Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Ready-to-Update: A Practical Approach for Mapping and Prioritizing Vulnerability Remediation of Third-Party Libraries

Wagner Emanuel S. Ferreira

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Ready-to-Update: A Practical Approach for Mapping and Prioritizing Vulnerability Remediation of Third-Party Libraries

Wagner Emanuel S. Ferreira

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Brasília, 20 de Setembro de 2025

Dedicatória

Dedico este trabalho para a Mariana, inspiração constante na busca por uma versão melhor de mim mesmo.

Agradecimentos

Quero agradecer primeiramente à minha família, por todo o apoio e suporte durante todo o percurso até aqui: à Mariana e Luisa pela compreensão nas minhas ausências - física ou psicológica - durante as aulas, trabalhos, pesquisas e por acreditar em mim, mesmo nos momentos que eu mesmo duvidei. Aos meus pais, por me fazerem acreditar na capacidade da educação de nos levar mais longe.

Quero agradecer também a todo o corpo da UnB que possibilitou de alguma forma que chegássemos a esse momento, desde o administrativo até os professores que contribuíram de forma primordial para o meu crescimento durante o processo, especialmente o meu Orientador, Professor Rodrigo Bonifácio, que desde o início acreditou na importância dessa pesquisa e com muita paciência e sabedoria guiou todo o processo de pesquisa e escrita. Gostaria de agradecer ainda ao Laerte Peotta pelos direcionamento ao longo da pesquisa e ao Luis Henrique Vieira Amaral pelas inúmeras discussões sobre o tema e por sempre acreditar no potencial desse trabalho.

Gostaria de agradecer, ainda, à Joanna C. S. Santos, em nome do quem agradeço aos diversos autores e acadêmicos com quem discutimos e contribuíram de alguma forma para o aprofundamento do estudo e melhorias do trabalho ao longo do processo.

Quero agradecer, por fim, aos meus gestores que tornaram factível participar da vida acadêmica quando, não muito raramente, os horários se chocavam com aquele esperado pela vida profissional. Luciano Lara, Charlene Soares e Carolina Beghelli, sem vocês não teria sido possível chegar até o final desse trabalho.

Ready-to-Update
Uma Abordagem para Mapear e Priorizar Vulnerabilidades Remediáveis em Bibliotecas Externas

Resumo

A disseminação do reuso de código aberto acelera o desenvolvimento de software, embora introduza riscos substanciais à segurança. Apesar das diversas abordagens acadêmicas para identificação e remediação de dependências vulneráveis, a ausência de soluções universais exige adaptações aos contextos organizacionais específicos. Esta dissertação apresenta e avalia o *Ready-to-Update*, uma metodologia concebida para priorizar e identificar vulnerabilidades conhecidas que demandam correção imediata. Desenvolvido a partir de um estudo de caso em uma instituição financeira de grande porte (**FinCompany**), o *Ready-to-Update* utiliza-se de métodos estabelecidos adaptados ao grau de maturidade de segurança e às necessidades da **FinCompany**. A metodologia estrutura-se em quatro etapas: (i) identificação de dependências; (ii) detecção de vulnerabilidades; (iii) localização de versões atualizadas; e (iv) avaliação de compatibilidade. A eficácia da proposta foi demonstrada por meio da análise de 2.140 artefatos Java, correspondentes a 585 projetos em produção na **FinCompany** entre janeiro e julho de 2024. Os resultados revelam uma prevalência crítica de vulnerabilidades, afetando 99,53% dos projetos analisados. Constatou-se que 92,40% das dependências diretas possuíam versões recentes, das quais 65,62% eram seguras. A atualização desses componentes permitiria reduzir as vulnerabilidades diretas em 98,63% dos projetos. Testes de compatibilidade binária e semântica confirmaram que a remediação é viável apenas via atualização de versão para 86,3% das dependências diretas. Em suma, esta dissertação evidencia que o *Ready-to-Update* possibilita uma abordagem estratégica para a remediação de vulnerabilidades, ao permitir que os desenvolvedores concentrem seus esforços em dependências que são diretamente gerenciáveis e para as quais existem alternativas mais seguras, o que, em última análise, leva a uma resolução mais rápida e eficaz das vulnerabilidades identificadas.

