the dependency's declaration order and scope.
The created database contains information and dependencies for 8.5 million artifacts hosted on Maven Central. Note that we failed for some artifacts to create nodes or resolve all dependencies if the `pom.xml` file hosted on Maven Central contained errors, circular dependencies, or invalid syntax.

Querying the Graph Database using Cypher For each found min-(s,t)-cut, UpCy queries our graph database if new versions of the root nodes of the sink partition T exist for updating the artifact t. In particular, UpCy asks Neo4j

i to find new versions of the root nodes of the sink partition T that already depend on the target version t' of the artifact to update.

ii to find versions of the returned nodes that do not introduce new incompatibilities—the (transitive) dependencies of the new versions of the root nodes should be compatible with each other.

iii to return the (transitive) dependencies of those updated root nodes (*update-Graph*).

To avoid the introduction of new incompatibilities, UpCy queries in (ii) for versions of the root nodes that share a duplicate or conflicting artifact with the same version, e.g., the node w is a duplicate since the artifact is included twice: once by the transitive dependencies starting from $u2$, and again by x in the sink partition of *min-cut 1*. Consider the min-cuts in Figure 4.3: For *min-cut 1*: (i) UpCy tries to find an update of the nodes $u2$ and x, where $u2$ depends on an updated version of t, (ii) where $u2$ and x depend on the same version of the (transitive) dependency w, and (iii) UpCy asks for all (transitive) dependencies of the updates of $u2$ and x. For *min-cut 2*, (i) UpCy tries to find an update of $u2$ and y where $u2$ depends on an updated version of t, (ii) $u2$ and y depend on the same version of the (transitive) dependency w, and (iii) UpCy asks for the (transitive) dependencies of the updates $u2$ and y.

To find update steps for a computed min-(s,t)-cut, UpCy maps the queries (i),(ii), and (iii) to Cypher, and then queries our graph database of Maven Central for artifacts that fulfill the generated queries. In the following, we present the Neo4j queries that UpCy generates.

(i) Find versions of the sink's root nodes that already depend on the updated artifact First, UpCy tries to find the target version of the artifact t in the Maven Central graph. To do so, UpCy generates for the artifact t in Figure 4.3 the Neo4j query shown in Listing 4.1.

```
1    MATCH (t:MvnArtifact)
2    WHERE t.group = "groupIdOfT" AND t.artifact = "artifactIdOfT" AND
3          t.version >= "targetVersionOfT"
```

Listing 4.1: Cypher Query for checking if a version greater than or equal to $targetVersion$ of the artifact to update exists.

The query finds (MATCH) all artifacts t with the groupId, artifactId, and version (GAV) *groupIdOfT:artifactIdOfT:targetVersionOfT*. To not limit the update options to the target version only and to ease matching with the following queries (ii) and (iii), we relax the version to be greater than or equal to the target version.

Second, UpCy tries to find new versions of the *root nodes* of the sink partition T that depend on the updated artifact t'. The root nodes are the artifacts that are the targets of the edges cut by the computed min-(s,t)-cut. UpCy generates for every root node r a Neo4j query as shown in Listing 4.2.

```
1  MATCH p = ((r:MvnArtifact) -[:DEPENDS_ON*1..]-> (t))
2  WHERE r.group = "groupIdOfR" AND r.artifact = "artifactIdOfR" AND
3        r.version >= "versionOfR"
4  RETURN p, r LIMIT 1
```

Listing 4.2: Cypher Query for checking if more recent versions of the artifacts that are potential update candidates as computed by the min-(s,t)-cut (the root nodes of sink partition T) exist.

The query finds (MATCH) all artifacts r, with the groupId and artifactId *groupIdOfR:artifactIdOfR* and a version greater than or equal to the already used version *versionOfR* that have a (transitive) dependency to the updated artifact t, matched in the first query. The `*1..` property in the relation instructs Neo4j to search for paths of unlimited length (direct and transitive dependencies of r). If the artifact t itself is a root node r, UpCy does not generate a query for that particular root node. Finally, the query returns the path p from the root node r to the dependency to update t, as well as the found nodes. Only one solution is selected if multiple exist (LIMIT 1).

(ii) Find compatible root nodes for duplicate or conflicting dependencies If a sink partition T has more than one root node, UpCy checks if duplicate or conflicting dependencies exist that are (transitively) included by two or more different root nodes. For nodes included by more than one root node, UpCy tries to find compatible root nodes that all depend on the same version. For example, in Figure 4.3, the dependency w is transitively included by the root nodes u and x of *min-cut 1*. Since an update of u and x should not introduce new incompatibilities, UpCy queries the graph database for versions of u and x that depend on the same version of w.

To identify nodes that are shared by multiple root nodes, UpCy traverses all nodes in the sink partition of the unified dependency graph backward to the *root nodes*. If the traversal ends in two or more root nodes, we refer to these nodes as *shared dependencies*—they are (transitively) included by multiple root nodes.

For instance, consider the nodes of *min-cut 1*. The backward traversal for the nodes in the sink partition yields the following paths: $U \leftarrow U$, $U \leftarrow t$, $U \leftarrow t \leftarrow v$, $\{u, x\} \leftarrow \{U, y\} \leftarrow \{t, z\} \leftarrow w$, and so on. Analogously, for *min-cut 2*. Since the traversal for node w ends in the root nodes $\{U, x\}$, w is a shared dependency: shared by the root nodes U and x.

```
1 MATCH p1 = ((u:MvnArtifact) -[:DEPENDS_ON*0..]-> (w:MvnArtifact)),
2       p2 = ((x:MvnArtifact) -[:DEPENDS_ON*0..]-> (w))
3 WHERE u.group = "groupIdOfU" AND u.artifact = "artifactIdOfU" AND
4       u.version >= "versionOfU" AND
5       x.group = "groupIdOfX" AND x.artifact = "artifactIdOfU" AND
6       x.version >= "versionOfX" AND
7       w.group = "groupIdOfW" AND w.artifact = "artifactIdOfW" AND
8       w.version >= "versionOfW"
9 RETURN p1, p2, u, x, w LIMIT 1
```

Listing 4.3: Cypher Query for finding compatible versions of the artifacts u and x. Compatible artifacts depend on the same version of the shared dependency w, avoiding the introduction of conflicts and binary incompatibilities.

For each shared dependency, UpCy creates additional queries for the root nodes that share a dependency, shown in Listing 4.3 for *min-cut 1*. The query finds (`MATCH`) all artifacts u and x, with their existing groupId and artifactId and a version greater than or equal to the already used versions (*versionOfU*, *versionOfX*), that have a (transitive) dependency to the same version of the shared artifact w, with a version greater than or equal to *versionOfW*.
As newer versions of the artifacts u or x may no longer depend on an instance of w, the matching is optional, defined by the relationship property `0..`. If the query finds a match, it returns the paths $p1$ and $p2$ from the artifacts (u and x) to the shared dependency w. If multiple solutions exist in the Maven Central dependency graph, UpCy only selects one (`LIMIT 1`).

(iii) Query the update graph If the queries (i) and (ii) yield results, UpCy queries the dependency trees for the found, new root nodes (*updateGraph*). The *updateGraph* contains the updated versions of the root nodes, as well as their (transitive) dependencies. For querying the *updateGraph*, UpCy generates for each root node r the query shown in Listing 4.4, and merges the results into a single *updateGraph*. Since the root nodes r are bound in the previous queries, no *WHERE* clauses are generated for the query. The merged update graph is returned to UpCy.

```
1 MATCH p = ((r) -[:DEPENDS_ON*0..]-> (:MvnArtifact))
2 RETURN p
```

Listing 4.4: Cypher Query for getting an artifact r and all of its (transitive) dependencies (the update-graph).

4.3 Evaluation

To evaluate in how many cases UpCy can effectively support developers, we compare it to the naïve update approach, state-of-the-art tools, such as Dependabot, apply.

4.3.1 Research Questions

As UpCy is especially beneficial in cases where naïve updates fail, we aim to study how often naïve updates fail, and complex updates are required. Thus, we investigate **RQ1: How often do naïve updates fail due to source-code, binary or semantic incompatibilities?** Here, compile failures show source-code incompatibilities, and failed tests that passed before the update show binary or semantic incompatibilities.

As UpCy tries to minimize binary, source-code, conflicting, duplicate, and blossom incompatibilities, we measure how often and how many of these compatibilities a safe backward compatible update has to fulfill. With an increasing number of compatibilities that an update has to fulfill, the complexity for developers to reason about update compatibility increases along with the helpfulness of UpCy. Thus, we investigate **RQ2: How many compatibilities does an update have to fulfill (source-code, binary, semantic, conflict, duplicate, and blossom)?** Here, we use the unified dependency graph to derive the number of compatibilities an update has to fulfill, which is equivalent to the edges in the graph.

Since we are interested in comparing the effectiveness of UpCy with the naïve approach for practical applications, we measure how often UpCy can provide better updates with fewer or no incompatibilities. Therefore, we check **RQ3: In how many cases does UpCy minimize the number of incompatibilities compared to a naïve update?** To identify compatibility issues, we use the ABI and source-code API-compatibility check of the tool SigTest [Tul22; Ora14c] and intersect the results with the API calls in the unified dependency graph.

4.3.2 Study Objects & Methodology

Data Set For the evaluation, we use the data set created by Hejderup et al. [HG22] of open-source projects sampled from GitHub that use Java as the primary language and Maven as the build-automation tool. Hejderup et al. already used the data set in a study to measure the effectiveness of automated update approaches. Thus, we

consider the data set equally well-suited for evaluating UpCy. The data set provides a representative sample of mid-sized, well-tested, open-source projects with a significant number of dependencies [HG22]. On average, each project assembles 668 methods, and 75% of the projects assemble around 588 or fewer declared methods. The median of direct dependencies is 7, and for transitive dependencies 16, indicating an expansion of transitive dependencies, which is in accordance with our study in Chapter 3 and other studies [Pas+18; WD14; Pas+22]. The downloadable data set [HG21] assembles commits of 462 different Maven projects.

Like Hejderup, we could not build all projects but were able to build (`mvn compile`) 380 projects successfully. These 380 projects constitute in total 2,047 different Maven modules. Typically, a Maven project constitutes one or more (independently) compilable sub-projects, so-called Maven modules, each with its own set of dependencies. 1,325 of these modules have OSS dependencies and a non-empty dependency graph.

Seeding Updates For developers, it is crucial to update an outdated or vulnerable dependency quickly when a new vulnerability has been discovered to eliminate that dependency from the dependency tree. For our evaluation, we had to derive adequate test updates to compare naïve updates and UpCy's updates. To create fair and unbiased test updates, we randomly choose from each Maven module up to 10 dependencies and then randomly up to 10 newer versions of those dependencies to seed test updates. Here, we only selected dependencies with the scope *compile* and excluded dependencies with the scope *testing, system, runtime, provided* since they usually do not specify a version and are unavailable during compile time, and thus cannot be statically checked.

In total, we created 29,698 test updates for 5,558 different artifacts in 1,325 modules: 8,327 updates of direct dependencies, and 21,371 (71.96%) updates of transitive dependencies. On average, we generated 22 (mean, 11.3 std) updates per module, with a positively skewed distribution. 75% of all modules in our sample had around 32 or fewer updates. The largest project had 85 updates.

4.3.3 Results

RQ1: How effective are naïve Updates?

State-of-the-art tools naïvely add the updated artifact as a direct dependency to the build-automation tool's configuration (`pom.xml`). To evaluate the effectiveness of this approach, we first performed each update as a naïve update, and then checked for compatibility issues. To check for source-code, and binary or semantic incompatibilities, we executed the Maven commands `mvn compile` and `mvn test` two times: on the original project and after applying the naïve update.

If the `mvn compile` command fails during the second invocation, the new artifact introduces source-code incompatibilities, as the project's code that invokes the updated artifact's API can no longer be compiled successfully. Because it is recommended practice to include artifacts whose API is invoked as direct dependencies, compile failures usually occur for direct dependencies only. If the `mvn test` command fails during the second invocation, the new artifact version introduces binary or semantic incompatibilities as formerly valid test cases fail to link or run successfully with the updated artifact.

Table 4.1 shows the results. 26,966 (90%) of the 29,698 updates compiled successfully and no tests failed. Only 2,732 updates actually result in compile (1,393) or test failures (1,339), shown in column *#failed updates*. The compile and test failures occurred in 372 (28.07%) of the 1,325 modules. The fact that only a few naïve updates resulted in errors is not surprising: The 29,698 updates *only* affect 5,558 different artifacts. Most seeded updates are successive version increments of the same artifacts in the same modules. The small number of failed naïve updates shows that the likelihood is high that if the update of an artifact from version 1.1 to version 1.1.5 is successful, updating that artifact to version 1.2 and version 1.4 will also be successful. Consequently, only a small number of the seeded updates fail.

Table 4.1: The table reports how many updates failed during the build process caused by source-code incompatibilities or during test execution, indicating binary or semantic incompatibilities.

failure type	#failed updates	per module				
		mean	std	min	median	max
build or test	2,732	7.34	8.11	1	6	36
build	1,393	5.71	7.34	1	4	36
test	1,339	7.52	8.03	1	6	36

Nevertheless, Hejderup et al. [HG22] found that a project's test cases typically cover less than 60% of the calls to direct dependencies, and the test coverage drops to 21% for calls to transitive dependencies, which was the majority of updates in our data set (71.96%). Since we used a project's test to detect binary incompatibilities and semantic breaking changes by executing the command `mvn test`, the number of failed updates is a lower bound only.

Findings from RQ1: 28.07% of the 1,325 modules suffered from compile or test failures when applying naïve updates of state-of-the-art tools.

RQ2: How many compatibilities does an update have to fulfill (source, ABI, blossom, semantic)?

To conduct a safe backward compatible update, developers must ensure that no incompatibilities occur between the updated artifact, the project, and the other artifacts in the dependency tree. To estimate how complex this reasoning is, we computed how many binary and potential semantic incompatibilities an update introduces on average.

To test for ABI incompatibilities between two versions of an artifact, we used Oracle's SigTest tool [Tul22; Ora14c], which is specifically designed to compare the signatures of the API types of two versions of the same artifact and identify binary and source incompatibilities.

To get an estimate of potential semantic incompatibilities, we compared the bytecode of the former method's body with the new method's bytecode using our tool SootDiff [DHB19]. SootDiff's comparison was specifically designed to be resistant to changes induced by various compilation schemes, and thus allows us to check for bytecode equivalence even if different Java compilers, Java versions, or source-code changes that do not change the bytecode have been applied to one of the classes, e.g., variable renaming. If SootDiff finds a difference, this indicates that the method's semantics *may* have changed. Nevertheless, only checking the bodies of API methods will cause transitive changes in the preconditions to be missed. For instance, if a public method m_{api} calls a private method m_{priv}, and only m_{priv} has changed, we would miss the fact that m_{api}'s API has changed if m_{priv} is excluded from the comparison [Foo+18]. Thus, we build the artifact's call graph using Soot's class hierarchy analysis and iterate over the call chain of each API method and check if the body of any method along the chain has changed [Foo+18]. This

approach yields an over-approximation: SootDiff will detect and report a difference if a potential semantic change exists. Note that the precise detection of semantic changes is impossible, as described in Section 4.1.

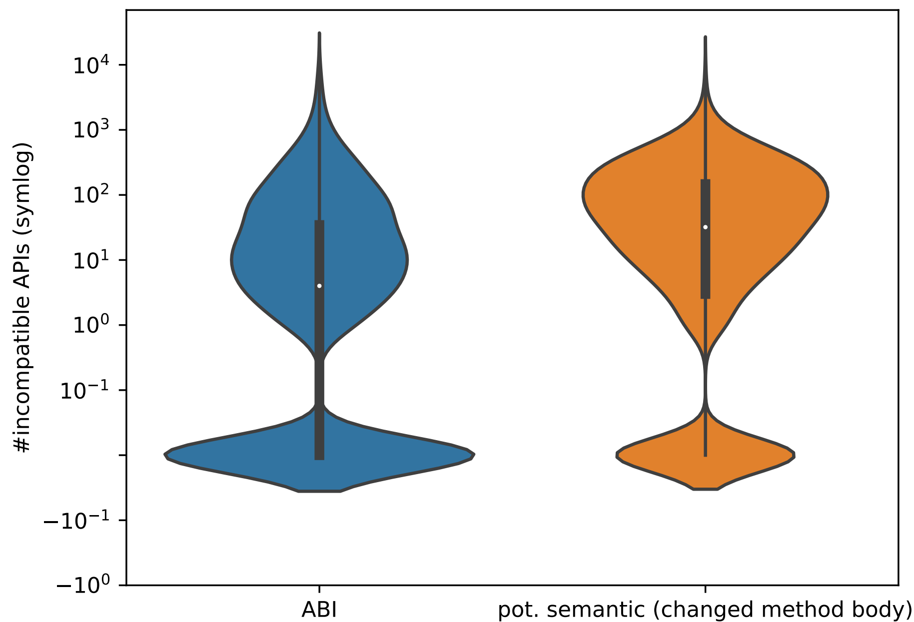


Figure 4.4: To understand to what extent an update introduces incompatibilities, this violin plot shows the amount of binary (left) and semantic (right) incompatibilities (Y-axis has a logarithmic scale), showing on average 83 binary incompatibilities and two peaks at 0 and 127 for semantic incompatibilities.

Figure 4.4 presents the results for binary and potential semantic incompatibilities as a violin plot, using a symlog transformation with base 10 to deal with outliers. Overall, 65.38% of all artifact updates suffer from binary incompatibilities issues. On average, an update introduces 83 binary incompatibilities. 75% of all updates have fewer or equal to 35 binary incompatibilities.

The distribution of potentially semantic incompatibilities (changed method bodies) suggests two classes of artifacts. In the first class—the peak at 0, we have artifact updates that do not introduce any changes in existing method bodies. In the second class—the peak at 127, we have artifact updates that contain changes in more than 100 methods, indicating larger refactorings.

To compute to how many other artifacts an update has to be binary and semantically compatible, we counted the number of incoming edges in the dependency graph. As described in Section 4.2, all edges in the unified dependency graph represent a use relation between two dependencies, thus an incoming edge shows that another artifact uses the artifact's API, and an outgoing edge shows that the artifact uses another artifact.

Table 4.2: The table shows the number of compatibilities an update has to satisfy due to blossoms, conflicting or duplicate dependencies, and binary-dependent artifacts.

artifact update	mean	std	min	median	max
size of blossoms	4.19	3.92	1	2	30
#conflicts	1.71	1.73	1	1	16
#duplicates	2.53	3.12	1	1	31
#binary dependents	1.95	2.33	1	2	32

Table 4.2 shows the results. A developer has, on average, to ensure binary compatibility to 1.9 other artifacts in the dependency graph, and consider 1.7 conflicts and 2.5 duplicates. If an artifact is part of a blossom, the blossom contains 4 artifacts on average. Thus, developers must not only reason about the relations between their project and the artifact but also respect duplicate and conflicting dependencies as they can induce inconsistent runtime behavior [Wan+22].

Findings from RQ2: When updating an artifact, on average, developers need to maintain binary compatibility to at least 2 further artifacts, and consider conflicts with 1.7 other instances of that library.

RQ3: How effective is UpCy compared to naïve Updates?

To evaluate the effectiveness of UpCy, we executed it on 20,610 updates of transitive dependencies; these are updates on which the naïve approach and UpCy completed (cf. Section 4.3.3), and checked in how many cases UpCy provides better updates. To increase the performance of the Neo4j database lookups, we restricted the length for transitive dependencies to 5 and set a timeout of 180 seconds for a query. Thus, UpCy may miss update options in scenarios where shared nodes have a path length greater than 5 or run exceptionally complex queries.

In total, UpCy could successfully compute for 16,884 (81.9%) of 20,610 updates if incompatibilities exist using SigTest and SootDiff. For the remaining updates, the computation of incompatibilities failed since either the compilation process of the unmodified module failed, the call graph was empty, e.g., for test projects, or the tools SigTest failed.

Table 4.3 shows the descriptive statistics for the updates that UpCy computed using the min-(s,t)-cut approach. Note that the table shows the number of incompatible artifacts. It does not contain the number of violated API calls.

Table 4.3: The table compares the efficiency of naïve and UpCy update in terms of the number of incompatibilities the update introduces.

update	count	mean	std	min	median	max
naïve	3,821	2.17	5.38	1	1	124
UpCy	3,821	1.15	1.29	0	1	15
fewer incompatibilities	1,572	0.62	1.38	0	0	15
more incompatibilities	14	2.36	0.49	2	2	3
incompatibility reduction		1.01	5.29	-2	0	124
complexity of min-(s,t)-cut		1.63	0.98	0	2	10

The table shows that 3,821 of the naïve updates produced incompatibilities, and thus UpCy computed alternative updates using the min-(s,t)-cut. The mean in row *UpCy* shows that UpCy's updates have, on average, fewer incompatibilities than naïve updates. In only 14 cases, the computed min-(s,t)-cut updates had more incompatibilities, shown in row *more incompatibilities*, resulting in the negative min value in row *reduction*.

For 1,572 (41.1%) updates, UpCy computed a min-(s,t)-cut update with fewer incompatibilities. Even 1,102 (70.1%) of these updates have zero incompatibilities. In cases where the naïve update introduced incompatibilities, UpCy computed update steps that require more than one artifact to update, which is given as min-(s,t)-cut complexity in Table 4.3, or found a compatible update of a preceding dependency. The high standard deviation of the naïve updates shows that the number of artifacts to which an update introduces incompatibilities heavily varies. The standard deviation for UpCy is lower; UpCy consistently reduces incompatibilities. While the number of updates for which UpCy computed a min-(s,t)-cut seems low at first glance, this is not surprising given the insights from *RQ1*: the majority of updates are version increments of the same artifact, and the numbers show that the likelihood is high that if an artifact update succeeded, further version increments, i.e., another naïve update, will succeed too.

On average, the updates computed by UpCy's min-(s,t)-cut approach require to update 1.63 (mean—in row *complexity of min-(s,t)-cut*) different artifacts. The biggest min-(s,t)-cut that minimizes the incompatibilities requires an update of 10 artifacts.

The results show that UpCy can effectively find updates with fewer incompatibilities than naïve updates. While other tools, which only apply naïve updates, yield incompatibilities, UpCy can successfully recommend compatible updates even if multiple

artifacts need to be updated. As the data set from Hejderup [HG22] exclusively contains well-tested, open-source GitHub projects, the results show that the need to update multiple artifacts to achieve compatibility is prevalent in practical cases.

> **Findings from RQ3:** UpCy can effectively reduce incompatibilities for 41.1% of the updates in which the naïve update lead to source or binary incompatibilities.

Example Update of Multiple Artifacts Figure 4.5 shows a simple example of the computed min-(s,t)-cut for the dependency graph of the project `mybatis-shards` in the data set, requiring the update of two dependencies to maintain compatibility. The outdated dependency in the example is `cglib:cglib:2.2.2`. The dependency should be updated to version 3.3.0. Instead of simply updating `cglib` to the target version, which is the naïve approach, UpCy computed the min-cut cutting the edge between the project and the target dependency `cglib:cglib`, and the edge between the project and the dependency `asm:asm:3.3.1`. The min-cut indicates that both dependencies `cglib` and `asm` need to be updated—the root nodes of the sink partition. Using our Maven Central graph database, UpCy found that the target version `cglib:cglib:3.3.0` already depends on `asm:7.1` and that an update for `asm` exists to version 7.1 by querying for the *updateGraph*, which exactly returns those artifacts. Thus, UpCy proposes to update `cglib` to version 3.3.0 and `asm` to version 7.1.

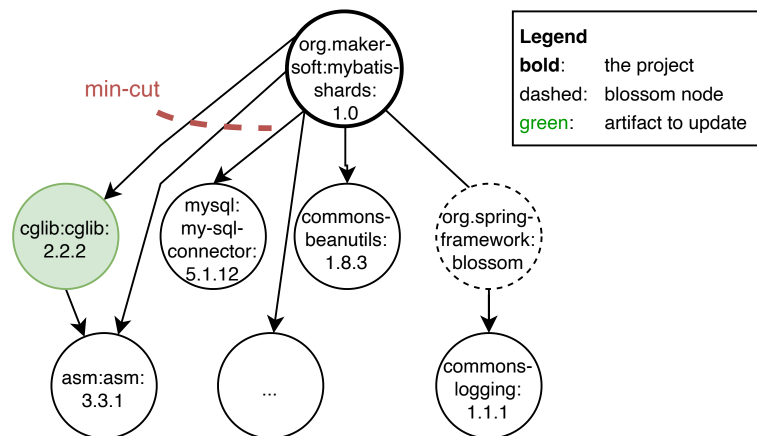


Figure 4.5: Example of min-(s,t)-cut computed by UpCy in project `mybatis-shards` to find compatible updates for `cglib`, suggesting to update the artifacts `cglib` and `asm`.