Palavras-chave: segurança de software, dependências vulneráveis, bibliotecas open source, gestão de vulnerabilidades, propagação de vulnerabilidades, segurança na cadeia de suprimentos de software

Abstract

The widespread adoption of open-source reuse accelerates software development, although it introduces substantial security risks. Despite various academic approaches for identifying and remediating vulnerable dependencies, the lack of universal solutions necessitates adaptations to specific organizational contexts. This dissertation presents and evaluates *Ready-to-Update*, a methodology designed to prioritize and identify known vulnerabilities requiring immediate correction. Developed from a case study at a large financial institution (**FinCompany**), *Ready-to-Update* employs established methods adapted to the organization's security maturity and requirements. The methodology is structured into four stages: (i) dependency identification; (ii) vulnerability detection; (iii) identification of updated versions; and (iv) compatibility assessment. The effectiveness of the proposal was demonstrated through the analysis of 2,140 Java artifacts, corresponding to 585 production projects at **FinCompany** between January and July 2024. Results reveal a critical prevalence of vulnerabilities, affecting 99.53% of the analyzed projects. It was found that 92.40% of direct dependencies had recent versions available, of which 65.62% were secure. Updating these components would reduce direct vulnerabilities in 98.63% of projects. Binary and semantic compatibility tests confirmed that remediation is feasible solely via version updates for 86.3% of direct dependencies. In summary, this dissertation demonstrates that *Ready-to-Update* enables a strategic approach to vulnerability remediation by allowing developers to focus efforts on directly manageable dependencies with safer alternatives, ultimately leading to a more efficient resolution of identified risks.

Keywords: software security, vulnerable dependencies, open-source libraries, vulnerability management, vulnerability propagation, security software supply chain

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software reuse via open-source component utilization is a prevalent practice attributed to factors including time and resource efficiency, expedited development cycles, and enhanced final software quality, as developers can concentrate on their domain-specific expertise. Nevertheless, deploying open-source components necessitates rigorous management, given their potential to introduce quality deficiencies that may adversely affect the final program, exemplified by security vulnerabilities.

The LOG4SHELL vulnerability [1] exemplifies the potential of issues introduced through reliance on external libraries. The vulnerability, affecting Java systems using particular Log4J library versions, a widely adopted Java logging framework, underscored the potential for widespread system compromise. The ensuing discourse among security experts, researchers, and news media emphasized the broad impact on projects, including Java application servers (e.g., Apache Tomcat and WildFly) and development frameworks (e.g., Spring Boot). To examine the practical implications of this scenario within an industry context, this thesis investigates the impact of vulnerable libraries within a large Brazilian financial institution (hereafter designated as **FinCompany**), which employs over 87,000 personnel. A preliminary investigation revealed that **FinCompany** lacked a strategy for managing external dependencies and the potential vulnerabilities inherent in such dependencies, a scenario that can also be observed in other organizations [2, 3].

A set of requirements related to this research topic has been identified from both the company and developer perspectives, with several of them being mutually relevant. From the **FinCompany** standpoint, the organization needs mechanisms to catalog the dependencies it employs and to monitor associated vulnerabilities. It also requires a systematic process to prioritize which vulnerabilities should be addressed first, balancing necessity and feasibility. From the developers' perspective, there is a need for clear visibility into project-specific dependency information—often aligned with the corporate view—as well as guidance in performing essential tasks such as selecting appropriate dependency versions and evaluating the potential impact of integrating updated versions into the project.

1.1 Problem statement

The scenario discussed in the previous section underscores the critical challenges addressed by *software supply chain security* [4], particularly the risks associated with relying on external libraries, and reinforces the need for robust dependency management practices.

In this context, it is essential to map the dependencies used by **FinCompany**, identify their associated vulnerabilities, and determine which of these vulnerabilities truly matter. Pashchenko et al. [5] define vulnerabilities that "matter" as those that directly enable developer remediation. Achieving this requires identifying and categorizing dependencies that are no longer actively maintained—both of which are critical for accomplishing this goal.

However, based on the situation observed at **FinCompany**, we found that the notion of "what matters" may vary across organizations, depending on factors such as company size, system complexity, and the maturity of development and security practices.

Given the scale of **FinCompany** 's systems — 585 projects were deployed at least once between January 1 and July 31, 2024 — and the absence of a prior process for monitoring vulnerable dependencies, we hypothesized that a substantial number of such dependencies would be discovered. Initially, prioritizing *quick wins* — defined here as maximizing the number of vulnerable dependencies with straightforward remediation paths — could prove more effective for **FinCompany**.

This strategy aims to achieve a broader reduction in overall exposure, enabling both development and strategy teams to plan more comprehensive interventions in the future, particularly for projects that depend on libraries without currently available secure versions.

1.2 Research objectives

Drawing upon this concept of "what matters", the primary objective of this study is **to develop a solution that provides FinCompany with the necessary insights to maximize the remediation of dependency vulnerabilities in a secure and rapid manner**. The research is guided by the following specific objectives:

- To catalog all vulnerable dependencies;
- To delineate which dependencies are eligible for transition to secure versions;

- To ascertain the compatibility of these secure versions with existing projects.

We present and evaluate *Ready-to-Update*, a technique designed to estimate how vulnerabilities propagate through libraries and to determine which vulnerabilities should be prioritized. *Ready-to-Update* builds upon the approach proposed by Pashchenko et al. [5], focusing on identifying vulnerabilities in dependencies that developers can remediate immediately and therefore consider more critical.

Furthermore, *Ready-to-Update* integrates design principles from the work of Dann, Hermann, and Bodden [6], which emphasizes determining valid update sequences that allow a dependency to reach a target version while minimizing incompatibilities with other libraries. Their study also investigates the broader implications of dependency updates, assessing whether updated versions preserve project compatibility, support successful recompilation, and ensure the correct execution of project tests.

To investigate both aspects of *Ready-to-Update* — identifying vulnerable libraries and suggesting update paths toward secure versions — we conducted two studies, which are outlined below and built upon two distinct modules, detailed in the Chapter 3: the first module resolves dependencies and iteratively identifies vulnerabilities, checking for and verifying the security of new versions, while the second module assesses the compatibility of the latest and most secure versions of previously identified vulnerable dependencies.

1.2.1 First Study: Identification of Vulnerable Dependencies

In the first study we apply *Ready-to-Update* to estimate how Java libraries propagate vulnerabilities across 585 enterprise systems in use by **FinCompany**. Our analysis aims to pinpoint **Vulnerable, Updatable, Direct Dependencies** (**VUDDs**), which are defined as direct dependencies of the analyzed systems that contain recognized vulnerabilities either in their direct or transitive dependencies and for which newer, more secure versions are available. These systems were deployed at least once between January 2024 and July 2024. Accordingly, in the first study we investigate the following research questions:

First Research Question (RQ1): What are the characteristics of the vulnerabilities discovered in the **FinCompany**'s project dependencies?

Second Research Question (RQ2): What is the impact of fixing VUDDs on the overall number of projects vulnerabilities?

The first results of this research bring the following contributions. First, *Ready-to-Update*, a novel approach for identifying and characterizing vulnerable dependencies, is designed to ensure scalability and meet the specific needs of sector organizations. Second, we present the results of an empirical study that offers a fresh perspective on an industry scenario, along with alternative strategies for addressing vulnerable dependencies. Third, we present a new decision-making process employed by **FinCompany** to prioritize which vulnerabilities should be addressed—highlighting the perspectives on vulnerabilities that are deemed most critical. The main outcomes of the first study are as follows:

- ⋆ We successfully mapped the dependencies associated with the 2,140 versions of the 585 projects deployed from January to July 2024, revealing a total of 30,856 unique deployed dependencies;
- ⋆ Vulnerabilities were identified in 99.53% of the projects, associated with 647 external dependencies, yielding an average of 63.51 high-severity and 28.46 critical vulnerabilities per project;
- ⋆ More than 60% of the vulnerable direct dependencies have corresponding updated versions that have no disclosed vulnerabilities; and

We disseminated the results of the first study to key divisions of **FinCompany** and recommended a set of process improvement measures. This initiative resulted in the establishment of a corporate cyber risk indicator to track **FinCompany**'s exposure and in follow-up consultations with relevant units to determine the most suitable methodological approach.