Computation Resources The computation resources for UpCy must be distinguished into resources required for creating the Neo4j database of Maven Central, which is only done once, and resources required for computing a list of updates, which is done for each dependency update. To create the Neo4j database, we executed UpCy's crawler on a machine with 68 GB RAM and 12 vCPUs for 10 days. The final database has a size of 22 GB.

To generate the list of compatible updates, UpCy executes two steps: computing the min-cut, and querying Neo4j. The min-(s,t)-cut computation takes only milliseconds on the developer's machine. The time for querying the Maven dependency graph depends on the machine hosting the database. In our case, queries ranged from milliseconds to a few minutes, depending on the number of dependencies to update and the size of the min-cut. However, the query performance can be further optimized, e.g., by using indices or by splitting large queries into smaller ones. Neo4j can handle millions of nodes; the Maven Central graph is relatively small with 8.5 million nodes.

In our evaluation, we set a timeout limit of 180 seconds per Cypher query to prevent long-running queries from blocking. Due to the timeout, 761 transitive updates $(21,371–20,610)$ across 8 Maven modules failed to complete, or no solution was found in the graph database.

4.4 Threats to Validity

In this section, we discuss threats to the design of UpCy and our evaluation.

4.4.1 Finding Compatible Updates with UpCy

The biggest threat to UpCy is the correctness, completeness, and recency of the Maven Central graph database; errors in the database cause UpCy to overlook valid, compatible updates. Similarly, the soundness of the call graph construction impacts UpCy's precision. Our current implementation uses the class hierarchy analysis of Soot, which is an over-approximation. Consequently, the unified dependency graph contains API edges between artifacts that are not invoked actually, unnecessarily constraining the number of compatible updates.

A further threat is that UpCy does not compute all min-(s,t)-cuts for the unified dependency graph. Nagamochi et al. [NNI00] give an algorithm that finds all minimum cuts efficiently. The existing implementation of the algorithm [Hen+20] is written in C++. We refrained from integrating the algorithm in Java applications ourselves. Nevertheless, our evaluation shows that UpCy can identify improved updates even if not all min-(s,t)-cuts are considered.

Another threat is the weight of edges for computing our min-(s,t)-cut. UpCy assigns the weight 1 to every edge, independently of the number of API calls between the artifacts connected by the edge. Thus, incompatibilities are minimized w.r.t. the number of incompatible artifacts but independently of the number of API calls. In some cases, minimizing the number of incompatible API calls might be more beneficial and require less effort for developers when choosing an update.

4.4.2 Evaluation

Sampling artifacts and versions randomly for creating updates poses a threat to our results, as we may select artifacts that are either relatively rarely or frequently updated. To mitigate this risk, we used the sampled data set from Hejderup [HG22] of mid-sized, well-tested, open-source projects from GitHub covering 5,558 different artifacts in 1,325 Maven modules to choose practical, realistic projects.

A threat to *RQ1* and *RQ3* is the fact that some of the seeded updates ask to update the selected artifact to the latest release. We found that for some dependent artifacts (those preceding the chosen artifact for updating in the dependency tree), no versions have been published yet that depend on the latest release. Consequently, the options for compatible updates were limited to direct updates of those dependencies only.

Similarly, some generated updates ask to update to a release that has been skipped by dependent artifacts. For instance, in our evaluation set, we seeded a test update from `org.slf4j:slf4j-api:1.7.21` to version 1.7.34 for a project. In the project's dependency tree `logback-classic:1.17` was a predecessor of `slf4j-api`. However, no version of `logback-classic` exists that depends on `slf4j-api:1.7.34`. Instead, recent versions depend on newer releases of `slf4j-api`. In such cases, UpCy could only identify naïve updates.

The biggest threat to *RQ3* is the completeness of our database, as described above. In cases where our database is incomplete, UpCy cannot identify valid update steps, making *RQ3* an under-approximation.

4.5 Related Work

In the following, we discuss studies investigating how developers update vulnerable or outdated open-source dependencies and approaches for computing the compatibility of dependency updates.

4.5.1 Studies: How Developers Update (Vulnerable) Dependencies

Researchers have conducted several studies investigating practices around updating (vulnerable) open-source dependencies [Kul+18; MP17; Bog+16; Pra+21]. The studies show that developers typically hesitate to update dependencies since they are afraid of introducing breaking changes and unnecessary refactoring efforts, and thus prioritize other tasks. Kula et al. [Kul+18] found that for 81.5% of the studied dependencies, developers did not update dependencies even though they contain known vulnerabilities.

Mirhosseini et al. [MP17] found that automated pull requests can encourage developers to update dependencies faster. However, they also suffer from high rates of rollbacks and missing support in continuous integration pipelines, deferring developers from automatically updating. Similarly, Bogart et al. [Bog+16] point out that developers perceive automated pull requests as information overload, which does not help evaluate an update's consequences.

Bavota et al. [Bav+15] studied the evolution of dependencies for 147 Java projects in the Apache ecosystem for a period of 14 years. The authors found that developers do not update to all available versions of a new dependency but tend to update their dependencies to more recent releases if the update contains substantial changes. However, the authors also found that developers are reluctant to conduct updates if the API changes. The authors conclude that tools that check the side effects of an update, such as API changes that require changes in the project, might be highly beneficial for developers.

Prana et al. [Pra+21] study the risk of vulnerabilities in used open-source dependencies for 450 software projects written in Java, Python, and Ruby sampled from GitHub. The authors found that vulnerable dependencies are a major issue in software projects and that it takes developers 4–5 months to incorporate an update into their application. Further, the authors suggest that this latency may be caused by the developers' perceived risk of breaking the application.

4.5.2 Update Compatibility Analysis

Dietrich et al. [DJB14] studied the effect of source code and binary compatibility issues in the Qualitas corpus. They found that 75% of the updates introduced breaking changes, but only a few resulted in actual errors in the project. The authors emphasize that their study only applies to the Qualitas corpus, which only has a few projects with intertwined dependencies.

In a follow-up study, Dietrich et al. [Die+19] investigate if open-source developers obey to the Semantic Versioning schema [Pre21] on over 70 million dependencies for different build-automation tools. The authors conclude that there is no evidence that the authors of OSS switch to semantic versioning on a large scale.

Wang et al. [Wan+22] conducted a study on whether dependency conflicts may break a project's semantics. The study finds that dependency updates introduce semantic breaking changes in 50 of the 128 sample Java projects. To detect semantic breaking changes between two versions, the authors present SENSOR. SENSOR applies call graph analysis to detect changed API and applies automated test generation to uncover semantic breaking changes.

Hejderup et al. [HG22] empirically investigate if test suites, which are used by automatic pull requests, are reliable in detecting breaking changes in updates. The study finds that test suits can only cover 58% of direct and 21% of transitive dependency calls, and thus are unreliable. The authors developed a static change impact analysis Uppdatera to alleviate this situation. Uppdatera statically computes the difference between two versions of an artifact based on the source codes' abstract syntax tree to identify methods with code changes. Using a heuristic, Uppdatera determines if the changes break the method's semantics. Then, Uppdatera computes the project's call graph and evaluates if the project invokes methods with breaking changes.

4.5.3 Repository Dependency Graphs

Hejderup et al. [HVG18] present a concept to create software repository call graphs for npm, also called ecosystem call graphs. A software ecosystem call graph specifies the methods an artifact invokes on its dependencies. The suggested ecosystem call graphs are similar to our unified dependency graph, except they are independent of a particular application and should scale to the whole npm repository. As ecosystem call graphs are independent of the application, they must conduct a standalone call graph constructing of each dependency, simulating all potential usages of the artifact; we discuss challenges for standalone analysis in Chapter 5.

Düsing and Hermann [DH22] investigate the impact of vulnerabilities on the public open-source artifact repositories Maven Central, NuGet, and npm. To do so, the authors created a Neo4j graph database of the repositories containing the artifacts and their dependencies. The presented approach and data model focus on the propagation of vulnerabilities through the repository. In contrast, our approach and model of Maven Central are tailored for querying compatible updates.

Fan et al. [Fan+20] present a complementary approach to capture inconsistency between declared dependencies (the dependency declared in the build-automation tool's configuration) and the actual dependency (the dependency needed during compile time) for GNU make, which they also call unified dependency graphs. The proposed unified dependency graphs are used to uniformly encode the dependencies between build targets to achieve reproducible GNU make builds, and address a different concern as our unified dependency graphs.

4.6 Conclusion

In this chapter, we presented UpCy, an approach to automatically compute updates for vulnerable or outdated dependencies with minimal incompatibilities. Recent research [Kul+18; Bav+15; MP17] highlights that developers hesitate to update dependencies and mistrust automated approaches, like Dependabot, since they are afraid of introducing incompatibilities that break their projects or lead to unwanted side effects. UpCy can support developers in finding updates even in complex scenarios where a compatible update requires updating multiple dependencies. To do so, UpCy investigates the project's dependency graph and explores multiple update options, whereas state-of-the-art approaches naïvely focus on the outdated library only. UpCy uses the min-(s,t)-cut algorithm to identify updates with minimal incompatibilities, and queries a graph database of Maven Central for new dependencies. To assess the impact of an update, UpCy determines incompatible API calls using the static analysis framework Soot. As an output, UpCy proposes a list of dependencies that developers can add as direct dependencies to eliminate a vulnerable or outdated dependency (especially a transitive one) from their project's dependency graph.

As our study in Chapter 3 and recent research [PPS18; Pas+22; HVG18] show (i) the use of vulnerable or outdated dependencies is an issue in both open-source and commercial applications, and (ii) the use of frameworks is widespread—most of

the studied projects included a framework. In such scenarios, UpCy is especially beneficial since frameworks usually require updating multiple dependencies, not only a single dependency.

Further, our evaluation on a representative data set of 1,325 well-tested, open-source Java projects sampled from GitHub shows that UpCy can effectively provide update suggestions that produce fewer incompatibilities than current, naïve approaches. In 41.1% of the cases where the naïve update leads to incompatibilities, UpCy could detect an update option with fewer incompatibilities to other libraries. Even 70.1% of the generated updates have zero incompatibilities. Additionally, our evaluation showed that naïve updates not only failed on a few projects, but we also found failed naïve updates in 28% of the projects.

5 Securely Integrating Open-Source Software with Java's Module System

Our study in Chapter 3 shows that vulnerabilities in included Open-Source Software (OSS) are an unfortunate reality for commercial and open-source applications. Even worse, our study reveals that typical development practices like forking, patching, re-compilation, or re-bundling can heavily decrease the effectiveness of vulnerability scanners. As a result, scanners may not detect known-vulnerable open-source dependencies in the project, leaving the application unknowingly exploitable. For cases in which developers could successfully identify outdated or vulnerable open-source dependencies, we introduced UpCy—an approach to automatically find compatible updates in Chapter 4. Nevertheless, another unfortunate reality is that 0-day vulnerabilities can appear in any used OSS at any time with different consequences, ranging from remote-code execution to denial-of-service attacks, as vulnerabilities like Log4Shell [NVD21] or de-serialization issues in `jackson` [Cau17] showed. Thus, it is desirable to defend against inadvertent vulnerabilities in used OSS by limiting their impact.

A possibility to limit the impact of vulnerable OSS is to limit its access to the host system, to the rest of the application, and to system classes of the Java platform by executing the OSS in an isolated environment. In Java, the security architecture and isolation mechanism rely on preventing the access of untrusted code to security-sensitive system classes, private methods, and fields [HB21]. If attackers obtain access to those fields or methods, they can turn off all security checks.

To prevent access to security-sensitive classes, methods, and fields, the Java platform hides them in system classes and restricts access to them using encapsulation. Encapsulation is a fundamental concept for wrapping classes and class members (fields and methods) inside their package or their declaring class [Ora96]. In Java, encapsulation is achieved by declaring access modifiers (private, protected, package private) for classes or their members to hide them from other classes.

The Java Virtual Machine (JVM) enforces encapsulation by a set of different security mechanisms [GED03], e.g., bytecode verifier, class loaders, and Java's security manager, which we introduce in Section 5.1.
However, following the general security principle of *Defense in Depth* [NSA12], whose idea is the implementation of multiple security layers such that if one layer fails or is circumvented, the other layers will still be up to protect the asset, the isolation of vulnerable OSS is only one layer. Consequently, vulnerable OSS should be guarded by further security measures such as monitoring and obfuscation [Sch11].

The Java programming language provides a built-in sandbox to enable the safe execution of untrusted third-party code isolated from the application. The sandbox aims to defend the application and host system from malicious behavior in potentially malicious code and inadvertent vulnerabilities [GED03; Cok+15]. Integral parts of Java's sandbox are (i) Java's encapsulation mechanism to prevent access to security-sensitive fields and methods in the Java Class Library (JCL), e.g., `sun.misc.Unsafe`, which can be (ab)used to bypass visibility and access constraints or deactivate security features, and (ii) Java's security manager to prevent access to security-sensitive features at runtime.

However, with the release of Java 17, Java's security manager has been marked as deprecated and for removal in future versions [Ora22; Ora21; Ora23b]. Reasons for removing the security manager are its rare usage, complex configuration, and performance overhead. These issues prevented its wide adaption and made its maintenance uneconomical. Crucially, no replacement for the security manager is planned. Instead, security mechanisms should rely, at least partially, on the Java module system [Ora22; Ora21; Ora23b]. The Java module system has been introduced with the release of Java 9. Its "primary goals are to make implementations of the Platform more easily scalable [...], improve security and maintainability" [Ora17b] by strongly encapsulating security-sensitive types and fields [Ora15b], making them inaccessible outside the Java Development Kit (JDK) and restricting access between modules. For example, security-sensitive classes such as `jdk.internal.misc.Unsafe`[1] are now only available within the JDK. In fact, a study by Holzinger et al. [Hol+16; HB21] on 87 Java exploits, conducted *before* the introduction of the module system, found that 61% of the exploits abuse flaws in the JCL to circumvent the encapsulation of JCL internal classes and methods to access security-sensitive types, methods, and fields to deactivate or bypass Java's sandbox and security mechanisms. The module system's strong encapsulation guarantees may serve as "security bulkheads that restrict a vulnerability in one component from affecting others" [Ora23b].

[1] The replacement for `sun.misc.Unsafe` in earlier versions.

To clarify to what extent the module system can help to integrate OSS securely, we first discuss Java's security architecture and sandbox in Section 5.1. Next, we present the Java module system and introduce strong encapsulation for restricting access to internal entities. Further, we discuss the limitations of the module system for confining sensitive data with an example in Section 5.1.3.

To benefit from modules' strong encapsulation and to use modules as *security bulkheads*, developers must design and implement their application modules so that an exploit executed in one module, e.g., an included OSS, only has limited access to sensitive types and data in the application's modules, e.g., encryption keys or database access.
A guiding principle for the constructive secure design of applications is the *Principle of Least Privilege* [SS75], which was defined in 1975. The principle states that "every program and every user of the system should operate using the least set of privileges necessary to complete the job" [SS75] in order to limit the damage of failures and errors or vulnerabilities an attacker abuses. Applied to the secure, modular development of Java applications integrating open-source artifacts, the Principle of Least Privilege requires that every module only has access to the privileges—at the Java language level, these are classes, types, objects, data, and functionalities—that are necessary to complete the job.

While the module system was designed to encapsulate internal types and classes, it cannot prevent the unintentional *escaping* of security-sensitive data and instances to the outside, e.g., secret keys [Ora14b]. Detecting such escapes requires reasoning about complex data flows between modules and which classes, methods, and fields a module actually exposes, which is hard to do manually.

With ModGuard, we present a novel static analysis based on Doop [SB11] to allow developers to benefit from the module system in Section 5.3. ModGuard complements the Java module system with an analysis to automatically identify what sensitive entities eventually *escape* the declaring module, and thus become reachable to other modules, e.g., an included OSS module. Thereby, ModGuard enables developers to leverage the module system security-wise by identifying unintended data flows that a secure, modular design should prevent. To assure the soundiness [Liv+15] of ModGuard, we introduce a formal specification of what we call *module entry points*, i.e., the set of method implementations a module defines, and which are invokable by other modules, either explicitly because their type is exported, or implicitly due to an exported supertype in Section 5.2. This entry-point definition can serve as a basis for further analyses supporting Java modules, e.g., Application Programming Interface (API) analysis, as it precisely defines what instances can eventually become

reachable, and thus must be API compatible. Further, we present a micro-benchmark suite MIC9Bench for evaluating Java module escape analysis and to facilitate their further development.

In Section 5.4, we conduct a case study on Apache Tomcat and check how ModGuard can effectively aid in identifying the escaping of sensitive entities to support developers in migrating to a module system. The case study also shows that migrating to modules is not straightforward but requires refactoring of an application's architecture to confine sensitive entities successfully. Further, the case study shows that modules cannot limit the impact of vulnerable OSS on a project completely. Thus, we discuss further required extensions and limitations of the module system for restring inadvertent vulnerabilities in OSS and conclude that modules can only serve partially for isolation in Section 5.5.

We complete the chapter with related work in Section 5.6 and a summary of Java's module system and its limitations in Section 5.7.

5.1 Java's Security Architecture & Module System

In the following, we present an overview of Java's security architecture, sandbox mechanisms, and encapsulation concept. In particular, we focus on the security mechanisms for isolation and preventing access to internal types, methods, and fields. Following the *Principle of Least Privilege*, these mechanisms are building blocks for the secure integration of OSS. Finally, we present the module system and its novel *strong encapsulation*.

5.1.1 Java 1.2 Security Model

The Java programming language and runtime environment were released to the public in 1995 to provide an easy, operating system-independent programming language with type and memory safety. With Java version 1.2, its security architecture [GED03; Gon+97] has been introduced, which we present in the following. A Java application is composed of a set of Java classes that are executed by the JVM in the Java Runtime Environment (JRE). Figure 5.1 gives an overview of the JRE's components. Class files can be bundled into JAR files for easier deployment and distribution, as presented in Chapter 2. Class files do not assemble actual machine code. Instead, Java source code is compiled into bytecode, a platform-independent

representation of the program behavior (cf. Section 3.8.1). To execute the platform-independent bytecode, the JRE performs a Just-In-Time (JIT) compilation on the class files [Ora96].

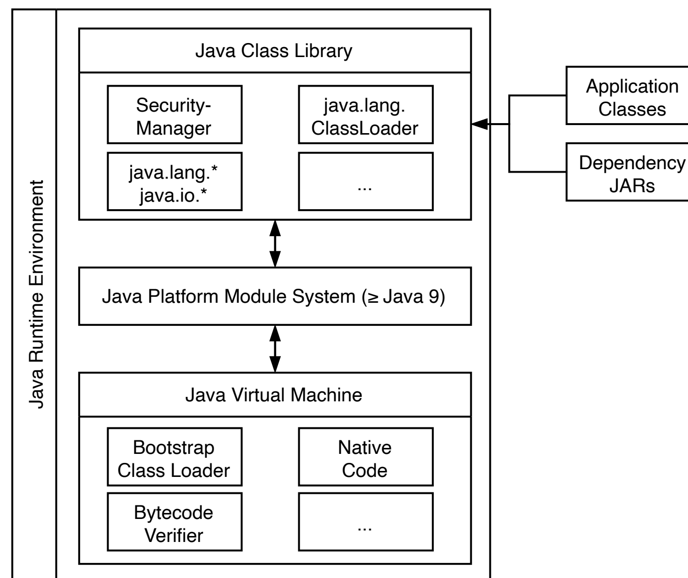


Figure 5.1: Overview of the components of the Java Runtime Environment.

Bytecode Verifier The first line of defense is the bytecode verifier, shown in Figure 5.1. When the JRE loads a class, the verifier checks if the class' bytecode obeys access restrictions, type-, and memory-safety properties [Ora96; Ler01]. In particular, the bytecode verifier checks if

- all variables are initialized before usage.
- there are no stack overflows or underflows.
- all types of the parameters of all bytecode instructions are correct.
- all class and class member access is legal, and access control is not violated.

The bytecode verifier's checks for type correctness, stack over- and underflows, and variable initialization guarantee type- and memory-safety [Ler01]. In particular, the type correctness checks guarantee the correctness of the bytecode blocks to prevent stack over- and underflows. The initialization checks prevent direct access to memory pointers and associated security weaknesses such as stack smashing [One96]. Also, all usages of arrays and strings are bound-checked, preventing writing beyond allocated memory. The access control check ensures that the bytecode obeys the *encapsulation* rules of the Java language: *public* types and members are available

to any other class, *protected* types and members are only accessible to subclasses, *private* types and members are accessible only from within the class, and *default* types and members are accessible to all types within the same package [Ora96].

Class Loaders & Classpath The next line of defense is the Java Class Loader, which loads the verified bytecode into the JRE [GED03; LB98]. The JRE assembles a hierarchy of different class loaders; each with different responsibilities and capabilities. The primary class loader is the *bootstrap class loader*, whose responsibility is to load the privileged system classes of the JCL from predefined, dedicated paths. The next class loader is the *extension class loader*, whose responsibility is to load classes that do not belong to the JCL but extend the Java platform. An example is SunJCE, an extension of different encryption and hashing routines. The *application class loader* is responsible for loading the application classes and their dependencies. The application class loader linearly searches the *classpath* for class files and loads them on demand. Since the application class loader linearly traverses the classpath, a single class loader can load exactly one—the first found—class with the given fully-qualified name (FQN).

For loading classes, the class loaders implement a class loading delegation strategy. When a class loader is asked to load a class, the class loader checks if the class has already been loaded. Otherwise, the class loader loads the class itself or delegates the request to its parent. If there is no parent, the primordial class loader is asked. Besides this default class delegation mechanism, frameworks and applications can define their own class loaders and delegation strategies, e.g., for loading conflicting versions of JARs in different class loaders or for loading and unloading JARs dynamically at runtime. Further, custom implementations can block access to certain classes by not delegating a loading request to a parent class loader or blocking the load request completely. An example of a framework implementing its own custom class loaders and delegation strategy is the OSGi platform [OSG23], which we shortly describe at the end of this section.

To ensure that classes are defined and loaded by the responsible loader with corresponding permissions, the class loaders only load classes from predefined paths: the boot class loader searches the *bootclasspath*, the extension loader searches Java's *ext* folder, and application class loaders linearly traverse the *classpath*. The class loader hierarchy is relevant for restricting access to security-sensitive classes and methods. A prominent example is the class `sun.misc.Unsafe`, which allows memory manipulation [Mas+15]. Its method `getUnsafe()`, which returns an instance of the class, is caller-sensitive to prevent illegal access from application classes [CGK15].

The method checks if its caller was loaded by the bootstrap class loader. If this is the case, the invocation is assumed to be legitimate, and the method returns an instance of `Unsafe`. If the caller was loaded from another class loader, the method throws an exception.

Java's Sandbox & Security Manager Further, Java code can be sandboxed by activating Java's built-in security manager [GED03; Cok+15]. When a class loader loads a class, the loader associates it with a *protection* domain based on the classes' origin, e.g., dynamically loaded from the network or a specific file. Developers can assign permissions to a protection domain, e.g., allowing network access or file system access, by specifying allowed or forbidden permissions for a protection domain in policy files using Java's policy language.

The security manager then enforces the policies. Whenever a class attempts to invoke a security-sensitive method, like writing to a file or gaining network access, the method triggers a permission check through a call to `SecurityManager.check*()`, e.g., `checkWrite(), checkAccess()`. The security manager then determines if the method invocation is permitted utilizing stack-based access control [BN05; WF98]. To do so, the security manager inspects the call stack, and checks whether all calling classes on the stack own the requested permission. If any caller does not have permission to execute the requested method, the security manager blocks the method invocation and throws an exception [Cok+15].

Class Introspection: Reflection & Dynamic-Language API With its reflection and dynamic-language API, e.g., MethodHandles and VarHandles, the Java language allows to inspect and change types, methods, and fields at runtime, enabling an application to adapt dynamically [LTX19]. For instance, reflection can be used to access class members that are unknown during compile time. A widespread use case is the (de-)serialization of classes implemented by serialization libraries like `jackson` or `protobuf`. Due to its dynamic nature and flexibility, reflection is widely used in Java applications, libraries, and frameworks [LTX19].

However, Java's reflection and dynamic-language API have also been widely used to break encapsulation, enabling the access and modification of private, protected, or default types and class members. Although Java's security manager checks if code is allowed to make use of the reflection API, Holzinger et al. [Hol+16] show in their study that exploits abuse reflection to circumvent Java's security mechanisms and access or modify security-sensitive entities regularly.

5.1.2 The Java Platform Module System

Up to Java 9, developers only had the option to structure applications, libraries, and frameworks into different packages and restrict the access to types and class members using access modifiers [Ora15a; Ora96]; where public types constitute the API and types with other modifiers are meant to be internal. As described above, developers, libraries, and frameworks regularly violate these access constraints using reflection to inspect types' internals, circumventing encapsulation [Sma+15; Bod+11; HB21; LTX19].

Modules Java 9 introduces modules to the platform as first-class constructs [Ora17c], providing new means to structure applications, libraries, frameworks, and their interfaces (API); classes are separated into packages, and packages are grouped into modules. Further, a module *strongly encapsulates* its internal implementation, preventing access to internal types during compile time and runtime [Ora15b]. In contrast to the previous security model, modules provide a *strong* encapsulation as even reflection is not permitted to access module-internal types.[2]

Like non-modularized Java artifacts, Java modules are distributed as JAR files, which assemble packages, classes, native code, and further resources. Yet, modules contain a static module descriptor (`module-info.java`) specifying the module's unique name, its dependency on other modules, its exported packages, and its re-exported module dependencies. The Java compiler and the JVM process the module descriptor, causing them to check and prevent access to the internal types of a module both at compile- and runtime. Listing 5.1 shows an excerpt of the module-descriptor of the JDK module `java.desktop`. The module `java.desktop` *requires* the modules `java.prefs`, `java.datatransfer`, and `java.xml`. The directive *requires transitive* specifies a dependency on another module and ensures that other modules reading the module also read that dependency. In this example, any module that reads `java.desktop` also implicitly reads the module `java.xml`. Further, the module descriptor specifies that the module `java.desktop` exports its packages `java.awt` and `javax.swing` to dependent modules—only modules that require `java.desktop` can access classes in the exported packages. In contrast to the Java 1.2 security model, modules are not loaded from the *classpath* but from the new *modulepath*.

[2] Java 17 forbids any reflective access to types that are not declared as open for reflection by default and even removes a formerly existing command-line option to enable it, enforcing strong encapsulation. `https://blogs.oracle.com/javamagazine/post/a-peek-into-java-17-continuing-the-drive-to-encapsulate-the-java-runtime-internals`

```
1  module java.desktop {
2      requires java.prefs;
3      requires transitive java.xml;
4      requires transitive java.datatransfer;
5      exports java.awt;
6      exports javax.swing; }
```

Listing 5.1: Example of the module-descriptor `module-info` of the module `java.desktop` of the Java Runtime Environment 1.9.

The module system is a fundamental shift from the previous classpath concept. Up to Java 8, every public class was visible to any other class on the classpath loaded by the same class loader. In Java's module system, a class contained in a module can only access the exported types of another module that it explicitly requires.

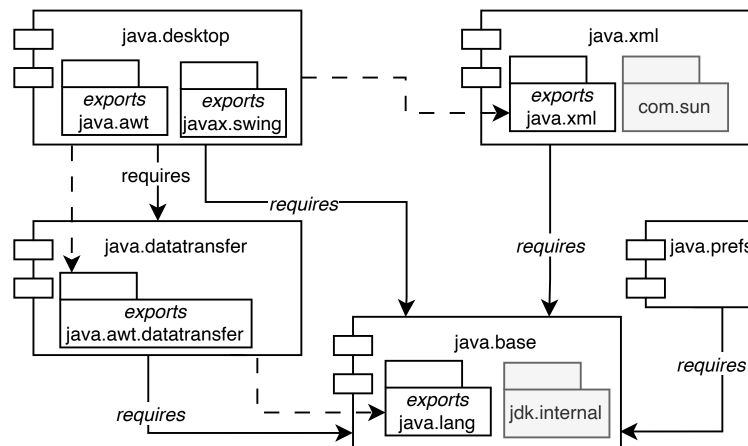


Figure 5.2: Subset of the module graph of the JRE module java.desktop. The solid arrows represent a requires dependency between modules. The dashed arrows show which packages a module can access. Only exported packages can be accessed by the dependent module. Internal packages (in gray) cannot be accessed.

The dependencies between modules specified in the module descriptors form an acyclic module graph. Figure 5.2 shows an exemplary subset of the module graph of the module java.desktop. The root of each module graph is the module java.base, which contains essential Java classes. The module `java.desktop` can only access classes of modules to which a require relation exists: `java.datatransfer` and `java.xml`. Correspondingly, the module `java.datatransfer` cannot access any class in the module `java.xml` as no require relation from `java.datatransfer` to `java.xml` exists.

In summary, a module can only access the public types in exported packages in required modules. For instance, in Figure 5.2, only the types of the exported package `java.xml` are visible to the module `java.desktop`, whereas types declared in the internal packages `com.sun` are invisible. All types that are neither explicitly declared public nor declared in an exported package are invisible outside their declaring module.

The access restrictions of the module system are enforced by the JVM during runtime, as every module and the package it declares are directly registered on top of the JVM in the Java platform module system outside the higher level JCL, shown in Figure 5.1. Consequently, the module system is independent of class loaders and the security manager.

Open Modules and Packages By default, the JVM denies runtime access to internal types via reflection or Java's dynamic-language API [Ora15b; Ora14a]. Nevertheless, modules can be declared as open. This grants compile-time access to exported packages only, but runtime access throughout [Ora17c]. Likewise, packages can be declared as open, thereby granting reflective access to all types in the package. Uses cases are to open data classes for (de-)serialization libraries.

Modules and Layers Since the module system is independent of class loaders, the class loading process is the same as in previous versions, and the same class loaders still exist. Nevertheless, the amount of system classes that are loaded by the bootstrap class loader has been reduced; essentially, only classes in the module `java.base` are loaded by the bootstrap class loader.

The module system introduces the concept of *module layers* [Ora14b] to allow the dynamic loading of modules at runtime. Module layers can be used for grouping modules, respectively module graphs. A module layer is created from a module graph and a function that maps each module to a class loader [Ora18]. A layer is responsible for finding a class loader to load classes from the module graph. Like class loaders, layers can be organized hierarchically. A module can read modules from its own layer and from layers lower in the hierarchy. Thus, the module system API allows the loading of multiple versions of a module if one organizes them in different class loaders and layers.

5.1.3 Motivating Example of Sensitive Entities Escaping a Module

Java's module system encapsulates module-internal types and members, and thus can be used to restrict access to sensitive members and types. However, a module cannot prevent the escaping of sensitive instances. In particular, a module cannot prevent instances of internal objects from escaping, e.g., an exported public factory $PubFactory$ in module A may instantiate and return an instance of an internal class $internalObjInst$ that implements an exported interface $PubInterface$.

Code outside of a module can invoke not only methods on object instances of exported types but can directly invoke all methods that are declared public (static or not), protected and static, or implement a method from an exported supertype on object instances, disregarding whether the types are declared in exported packages or not. The only condition is that code outside of module A must obtain access to an *object instance* to invoke the method on, e.g., all methods of $PubInterface$ that $internalObjInst$ implements can be invoked from any module depending on A.

Although escaping instances cannot be downcasted to the internal type, this behavior makes it complex for developers to reason about (unintended) data leaks. It requires reasoning about pointers and types, which is hard to discern in manual reviews.

```
1  module my.mod{ exports api; requires java.base; }
2
3  package api;
4  public class KeyProvider {
5    public static Key getKey(){
6      return new SecretKey();}
7   }
8
9  abstract class Key{
10   private byte[] key;
11   protected Key(byte[] key){this.key = key;}
12   public getKey(){return key;}
13 }
14
15 package internal;
16 public class SecretKey extends Key {
17   private static byte[] keyMaterial = {1,2,3,4};
18   public SecretKey() { super(keyMaterial); }
19 }
```

Listing 5.2: Example of the sensitive field `SecretKey.keyMaterial` escaping the module-internal package `internal` through the exported, overridden method `getKey()` of superclass `Key`. In green exported types and methods, in yellow internal types. Marked in red the sensitive field `keyMaterial`.

Listing 5.2 shows a simplified example in which access to an internal object instance results in an unintended data leak. The module `my.mod` exports the package `api`, making the types `KeyProvider` and `Key` visible to other modules but keeping the type `SecretKey` internal. The class `KeyProvider` in the exported package `api` creates an instance of the internal type `SecretKey` and returns it as the exported abstract class `Key`. Although `SecretKey` is in the module-internal package `internal` and stores its key material in the private field `keyMaterial`, classes outside the module `my.mod` can access the stored key using the inherited method `Key.getKey()` of the public exported type `Key`.

Since the Java module system confines types only, invoking methods of the exported supertype `Key` is permitted. The problem, in this example, is the leak of the internal `keyMaterial`. To detect such leaks, developers must reason about complex pointers and type hierarchy. Even for this simple example, a developer needs to reason that the constructor of the internal class `SecretKey` hands the internal, sensitive `keyMaterial` to the constructor of its exported superclass `Key`, which assigns it to the field `key`, which can be retrieved by the exported, inherited method `Key.getKey()`.

The inherited method `SecretKey.getKey()` is, in our terminology, an *entry point* of the declaring module (in addition to the directly exported entry points `KeyProvider.getKey()` and `Key.getKey()`). Such entry points must remain API (and semantically) compatible as they are part of a module's API. Manual reasoning about these pointer relations becomes virtually impossible in real-world scenarios with more complex pointers and types.

5.1.4 Excursion: The OSGi Platform

The Java module system [Ora17b] is not the first approach for introducing modules into the Java programming language. As we describe in Section 5.6, multiple modularization approaches for Java have been developed. A popular and widespread approach is the OSGi platform [OSG23] released in 2000. An OSGi module, so-called OSGi bundles, is a JAR file enhanced with a `META-INF/MANIFEST.MF` file declaring: the bundle's name, its dependencies to other bundles, and the packages it exports. The OSGi platform supports dynamic loading, installation, and de-installation of bundles at runtime. To do so, the OSGi platform implements on top of the JCL a set of custom class loaders with complex and dynamic class loader delegation strategies to support the dynamic loading and separation into namespaces.

The Java module system and OSGi serve different purposes: The module system aims to support the modularization of the JDK and applications by implementing strong encapsulation implemented as a separate platform module layer on top of the JVM, whereas the OSGi platform aims to support the dynamic loading or deinstallation of (concurrent) application modules at runtime implemented using Java class loaders.

5.2 Precisely Defining a Module's Entry Points

In the following, we precisely define what constitutes a module's entry points and which module-internal instances can be invoked from outside the declaring module—these are the classes that must be semantically API compatible or may lead to unintended leaks, as shown in the example in Listing 5.2. This model serves as a basis for our analysis ModGuard for identifying to which extent a module confines internal types and data. Further, the given entry-point definition can also serve to detect incompatible API types and methods, as it precisely defines which types and methods of a module can be invoked.

5.2.1 Explicitly vs. Implicitly Reachable Entry Points

As the KeyProvider example in Listing 5.2 shows, invocations crossing module boundaries are not restricted to explicitly exported types. Instead, external code, such as included OSS modules, may interact with many methods defined in the module. We call those methods *entry points*. A module's entry points constitute its API: all methods that code outside the module can invoke. Entry points comprise two kinds of methods: *explicitly* and *implicitly* reachable methods.

Initially, external code can only invoke *explicitly* reachable methods. Explicit methods are methods that are declared public (or protected) and whose declaring type and package are exported. In Listing 5.2, the methods `KeyProvider.getKey()` and `Key.getKey()` are *explicitly* reachable; their types `Key` and `KeyProvider` are public and declared in the exported package `api`. The use of *explicitly* reachable methods may grant access to further methods, which we call *implicitly* reachable.

Implicitly reachable methods are declared by internal types, inherit, implement, or override *explicitly* reachable methods of exported supertypes. In the example, `SecretKey.getKey()` is an *implicitly* reachable method: its declaring type `SecretKey` is module-internal, yet code outside the module can directly invoke the method on the object that `KeyProvider.getKey()` returns.

Note that *implicit* reachability is different from the usual notion of *indirect* reachability, i.e., the ability to invoke a method indirectly through a chain of calls.

The union of *all explicitly* and *all implicitly* reachable methods constitutes an upper bound of a module's entry points. This upper bound is naïve since it ignores whether objects of internal types, which declare the implicit method, actually become reachable outside of the module or not. To gain a tighter bound, ModGuard identifies which instances of internal types actually become reachable outside a module utilizing points-to analysis in Section 5.3.

5.2.2 Logic-based Specification of the Entry-Point Model

We realized our entry-point definition as logic-based specification in the syntax of declarative, Datalog-based analysis rules, extending the static analysis framework Doop [SB11; SKB14]. Listing 5.3 shows our entry-point model. In the following, we introduce the entry-point model's rules in detail.

A Datalog program consists of a set of facts and rules. Facts are represented as predicate values that are held to be true. Rules infer new facts (also called derived relations) from the conjunction of previously established facts already known to be true until no new facts can be extracted, separated by the left arrow symbol (←). For instance, the facts EXTENDS("A", "OBJECT") and EXTENDS("B", "A") mean that A extends the class *Object* and B extends the class A. The rules ISSUPERTYPE(X, Y) ← EXTENDS(Y, X) and ISSUPERTYPE(X, Y) ← EXTENDS(Y, Z), ISSUPERTYPE(X, Z) mean that X is a supertype of Y if Y extends X, and X is a supertype of Y if Y extends some class Z, and X is a supertype of Z.

In our Datalog entry-point model, in Listing 5.3, some relations are functions, written as RELATION[DOMAINVARIABLE] = VALUE. The notation is equivalent to RELATION(DOMAINVARIABLE, VALUE) but is required by Datalog, which throws an error if a computation yields multiple values for the same domain variable [SBL11].

Domain:

T	set of class types
D	set of module descriptors
M	set of method identifier
H	set of heap abstractions (e.g., allocation sites)
F	set of fields
V	set of program variables
HC	set of heap contexts

Input Relations:

METHOD:DECLARINGTYPE[*method* : *M*] = *type* : *T*

METHOD:MODIFIER(*modifier* : *String*, *method* : *M*)

RETURNVAR(*var* : *V*, *method* : *M*)

CLASSMODIFIER(*modifier* : *String*, *class* : *T*)

SUPERTYPEOF(*supertype* : *T*, *type* : *T*)

OVERRIDESMETHOD(*method* : *M*, *type* : *T*, *supertype* : *T*)

IMPLEMENTSINTERFACE(*method* : *M*, *type* : *T*, *supertype* : *T*)

MODULEDECLTYPE(*module* : *D*, *type* : *T*)

MODULEEXPORTS(*fromModule* : *D*, *package* : *String*, *toModule* : *D*)

EXPORTEDTYPE(*type* : *T*)

MODULEFORANALYSIS(*module* : *D*)

VARPOINTSTO(*var* : *V*, *ctx* : *C*, *heap* : *H*, *hctx* : *HC*)

INSTANCEFIELDPOINTSTO(*base* : *H*, *baseCtx* : *HC*, *fld* : *F*, *heap* : *H*, *ctx* : *HC*)

STATICFIELDPOINTSTO(*ctx* : *HC*, *fld* : *F*, *heap* : *H*)

Output Relations:

EXPLICITMETHOD(*class* : *T*, *method* : *M*)

IMPLICITMETHOD(*class* : *T*, *method* : *M*)

ENTRYPOINT(*method* : *M*)

CLASSHASPOSSIBLEENTRYPOINT(*class* : *T*)

Listing 5.3: Module entry-point model: domain, input, and output relations. Doop's [SKB14; SB11] default rules are gray.

Listing 5.3 shows the domain (the different value sets that constitute the space of our computation) of our entry-point model, its input relations, and output relations.

Domain To describe modules, we introduced module descriptors (D) to Doop's domain, representing a module, its name, its exported packages, its internal packages, and its (re-exported) module dependencies.

Input Relations The input relations, shown in Listing 5.3, are logically grouped: relations representing Doop's intermediate representation, modules, and points-to information. The built-in relations represent the code as Datalog facts [SBL11]: RETURNVAR represents a *method's* return variable, METHOD:DECLARINGTYPE represents its declaring type, METHOD:MODIFIER and CLASSMODIFIER represent the modifier of a *method* or *class*, and SUPERTYPEOF represents all *supertypes* of a *type*. The added input relations IMPLEMENTSINTERFACE and OVERRIDES-METHOD represent every *method* of a *type* that implements or overrides a method of an exported *supertype*.

The input relations MODULEDECLTYPE, MODULEEXPORTS, and EXPORTED-TYPE are module-specific: MODULEDECLTYPE represents in which unique *module* a *type* is declared, MODULEEXPORTS specifies which module (*fromModule*) ex-

ports which *package* to which other module (*toModule*), and EXPORTEDTYPE represents every publicly exported *type*. The input relation MODULEFORANALYSIS represents the module for which the entry points should be computed, as an application typically consists of multiple modules.

VARPOINTSTO, INSTANCEFIELDPOINTS, and STATICFIELDPOINTSTO encode points-to information: they link a return variable *var* or a field *fld* to a heap object *heap*.

Output Relations The output relations ENTRYPOINT and CLASSHASPOSSIBLEENTRYPOINT encode the computed entry-point model. The relations EXPLICITMETHOD and IMPLICITMETHOD represent the *explicitly* and *implicitly* reachable methods.

Datalog Rules: Entry-Point Model The Datalog entry-point model is shown in Listing 5.4. The main rules CLASSHASPOSSIBLEENTRYPOINT and ENTRYPOINT (in duplicate) state that the set of entry points constitutes every implicitly and explicitly reachable *method*.

The rule EXPLICITMETHOD states that a *method* is explicitly reachable if it has the modifiers public or static protected, and its declaring *class* is exported.

The rule IMPLICITMETHOD represents all implicit methods; overriding, implementing, or inheriting a supertype's method. To express implicit methods in Datalog, the rule IMPLICITMETHOD is defined twice; once for overridden methods, and once for inherited methods. Note that Datalog executes multiple definitions of the same rule independently and merges the results.

The first definition states that every *method* implementing or overriding a *method* of an exported *supertype* constitutes an implicitly reachable method. We constrain this set by the rules ENTRYPOINT and METHODRETURNSTYPE or FIELDSTYPE; requiring that an object of type *class* actually becomes reachable outside of the module as the result of a return statement (METHODRETURNSTYPE) or becomes reachable through a static or instance field (FIELDSTYPE) of a previously established *entrypoint*.

The second definition of the rule IMPLICITMETHOD states that every *method* declared in an exported *supertype* constitutes an implicitly reachable method too. Similar to the first definition, the rule constraints this to methods and fields of types *retType* that actually become reachable through a previously established *entrypoint*, encoded by METHODRETURNSTYPE(ENTRYPOINT, RETTYPE) and FIELDSTYPE(ENTRYPOINT, RETTYPE). Moreover, we limit implicit methods to those

whose declaring types *retType* are defined in the module itself (MODULEDECLTYPE(_,RETTYPE), or whose declaring types are directly returned from another entry point of the module, encoded by the conjunction METHOD:DECLARINGTYPE and MODULEDECLTYPE.

Since both rules EXPLICITMETHOD and IMPLICITMETHOD recursively refer to the main rule ENTRYPOINT, they are repeatedly applied to newly discovered entry points, until no further entry points are found. Thus, both rules compute entry points that become transitively reachable, e.g., if an internal type's method grants access to other internal types.

The rule METHODRETURNSTYPE determines the possible concrete runtime types of objects flowing into method returns. The rule states that a *method* grants access to a *type* if the return variable *var* points-to a heap object *heap* of the *type*.

Similarly, the rule FIELDSTYPE determines the possible concrete runtime types of instance and static fields of objects that become reachable through an established entry point. The rule states that a *method* grants access to a type (*fieldType*) if the returned object's (*var*) instance fields points-to a heap object *value* of type *fieldType*, the second definition is defined analog for static fields.

```
CLASSHASPOSSIBLEENTRYPOINT(class),
ENTRYPOINT(method) ←
  EXPLICITMETHOD(class, method).
CLASSHASPOSSIBLEENTRYPOINT(class),
ENTRYPOINT(method) ←
  IMPLICITMETHOD(class, method).
// identify explicitly reachable methods
EXPLICITMETHOD(class, method) ←
  METHOD:DECLARINGTYPE][method] = class,
  CLASSMODIFIER("public", class),
  MODULEFORANALYSIS(module),
  MODULEDECLTYPE(module, class),
  EXPORTEDTYPE(class),
  (METHOD:MODIFIER("public", method);
  (METHOD:MODIFIER("protected", method), METHOD:MODIFIER("static", method))).
```

```
// identify implicitly reachable methods recursively
ImplicitMethod(class, method) ←
  Method:DeclaringType][method] = class,
  ModuleForAnalysis(module),
  ModuleDeclType(module, class),
  Method:Modifier("public", method),
  (OverridesMethod(method, class, supertype) ;
  ImplementsInterface(method, class, supertype)),
  ExportedType(supertype),
  EntryPoint(entrypoint),
  (MethodReturnsType(entrypoint, class) ; FieldReturnsType(entrypoint, class)).
ImplicitMethod(supertype, method) ←
  SupertypeOf(supertype, retType),
  ExportedType(supertype),
  Method:DeclaringType[method] = supertype,
  Method:Modifier("public", method),
  ClassModifier("public", supertype),
  EntryPoint(entrypoint),
  ModuleForAnalysis(module),
  (ModuleDeclType(module, retType);
  (Method:DeclaringType[entrypoint] = classOfEntryPoint,
  ModuleDeclType(module, classOfEntryPoint)),
  (MethodReturnsType(entrypoint, class) ; FieldReturnsType(entrypoint, class)).
// identify return values and types of methods
MethodReturnsType(method, type) ←
  ReturnVar(var, method),
  VarPointsTo(_, heap, _, var),
  Value:Type[heap] = type.
// identify instance fields and types of returned objects
FieldsType(method, fieldType) ←
  ReturnVar(var, method),
  VarPointsTo(_, heap, _, var),
  Value:Type[heap] = retType,
  ExportedType(?retType),
  Field:DeclaringType[field] = type,
  Field:Modifier("public", field),
  !Field:Modifier("static", field),
  InstanceFieldPointsTo(_, value, field, heap, _),
  Field:Type[value] = fieldType.
```

```
// identify static fields and types
FieldsType(method, fieldType) ←
  ExportedType(?type),
  Field:DeclaringType[field] = type,
  Field:Modifier("public", field),
  Field:Modifier("static", field),
  StaticFieldPointsTo(_, value, field),
  Field:Type[value] = fieldType.
```

Listing 5.4: Datalog rules for detecting explicit and implicit entry points of a module, constituting the module's API.

The formal definition of module's entry points serves as a basis for our static analysis tool ModGuard to analyze to what extent a module confines internal types and data by computing all potential data flows between modules in Section 5.3—that are the methods a potentially vulnerable OSS module can invoke on the module. To do so, ModGuard checks for all user-defined sensitive classes, methods, and fields if they become reachable outside the module or can be manipulated through invocations of the module's entry points.

5.2.3 Limitations of the Entry-Point Model

Our entry-point model is based on the following design decisions. First, the entry points are sound w.r.t. explicitly reachable methods: all API methods that a module explicitly exports are captured in the model. Second, the computation of the entry points should be scalable. Thus, the entry-point model may only contain a subset of *all* implicitly reachable methods.

To be sound w.r.t. explicit methods (the explicitly declared API), our entry-point model computes all reachable methods of the exported types and supertypes. To be scalable, the entry-point model only computes implicit methods of module-internal types that become reachable *directly* through an exported type. Thus, the entry-point model captures the implicit methods of types that become reachable outside the module only if they are returned from an accessible method or are referenced by (static-) fields of an exported class.