1.2.2 Secondy Study: Compatibility Assessment and Risk Mitigation Outcomes

The second study focused on assessing the compatibility between projects and the recommended versions of libraries identified as free from known vulnerabilities. Its objective was to determine which projects could be rapidly updated, thereby mitigating organizational risks while enabling developers to confidently apply timely updates. For this purpose, we identified which projects from the first study had been deployed between December-2024 and July 2025 and applied the second module of *Ready-to-Update* to these projects. Accordingly, the second study aimed to address the following research question:

Fourth Research Question (RQ4): Given how vulnerabilities propagate, what strategies can be adopted to update libraries to secure versions?

The implementation of the second module of *Ready-to-Update* represents a key contribution of this study. Moreover, the second study provides a follow-up evaluation of external dependency vulnerabilities within the same company, conducted roughly one year after the initial assessment. The results indicate a reduction in the number of critical vulnerabilities; however, the overall project profile remains fundamentally exposed. The main outcomes of the second study include:

⋆ Comparing the update of only a vulnerable dependency with the update of all dependencies within the same group revealed that the former approach yields higher project compatibility, contradicting the recommendations of some previous studies.

⋆ Projects built with more recent Java versions exhibit greater compatibility with new dependency releases. The *Ready-to-Update* approach successfully identified updated, more secure, and compatible versions for 86.3% of vulnerable direct dependencies.

1.3 Organization of the dissertation

This document is organized according to the following description.

Chapter 2 presents the necessary background for the work, covering key topics such as code reuse, the software supply chain and its security implications, and software compatibility verification. It also outlines the foundational research used to create the RTU and their main contributions.

Chapter 3 introduces the RTU and its two component modules, which were designed based on the studies mentioned in the introduction. For each stage of the methodology, the decisions made during its creation are detailed, along with specifics on its deployment within the company.

Chapter 4 provides the results of each study, preceded by an exploratory analysis of the collected data. For each study, the corresponding research questions are articulated, as are the key findings from the data exploration.

Chapter 5 analyzes the work's main contributions, identifies potential threats to its validity and concludes the document by restating the final remarks and suggesting future work possibilities in the area, taking into account the research findings.

Chapter 2

Background and Related Work

This section provides the necessary background to help readers better understand our research. We begin with an overview of code reuse, followed by key concepts related to software supply chains, their associated security concerns, and compatibility challenges arising from software versioning. Throughout this chapter, we also relate our research to existing work.

2.1 Code Reuse

The topic of code reuse has been discussed in software engineering for nearly six decades [7]. It is commonly defined as the process of developing new software systems using pre-existing code rather than constructing them entirely from scratch [8]. In his seminal report on software reuse, Krueger [8] examined multiple dimensions of reuse-related research, focusing on processes such as abstraction, selection, specialization, and integration of software artifacts [9]. Around the same time, Capilla et al. [10] organized the body of research on software reuse into four main categories: domain analysis; generators and transformation models; components, languages, and code reuse; and reuse cost models.

Arvanitou et al. [11] later observed that, in contemporary software engineering practice, reuse typically occurs not through direct source code reuse but via the development of reusable artifacts. These artifacts are most often distributed as libraries designed to address recurring problems within a specific domain. In some cases, researchers have discussed how application programming interfaces (APIs) can facilitate the reuse of such third-party libraries [12]. Likewise, the reuse of source code and software components has been recognized as one of the principal forms of reusable assets [13]. In their study, Haefliger, Von Krogh, and Spaeth [13] observed that code reuse is pervasive across the analyzed projects. They found that both open-source and enterprise software developers rely on mechanisms that lower the search costs for knowledge and code, help assess the quality of software components, and motivate the practice of reuse. Open-source developers, in particular, engage in code reuse to rapidly integrate new functionality, to incorporate preferred code patterns, to mitigate constraints related to time and expertise, and to reduce development costs.