Limiting the entry-point model to implicit methods of objects that are (directly) returned from the module bounds the set to a reasonable size. Without these constraints, the size of the entry point set would explode quickly, for instance, consider that the class `Object` is a supertype of every class, and thus all its public methods would be entry points too. Consequently, all types returned by these entry points

would also be an entry point, e.g., the method `getClass()`, which returns an instance of type `Class`, which in turn declares further public methods would be entry points also, and so on. Our static analysis ModGuard refines this computation of further entry points w.r.t. its goal of detecting what internal types and data become eventually accessible from the outside.

5.3 ModGuard: Identify Confidentiality or Integrity Violations of Modules

To complement the module system with means to identify and analyze unintended data flows, we designed and implemented ModGuard, a novel static analysis for Java modules that automatically identifies instances, fields, or methods that can become accessible outside their declaring module, and thus undermine the confinement of sensitive types or data. ModGuard enables developers to leverage the module system security-wise by identifying unintended data flows that the modularization should prevent. We call such unintended data flows that make an internal type or data accessible or modifiable to the outside *escapes*.

To restrict inadvertent vulnerabilities in one module, e.g., an included OSS, from affecting others, security-sensitive entities, e.g., methods or encryption keys, should not be accessible outside their declaring module but should be well isolated [Ora21; Ora22; SS75]. To detect if sensitive entities can become reachable outside their module (confidentiality violation) or if sensitive entities can be manipulated (integrity violation), an analysis must foresee all ways in which other modules can invoke the module's functionality. Implementing a static analysis on individual modules is challenging, as the analysis must be conducted on open code, much alike call-graph construction for libraries [Rei+16]. Using the former defined entry-point model, ModGuard precisely models those possible invocations and data flow using the static analysis frameworks Doop [SB11] and Soot [Lam+11]. The analysis and our extensions to Doop and Soot are independent of the particular points-to analysis. They can be used on top of any other points-to analyses Doop offers.

5.3.1 Algorithm

Figure 5.3 gives an overview of ModGuard's analysis steps.

To detect unintended data flows, like in the motivating example in Listing 5.2, that lead to the escaping of sensitive types and data, ModGuard requires the list of sensitive entities as user input either in the form of our Java annotation `@Critical` or as a separate text file, since the information which types and data are sensitive is domain-specific and depends on the implementation. The input relations SENSITIVEMETHOD, and SENSITIVECLASS in Listing 5.5 represent the user input.

Sensitive information is often stored using primitive types or arrays, e.g., salts and secret keys are stored in `byte` arrays, while passwords are stored in `char` arrays [Krü+21]. To track such primitive-typed data in addition to regular pointers, we use P/Taint [GS17], an extension to Doop that augments its points-to analysis with additional rules for a context-sensitive, flow-insensitive propagation of primitive-typed data.

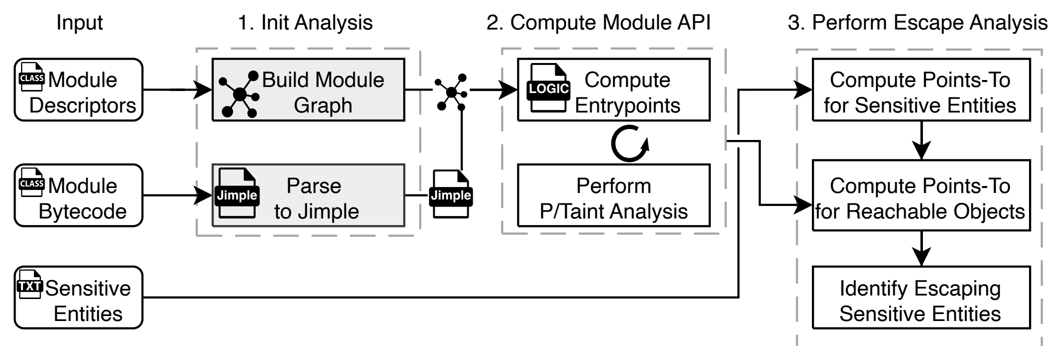


Figure 5.3: ModGuard's process steps for identifying escapes of sensitive entities using the Doop framework and Datalog rules. Gray steps are executed in Soot and white steps are executed in Doop.

Input To compute the module's entry points, ModGuard first reads the module descriptors and bytecode. ModGuard uses Soot to transform the bytecode from all modules into the Jimple intermediate representation [Lam+11], on which Doop operates.

1. Building the Module Graph Next, ModGuard constructs the module graph. The construction of the complete module graph is necessary to detect exported supertypes declared in other modules that the module under analysis implements. We adapted Soot to parse the descriptor of the module and its (transitive) module

dependencies using `requires`, `exports`, and `opens` declarations. Furthermore, ModGuard marks primitive-typed sensitive entities as tainted for Doop's P/Taint analysis [GS17].

2. Precise Modeling of Module's Entry Points In contrast to a Java application, modules do not have a single entry point (e.g., a main method). Instead, modules compromise many entry points—the *explicitly* and *implicitly*—reachable methods, as described in Section 5.2. ModGuard computes the set of entry points based on the Datalog rules described in Listing 5.4. As our entry-point model only captures *implicit* entry points that are directly reachable, ModGuard's data flow analysis completes the entry-point set. To do so, ModGuard identifies objects that escape transitively through static or instance fields of reachable objects, arguments passed into the module, and arguments passed to callback methods outside the module. To compute points-to information for the arguments of implicit and explicit entry points, ModGuard models those arguments as mock objects for Doop's points-to analysis, P/Taint Analysis in Figure 5.3. These mock objects simulate the allocation side of an entry point's arguments by code outside the module.

To identify data flows that allow code outside the module to access or modify sensitive types or data, ModGuard checks if the points-to sets of sensitive entities and the points-to sets of reachable objects intersect. If the two points-to sets intersect, the elements in the intersection are (i) the sensitive entities that escape the module, marking a *confidentiality violation*; or (ii) objects that have been passed into the module and have been assigned to a sensitive entity, marking an *integrity violation*. The input relations in Listing 5.5 represent the user-defined sensitive entities.

3. Escape Analysis To identify whether the values of sensitive fields escape or can be manipulated, ModGuard implements an escape analysis specific to modules.

Domain Listing 5.5 shows the domain, input, and output relations of ModGuard's module escape analysis. The queries for identifying if sensitive fields, methods, and classes escape a module are defined in Listing 5.6.

Input relations The input relations, shown in Listing 5.5, are logically grouped and represent the sensitive entities, the points-to information of sensitive entities, and the points-to information of reachable objects. The rules SENSITIVEFIELD, SENSITIVEMETHOD, and SENSITIVECLASS represent the user-defined sensitive entities. The input relation SENSITIVEFIELD represents the fields that should not

be leaked or manipulated from outside the module. The relation SensitiveField-PointsTo represents the points-to information of these sensitive fields. The rules SensitiveMethod and SensitiveClass are defined analog.

The remaining input relations represent the points-to sets of objects that become reachable outside the module. VisibleFieldPointsTo represents the points-to set of all fields that are either public or protected and reside in classes the module exports. The relation ReturnVarPointsTo represents the points-to set of the return variables of all reachable implicit and explicit methods. Similarly, the relations MockObjectPointsTo and ArgumentOfCallbackPointsTo represent the points-to set of receiver mock-objects on which an entry-point method is invoked, of mock-objects that are passed as arguments into the module, and of arguments that are passed to callback methods outside the module.

Output Relations The output relations EscapingField, EscapingMethod, and EscapingClass represent an escape of the corresponding sensitive entity.

Domain:

T	set of class types
D	set of module descriptors
M	set of method identifier
H	set of heap abstractions (e.g., allocation sites)
F	set of fields
V	set of program variables
HC	set of heap contexts

Input Relations:

SensitiveField(*field : F*)
SensitiveMethod(*method : M*)
SensitiveClass(*class : T*)

SensitiveFieldPointsTo(*f: field, heap : H, ctx : HC*)
VisibleFieldPointsTo(*f : field, heap : H, ctx : HC*)
ReturnVarPointsTo(*var : V, type : T, heap : H, ctx : HC*)
MockObjectPointsTo(*type : T, heap : H*)
ArgumentOfCallbackPointsTo(*type : T, heap : H*)

Output Relations:

EscapingField(*f : field*)
EscapingMethod(*type : T, m : M*)
EscapingClass(*sensitiveType : T, escapingClass : T*)

Listing 5.5: ModGuard Escape Analysis: domain, input, and output relations.

Datalog Rules of the Escape Analysis Listing 5.6 shows the escape analysis using Datalog rules. For identifying escaping fields, methods, and classes, ModGuard intersects the points-to set of sensitive fields and the sensitive entities' types with the

input relations that represent the points-to set of reachable objects: the points-to set of visible fields, return variables, and mock objects in the queries EscapingField, EscapingMethod, and EscapingClass. If a query results in a non-empty intersection, ModGuard reports an escape.

Each rule is defined three times in Listing 5.6: first for checking if a sensitive entity escapes over accessible fields (SensitiveFieldPointsTo), second to check if it escapes as a return value of a method (ReturnVarPointsTo), and third to check if it escapes over a mock object (MockObjectPointsTo). We also defined the rules to check if escapes occur through exported supertypes of fields and return values. Since the rules are defined analog, we omitted them in Listing 5.6.

```
//check for escaping fields
EscapingField(field, fieldValue)←
  SensitiveFieldPointsTo(field, fieldValue, ctx),
  VisibleFieldPointsTo(field, fieldValue, ctx).
EscapingField(field, fieldValue) ←
  SensitiveFieldPointsTo(field, fieldValue, ctx),
  ReturnVarPointsTo(var,_, fieldValue, ctx ).
EscapingField(field, fieldValue) ←
  SensitiveFieldPointsTo(field, fieldValue, _),
  MockObjectPointsTo(_, fieldValue).
//check for escaping methods
EscapingMethod(type, method)←
  SensitiveMethod(method),
  VisibleFieldPointsTo(field, fieldValue, ctx),
  Field:Type[fieldValue] = fieldType,
  Method:DeclaringType[fieldValue] = fieldType,
  MethodInvokable(method).
EscapingMethod(type, method)←
  SensitiveMethod(method),
  ReturnVarPointsTo(var, type, _, ctx ),
  Method:DeclaringType[method] = type,
  MethodInvokable(method).
EscapingMethod(type, method)←
  SensitiveMethod(method),
  MockObjectPointsTo(type, value),
  Method:DeclaringType[method] = type,
  MethodInvokable(method).
```

```
//check for escaping classes
EscapingClass(sensitiveType, escapingType)←
  SensitiveClass(sensitiveType),
  VisibleFieldPointsTo(field, fieldValue, ctx),
  Field:Type[fieldValue] = fieldType,
  SupertypeOf(escapingType, sensitiveType).
EscapingClass(sensitiveType, escapingType)←
  SensitiveClass(sensitiveType),
  ReturnVarPointsTo(var, escapingType, _, ctx ),
  SupertypeOf(escapingType, sensitiveType)
EscapingClass(sensitiveType, escapingType)←
  SensitiveClass(sensitiveType),
  MockObjectPointsTo(escapingType, value),
  SupertypeOf(escapingType, sensitiveType)
```

Listing 5.6: ModGuard's Datalog rules for detecting escaping fields, methods, and classes.

Analysis of the Motivating Example For the example in Listing 5.2, ModGuard's analysis works as follows: First, ModGuard computes the entry-point model for the module. As defined by the rules in Section 5.2, the model contains the explicit method `KeyProvider.getKey()`. To represent the return value of this method, ModGuard creates a mock object for the internal type `SecretKey`, representing the returned object `new SecrectKey()`. Second, ModGuard's entry-point definition detects, based on the entry-point rule ImplicitMethod, that the returned object's type `SecretKey` constitutes the implicit entry point `getKey()`, declared in the public exported supertype `Key`. Third, ModGuard binds the created mock object `new SecrectKey()` to the `this` parameter of the implicit method `getKey()` and creates an additional mock object representing the returned key `key`, which aliases with the sensitive field `keyMaterial`. Finally, ModGuard checks if the points-to sets of the private field `keyMaterial` and the points-to set of the mock objects intersect in the rule EscapingField. Since ModGuard bound the `this` parameter of the method `getKey()` to the returned mock-object `new SecretKey()`, the points-to set of the returned object and sensitive field intersect, and ModGuard successfully detects that the field escapes the module.

5.3.2 Limitations

ModGuard shares some inherent limitations with other static analyses. For instance, ModGuard does not identify violations resulting from using `MethodHandles.Lookup`, `MethodHandle`, or `VarHandle` from Java's dynamic-language API. Although Mod-

Guard can detect exposed `MethodHandles.Lookup`, `MethodHandle`, and `VarHandle` instances, simply reporting their escapes results in false positives in real-world applications. While Java's `Handles` allows, in principle, inspecting internal classes, methods, and fields the returned Handle instance may only grant access to exported types. Thus, an exposed instance of a `Handle` does not necessarily indicate a violation. Handling precisely Java's invoke API requires the inspection of a `Handle` using typestate analyses to identify to which entities it grants access. Consequently, ModGuard currently under-approximates w.r.t. Java's dynamic-language API leading to false negatives. To deal with reflection, we use Doop's extensions to resolve reflective calls and compute points-to information in conjunction with our rules and queries [Sma+15].

Additionally, ModGuard fails to detect violations in native code, resulting in false negatives, e.g., if sensitive entities are written or read from files, environment variables, network sockets, or passed to native code. Detecting violations in arbitrary native methods requires analyses for languages such as C and C++ or native binaries, which is out of scope. However, for commonly used native methods, like `System.arraycopy()`, ModGuard's analysis can be supplemented with hand-crafted summaries (Datalog facts) that specify the impact of native methods on data flows, as proposed by Sălcianu and Rinard [SR05].

5.4 Evaluation

ModGuard provides means for developers to benefit from Java's module system by detecting unintended data flows that violate the confidentiality or integrity of a module. In the following, we evaluate ModGuard's effectiveness for detecting escaping sensitive entities and ModGuard's use on applications within a case study.

5.4.1 Research Questions

Our evaluation is twofold. In the first part, we evaluate **RQ1: How effectively can our API entry-point model and ModGuard detect escaping internal types and data?** We analyze on a benchmark for Java modules how effectively and in what cases our entry-point model captures (unintended) data flows violating the confinement of sensitive entities in a given module.

In the second part, we analyze the consequences of strong encapsulation for Java applications by investigating **RQ2: If Java applications benefit from migrating to modules w.r.t. escapes and if ModGuard can properly identify those?** Since ModGuard aims to support developers in structuring their application to potentially restrict vulnerabilities in an OSS module from affecting others by limiting the data the vulnerable module can access, we conduct a case study on the widespread Java web server Tomcat in which we analyze two modularization options. Finally, we discuss to what extent modules can limit the impact of inadvertent vulnerabilities in modules w.r.t. the vulnerability CVE-2017-5648, which was reported for Tomcat in 2017.

5.4.2 Study Objects

MIC9Bench: A test suite for Java modules While benchmark suites exist for detecting Java coding vulnerabilities [WR99], e.g., SQL injection, path traversal, cross-site scripting, or evaluating the precision and soundness of points-to analyses [Spä+16], there currently exists no benchmark for Java modules. Existing generic benchmark suites for Java cannot be used to assess the effectiveness of module analyses (*RQ1*), as they comprise no module declarations nor any ground truth w.r.t. the types and data the module should confine.

Specifically for Java module analyses, we developed the test suite MIC9Bench (**m**odule, **i**ntegrity, **c**onfidentiality for Java **9**) modules: integrity is violated if code outside of a module can change the value of a sensitive field, confidentiality is violated if code outside the module can access an internal type, method, or the value of an internal field.

The test suite contains 22 small hand-crafted Java modules in which a sensitive internal type or data can be accessed or modified by code outside of the module through the module's entry points, shown in Figure 5.4.

Sensitive Entities in Fields As Figure 5.4 shows, sensitive entities referenced by API *fields* can be accessed or modified from external code. The value of these fields is directly accessible if the declaring type is exported and the field is public or protected.

Sensitive Entities in Method Returns A sensitive entity can be returned through a chain of method invocations.

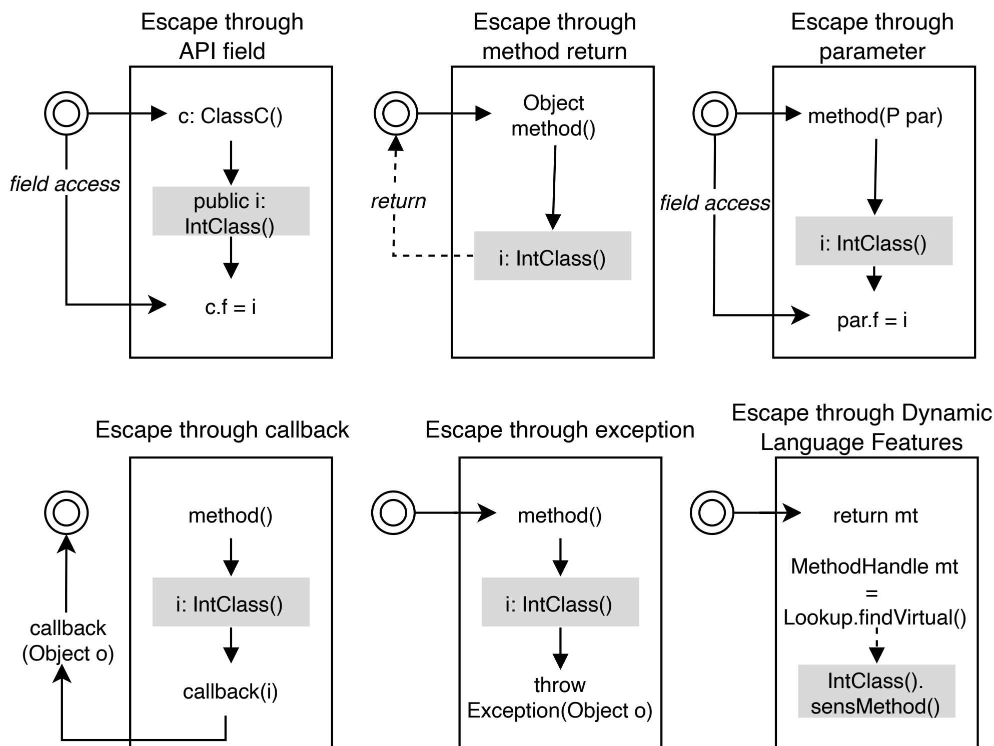


Figure 5.4: Scenarios leading to the escape of sensitive entities.

Sensitive Entities in Arguments Sensitive entities can be disclosed through arguments that are passed as *parameters* into the module, i.e., if sensitive entities are assigned to a parameter or added to a collection, accessible to code outside the model. Similarly, the argument of a method can be assigned to a sensitive field, changing its value.

Sensitive Entities in Callbacks Similarly, sensitive entities can escape as an argument of a callback method outside the module.

Sensitive Entities in Exceptions Sensitive entities can escape through exceptions bypassing the normal control flow.

Sensitive Entities in Java Dynamic Features Sensitive entities can escape if they are referenced by Java's dynamic language constructs, which are typed and executable references and grant access to the underlying entities (types, methods, fields). Java's `Method`-, `VarHandle`, or `MethodHandles.Lookup` are typed, directly executable references to any method, constructor, or field [Ora15a]. Since access checks for `Handle` instances are made at creation-time rather than runtime, code outside the module can invoke any operation freely on an escaping `Handle` object circumventing the module systems access checks.

Sensitive Entities in Side Effects and Native Code Additionally, sensitive entities can also escape through *side effects* occurring in native code, e.g., modification of the JVM memory or reading/writing sensitive entities to the file system.

MIC9Bench also comprises test cases for the above scenarios for collections and arrays. To evaluate which escapes ModGuard correctly detects, we execute it on the test suite.

Organization and Distribution To allow the extension of MIC9Bench, we make MIC9Bench publicly available on GitHub (cf. Chapter 6).

The organization of MIC9Bench is shown in Figure 5.5. Each test case is located in a separate top-level folder. The test cases are further separated into scenarios, e.g., escapes through a getter method or public field.

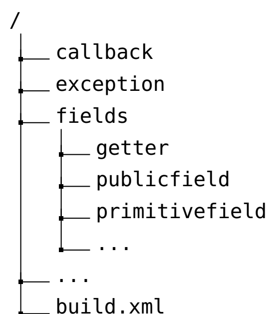


Figure 5.5: Layout of MIC9Bench

Case Study: Tomcat Web Server To evaluate the effectiveness of ModGuard on real-world applications and to check if applications suffer from escaping sensitive entities (*RQ2*), we executed ModGuard on an established Java application with a reasonably sized codebase. However, applications leveraging Java modules are still scarce. In fact, we searched Maven Central for OSS declaring modules and module-internal packages but could not find any artifact in December 2021. Only 215 different artifacts on Maven Central, at least, contain a module descriptor (`module-info.class`), and only 83 of those 215 include their own `module-info.java` file in their source-jar. The 83 artifacts and module descriptors export all packages publicly, rendering the analysis pointless. The remaining artifacts only include a copy of the `module-info.class` of a beta version of the library `org.slf4j`, which was included in the artifact during re-bundling.

Thus, we choose to follow the approach of Corwin et al. [Cor+03] for modularizing existing Java applications. In their work, Corwin et al. [Cor+03] propose a new module system for the Java platform and present a structured approach for modularizing the Java web server Tomcat to modules. We apply the presented approach for our evaluation and use the well-established, widespread OSS Apache Tomcat 8.5.21 as a case study. Following the approach, we created a separate module for each of Tomcat's 26 JAR files, maintaining the original grouping of classes into logical units.

For creating modules and deciding what packages are module-internal and what packages a module exports, we applied two different modularizations to the 26 Tomcat modules: a *naïve modularization*, all modules export all packages publicly, and a *strict modularization*, each module only exports packages that are necessary for compilation. To create the naïve modularization, we used the tool jdeps [Ora17a]. Jdeps is the Java dependency analysis tool that statically analyzes declared dependencies between classes and aggregates them at the package or JAR level. With Java 9, jdeps has been extended to help developers to migrate to the new module system by automatically generating module descriptors with all packages publicly exported. To create the strict modularization, we restricted the naïve modularization as follows. We removed each package export statement for each module and checked if Tomcat still compiles; we continued for all packages and modules until the compilation failed, retaining exactly those package exports that Tomcat requires to compile.

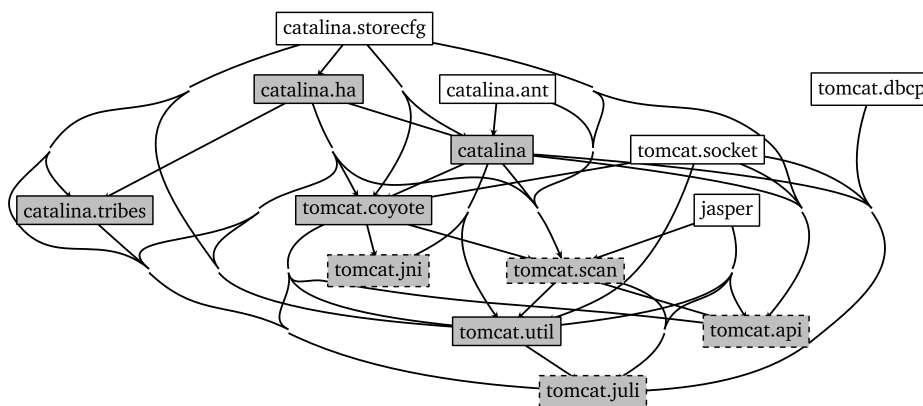


Figure 5.6: For modularizing Tomcat 8.5.21, we created a module per JAR. The figure shows the resulting module graph. In the strict modularization (all exports not required for compilation removed), the white modules do not export any packages. The dashed modules have the same exports in the naïve and strict modularization.

The resulting module graph is shown in Figure 5.6. In the strict modularization, the modules on top of the graph (in white) `catalina.ant`, `catalina.storeconfig`, `jasper`, `tomcat.dbcp`, and `tomcat.websocket` do not export any packages. Since the modules on top of the module graph are not required by any other other module, all their export statements were removed. Note that the Tomcat web server does not contain any client code using the modules on top of the module graph, e.g., a web application. The modules marked with a dashed line `tomcat.api`, `tomcat.jni`, `tomcat.juli`, and `tomcat.util.scan` are identical in the naïve and strict modularization. For the remaining modules (marked in gray), we could reduce the exported packages to the minimum necessary to compile. Note that the strict modularization is more restrictive than a regular one, as web applications running on Tomcat need access to the modules on top of the graph. However, by applying the strict modularization, we do not risk exposing internal types unnecessarily, and as we aim to investigate the effect of modules w.r.t. confining sensitive entities, a more strict modularization only provides a lower bound for those modules.

To check if sensitive entities escape a module, ModGuard requires the sensitive methods and fields as input. To determine the sensitive methods, we applied the framework SuSi [RAB14] on Tomcat's codebase. SuSi [RAB14] is an automated machine-learning tool for identifying sources and sinks in Java and Android binaries from code using syntactic features, e.g., method, class, and parameter names, and semantic features, e.g., data flow to return statements. SuSi has been successfully applied and extended to detect security-sensitive entities in Java applications, particularly the spring-framework [Pis22]. Thus, we consider it well-suited for detecting sensitive entities in Tomcat for our evaluation. We used security-sensitive methods of the JDK 9 as a training set. In particular, we trained SuSi on methods that throw `SecurityExceptions`, e.g., methods accessing or modifying class loaders or the file system. The functionality implemented by these methods is guarded by permissions checks, triggering Java's `SecurityManager` to check whether all classes on the call stack possess the required permissions. As a result, their set of calling classes is restricted, and these methods should not be accessible to any code, analog to security-sensitive methods in modules. We used these methods as the training set for SuSi to identify similar security-sensitive methods in Tomcat to which access should be restricted too.

Based on this training set, SuSi reported 3,300 sensitive methods in 12 Tomcat modules. We further manually added 90 classes and 25 fields as sensitive entities whose JavaDoc comments state that they are "internal". Note that access to these sensitive entities *does not* necessarily imply the existence of security vulnerabilities, yet it indicates privileged classes, fields, and methods whose caller may be limited.

5.4.3 Results

RQ1: How Effectively can ModGuard Detect Escapes?

To evaluate the effectiveness of ModGuard in different scenarios, we executed it on the benchmark MIC9Bench and evaluated the results, shown in Table 5.1. The table shows that ModGuard successfully detects confidentiality and integrity violations in 18 of 25 test cases.

Table 5.1: Evaluation of ModGuard on MIC9Bench for detecting escaping sensitive entities over a module's API and different Java language features. Integrity violations occur if external code can change the value of a sensitive field, and confidentiality violations occur if external code can access a sensitive entity.

Entity Escapes through	Scenario	Detected by ModGuard
Accessible Field	Integrity/Confidentiality primitive field	✓
	Integrity/Confidentiality non-primitive field	✓
	Integrity/Confidentiality field array, collection	✓
	Getter/Setter for field	✓
Invokable Method	Access to explicit method	✓
	Access to implicit interface/ abstract method	✓
Parameter	Entity added to parameter array, collection	✓
	Static method returns internal field	✓
Callback	Entity/Class referencing Entity as argument	✓
Exception	Declared/Undeclared exception	✓
Reflection & Invoke API	Referenced by `VarHandle`/ `MethodHandle`	✗
	Access to privileged `MethodHandles.Lookup`	✗
	Return field via reflection	✓
Side Effect	Pass entity to native code	✗

✓ true positives
✗ false negatives
no false positives observed

The table shows that ModGuard detects if sensitive entities leak through API fields. Similarly, ModGuard detects if sensitive entities leak or can be manipulated through API methods and if sensitive methods are eventually reachable from outside the module, e.g., if packages or supertypes are erroneously exported. ModGuard also

detects if sensitive fields can be modified through method arguments and if sensitive entities escape through arguments of exported methods, callback methods, or exceptions. Further, the table shows that ModGuard detects if an exposed collection or array discloses sensitive entities. To check arrays and collections, ModGuard uses Doop's partially 1-object-sensitive points-to analysis context-insensitive++. Doop's object-sensitive analysis distinguishes instances of the same collection, and thus can detect if an exposed instance contains sensitive entities. Omitting object-sensitivity would lead to false positives, since entities added to one particular collection would appear to be retrievable by any other instance of the collection [MRR05].

The evaluation also shows that ModGuard fails to detect if a module exposes references to internal types, methods, and fields in the form of `Method-`, `VarHandle`, or `MethodHandles.Lookup` instances. Equally, ModGuard fails to detect if sensitive instances escape through side effects or native code. As discussed in detail in Section 5.3.2, dealing with side effects and native code is still an open issue for static code analysis.

Computation Resources For each test case, ModGuard runs 22.4 minutes, and consumes 3.4 GB RAM, including repeated analysis of the full JDK on an Intel i7 2.60 GHz per Mic9Bench module. If ModGuard does not use Doop's reflection handling, its runtime is reduced to 7.4 minutes and the resource consumption to 3.1 GB RAM.

> **Findings from RQ1:** ModGuard effectively identifies confidentiality and integrity violations of sensitive entities in a module, unless Java's dynamic-language API is used.

RQ2: Case Study - Can Java Applications Benefit from Migrating to Modules?

To check to what extent Java applications suffer from escaping sensitive entities, we executed ModGuard on the naïve and strict modularization of Tomcat. Table 5.2 shows the analysis results.

Even in the strict modularization, thousands of sensitive entities escape the modules, allowing code from other modules to access them (confidentiality violation) or to modify them (integrity violation). The table shows that even in the strict modularization, the number of violations is only slightly reduced or is unchanged compared to the naïve one.

In the results, the escapes reported for one module intersect with the reported violations of other modules. Since Tomcat's modules cross-reference types between each other, a sensitive entity might escape through module A and also through modules B and C. For instance, the escaping of a sensitive method of an exported supertype in one module is reported for all dependent modules that declare escaping subtypes. Thus, removing the export of such supertype also reduces escapes in dependent modules. In fact, the majority of escapes occur due to sensitive types that are declared in exported packages whose export cannot be removed without compilation errors.

Table 5.2: The table shows the number of escaping of sensitive entities that ModGuard identifies in the modularization of Tomcat 8.5.21.

Tomcat Module	#Violations strict / Δ naïve modularization			Runtime in min
	Methods	Fields	Classes	
catalina	2,556 / −236	8 / −21	31 / −17	11:13
catalina.ant ⊖	0 / –	0 / –	0 / −7	00:43
catalina.ha	1,081 / −391	3 / −1	15 / –	03:48
catalina.storeconfig ⊖	0 / −79	0 / –	0 / –	01:49
catalina.tribes	0 / –	3 / –	4 / −1	01:13
jasper ⊖	0 / −6	0 / –	0 / –	01:30
tomcat.coyote	2,020 / −294	0 / −3	20 / −14	01:25
tomcat.dbcp ⊖	0 / –	0 / −3	0 / –	00:44
tomcat.jni ⊙	1 / –	2 / –	0 / –	00:38
tomcat.util	78 / –	0 / –	1 / –	00:39
tomcat.util.scan ⊙	449 / –	0 / –	0 / –	00:44
tomcat.websocket ⊖	0 / –	0 / –	0 / −6	00:46

⊖ The modules do not export any package

⊙ The modules export the same packages in the strict and naïve modularization

Table 5.2 shows that in the strict modularization, no violations occur in the 5 modules: `catalina.ant`, `catalina.storeconfig`, `jasper`, `tomcat.dbcp`, and `tomcat.websocket`. Since these modules are at the top of the module graph (cf. Figure 5.6) all their export packages are removed, and thus no data flows exist over which sensitive entities can escape. The number of violations is unchanged in the 3 modules: `tomcat.api`, `tomcat.jni`, and `tomcat.util.scan` since their export packages

are identical in the naïve and strict modularization. The violations for the module `tomcat.util` are unchanged since the removed package exports do not contain any sensitive entity.

Only in the modules `catalina`, `catalina.ha`, `catalina.tribes` and `tomcat.coyote` fewer violations occur in the strict modularization, excluding modules whose exports are removed completely or unchanged.

Although 18 export statements out of 30 are removed in the module `catalina`, the number of violations is only reduced to a small extent. The removal of public exported packages (for the modules in gray in Figure 5.6) only has a small impact on the number of found escapes. For instance, the package `org.apache.catalina.webresources` contains 140 internal, sensitive methods, and its export was removed in the strict modularization. However, 32 sensitive methods that are declared in `WebResource` are still accessible. `WebResource` is contained in the package `org.apache.catalina` whose exports cannot be removed without compilation errors. Also, the module `catalina.ha` does not benefit much from the stricter encapsulation of internal types; 70% of its sensitive methods still escape and can be invoked from code outside the module.

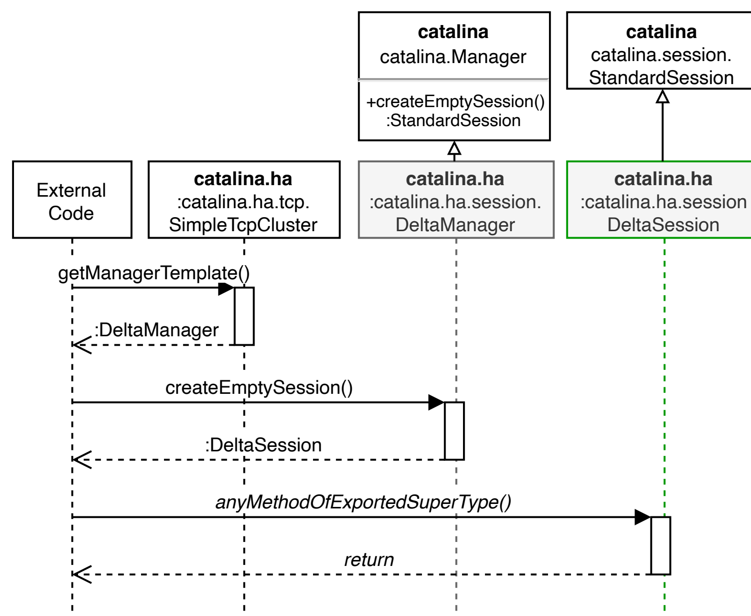


Figure 5.7: Example escape of the sensitive methods of `DeltaSession` from the module `catalina.ha` in the strict modularization of Tomcat. External code can invoke `SimpleTcpCluster.getManagerTemplate()` to receive an instance of `DeltaManager`, which leaks the sensitive `DeltaSession` via the method `createEmptySession()`. The gray types are module-internal, the white types are exported; in bold is the module declaring the type.

Example Escape in Tomcat Module catalina.ha An instance of a data flow over which the sensitive methods of class `DeltaSession` in module `catalina` escape is depicted as a sequence diagram in Figure 5.7. The escape occurs in the strict modularization, as follows: First, code outside the module acquires an instance of `DeltaManager`, returned from the public method `getManagerTemplate()`. Second, code outside the module invokes the implicit method `createEmptySession()` on the acquired instance, which is overridden in `DeltaManager` from its exported supertype `Manager`. Third, the method `createEmptySession()` returns an instance of `DeltaSession` to the outside. Fourth, on the such acquired instance of `DeltaSession` external code can invoke all overridden sensitive methods of the supertype `StandardSession` (exported by the module `catalina`). This exposes the internal, sensitive methods of `DeltaSession` in `catalina.ha` to the outside.

Findings from RQ2: Escaping of sensitive entities over modules' API is indeed a problem when modularizing real-world applications. ModGuard identifies data flows leading to escapes, and thus helps developers to assess how successfully a module confines sensitive entities.

In result, Table 5.2 shows that neither the naïve nor the strict modularization of Tomcat effectively limits that sensitive entities escape through complex (unintended) data flows. While an automatic migration using jdeps is possible without major effort, the resulting modules do not benefit from the module system security-wise. Although internal types are encapsulated, modularizing w.r.t. forbidden data flows can prevent data leaks, and thus results in a security benefit. To limit violations, it is insufficient to simply limit package exports. Even modules whose exports we were able to reduce (`catalina.ha`, `catalina.tribes`, and `tomcat.coyote`), effectively confine a small subset of sensitive entities only. Instead, applications must be migrated carefully to prevent data flows leading to integrity and confidentiality violations. For this migration, developers should be supported by appropriate tools. Our case study shows that ModGuard can support the migration to Java modules, as well as refactorings, by revealing integrity or confidentiality violations.

5.4.4 Case Study: CVE-2017-5648 in Tomcat (modules)

In 2017, the Tomcat web server suffered from a vulnerability reported as CVE-2017-5648 [NVD17b] in the component `catalina`, shown in Listing 5.7. The vulnerability enables any web application running on the server—even with Java's Security

Manager activated—to retain a reference to a Request object, exposing access to security-sensitive methods, which allow to access and to modify information belonging to other web applications running on the same server. The vulnerability shows that programming mistakes involving the *unintentional* escape of sensitive entities occur in real-world applications and may significantly impact an application's security. Line 9 and Line 12 contain the vulnerable code. The fix is shown in Line 10 and Line 13 [Apa15].

```
1  class Request implements HttpServletRequest{
2    public HttpServletRequest getRequest(){
3      return new RequestFacade(this); }
4  }
5
6  class RequestFacade implements HttpServletRequest {}
7
8  class FormAuthenticator extends AuthenticatorBase {
9  - if(context.fireRequestInitEvent(request)){
10 + if(context.fireRequestInitEvent(request.getRequest())){
11         disp.forward(request.getRequest(), response);
12 - context.fireRequestDestroyEvent(request);
13 + context.fireRequestDestroyEvent(request.getRequest());}}
14 }
```

Listing 5.7: Excerpt of the fix for CVE-2017-5648 in class FormAuthenticator.java in Tomcat revision 1785776 [Apa15].

In the vulnerable version of Tomcat any Request object received by the FormAuthenticator was dispatched directly to the web application via the context object (cf. Line 9, Line 12), which one must assume to be attacker-controlled.

To mitigate the vulnerability, the fixed version confines the Request object in the component catalina. Instead, a RequestFacade is passed to the web application, wrapping the original Request object and denying access to security-sensitive methods.

Although CVE-2017-5648 was contained in a version of Tomcat that did not yet use Java modules, it was caused by an improper confinement of an internal Request object, which malicious code could abuse. Our case study shows that while the module system can encapsulate the internal Request type, its encapsulation is too weak to prevent such vulnerabilities since it only considers types: both classes Request and RequestFacade implement the interface ServletRequest. Crucially, this interface *has to be exported* to be usable by web applications. Thus, any code outside the component catalina may invoke any method on any object of type RequestFacade,

but also on `Request`, as long as the method is defined in the interface `ServletRequest`. Unfortunately, `ServletRequest` defines several such methods, e.g., `getParameter`, `getLocale`, etc., and because `ServletRequest` is exported, attackers can invoke those methods on the unprotected `Request` object *even if the type `Request` is declared as internal*.
Vulnerabilities of this kind cannot be remedied solely by relying on the encapsulation of types, but also require reasoning about escaping instances, modules' API, and the data flow between modules.

CVE-2017-5648 shows that ModGuard complements the module system with an analysis that detects if security-sensitive objects like `Request` escape. This is generally non-trivial, however, as ModGuard must reason about pointers while being aware of a module's API and boundaries in the absence of client code invoking the module. In the example, the `context` object represents the untrusted web application that runs on top of Tomcat, and outside of the component `catalina`. Crucially, ModGuard and its entry-point model detect that values passed to `context.fireRequestInitEvent(..)` in Line 9 and `fireRequestDestroyEvent(..)` in Line 12 in Listing 5.7 is a confidentiality violation: because the objects are passed to a `context` object outside the module.

To detect such escapes, ModGuard takes into consideration that there are two internal subtypes of the exported interface `ServletRequest`: one which is security-sensitive (`Request`) and one which is safe to be used outside the module (`RequestFacade`). To do so, users must provide ModGuard with a list of sensitive entities since the information which of these two classes is sensitive is domain specific.

5.5 Limitations of Modules for the Secure Integration of Open-Source Software

Our case study of Apache Tomcat in Section 5.4 shows that modules can successfully prevent access to security-sensitive types and that ModGuard can identify data flows that leak data or allow their manipulation. The study also found that a secure design of application modules requires significant refactoring to benefit from strong encapsulation. However, the module system is insufficient to restrict a vulnerability in a module from affecting others. Strong encapsulation is only part of a solution—as also stated by the developers of the Java platform [Ora22; Ora23b; Ora21] and the principle of Defense in Depth [NSA12]. In the following, we discuss features and security mechanisms that are not covered by the module system.

Fine-Grained Access Control The module system does not allow to grant only limited permissions to modules similar to the security manager, e.g., disallowing access to specific files or network addresses. At the root of a module graph resides the module `java.base`, which is accessible by every other module and publicly exports classes for file system operations or network access, etc. Thus, vulnerabilities that abuse exported classes of the `java.base` module cannot be restricted by the module system alone.

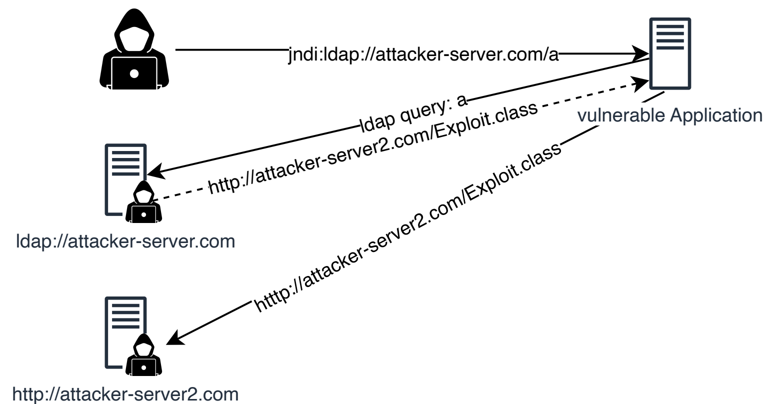


Figure 5.8: Overview on how to exploit the Log4Shell vulnerability. Based on [Jun21].

We illustrate the issue exemplary with the Log4Shell vulnerability, published as CVE-2021-44228 [NVD21]. The Log4Shell vulnerability allowed an attacker to execute arbitrary code on the host system running a vulnerable Java application. An overview of the steps for exploiting the Log4Shell vulnerability is given in Figure 5.8, and an example exploit is given in Listing 5.8. The root cause of the vulnerability is the fact that the Log4j library supported a feature called "lookup" that enabled the replacement of variable-like values with dynamic strings at the time of logging, e.g., the string `$java:version` was evaluated and replaced with the Java version running on the host, e.g., `java version 1.8.0_231` [ECZ22]. Crucially, Log4j supported the use of Java Naming and Directory Interface (JNDI) lookups, which is an API for providing naming and directory services like Lightweight Directory Access Protocol (LDAP) or Domain Name System (DNS), allowing to load and execute Java objects from remote locations. If a JNDI expression is logged with Log4j, the expression is evaluated, which results in the exploitation of the complete application.

As a prerequisite for the exploit to work, the application must log an attacker-controlled string with the Log4j library. First, an attacker forces the application to log a malicious string of the form `$jndi:ldap://attacker-server.com/a`, with the server `attacker-server.com` being controlled by the attacker. Second, the Log4j library evaluates the string and triggers a request via JNDI to the attacker-controlled

server attacker-server.com. Third, the server attacker-server.com responds with a Java class containing the exploit, e.g., attacker-server2.com/Exploit.class. Fourth, JNDI downloads the class Exploit.class and executes it in the application's context. The exploit then allows an attacker to execute arbitrary code on the host system.

An example exploit opening a reverse shell on the attacked host system is shown in Listing 5.8. The exploit constructs a java.lang.ProcessBuilder object and a java.net.Socket object to open a reverse shell /bin/sh in a separate process.

```
1  import java.io.IOException;
2  import java.io.InputStream;
3  import java.io.OutputStream;
4  import java.net.Socket;
5
6  public class Exploit {
7    public Exploit() throws Exception {
8      String host="%s";
9      int port=%d;
10     String cmd="/bin/sh";
11     Process p=new ProcessBuilder(cmd).redirectErrorStream(true).start();
12     Socket s=new Socket(host,port);
13     InputStream pi=p.getInputStream(), pe=p.getErrorStream(),
14         si=s.getInputStream();
15     OutputStream po=p.getOutputStream(),so=s.getOutputStream();
16     while(!s.isClosed()) {
17         while(pi.available()>0)
18             so.write(pi.read());
19         while(pe.available()>0)
20             so.write(pe.read());
21         while(si.available()>0)
22             po.write(si.read());
23         so.flush();
24         po.flush();
25         Thread.sleep(50);
26         try {
27             p.exitValue();
28             break;
29         }
30         catch (Exception e){}
31     };
32     p.destroy();
33     s.close();
34   }
35 }
```

Listing 5.8: Example exploit for the Log4Shell vulnerability, opening a reverse shell. Taken from [Git23].

The exploit illustrates the shortcomings of the module system for restricting a vulnerable module. First, once an attacker can open a shell on the attacked host system, the whole system is compromised, and the attacker may be free to manipulate the

JVM and host, completely bypassing Java's security architecture. Second, the classes in the packages `java.lang.*`, `java.net.*`, `java.io.*`, allowing file system access, network access, and process creation, are contained in the fundamental module `java.base`. Critically, the base module `java.base` exports all security-relevant Java classes publicly to all modules on the modulepath. The module system does not support the encapsulation (or hiding) of a subset of packages or classes for a specific module, e.g., an included OSS. The module system's encapsulation is coarse-grained compared with the security manager (cf. Section 5.1), which can deny or allow access to the file system, specific files, or network addresses. While a module can confine its own internal types and classes, the module system cannot effectively limit a module's access to sensitive classes, types, methods, and functions in other modules. In particular, the module `java.base` provides too broad access to security-sensitive functions, violating the secure design Principle of Least Privilege [SS75]. Consequently, a vulnerability like Log4shell could not be restricted using the module system. However, if malicious code is executed in the context of the affected module, strong encapsulation only gives access to entities that escape.

An option to deny access to security-relevant classes in the module `java.base` is the use of module layers and class loaders (cf. Section 5.1), as suggested by the Java Platform Group [Ora21]. To do so, a module is loaded into a separate layer with a custom class loader that blocks the module from assessing specific packages and classes using class loader isolation [Gon+97; GED03]. In addition, as suggested by the Java Platform Group, potential refinements are the use of bytecode modification libraries like ASM to remove and replace invocations to security-relevant methods in the untrusted module during class loading.
An alternative is to apply isolation on the whole application using containers and virtual machines, which also allows restricting the operations native code can execute.

Isolation of Native Code The Java Native Interface (JNI) enables Java code to invoke native code written in low-level languages such as C, C++, or assembly. To interact with the operating system for user interactions, file system, or network access; the JCL 1.6 compromises more than 800k lines of native code written in C and C++ [ST12]. Native code is executed in the same memory segment as the JVM and has full access to its heap. This enables native code to freely read and write the JVM's heap, ignoring any Java access control or visibility restrictions. Thus, native code has full control and access to the JVM. Since native code is executed outside

Java's security model without any type- and memory-safety guarantees, vulnerable or malicious native code poses a severe threat to the host system running the Java application.

Consequently, multiple approaches for sandboxing native code in Java exist, which we discuss in the related work in Section 5.6. Since native code resides outside Java's security architecture, the module system provides no means for encapsulation w.r.t. native code.

5.6 Related Work

In this section, we discuss related work concerning the isolation of included OSS. We first introduce approaches for sandboxing native code for Java. Next, we present approaches for encapsulating and isolating faults in OSGi modules (so-called OSGi bundles). Afterward, we discuss related work for detecting (unintended) data flows through which objects escape their intended scope, as we do in ModGuard. The detection of escaping objects is a known problem in static code analysis. Escape analysis has already been applied in the context of memory optimization and synchronization—objects that do not escape their scope do not need to be synchronized, and their memory allocation can be optimized. Finally, we present approaches to enforce information flow policies, e.g., ensuring that certain data is only accessible by a specific code segment.

5.6.1 Sandboxes for Native Code

Since native code is executed outside Java's security model, any flaws or vulnerabilities in native libraries grant full control over the JVM or even the host system. Thus, several approaches for sandboxing native code [STM10; ST12; ST14; Ora23c] have been developed.

Siefers et al. [STM10] present a sandbox called Robusta. Robusta applies Software-Based Fault Isolation (SFI) [Wah+93] and separates native libraries into their own memory segment, the so-called sandbox, within the JVM's memory space. Robusta's sandbox relies on Google's Native Client tool [Yee+10], which prevents native code from reading or writing outside a specified memory address range. The authors extend Google's Native Client with support for dynamic linking and loading since native libraries are dynamically loaded by the JVM. Further, to place native code

under the same security restrictions as Java code, Robusta intercepts JNI invocations and queries Java's security manager to decide if a system call should be blocked or denied based on a predefined security policy.

Sun et. al [ST12; ST14] present Arabica [ST12] an extension of Robusta. Since the implementation of Robusta required significant changes to the JVM, the authors provide with Arabica a sandbox that does not require such modifications, and thus is portable between JVMs. To do so, Arabica uses Java's built-in Java Virtual Machine Tool (JVMTI) interface, which allows to control the JVM's execution at runtime, and generates library stubs that redirect JNI calls to the real libraries in Arabica's sandbox. Further, the authors adapt their concepts with NativeGuard [ST14] to Android by separating an Android app and its used native libraries into different processes to achieve SFI.

An approach that allows an application written in a JVM-based language, like Java, to execute untrusted code written in a sandbox on the GraalVM are polyglot sandboxes [Ora23c]. The GraalVM [Ora23a] was released by Oracle in 2019 as an alternative to the JVM that allows not only the execution of bytecode languages, like Java, Kotlin, Scala, but also the execution of JavaScript, LLVM-based languages such as C, and other dynamic languages on a single platform. GraalVM's polyglot sandbox allows fine-grained access control, such as restricting the loading of native code, limiting resource consumption, forbidding file and network access, and disallowing access to classes and interfaces, etc. The sandbox is based on so-called GraalVM isolates [Wim19]. In contrast to Java's security manager, isolates are not executed in the same JVM environment, sharing the same JCL classes, but in dedicated virtual machine instances. GraalVM isolates allow spanning multiple virtual machine instances in the same process with a separate heap per isolate. This ensures a secure separation between two isolates: objects from one isolate cannot be referenced by another isolate; instead, they must be copied.

5.6.2 Encapsulation and Isolation of OSGi Bundles

Since the Java module system was recently introduced in 2017, existing work on module models for Java focuses on OSGi.

Parrend and Frénot [PF07; PF09] study to what extent OSGi bundles can be isolated from each other and investigate techniques for bypassing the OSGi's encapsulation mechanisms, e.g., modifying a bundle's private data through its API or shared variables, injecting untrusted code into a bundle. The authors generalize the identified

techniques into 25 different vulnerability patterns for the OSGi module systems. While 8 of these patterns originate from issues in the OSGi platform and JVM (e.g., exhaustive memory or CPU-time consumption), 17 of them originate in the implementation of OSGi module system, e.g., allowing duplicate package imports or erroneous bundle declarations. Similar to ModGuard, Parrend and Frénot also investigate patterns in which cases a bundle's internal data can be modified from the outside through its API or shared variables. They also investigate if a bundle may leak internal data by allowing access to methods contained in non-exported packages.

In their following work [Par09; Goi+13], Parrend and Frénot introduce a points-to analysis, which is similar to ModGuard. The presented analysis does not aim to detect escaping instances but aims to detect if objects can be passed from untrusted code into a trusted bundle, thereby risking denial of service attacks when untrusted, malicious code is executed.

Geoffray et al. [Geo+08; Geo+09] introduce I-JVM, a JVM to isolate vulnerable or malicious OSGi bundles from each other. For the secure execution of OSGi bundles, the authors identify vulnerabilities in the OSGi platform and OSGi bundles. They subdivide the found vulnerabilities into three categories: lack of bundle isolation, lack of bundle resource accounting, and failure in bundle termination. For the first category, the authors argue that bundles are not properly isolated in OSGi since `java.lang.Class` objects, static variables, and strings are shared by all bundles in the JVM. These shared instances can be modified by malicious bundles leading to faults in the execution of trusted bundles, e.g., by setting a shared variable to null. For the second category, Geoffray et al. argue that the lack of resource accounting may lead to the exhaustion of memory and CPU-time by malicious bundles. Finally, they argue that the lack of bundle termination may lead to situations where malicious bundles cannot be properly unloaded by the OSGi platform resulting in further abuse of memory or CPU-time. To overcome these limitations, I-JVM executes each bundle in a separate thread containing a private copy of all static variables, strings, and `java.lang.Class` objects, thereby achieving isolation.

Similarly, Gamma and Donsez [GD09; GD10] propose to load untrusted OSGi bundles in separate sandboxes to achieve fault isolation. In contrast to Geoffray et al. [Geo+09], the authors do not provide a customized JVM but rely on the process isolation of the operating system to execute OSGi bundles in separated memory regions. The authors implement a custom proxy-based communication protocol between the isolated bundles that checks invoked methods, arguments, and return values flowing across bundle boundaries at runtime.

Analogously, Huang et al. [HWH07] introduce an own OSGi security layer to prevent malicious bundles from performing security-sensitive operations, e.g., modifying files or probing the API of other bundles. To prevent and detect malicious or faulty bundles, the proposed layer continuously inspects the state of the JVM and stops misbehaving bundles.

While the approaches isolate bundles from each other and investigate possibilities to confine sensitive data within a bundle, they do not investigate unintended data flows (escapes) between bundles. On the contrary, ModGuard aims to support the encapsulation of sensitive entities within modules. To do so, ModGuard precisely defines a module's entry points and statically analyzes the data flow over it to detect confidentiality and integrity violations. While ModGuard can detect confidentiality and integrity violations, it cannot detect availability violations of a module. However, as the module system does not support the dynamic (un-)loading of modules, it provides no mitigation for stopping misbehaving modules.

5.6.3 Escape Analysis

Several approaches apply static analysis to determine whether objects escape a dedicated scope, e.g., they check if an object instance becomes accessible outside of a method or a thread, to improve memory allocation [GS00; VR01; WR99] or remove synchronization: objects that do not escape a method are bound to the method's lifetime, and can therefore be allocated on the stack, and objects that do not escape a thread are bound to the thread's lifetime, and therefore synchronization for these objects can be erased. Similar to our analysis, the escape analyses rely on points-to analyses but operate on a complete code base rather than on modules where the invocations are unknown.

The most related approaches are by Whaley et al. [WR99] and its extension by Viven et al. [VR01]. The authors present an abstract points-to and escape analysis based on so-called points-to escape graphs. In an escape graph, nodes represent objects, and edges represent references between those objects. The analysis separates the code under analysis into unanalyzed and analyzed regions and uses the escape graph to record escape paths into unanalyzed regions [VR01].
Using the graph, the algorithm tracks all escaping instances: if an object escapes, all objects that the escaping object references also escape. The approaches can identify escaping objects, and, additionally, methods, (static) fields, parameters, exceptions, and callbacks like ModGuard.

The Watson Libraries for Analysis (WALA) [IBM06] framework implements a state-of-the-art escape analysis. The algorithm respects fields, thread constructor parameters, and all objects transitively reachable from fields of escaping objects, but solely focuses on threads.

Blanchet [Bla03] and Choi et al. [Cho+99] propose fast escape analyses for optimizing memory allocation. The approach omits the computation of sophisticated points-to information but uses integer vectors or type hierarchy analysis to represent types. These simplified type representations are insufficient in the context of modules. Determining whether a particular entity escapes through the API depends on the object's type, its super-, and subtypes, which the presented approaches tune out in the simplified representation.

Current escape analyses check if objects escape methods or threads but do not consider escapes in structural contexts like modules. In addition, they analyze concrete implementations, including callers, whereas our analysis, in the absence of such callers, analyzes all potential usage scenarios of a module, including potentially malicious ones. Consequently, the presented approaches do not require nor provide a set of formalized entry points through which the confidentiality or integrity of the module may be violated.

5.6.4 Information-Flow Control

To cope with unintended data flows, several approaches [BVR15; SM03; Enc+14; Gif+17; Yip+09] exist for analyzing information-flow using runtime-monitoring, static analysis, or language-based mechanisms.

Sabelfeld and Myers [SM03] state that visibility constraints and access controls are insufficient to protect confidential data within modules. Instead, the authors advocate the introduction of security-type systems into programming languages to augment the types with annotations that specify policies on the use of the data. These security policies are enforced by compile-time type checking.

Buiras et al. [BVR15] introduce the library Hybrid LIO for the programming language Haskell to enforce information-flow policies both statically and at runtime. The authors extend Haskell's type system to distinguish public and confidential data. Based on the extended type system, the library Hybrid LIO checks statically and at runtime if confidential data flows into public objects or methods.

In contrast, ModGuard does not require a security-type system. Instead, ModGuard analyzes all potential data flows through a module's API and checks if they leak or manipulate sensitive entities. Nevertheless, the presented security-type systems are more powerful since they enable checking for non-interference and data flows in distributed systems, e.g., web servers or files.

Enck et al. [Enc+14] introduce TaintDroid, a runtime monitor for Android to limit data flow between apps. To do so, TaintDroid instruments the Android VM and taints sensitive information on the level of variables, methods, files, and inter-application messages. TaintDroid traces the data flow at runtime and reports violations whenever data flows into a method, variable, or application with a lower security level.

Yip et al. [Yip+09] propose Resin a runtime for PHP and Python to prevent data leaks in web applications. Resin allows developers to specify application-level data flow policies, which are then enforced at runtime. Similarly, Giffin et al. [Gif+17] present a novel web framework for Haskell that allows the specification of data flow policies for sensitive entities and enforces them at runtime.
ModGuard does not provide checks for enforcing data flow policies at runtime but is exclusively a static analysis for checking the encapsulation of sensitive entities through APIs.

5.7 Conclusion

In this chapter, we investigated to what extent developers can benefit from the Java module system for restricting vulnerabilities in one module from affecting others by means of strong encapsulation. To support developers in applying strong encapsulation, we first capture in a formal definition of the module's entry points what types, methods, and fields may become *explicitly* or *implicitly* accessible to code outside the module. Our entry-point model can also serve as a basis for future static analyses for Java's modules since it specifies which methods or types of a module may become accessible, thereby computing the set of methods that are directly invokable on a given module.

Based on our entry-point model, we introduce the static analysis ModGuard for detecting the escaping of sensitive entities. We illustrate that escaping sensitive entities occur in real-world applications and that ModGuard can identify them in a case study of Tomcat 8.5.21. Our case study shows that using the standard tool jdeps for naïvely migrating to modules does not allow developers to benefit from the module system. Yet, it shows that simply restricting modules by strictly limiting

export statements only has a small effect on the number of violations. Even with only a few exported packages, a significant number of sensitive entities can leak. Hence, the case study shows that if developers want to benefit from modules and limit the escaping of sensitive entities, developers must refactor the application's type hierarchy, as our discussion of CVE-2017-5648 shows. Nevertheless, the case study also shows that ModGuard can support migrating to modules by identifying problematic data flows leading to confidentiality and integrity violations of security-relevant entities, complementing the module system.

Our discussion on the limitations of the module system for the secure integration of OSS shows that Java modules are insufficient to isolate inadvertent vulnerabilities in modules. Especially the Log4Shell vulnerability shows that despite its strong encapsulation and access prevention to JDK internal types, the permissions granted by the base module java.base are too broad, violating the *Principle of Least Privilege*. The module system does not provide fine-grained access control mechanisms like the security manager. Further, it provides no means for isolating native code. Given that the security manager has been marked for removal with Java 17, GraalVM isolates are a candidate for further research in isolating included OSS as they provide strong isolation by maintaining separate memory heaps per isolate.

6 Conclusion and Outlook

Using community-developed, well-tested Open-Source Software (OSS) provides several benefits for software development; it increases speed and quality while decreasing costs. Consequently, today's open-source and commercial applications are shipped with several open-source libraries and frameworks. A single vulnerability in any of those open-source artifacts poses a severe threat to the application, as multiple exploits of vulnerabilities in OSS like Log4Shell or Struts2 have shown. Thus, developers must react quickly when a new vulnerability has been published in one of the included open-source artifacts by updating the vulnerable artifact to a more recent, non-vulnerable version. To do so, developers must regularly check if any of the included open-source artifacts are affected by a published vulnerability. Since modern build-automation and dependency-management tools transparently include dependencies (and their dependencies), developers often need to be made aware of all the artifacts their application includes. On top of that, developers need to conduct any updates carefully to avoid breaking the applications. In the case of enterprise systems that serve business-critical functions, any downtime or malfunction comes with high costs.
These are severe challenges that developers have to cope with. Prior research shows that developers are, in fact, unaware that an included open-source artifact was affected by a known vulnerability or are hesitant to update an artifact as they are afraid of breaking the application.

In this work, we have presented approaches and tools to address these issues. In our first contribution, our study on the use of OSS at SAP, we investigate developer practices regarding the use of open-source artifacts in commercial applications. Our study revealed challenges for detecting known-vulnerable open-source artifacts faced by developers and automated tools. In particular, we found that developers commonly included open-source artifacts that have been modified in some aspect by forking, re-compilation, or re-bundling multiple artifacts into one. Further, our study showed that current tools fail to detect known vulnerabilities if any modification has been applied to artifacts. This poses the risk that vulnerabilities remain undetected. Our results show that such modifications are not exclusive to commercial applications but also occur in open-source projects hosted on the public open-source repository Maven Central. To facilitate future research and allow the comparison

of tools for detecting known-vulnerable open-source artifacts, we derived the test suite Achilles from our study. Our study and test suite aim to foster future research to improve the detection capability of tools, especially—but not exclusively—for modified artifacts.

With SootDiff, we present a tool to check if two classes originate from the same artifact, even if one has been subject to modifications. SootDiff uses Soot's intermediate representation Jimple and static analyses to transform the classes' bytecode into a unified representation, canceling out dissimilarities introduced by different Java compilers and versions. SootDiff can be considered a first step for successfully detecting modified open-source artifacts. To adapt it to practical environments, SootDiff can be used to construct databases containing the bytecode (or the fingerprints) of the original, unmodified open-source artifacts. Using SootDiff, local dependencies in the application can be matched against fingerprints in the created database or bytecode. Combined with local sensitive hashing, such as TLSH, the computation of robust fingerprints of the bytecode generated by SootDiff can be further optimized for database storage and resilience against modifications, e.g., re-packaging and re-naming.

Our second contribution, UpCy, an approach for finding compatible updates automatically, solves the challenges developers face when trying to update a vulnerable open-source artifact. In contrast to state-of-the-art tools, UpCy considers all dependencies an application includes and suggests—if required—an update of (multiple) artifacts to achieve maximum compatibility with the application and other libraries and frameworks. To do so, UpCy merges the application's call graph with the application's dependency graph into a new representation—the unified dependency graph. On this graph, UpCy applies the min-(s,t)-cut algorithm to identify update options with minimal incompatibilities and queries our graph database of Maven Central for concrete instances of these update options. If UpCy finds a concrete set of updates, it reports them to developers. Then, developers can add the proposed updates as direct dependencies to their application to eliminate the vulnerable artifact from the dependency tree. Our empirical evaluation revealed that UpCy effectively finds updates for artifacts with less or no incompatibilities in cases where state-of-the-art approaches fail. However, further work is needed to evaluate the acceptance and understandability of the generated updates from the developers' perspective.

In our third contribution, we checked to what extent modules can be used for the secure integration of OSS using the strong encapsulation of the Java module system. We developed the static analysis ModGuard to allow developers to implement modules securely so that modules confine security-relevant entities. ModGuard detects

which internal types and data can become accessible through a module's Application Programming Interface (API) to outside code, like included OSS. To facilitate the development of further module analyses, we introduced a formal module model specified in the logic-based language Datalog. The model precisely formulates what constitutes a module's API and what types are meant to be internal. Our case study on Apache Tomcat and the discussions of the limitations of the module system show that modules are insufficient for the secure integration of OSS as they provide no means for controlling access to host resources, e.g., file system or network, and no isolation of native code. Nevertheless, modules successfully prevent access to security-relevant Java Development Kit (JDK) internal classes, and thus can support information hiding by restricting access to sensitive classes and methods.

The use of (vulnerable) OSS is not exclusive to Java but is common practice in most programming languages, as shown in several studies [HVG18; Pit16; Mar+18]. Equally, the challenges for identifying and updating open-source artifacts are similar in other programming languages and tools, as we discussed for npm and pip. The developed module model and UpCy are general enough to be applied to other programming languages. However, the adaptation of the tools and concepts to programming languages that do not have a global dependency graph but allow conflicting and duplicate dependencies should be evaluated in future work.

The secure integration of (untrusted) OSS in applications remains an open challenge. With the deprecation of the security manager in Java, the deprecation of the security manager and code access security in .NET, as well as the deprecation of the portable NativeClient in Google's Chrome web browser, other mechanisms for limiting the impact of inadvertent vulnerabilities in OSS are required; strong encapsulation can only serve as a basis. To enable a wide adaption, these mechanisms should be easy to implement for developers. The constructive *Principle of Least Privilege* and language-based security mechanisms can serve as guidelines to achieve this goal. For instance, extensions to the module system that allow fine-grained export of certain packages and classes of the module `java.base` to specific modules only are potential candidates.

Implementations and Data

While conducting the research presented in this thesis, we have created several data sets and prototypical implementations. We made these data and implementations available to enable other researchers to reproduce and extend our results. All projects contain detailed documentation and instructions to validate our results.

Study on the Use of Open-Source Software and Achilles

The Achilles test suite for evaluating the precision and recall of open-source vulnerability scanners presented in Chapter 3 is published at `https://github.com/secure-software-engineering/achilles-benchmark-depscanners`. Achilles provides a graphical user interface for applying the modifications: re-compilation, re-packaging, and re-bundling to open-source artifacts and enables the automatic creation of Maven projects that can serve as test cases for vulnerability scanners. Further, the test suite comprises the manual classification of the 2,558 vulnerability reports presented in Section 3.3. The case study and results presented in Section 3.4 are also contained in the repository.

UpCy: Updating Open-Source Dependencies

The source code of UpCy presented in Chapter 4 is available at `https://github.com/secure-software-engineering/upcy`.

The Java projects used in the evaluation of UpCy and the results presented in Section 4.3 are hosted at `https://doi.org/10.5281/zenodo.7037673`. The archive also contains the complete snapshot of the Maven Central dependency graph and docker containers to re-run the evaluation.

ModGuard: Module Escape Analysis

The implementation of ModGuard presented in Chapter 5 is available at `https://github.com/secure-software-engineering/modguard`. In addition to the analysis' source code, the repository also contains the naïve and strict modularization of the Tomcat 8.5.21 and the evaluation results presented in Section 5.4.

The test suite for Java modules MIC9Bench is publicly available at `https://github.com/secure-software-engineering/mic9bench`. The repository comprises 22 test cases as well as detailed documentation to facilitate reproducibility and further module-based static code analyses. To re-run our evaluation, the repository contains an interactive user interface.

Bibliography

[BM98] Brenda S. Baker and Udi Manber. “Deducing Similarities in Java Sources from Bytecodes”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’98. New Orleans, Louisiana: USENIX Association, 1998, pp. 15–15. DOI: 10.5555/1268256.1268271 (cit. on p. 66).

[BN05] Anindya Banerjee and David A. Naumann. “Stack-based access control and secure information flow”. In: *Journal of Functional Programming* 15.2 (Mar. 2005), pp. 131–177. DOI: 10.1017/s0956796804005453 (cit. on p. 107).

[BHD12] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. “A structured approach to assess third-party library usage”. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sept. 2012, pp. 483–492. DOI: 10.1109/ICSM.2012.6405311 (cit. on pp. 1, 22, 23).

[Bav+15] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. “How the Apache community upgrades dependencies: an evolutionary study”. In: *Empirical Software Engineering* 20.5 (Oct. 2015), pp. 1275–1317. DOI: 10.1007/s10664-014-9325-9 (cit. on pp. 1, 3, 22, 23, 46, 64, 69, 96, 98).

[Bax+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. “Clone Detection Using Abstract Syntax Trees”. In: *Proceedings. International Conference on Software Maintenance*. ICSM ’98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–377. DOI: 10.1109/ICSM.1998.738528 (cit. on p. 67).

[Bla+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, et al. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. In: *ACM SIGPLAN Notices* 41.10 (Oct. 2006), pp. 169–190. DOI: 10.1145/1167515.1167488 (cit. on pp. 49, 65).

[Bla03] Bruno Blanchet. “Escape Analysis for Java: Theory and Practice”. In: *ACM Transactions on Programming Languages and Systems* 25.6 (Nov. 2003), pp. 713–775. DOI: 10.1145/945885.945886 (cit. on p. 146).

[Bod+11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. “Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders”. In: *Proceedings of the 33rd international conference on Software engineering - ICSE ’11*. ICSE ’11. New York, New York, USA: ACM, May 2011, pp. 241–250. DOI: 10.1145/1985793.1985827 (cit. on p. 108).

[BKH15] Christopher Bogart, Christian Kästner, and James Herbsleb. “When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, Nov. 2015, pp. 86–89. DOI: 10.1109/ASEW.2015.21 (cit. on pp. 2, 6).

[Bog+16] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. “How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, Nov. 2016, pp. 109–120. DOI: 10.1145/2950290.2950325 (cit. on pp. 2–4, 6, 64, 73, 96).

[BVR15] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. “HLIO: Mixing Static and Dynamic Typing for Information-flow Control in Haskell”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, Aug. 2015, pp. 289–301. DOI: 10.1145/2784731.2784758 (cit. on p. 146).

[Cho+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. “Escape Analysis for Java”. In: *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA ’99. Denver, Colorado, USA: ACM, Oct. 1999, pp. 1–19. DOI: 10.1145/320384.320386 (cit. on p. 146).

[CGK15] Cristina Cifuentes, Andrew Gross, and Nathan Kynes. “Understanding Caller-Sensitive Method Vulnerabilities A Class of Access Control Vulnerabilities in the Java Platform”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2015. New York, NY, USA: ACM, June 2015, pp. 7–12. DOI: 10.1145/2771284.2771286 (cit. on p. 106).

[Cok+15] Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. “Evaluating the Flexibility of the Java Sandbox”. In: *Proceedings of the 31st Annual Computer Security Applications Conference*. New York, New York, USA: ACM, Dec. 2015, pp. 1–10. DOI: 10.1145/2818000.2818003 (cit. on pp. 3, 102, 107).

[Cor+03] John Corwin, David F. Bacon, David Grove, and Chet Murthy. “MJ: a rational module system for Java and its applications”. In: *ACM SIGPLAN Notices* 38.11 (Oct. 2003), pp. 241–254. DOI: 10.1145/949343.949326 (cit. on p. 130).

[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*. SOAP 2019. Phoenix, AZ, USA: ACM, June 2019, pp. 14–19. DOI: 10.1145/3315568.3329966 (cit. on pp. ix, 89).

[DHB21] Andreas Dann, Ben Hermann, and Eric Bodden. “ModGuard: Identifying Integrity & Confidentiality Violations in Java Modules”. In: *IEEE Transactions on Software Engineering* 47.8 (Aug. 2021), pp. 1656–1667. DOI: 10.1109/TSE.2019.2931331 (cit. on p. ix).

[DHB23] Andreas Dann, Ben Hermann, and Eric Bodden. “UPCY: Safely Updating Outdated Dependencies”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. ICSE ’23. Melbourne, Australia: IEEE, May 2023, pp. 233–244. DOI: 10.1109/icse48619.2023.00031 (cit. on p. ix).

[Dan+22] 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 (Sept. 2022), pp. 3613–3625. DOI: 10.1109/TSE.2021.3101739 (cit. on pp. ix, 46, 47).

[DBM19] Stanislav Dashevskyi, Achim D. Brucker, and Fabio Massacci. “A Screening Test for Disclosed Vulnerabilities in FOSS Components”. In: *IEEE Transactions on Software Engineering* 45.10 (Oct. 2019), pp. 945–966. DOI: 10.1109/TSE.2018.2816033 (cit. on p. 21).

[DMG19] Alexandre Decan, Tom Mens, and Philippe Grosjean. “An empirical comparison of dependency network evolution in seven software packaging ecosystems”. In: *Empirical Software Engineering* 24.1 (Feb. 2019), pp. 381–416. DOI: 10.1007/s10664-017-9589-y (cit. on p. 1).

[Der+17] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. “Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: ACM, Oct. 2017, pp. 2187–200. DOI: 10.1145/3133956.3134059 (cit. on pp. 2–4).

[DJB14] Jens Dietrich, Kamil Jezek, and Premek Brada. “Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades”. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, Feb. 2014, pp. 64–73. DOI: 10.1109/CSMR-WCRE.2014.6747226 (cit. on pp. 3, 71, 73, 97).

[Die+19] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. “Dependency Versioning in the Wild”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2019, pp. 349–359. DOI: 10.1109/MSR.2019.00061 (cit. on p. 97).

[Die+17] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. “XCorpus - An executable corpus of Java programs”. In: *The Journal of Object Technology* 16.4 (2017), pp. 1–24. DOI: 10.5381/jot.2017.16.4.a1 (cit. on p. 65).

[DH22] Johannes Düsing and Ben Hermann. “Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories”. In: *Digital Threats* (June 2022). Just Accepted. DOI: 10.1145/3472811 (cit. on pp. 2, 4, 98).

[Enc+14] William Enck, Peter Gilbert, Byung-Gon Chun, et al. “TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *ACM Transactions on Computer Systems*. OSDI’10 32.2 (June 2014), pp. 1–29. DOI: 10.1145/2619091 (cit. on pp. 146, 147).

[ECZ22] Douglas Everson, Long Cheng, and Zhenkai Zhang. “Log4shell: Redefining the Web Attack Surface”. In: *Proceedings 2022 Workshop on Measurements, Attacks, and Defenses for the Web*. Internet Society, 2022. DOI: 10.14722/madweb.2022.23010 (cit. on p. 139).

[Fan+20] Gang Fan, Chengpeng Wang, Rongxin Wu, et al. “Escaping dependency hell: Finding build dependency errors with the unified dependency graph”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, July 2020, pp. 463–474. DOI: 10.1145/3395363.3397388 (cit. on p. 98).

[Foo+18] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. “Efficient Static Checking of Library Updates”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: ACM, Oct. 2018, pp. 791–96. DOI: 10.1145/3236024.3275535 (cit. on p. 89).

[FF56] L. R. Ford and D. R. Fulkerson. “Maximal Flow Through a Network”. In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. DOI: 10.4153/CJM-1956-045-5 (cit. on p. 80).

[GD09] Kiev Gama and Didier Donsez. “Towards Dynamic Component Isolation in a Service Oriented Platform”. In: *Component-Based Software Engineering*. Ed. by Grace A Lewis, Iman Poernomo, and Christine Hofmeister. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 104–120. DOI: 10.1007/978-3-642-02414-6_7 (cit. on p. 144).

[GD10] Kiev Gama and Didier Donsez. “A Self-healing Component Sandbox for Untrustworthy Third Party Code Execution”. In: *Component-Based Software Engineering*. Ed. by Lars Grunske, Ralf Reussner, and Frantisek Plasil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 130–149. DOI: 10.1007/978-3-642-13238-4_8 (cit. on p. 144).

[GS00] David Gay and Bjarne Steensgaard. “Fast Escape Analysis and Stack Allocation for Object-Based Programs”. In: *Compiler Construction*. CC ’00. London, UK, UK: Springer Berlin Heidelberg, 2000, pp. 82–93. DOI: 10.1007/3-540-46423-9_6 (cit. on p. 145).

[Geo+09] Nicolas Geoffray, Gael Thomas, Gilles Muller, et al. “I-JVM: a Java Virtual Machine for component isolation in OSGi”. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, June 2009, pp. 544–553. DOI: 10.1109/DSN.2009.5270296 (cit. on p. 144).

[Geo+08] Nicolas Geoffray, Gaël Thomas, Bertil Folliot, and Charles Clément. “Towards a New Isolation Abstraction for OSGi”. In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. IIES ’08. Glasgow, Scotland: ACM, Apr. 2008, pp. 41–45. DOI: 10.1145/1435458.1435466 (cit. on p. 144).

[Gif+17] Daniel B. Giffin, Amit Levy, Deian Stefan, et al. "Hails: Protecting Data Privacy in Untrusted Web Applications". In: *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)* 25.4-5 (July 2017). Ed. by Toby Murray, Andrei Sabelfeld, and Lujo Bauer, pp. 427–461. DOI: 10.3233/jcs-15801 (cit. on pp. 146, 147).

[Goi+13] François Goichon, Guillaume Salagnac, Pierre Parrend, and Stéphane Frénot. "Static Vulnerability Detection in Java Service-oriented Components". In: *Journal of Computer Virology and Hacking Techniques* 9.1 (Sept. 2013), pp. 15–26. DOI: 10.1007/s11416-012-0172-1 (cit. on p. 144).

[GED03] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. 2. ed., 1. print. Boston, Mass.: Addison-Wesley Professional, 2003 (cit. on pp. 13, 102, 104, 106, 107, 141).

[Gon+97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. "Going beyond the Sandbox: An Overview of the New Security Architecture in the JavaTM Development Kit 1.2". In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*. USITS'97. Monterey, California: USENIX Association, 1997, p. 10 (cit. on pp. 104, 141).

[GS17] Neville Grech and Yannis Smaragdakis. "P/Taint: Unified Points-to and Taint Analysis". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–28. DOI: 10.1145/3133926 (cit. on pp. 121, 122).

[Har22] Nicolas Harrand. "Software Diversity for Third-Party Dependencies". PhD thesis. SE-10044 Stockholm, Sweden: KTH Royal Institute of Technology, School of Electrical Engineering, Computer Science, Division of Software, and Computer Systems, 2022 (cit. on p. 3).

[Hei+11] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. "On the Extent and Nature of Software Reuse in Open Source Java Projects". In: *Top Productivity through Software Reuse*. Ed. by Klaus Schmid. Vol. 6727 LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 207–222. DOI: 10.1007/978-3-642-21347-2_16 (cit. on pp. 1, 22, 23, 64).

[HG22] Joseph Hejderup and Georgios Gousios. "Can we trust tests to automate dependency updates? A case study of Java Projects". In: *Journal of Systems and Software* 183 (Jan. 2022), p. 111097. DOI: 10.1016/j.jss.2021.111097 (cit. on pp. 2–4, 6, 7, 64, 70, 74, 78, 86, 87, 89, 93, 95, 97).

[HVG18] Joseph Hejderup, Arie Van Deursen, and Georgios Gousios. "Software ecosystem call graph for dependency management". In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, May 2018, pp. 101–104. DOI: 10.1145/3183399.3183417 (cit. on pp. 7, 97, 98, 151).

[Hen+20] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. “Finding All Global Minimum Cuts in Practice”. en. In: *28th Annual European Symposium on Algorithms (ESA 2020)*. Ed. by Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders. Vol. 173. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 1–20. DOI: `10.4230/LIPIcs.ESA.2020.59` (cit. on p. 95).

[HB21] Philipp Holzinger and Eric Bodden. “A Systematic Hardening of Java’s Information Hiding”. In: *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*. ACM, May 2021, pp. 11–22. DOI: `10.1145/3457340.3458300` (cit. on pp. 101, 102, 108).

[Hol+16] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. “An In-Depth Study of More Than Ten Years of Java Exploitation”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. New York, New York, USA: ACM, Oct. 2016, pp. 779–790. DOI: `10.1145/2976749.2978361` (cit. on pp. 102, 107).

[Hu+17] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. “Binary Code Clone Detection across Architectures and Compiling Configurations”. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 2017, pp. 88–98. DOI: `10.1109/ICPC.2017.22` (cit. on p. 8).

[HWH07] Chi-Chih Huang, Pang-Chieh Wang, and Ting-Wei Hou. “Advanced OSGi Security Layer”. In: *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW’07)*. Vol. 2. IEEE, May 2007, pp. 518–523. DOI: `10.1109/AINAW.2007.70` (cit. on p. 145).

[Hua+20] Kaifeng Huang, Bihuan Chen, Bowen Shi, et al. “Interactive, effort-aware library version harmonization”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov. 2020, pp. 518–529. DOI: `10.1145/3368089.3409689` (cit. on p. 70).

[J H94] J. Howard Johnson. “Substring matching for clone detection and change tracking”. In: *Proceedings International Conference on Software Maintenance ICSM-94*. IEEE Comput. Soc. Press, 1994, pp. 120–126. DOI: `10.1109/ICSM.1994.336783` (cit. on p. 67).

[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 (July 2002), pp. 654–670. DOI: `10.1109/TSE.2002.1019480` (cit. on p. 67).

[KRR14] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. “SeByte: Scalable clone and similarity search for bytecode”. In: *Science of Computer Programming* 95 (Dec. 2014), pp. 426–444. DOI: `10.1016/j.scico.2013.10.006` (cit. on p. 66).

[Kos07] Rainer Koschke. “Survey of Research on Software Clones”. en. In: Dagstuhl Seminar Proceedings 06301 (2007). Ed. by Rainer Koschke, Ettore Merlo, and Andrew Walenstein. DOI: `10.4230/DagSemProc.06301.13` (cit. on pp. 66, 67).

[Krü+21] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs". In: *IEEE Transactions on Software Engineering* 47.11 (Nov. 2021), pp. 2382–2400. DOI: 10.1109/TSE.2019.2948910 (cit. on p. 121).

[Kul+18] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. "Do Developers Update Their Library Dependencies?" In: *Empirical Software Engineering* 23.1 (Feb. 2018), pp. 384–417. DOI: 10.1007/s10664-017-9521-5 (cit. on pp. 1, 2, 4, 22, 23, 46–48, 64, 69, 96, 98).

[Lam+11] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*. Vol. 15. 35. 2011 (cit. on pp. 54, 58, 60, 120, 121).

[Lat+23] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. "Not All Dependencies Are Equal: An Empirical Study on Production Dependencies in NPM". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3551349.3556896 (cit. on p. 7).

[Ler01] Xavier Leroy. "Java Bytecode Verification: An Overview". In: *Computer Aided Verification*. Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 265–285. DOI: 10.1007/3-540-44585-4_26 (cit. on p. 105).

[LTX19] Yue Li, Tian Tan, and Jingling Xue. "Understanding and analyzing Java reflection". In: *ACM Transactions on Software Engineering and Methodology* 28.2 (Feb. 2019), pp. 1–50. DOI: 10.1145/3295739 (cit. on pp. 107, 108).

[LB98] Sheng Liang and Gilad Bracha. "Dynamic Class Loading in the Java Virtual Machine". In: *SIGPLAN Not.* 33.10 (Oct. 1998), pp. 36–44. DOI: 10.1145/286942.286945 (cit. on pp. 13, 106).

[Lim94] W.C. Lim. "Effects of reuse on quality, productivity, and economics". In: *IEEE Software* 11.5 (Sept. 1994), pp. 23–30. DOI: 10.1109/52.311048 (cit. on p. 1).

[Liv+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, et al. "In defense of soundiness: A manifesto". In: *Communications of the ACM* 58.2 (Jan. 2015), pp. 44–46. DOI: 10.1145/2644805 (cit. on p. 103).

[Lop+17] Cristina V. Lopes, Petr Maj, Pedro Martins, et al. "DéjàVu: a map of code duplicates on GitHub". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–28. DOI: 10.1145/3133908 (cit. on p. 64).

[Mar+18] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. "Docker ecosystem - Vulnerability Analysis". In: *Computer Communications* 122 (June 2018), pp. 30–43. DOI: 10.1016/J.COMCOM.2018.03.011 (cit. on pp. 64, 151).

[Mas+15] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, et al. "Use at Your Own Risk: The Java Unsafe API in the Wild". In: *ACM SIGPLAN Notices* 50.10 (Oct. 2015), pp. 695–710. DOI: 10.1145/2858965.2814313 (cit. on p. 106).

[MLM96] Mayrand, Leblanc, and Merlo. "Experiment on the automatic detection of function clones in a software system using metrics". In: *Proceedings of International Conference on Software Maintenance ICSM-96*. IEEE, 1996, pp. 244–253. DOI: 10.1109/ICSM.1996.565012 (cit. on p. 67).

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. "Parameterized Object Sensitivity for Points-to Analysis for Java". In: *ACM Transactions on Software Engineering and Methodology* 14.1 (Jan. 2005), pp. 1–41. DOI: 10.1145/1044834.1044835 (cit. on p. 133).

[MP18] André Miranda and João Pimentel. "On the Use of Package Managers by the C++ Open-Source Community". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France: Association for Computing Machinery, 2018, pp. 1483–1491. DOI: 10.1145/3167132.3167290 (cit. on p. 16).

[MP17] Samim Mirhosseini and Chris Parnin. "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Oct. 2017), pp. 84–94. DOI: 10.1109/ASE.2017.8115621 (cit. on pp. 2, 4, 6, 69, 96, 98).

[MNT20] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. "Detecting locations in JavaScript programs affected by breaking library changes". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (Nov. 2020), pp. 1–25. DOI: 10.1145/3428255 (cit. on p. 7).

[Mye86] Eugene W. Myers. "AnO(ND) difference algorithm and its variations". In: *Algorithmica* 1.1-4 (Jan. 1986), pp. 251–266. DOI: 10.1007/BF01840446 (cit. on p. 60).

[NNI00] Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. "A Fast Algorithm for Cactus Representations of Minimum Cuts". In: *Japan Journal of Industrial and Applied Mathematics* 17.2 (June 2000), pp. 245–264. DOI: 10.1007/BF03167346 (cit. on p. 95).

[NDM16] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. "An automatic method for assessing the versions affected by a vulnerability". In: *Empirical Software Engineering* 21.6 (Dec. 2016), pp. 2268–2297. DOI: 10.1007/s10664-015-9408-2 (cit. on p. 21).

[NM13] Viet Hung Nguyen and Fabio Massacci. "The (Un)Reliability of NVD Vulnerable Versions Data: An Empirical Experiment on Google Chrome Vulnerabilities". In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS '13. Hangzhou, China: Association for Computing Machinery, 2013, pp. 493–98. DOI: 10.1145/2484313.2484377 (cit. on p. 21).

[Nil+13] Gary Nilson, Kent Wills, Jeffrey Stuckman, and James Purtilo. "BugBox: A Vulnerability Corpus for PHP Web Applications". In: *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*. Washington, D.C.: USENIX, 2013 (cit. on p. 65).

[OCC13] Jonathan Oliver, Chun Cheng, and Yanggui Chen. "TLSH – A Locality Sensitive Hash". In: *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, Nov. 2013, pp. 7–13. DOI: 10.1109/CTC.2013.9 (cit. on p. 39).

[One96] Aleph One. "Smashing the stack for fun and profit". In: *Phrack magazine* 7.49 (1996), pp. 14–16 (cit. on p. 105).

[Ora17c] Oracle Corporation. *The Java Language Specification Java SE 9 Edition*. https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf. 2017 (cit. on pp. 108, 110).

[OSG23] OSGi Alliance. *OSGi Core, Release 8*. Tech. rep. https://osgi.github.io/osgi/core/. Eclipse Foundation, 2023 (cit. on pp. 14, 106, 112).

[Par09] Pierre Parrend. "Enhancing Automated Detection of Vulnerabilities in Java Components". In: *2009 International Conference on Availability, Reliability and Security*. IEEE, Mar. 2009, pp. 216–223. DOI: 10.1109/ARES.2009.9 (cit. on p. 144).

[PF07] Pierre Parrend and Stephane Frénot. "Supporting the Secure Deployment of OSGi Bundles". In: *2007 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*. IEEE, June 2007, pp. 1–6. DOI: 10.1109/WOWMOM.2007.4351681 (cit. on p. 143).

[PF09] Pierre Parrend and Stephane Frénot. "Security benchmarks of OSGi platforms: toward Hardened OSGi". In: *Software: Practice and Experience* 39.5 (Apr. 2009), pp. 471–499. DOI: 10.1002/spe.906 (cit. on p. 143).

[Pas+18] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. "Vulnerable Open Source Dependencies: Counting Those That Matter". In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '18. Oulu, Finland: ACM, Oct. 2018, pp. 1–10. DOI: 10.1145/3239235.3268920 (cit. on pp. 20, 23, 28, 46–48, 64, 74, 75, 77, 87).

[Pas+22] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. "Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies". In: *IEEE Transactions on Software Engineering* 48.5 (May 2022), pp. 1592–1609. DOI: 10.1109/TSE.2020.3025443 (cit. on pp. 2, 4, 22, 47, 64, 87, 98).

[PVM20] Ivan Pashchenko, Duc-Ly Ly Vu, and Fabio Massacci. "A Qualitative Study of Dependency Management and Its Security Implications". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. Virtual Event, USA: ACM, Oct. 2020, pp. 1513–1531. DOI: 10.1145/3372297.3417232 (cit. on p. 7).

[Pis22] Goran Piskachev. "Adapting taint analyses for detecting security vulnerabilities". eng. PhD thesis. Paderborn University, Warburger Straße 100, 33098 Paderborn: Faculty for Computer Science, Electrical Engineering and Mathematics, Department of Computer Science, Research Group Secure Software Engineering, 2022. DOI: 10.17619/UNIPB/1-1665 (cit. on p. 131).

[Pit16] Mike Pittenger. *The State of Open Source Security in Commercial Applications*. Tech. rep. https://info.blackducksoftware.com/rs/872-OLS-526/images/OSSAReportFINAL.pdf. BlackDuck Software, 2016, pp. 1–5 (cit. on pp. 1, 22, 46–48, 64, 151).

[Pla23] Henrik Plate. *State of Dependency Management 2023*. Tech. rep. https://www.endorlabs.com/state-of-dependency-management-2023. Endor Labs, 2023 (cit. on p. 1).

[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, Sept. 2018, pp. 449–460. DOI: 10.1109/ICSME.2018.00054 (cit. on pp. 21, 23, 24, 34, 36, 46, 64, 98).

[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 (Sept. 2020), pp. 3175–3215. DOI: 10.1007/s10664-020-09830-x (cit. on pp. 21, 23, 36).

[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)*. MSR '19. Montreal, Quebec, Canada: IEEE, May 2019, pp. 383–387. DOI: 10.1109/MSR.2019.00064 (cit. on pp. 27, 46–48).

[Pra+21] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, et al. "Out of sight, out of mind? How vulnerable dependencies affect open-source projects". In: *Empirical Software Engineering* 26.4 (Apr. 2021). DOI: 10.1007/s10664-021-09959-3 (cit. on pp. 2–4, 7, 96).

[RVV14] Steven Raemaekers, Arie Van Deursen, and Joost Visser. "Semantic versioning versus breaking changes: A study of the maven repository". In: *Proceedings - 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014* (Sept. 2014), pp. 215–224. DOI: 10.1109/SCAM.2014.30 (cit. on p. 3).

[RKC18] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. "A Comparison of Code Similarity Analysers". In: *Empirical Software Engineering* 23.4 (Aug. 2018), pp. 2464–2519. DOI: 10.1007/s10664-017-9564-7 (cit. on p. 67).

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks". In: *Proceedings 2014 Network and Distributed System Security Symposium* February (2014), pp. 23–26. DOI: 10.14722/ndss.2014.23039 (cit. on p. 131).

[Rei+16] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. "Call Graph Construction for Java Libraries". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, Nov. 2016, pp. 474–486. DOI: 10.1145/2950290.2950312 (cit. on p. 120).

[RH09] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (Apr. 2009), pp. 131–164. DOI: 10.1007/s10664-008-9102-8 (cit. on p. 24).

[SM03] Andrei Sabelfeld and Andrew C. Myers. “Language-based information-flow security”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (Jan. 2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121 (cit. on p. 146).

[Saj+14] Hitesh Sajnani, Vaibhav Saini, Joel Ossher, and Cristina V. Lopes. “Is Popularity a Measure of Quality? An Analysis of Maven Components”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, Sept. 2014, pp. 231–240. DOI: 10.1109/ICSME.2014.45 (cit. on p. 28).

[SR05] Alexandru Sălcianu and Martin Rinard. “Purity and Side Effect Analysis for Java Programs”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 199–215. DOI: 10.1007/978-3-540-30579-8_14 (cit. on p. 126).

[SS75] Jerome H. Saltzer and Michael D. Schroeder. “The protection of information in computer systems”. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939 (cit. on pp. 103, 120, 141).

[Sch11] Fred B. Schneider. *Blueprint for a Science of Cybersecurity*. Tech. rep. https://www.cs.cornell.edu/fbs/publications/SoS.blueprint.pdf. Ithaca, New York 14853: Department of Computer Science, Cornell University, 2011 (cit. on p. 102).

[Sch+23] Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. “Java Bytecode Normalization for Code Similarity Analysis”. submitted to IEEE Transactions on Software Engineering. 2023 (cit. on pp. 67, 68).

[SE13] Widura Schwittek and Stefan Eicker. “A Study on Third Party Component Reuse in Java Enterprise Open Source Software”. In: *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*. CBSE ’13. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2013, pp. 75–80. DOI: 10.1145/2465449.2465468 (cit. on pp. 1, 64).

[SFZ10] Gehan M.K. Selim, King Chun Foo, and Ying Zou. “Enhancing Source-Based Clone Detection Using Intermediate Representation”. In: *2010 17th Working Conference on Reverse Engineering*. IEEE, Oct. 2010, pp. 227–236. DOI: 10.1109/WCRE.2010.33 (cit. on p. 67).

[STM10] Joseph Siefers, Gang Tan, and Greg Morrisett. “Robusta: Taming the Native Beast of the JVM”. In: *Proceedings of the 17th ACM conference on Computer and communications security*. New York, New York, USA: ACM, Oct. 2010, pp. 201–211. DOI: 10.1145/1866307.1866331 (cit. on pp. 3, 142).

[Sma+15] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. “More Sound Static Handling of Java Reflection”. In: *Programming Languages and Systems*. Ed. by Xinyu Feng and Sungwoo Park. Cham: Springer International Publishing, 2015, pp. 485–503. DOI: 10.1007/978-3-319-26529-2_26 (cit. on pp. 108, 126).

[SB11] Yannis Smaragdakis and Martin Bravenboer. "Using Datalog for Fast and Easy Program Analysis". In: *Proceedings of the First International Conference on Datalog Reloaded*. Datalog'10. Oxford, UK: Springer Berlin Heidelberg, 2011, pp. 245–251. DOI: 10.1007/978-3-642-24206-9_14 (cit. on pp. 103, 114, 115, 120).

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. "Pick Your Contexts Well: Understanding Object-sensitivity". In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '11. Austin, Texas, USA: ACM, Jan. 2011, pp. 17–30. DOI: 10.1145/1926385.1926390 (cit. on pp. 114, 115).

[SKB14] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. "Introspective Analysis: Context-sensitivity, Across the Board". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, June 2014, pp. 485–495. DOI: 10.1145/2594291.2594320 (cit. on pp. 114, 115).

[Spä+16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java". en. In: Leibniz International Proceedings in Informatics (LIPIcs) 56 (2016). Ed. by Shriram Krishnamurthi and Benjamin S. Lerner, pp. 1–26. DOI: 10.4230/LIPIcs.ECOOP.2016.22 (cit. on p. 127).

[ST12] Mengtao Sun and Gang Tan. "JVM-Portable Sandboxing of Java's Native Libraries". In: *Computer Security – ESORICS 2012*. Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 842–858. DOI: 10.1007/978-3-642-33167-1_48 (cit. on pp. 141–143).

[ST14] Mengtao Sun and Gang Tan. "NativeGuard: Protecting Android Applications from Third-Party Native Libraries". In: *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, July 2014, pp. 165–176. DOI: 10.1145/2627393.2627396 (cit. on pp. 142, 143).

[Tan+23] Wei Tang, Zhengzi Xu, Chengwei Liu, et al. "Towards Understanding Third-Party Library Dependency in C/C++ Ecosystem". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3551349.3560432 (cit. on pp. 8, 16).

[Tem+10] Ewan Tempero, Craig Anslow, Jens Dietrich, et al. "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies". In: *2010 Asia Pacific Software Engineering Conference*. IEEE, Nov. 2010, pp. 336–345. DOI: 10.1109/APSEC.2010.46 (cit. on p. 65).

[Val+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, et al. "Soot: A Java Bytecode Optimization Framework". In: *CASCON First Decade High Impact Papers on - CASCON '10*. CASCON '10. Toronto, Ontario, Canada: ACM Press, 2010, pp. 214–224. DOI: 10.1145/1925805.1925818 (cit. on p. 77).

[VH98] Raja Vallée-Rai and Laurie Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. Tech. rep. http://www.sable.mcgill.ca/publications/techreports/sable-tr-1998-4.ps. Montreal, Canada: McGill University, 1998, pp. 1–15 (cit. on pp. 54, 58, 60).

[VR01] Frédéric Vivien and Martin Rinard. "Incrementalized Pointer and Escape Analysis". In: *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. PLDI '01. Snowbird, Utah, USA: ACM, May 2001, pp. 35–46. DOI: 10.1145/378795.378804 (cit. on p. 145).

[Wah+93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient software-based fault isolation". In: *ACM SIGOPS Operating Systems Review* 27.5 (Dec. 1993), pp. 203–216. DOI: 10.1145/173668.168635 (cit. on p. 142).

[WF98] Dan S. Wallach and Edward W. Felten. "Understanding Java stack inspection". In: *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*. IEEE. IEEE Comput. Soc, 1998, pp. 52–63. DOI: 10.1109/SECPRI.1998.674823 (cit. on p. 107).

[Wan+20] Ying Wang, Bihuan Chen, Kaifeng Huang, et al. "An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2020, pp. 35–45. DOI: 10.1109/ICSME46990.2020.00014 (cit. on pp. 2–4, 7).

[Wan+22] Ying Wang, Rongxin Wu, Chao Wang, et al. "Will Dependency Conflicts Affect My Program's Semantics?" In: *IEEE Transactions on Software Engineering* 48.7 (July 2022), pp. 2295–2316. DOI: 10.1109/TSE.2021.3057767 (cit. on pp. 71, 74, 91, 97).

[WR99] John Whaley and Martin Rinard. "Compositional Pointer and Escape Analysis for Java Programs". In: *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '99. Denver, Colorado, USA: ACM, Oct. 1999, pp. 187–206. DOI: 10.1145/320384.320400 (cit. on pp. 127, 145).

[WD14] Jeff Williams and Arshan Dabirsiaghi. *The unfortunate reality of insecure libraries*. Tech. rep. https://cdn2.hubspot.net/hub/203759/file-1100864196-pdf/docs/contrast_-_insecure_libraries_2014.pdf. Contrast Security, 2014 (cit. on pp. 46, 47, 64, 87).

[WDD22] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. "What the Fork? Finding Hidden Code Clones in Npm". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2415–2426. DOI: 10.1145/3510003.3510168 (cit. on pp. 7, 8).

[Yee+10] Bennet Yee, David Sehr, Gregory Dardyk, et al. "Native Client: A Sandbox for Portable, Untrusted X86 Native Code". In: *Commun. ACM* 53.1 (Jan. 2010), pp. 91–99. DOI: 10.1145/1629175.1629203 (cit. on p. 142).

[Yip+09] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. “Improving Application Security with Data Flow Assertions”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP ’09. Big Sky, Montana, USA: ACM, Oct. 2009, pp. 291–304. DOI: 10.1145/1629575.1629604 (cit. on pp. 146, 147).

[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. DOI: 10.1109/ACCESS.2019.2898411 (cit. on p. 66).

Webpages

[Apa15] Apache Software Foundation. *Tomcat tc8.0.x SVN Repository*. 2015. URL: https://svn.apache.org/viewvc/tomcat/tc8.0.x/trunk/java/org/apache/catalina/authenticator/FormAuthenticator.java?r1=1785776&r2=1785775&pathrev=1785776 (visited on July 28, 2023) (cit. on p. 137).

[Apa22] Apache Software Foundation. *Ant*. 2022. URL: https://ant.apache.org/ (visited on June 28, 2023) (cit. on p. 11).

[Apa23a] Apache Software Foundation. *Introduction to the Dependency Mechanism*. 2023. URL: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html (visited on Oct. 26, 2023) (cit. on pp. 14, 79).

[Apa23b] Apache Software Foundation. *Maven*. 2023. URL: https://maven.apache.org (visited on June 8, 2023) (cit. on pp. 11, 15).

[Cau17] Adam Caudil. *Exploiting the Jackson RCE: CVE-2017-7525*. 2017. URL: https://adamcaudill.com/2017/10/04/exploiting-jackson-rce-cve-2017-7525/ (visited on June 22, 2023) (cit. on pp. 1, 101).

[Ecl07] Eclipse Foundation. *Evolving Java-based APIs*. 2007. URL: https://wiki.eclipse.org/Evolving_Java-based_APIs (visited on May 5, 2023) (cit. on pp. 73, 74).

[Ecl20] Eclipse Foundation. *Eclipse Steady*. 2020. URL: https://projects.eclipse.org/projects/technology.steady (visited on July 22, 2023) (cit. on pp. 3, 48).

[Fer22] Stefan Ferstl. *depgraph-maven-plugin*. 2022. URL: https://github.com/ferstl/depgraph-maven-plugin (visited on Aug. 26, 2023) (cit. on p. 77).

[For17] Forbes. *Equifax*. Feb. 2017. URL: https://www.forbes.com/sites/thomasbrewster/2017/09/14/equifax-hack-the-result-of-patched-vulnerability/ (visited on Feb. 20, 2023) (cit. on p. 2).

[Git23] Github Repository. *A Proof-Of-Concept for the CVE-2021-44228 vulnerability*. 2023. URL: https://github.com/kozmer/log4j-shell-poc (visited on June 28, 2023) (cit. on p. 140).

[Git20] GitHub, Inc. *Security Alerts*. 2020. URL: https://help.github.com/articles/about-security-alerts-for-vulnerable-dependencies/ (visited on Feb. 20, 2023) (cit. on pp. 3, 21).

[Git22] GitHub, Inc. *DependaBot*. 2022. URL: https://DependaBot2022.com/ (visited on July 26, 2023) (cit. on pp. 4, 78).

[Gra23] Gradle, Inc. *Gradle*. 2023. URL: https://gradle.org/ (visited on June 8, 2023) (cit. on p. 11).

[HG21] Joseph Hejderup and Georgios Gousios. *Dataset. Can We Trust Tests To Automate Dependency Updates? A Case Study of Java Projects*. 2021. URL: https://zenodo.org/record/4479015 (visited on July 26, 2023) (cit. on p. 87).

[IBM06] IBM T.J. Watson Research Center. *Watson Libraries for Analysis (WALA)*. 2006. URL: http://wala.sourceforge.net/wiki/index.php (visited on Jan. 2, 2022) (cit. on p. 146).

[JFr23] JFrog Ltd. *Conan, software package manager for C and C++ developers*. 2023. URL: https://conan.io/ (visited on June 28, 2023) (cit. on p. 16).

[Jun21] Juniper Networks, Inc. *Apache Log4j Vulnerability CVE-2021-44228 Raises widespread Concerns*. 2021. URL: https://blogs.juniper.net/en-us/security/apache-log4j-vulnerability-cve-2021-44228-raises-widespread-concerns (visited on June 28, 2023) (cit. on p. 139).

[Kit23] Kitware Inc. *CMake*. 2023. URL: https://cmake.org/ (visited on Oct. 26, 2023) (cit. on p. 16).

[Kre18] Brian Krebs. *Equifax Breach*. 2019-03-16. 2018. URL: https://krebsonsecurity.com/tag/equifax-breach/ (visited on Feb. 20, 2023) (cit. on pp. 1, 2).

[Liv12] Benjamin Livshits. *SecuriBench*. 2012. URL: https://suif.stanford.edu/~livshits/securibench/ (visited on June 22, 2023) (cit. on p. 65).

[Men23a] Mend.io. Feb. 2023. URL: https://mend.io (visited on Apr. 20, 2023) (cit. on pp. 3, 22).

[Men23b] Mend.io. *Renovate*. 2023. URL: https://www.mend.io/Mend.io2023b/ (visited on Aug. 26, 2022) (cit. on p. 4).

[Mvn20] MvnRepository. *100 Popular Projects*. 2020. URL: https://mvnrepository.com/popular (visited on Feb. 20, 2023) (cit. on pp. 24–26).

[Neo23] Neo4j, Inc. *Cypher Query Language*. 2023. URL: https://neo4j.com/developer/cypher/ (visited on July 20, 2023) (cit. on p. 82).

[NIS17] NIST. *Juliet Test Suite*. 2017. URL: https://samate.nist.gov/SARD/testsuite.php (visited on Feb. 20, 2020) (cit. on p. 65).

[NIS18] NIST. *SAMATE - Software Assurance Metrics And Tool Evaluation*. 2018. URL: https://samate.nist.gov/ (visited on Feb. 20, 2023) (cit. on p. 65).

[NIS20] NIST. *NVD*. 2020. URL: https://nvd.nist.gov/ (visited on Feb. 20, 2020) (cit. on pp. 20, 27, 34, 35, 48, 65).

[NSA12] NSA. *Defense in Depth: A practical strategy for achieving Information Assurance in today's highly networked environments*. 2012. URL: https://gtldresult.icann.org/applicationstatus/applicationdetails:downloadattachment/131817?t:ac=13 (cit. on pp. 102, 138).

[NVD17a] NVD. *CVE-2017-5638*. Feb. 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-5638 (visited on Feb. 20, 2023) (cit. on p. 2).

[NVD17b] NVD. *CVE-2017-5648*. Apr. 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-5648 (visited on Oct. 20, 2023) (cit. on p. 136).

[NVD21] NVD. *CVE-2021-44228*. 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-44228 (visited on Apr. 18, 2023) (cit. on pp. 1, 2, 4, 69, 101, 139).

[NVD22] NVD. *CVE-2022-33980*. Feb. 2022. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-33980 (visited on May 20, 2023) (cit. on p. 1).

[Off20] OffSec Services. *Exploit Database*. 2020. URL: https://www.exploit-db.com/ (visited on Feb. 20, 2023) (cit. on p. 65).

[Ora15a] Oracle Cooperation. *JavaSE Specification*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/html/ (visited on Aug. 26, 2023) (cit. on pp. 73, 108, 128).

[Ora17a] Oracle Cooperation. *Java Platform, Standard Edition Tools Reference - jdeps*. Sept. 2017. URL: https://docs.oracle.com/javase/9/tools/jdeps.htm (visited on Oct. 26, 2023) (cit. on p. 130).

[Ora18] Oracle Cooperation. *JDK 11 - JavaDoc ModuleLayer*. Sept. 2018. URL: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/ModuleLayer.html (visited on Oct. 26, 2023) (cit. on p. 110).

[Ora23a] Oracle Cooperation. *GraalVM*. 2023. URL: https://www.graalvm.org/ (visited on June 28, 2023) (cit. on p. 143).

[Ora96] Oracle Corporation. *The Java Language Environment - A White Paper*. 1996. URL: https://www.oracle.com/java/technologies/language-environment.html (visited on June 2, 2023) (cit. on pp. 101, 105, 106, 108).

[Ora14a] Oracle Corporation. *JEP 261: Module System*. 2014. URL: http://openjdk.java.net/jeps/261 (visited on May 5, 2023) (cit. on p. 110).

[Ora14b] Oracle Corporation. *Project Jigsaw*. 2014. URL: http://openjdk.java.net/projects/jigsaw/ (visited on Nov. 2, 2022) (cit. on pp. 103, 110).

[Ora14c] Oracle Corporation. *SigTest User Guide*. 2014. URL: https://docs.oracle.com/javacomponents/sigtest-3-1/user-guide/index.html (visited on Aug. 26, 2022) (cit. on pp. 86, 89).

[Ora15b] Oracle Corporation. *JEP 260: Encapsulate Most Internal APIs*. 2015. URL: http://openjdk.java.net/jeps/260 (visited on Nov. 2, 2017) (cit. on pp. 7, 102, 108, 110).

[Ora17b] Oracle Corporation. *JEP 200: The Modular JDK*. 2017. URL: https://openjdk.java.net/jeps/200 (visited on Apr. 28, 2019) (cit. on pp. 102, 112).

[Ora21] Oracle Corporation. *Security and Sandboxing Post SecurityManager*. 2021. URL: https://inside.java/2021/04/23/security-and-sandboxing-post-securitymanager/ (visited on June 2, 2023) (cit. on pp. 5, 102, 120, 138, 141).

[Ora22] Oracle Corporation. *JEP 411: Deprecate the Security Manager for Removal*. 2022. URL: https://openjdk.org/jeps/411 (visited on June 2, 2023) (cit. on pp. 5, 102, 120, 138).

[Ora23b] Oracle Corporation. *JEP draft: Integrity and Strong Encapsulation*. 2023. URL: https://openjdk.org/jeps/8305968 (visited on June 2, 2023) (cit. on pp. 5, 7, 102, 138).

[Ora23c] Oracle Corporation. *Polyglot Sandboxing*. 2023. URL: https://www.graalvm.org/dev/security-guide/polyglot-sandbox/ (visited on June 28, 2023) (cit. on pp. 142, 143).

[Ora23d] Oracle Corporation. *The Java programming language Compiler Group*. 2023. URL: http://openjdk.java.net/groups/compiler/ (visited on May 2, 2023) (cit. on p. 55).

[OSG11] OSGi Alliance. *What You Should Know about Class Loaders*. May 2011. URL: https://blog.osgi.org/2011/05/what-you-should-know-about-class.html (visited on Oct. 28, 2023) (cit. on p. 14).

[OWA20] OWASP. *OWASP Dependency-Check*. 2020. URL: https://owasp.org/www-project-dependency-check/ (visited on June 20, 2023) (cit. on p. 3).

[Pip23] Pip Documentation. *Dependency Resolution*. 2023. URL: https://pip.pypa.io/en/stable/topics/dependency-resolution/ (visited on July 20, 2023) (cit. on p. 15).

[Piv20] Pivotal Software. *CVE-2018-1271*. Dec. 2020. URL: https://pivotal.io/security/cve-2018-1271 (visited on Dec. 28, 2022) (cit. on p. 21).

[Pre21] Tom Preston-Werner. *Semantic Versioning 2.0.0*. 2021. URL: https://semver.org/lang/de/ (visited on July 20, 2023) (cit. on pp. 15, 16, 73, 97).

[Pro04] Prof. Robert H. (Bob) Sloan - University Illion. *Java Example Program*. 2004. URL: https://www.cs.uic.edu/~sloan/CLASSES/java/ (visited on Mar. 16, 2023) (cit. on pp. 61, 62).

[SAP20] SAP. *Vulnerability Assessment Knowledge Base*. 2020. URL: https://github.com/SAP/project-kb (visited on June 20, 2023) (cit. on p. 27).

[Sny23] Snyk Limited. 2023. URL: https://snyk.io/ (visited on Apr. 28, 2023) (cit. on pp. 3, 22).

[Son20] Sonatype. *Central download statistics for OSS projects*. 2020. URL: https://blog.sonatype.com/2010/12/now-available-central-download-statistics-for-oss-projects/ (visited on Sept. 20, 2019) (cit. on p. 11).

[Son22] Sonatype. *Maven Central*. 2022. URL: https://search.maven.org/stats (visited on Dec. 4, 2022) (cit. on p. 11).

[Sou20] SourceClear. *Evaluation Framework for Dependency Analysis*. 2020. URL: https://github.com/srcclr/efda (visited on Feb. 20, 2023) (cit. on p. 65).

[Syn23] Synopsys. Feb. 2023. URL: https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html (visited on Feb. 20, 2021) (cit. on pp. 3, 22, 64).

[The22] The Neighbourhoodie Software GmbH. *Greenkeeper*. 2022. URL: https://greenkeeper.io/ (visited on Aug. 26, 2022) (cit. on p. 4).

[Tul22] Jaroslav Tulach. *SigTest GitHub Repository*. 2022. URL: https://github.com/jtulach/netbeans-apitest (visited on Aug. 26, 2022) (cit. on pp. 73, 82, 86, 89).

[Wim19] Christian Wimmer. *Isolates and Compressed References: More Flexible and Efficient Memory Management via GraalVM*. Jan. 2019. URL: https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67e (visited on Oct. 26, 2023) (cit. on p. 143).

[Yah21] Yahoo. *Apple iCloud, Twitter and Minecraft vulnerable to ubiquitous zero-day flaw*. 2021. URL: https://techcrunch.com/2021/12/10/apple-icloud-twitter-and-minecraft-vulnerable-to-ubiquitous-zero-day-exploit/ (visited on June 26, 2023) (cit. on p. 2).

List of Figures

2.1 Example of a Maven dependency graph containing a duplicate dependency. 13

3.1 To understand how the open-source community also uses the selected sample, this graph reports the #usages of the 723 artifacts (GAVs) as reported by `mvnrepository.com`, showing a log-normal distribution (X-axis has logarithmic scale). 26

3.2 To understand to what extent and how developers include open-source artifacts, this graph reports: the average ratio of OSS to proprietary dependencies, the ratio of direct to transitive dependencies, and the scopes developers use; grouped by groupId:artifactId (GA). 29

3.3 Achilles' process steps for evaluating OSS-vulnerability scanners. 52

3.4 Feature diagram of the options that the Achilles generator provides for re-creating the modifications on JAR files. By default, the features in bold are selected, generating unmodified JAR files. 53

3.5 Layout of Achilles 53

3.6 SootDiff's process steps for comparing the bytecode of two classes using Jimple and optimizers to be resilient against bytecode modifications. Soot's optimizers are in gray. 59

4.1 The options for updating the transitive dependency t and resulting changes in the dependency graph. 72

4.2 UpCy's process steps for computing compatible updates using min-(s,t)-cut algorithm and a dependency graph of the Maven Central repository. . 76

4.3 Example min-(s,t)-cuts computed by UpCy on the unified dependency graph for updating t with minimal incompatibilities. Each edge represents a source or a binary compatibility relation between the connected artifacts. 78

4.4 To understand to what extent an update introduces incompatibilities, this violin plot shows the amount of binary (left) and semantic (right) incompatibilities (Y-axis has a logarithmic scale), showing on average 83 binary incompatibilities and two peaks at 0 and 127 for semantic incompatibilities. 90

4.5 Example of min-(s,t)-cut computed by UpCy in project `mybatis-shards` to find compatible updates for `cglib`, suggesting to update the artifacts `cglib` and `asm`. 93

5.1 Overview of the components of the Java Runtime Environment. 105
5.2 Subset of the module graph of the JRE module java.desktop. The solid arrows represent a requires dependency between modules. The dashed arrows show which packages a module can access. Only exported packages can be accessed by the dependent module. Internal packages (in gray) cannot be accessed. 109
5.3 ModGuard's process steps for identifying escapes of sensitive entities using the Doop framework and Datalog rules. Gray steps are executed in Soot and white steps are executed in Doop. 121
5.4 Scenarios leading to the escape of sensitive entities. 128
5.5 Layout of MIC9Bench 129
5.6 For modularizing Tomcat 8.5.21, we created a module per JAR. The figure shows the resulting module graph. In the strict modularization (all exports not required for compilation removed), the white modules do not export any packages. The dashed modules have the same exports in the naïve and strict modularization. 130
5.7 Example escape of the sensitive methods of `DeltaSession` from the module `catalina.ha` in the strict modularization of Tomcat. External code can invoke `SimpleTcpCluster.getManagerTemplate()` to receive an instance of `DeltaManager`, which leaks the sensitive `DeltaSession` via the method `createEmptySession()`. The gray types are module-internal, the white types are exported; in bold is the module declaring the type. 135
5.8 Overview on how to exploit the Log4Shell vulnerability. Based on [Jun21].139

List of Tables

3.1 The selected sample set of the 20 most-used artifacts in the 7,024 projects developed at SAP, grouped by groupId:artifactId (groupId:artifactId (GA)). The table shows how many different projects use that artifact and how popular—based on its usage—the artifact is on Maven Central. 25

3.2 To identify known vulnerabilities in our sample set of the 20 most-used dependencies (723 artifacts), we used the scanners Steady, OWASP Dependency-Check (DepCheck), and C3. The table gives an overview of the findings that the scanners reported. Highlighted are cases in which the artifact itself is reported as vulnerable. 31

3.3 To understand how prevalent the modifications are on Maven Central, this table reports how often the identified 254 vulnerable classes were subject to the modifications and in how many different artifacts they occurred. . 37

3.4 Test cases for evaluating the precision and recall of vulnerability scanners. The column on the right shows which scanners reported the vulnerability in *RQ2*. 40

3.5 To understand how the modifications impact the vulnerability scanners' effectiveness, this table reports the vulnerability scanners' precision, recall and F1-score for type 1–4 modifications on the test cases. The scanners marked with * were used in the construction of the test cases; in bold the highest score. 43

3.6 The table shows the artifacts that the different scanners failed detect in the scenarios in the form `artifact:version`. In the unmodified scenario the scanners detect the artifacts and the versions from the test fixtures correctly (cf. Table 3.4). The scanners marked with * were used in the construction of the test cases. 44

3.7 Evaluation of the efficiency of bytecode and SootDiff's comparison on bytecode generated by different Java compilers. 62

4.1 The table reports how many updates failed during the build process caused by source-code incompatibilities or during test execution, indicating binary or semantic incompatibilities. 88

4.2 The table shows the number of compatibilities an update has to satisfy due to blossoms, conflicting or duplicate dependencies, and binary-dependent artifacts. 91
4.3 The table compares the efficiency of naïve and UpCy update in terms of the number of incompatibilities the update introduces. 92

5.1 Evaluation of ModGuard on MIC9Bench for detecting escaping sensitive entities over a module's API and different Java language features. Integrity violations occur if external code can change the value of a sensitive field, and confidentiality violations occur if external code can access a sensitive entity. 132
5.2 The table shows the number of escaping of sensitive entities that ModGuard identifies in the modularization of Tomcat 8.5.21. 134

Listings

2.1 Example declaration of an artifact as a dependency in a project's `pom.xml` for the build-automation and dependency-management tool Maven. . . . 12
2.2 Example declaration of artifacts with version constraints as dependencies in a Python project's `Pipfile` for the tool pip. 15
2.3 Example declaration of artifacts with version constraints as dependencies in a `package.json` for the tool npm. 16
2.4 Example declaration of artifacts as dependencies in a `conanfile.txt` for the tool Conan. 16

3.1 Example test fixture of Achilles for the artifact jackson-dataformat-xml:2.4.3 and the vulnerability CVE-2016-3720. The test fixture contains the result of the manual classification (*vulnerable*), and—if available—information about the vulnerable classes and the commit fixing the vulnerability. . . . 51
3.2 Example source class `Point2d.java`. 55
3.3 Comparison of the (decompiled) bytecode generated with ECJ compiler and Javac, both with target version 1.8, from the source class `Point2d.java`. References to the constant pool are resolved and typeset as comments in green. 56
3.4 Comparison of the Jimple code parsed from the bytecode generated with ECJ compiler and Javac, both with target version 1.8, from the source class `Point2d.java`. 57
3.5 Comparison of compiler optimizations of Javac and ECJ from the source class `DivZero.java`. Javac and ECJ optimize unused locals differently, resulting in bytecode dissimilarities. 63

4.1 Cypher Query for checking if a version greater than or equal to $targetVersion$ of the artifact to update exists. 83
4.2 Cypher Query for checking if more recent versions of the artifacts that are potential update candidates as computed by the min-(s,t)-cut (the root nodes of sink partition T) exist. 84
4.3 Cypher Query for finding compatible versions of the artifacts u and x. Compatible artifacts depend on the same version of the shared dependency w, avoiding the introduction of conflicts and binary incompatibilities. 85

4.4 Cypher Query for getting an artifact r and all of its (transitive) dependencies (the update-graph). 85

5.1 Example of the module-descriptor `module-info` of the module `java.desktop` of the Java Runtime Environment 1.9. 109
5.2 Example of the sensitive field `SecretKey.keyMaterial` escaping the module-internal package `internal` through the exported, overridden method `getKey()` of superclass `Key`. In green exported types and methods, in yellow internal types. Marked in red the sensitive field `keyMaterial`. . . . 111
5.3 Module entry-point model: domain, input, and output relations. Doop's [SKB14; SB11] default rules are gray. 115
5.4 Datalog rules for detecting explicit and implicit entry points of a module, constituting the module's API. 119
5.5 ModGuard Escape Analysis: domain, input, and output relations. 123
5.6 ModGuard's Datalog rules for detecting escaping fields, methods, and classes. 125
5.7 Excerpt of the fix for CVE-2017-5648 in class `FormAuthenticator.java` in Tomcat revision 1785776 [Apa15]. 137
5.8 Example exploit for the Log4Shell vulnerability, opening a reverse shell. Taken from [Git23]. 140