The use of reusable components—a practice originally advocated by McIlroy [14]—is widely discussed in academic literature under the term *Component-Based Software Engineering (CBSE)* [15]. Szyperski [16] argues that the adoption of components is inevitable, since avoiding their use entails reinventing existing solutions—a course of action justifiable only when the newly developed solution offers substantial advantages over available alternatives. In their investigation, Chen, Usman, and Badampudi [17] conducted a systematic literature review to identify the benefits and costs associated with software reuse. Their findings indicate that the main benefits include improved quality and increased development productivity, whereas the main costs are related to reduced quality and maintainability. In particular, the metrics used to characterize costs associated with reduced quality also capture impacts on the security of reused software [18]. The findings of Gkortzis, Feitosa, and Spinellis [19] corroborate the assertion by Gupta et al. [18], based on their evaluation of open-source projects, that security vulnerabilities may exist in code incorporated into projects through library reuse.

Correspondingly, among the practices highlighted by Cox[20], the implementation of best practices is deemed essential for dependency management within the current resource constraints. This necessitates the development of processes that assess, mitigate, and monitor risks, from the initial adoption decision to production deployment.

2.2 Software Supply Chain

The software supply chain encompasses the entire lifecycle of digital product development, from creation to distribution. It comprises the network of people, processes, software artifacts (such as applications, libraries, and firmware), and technologies that enable and support this complex development process [21]. Nocera et al. [3] offer a more concise definition, describing the software supply chain as "anything needed (. . .) to develop and deliver a software product."

Within this context, the concept of a *Software Bill of Materials* (SBOM)—an adaptation of the engineering bill of materials—emerges as a means to provide a comprehensive record of all components that constitute a software product, including third-party dependencies. By doing so, it establishes itself as a key instrument for software supply chain security (SCSS) [22]. To support this goal, several standardized formats have been developed for SBOM generation. Among them, CycloneDX, the *Software Package Data*

Exchange (SPDX), and *Software Identification* have been recognized by the United States National Telecommunications and Information Administration (NTIA) [23] as meeting the baseline criteria defined in Executive Order 14028 [24].

Nevertheless, implementing SBOMs presents several inherent challenges. Two papers from the *Special Issue on Software Supply Chain Security* [4] address this topic in depth. Balliu et al. [22] evaluated six different SBOM generation tools and identified multiple challenges in their creation. These include determining which tools were used during project development (e.g., version control, testing, and build systems), identifying supply chain vulnerabilities (such as an excessive number of developers), and performing runtime validation of SBOM accuracy to ensure that only declared components are executed. Torres-Arias et al. [25], in turn, argue that research should move beyond the mere adoption of SBOM generation and instead emphasize the development and validation of quality metrics for SBOMs. Such metrics would enable greater visibility into the quality characteristics of SBOMs—beyond, for instance, the completeness of their data fields.

Stalnaker et al. [2], in turn, conducted an extensive survey involving 138 participants drawn from diverse groups relevant to the topic, including contributors to critical open-source projects, professionals working on cyber-physical systems, and legal practitioners. Consistent with the findings of previous studies, they identified several challenges associated with SBOM adoption, such as the complexity of requirements, the burden of long-term maintenance, and the difficulty of verifying the accuracy and completeness of SBOMs, among others.

Of particular relevance to our investigation is the area of software supply chain security (SSCS), which is concerned with the governance of security risks related to the incorporation of external software at any point within the software production pipeline, through the identification and mitigation of vulnerabilities and threats across all constituents of the software production process [21].

2.3 Software Supply Chain Security

In this context, Wang et al. [26] observe that in modern software development, third-party libraries—particularly open-source components—serve as essential building blocks for constructing software systems. A recent report by Synopsys [27] reinforces this observation, noting that open-source libraries were present in 96% of all audited codebases, and that 74% of these contained high-risk vulnerabilities. Indeed, the use of vulnerable or outdated dependencies ranks sixth in the OWASP Top Ten [28], a widely recognized list that identifies the most critical security risks affecting web applications. The issue has also been examined extensively by organizations such as NIST [29, 30] and SAFEcode [31, 32], discussed in a special issue of *IEEE Security & Privacy Magazine* [4], and addressed by regulatory initiatives around the world, including in the United States [24] and Europe [33].

In this context, the *Software Bill of Materials (SBOM) and Cybersecurity Readiness* report highlights that the implementation of SBOMs can play a key role in addressing three major challenges associated with the integration of third-party software: security, provenance, and licensing [34]. Donoghue, Reinhold, and Izurieta [35] further demonstrate that SBOMs can be effectively employed in supply chain security assessments and in illustrating the risks inherent in using vulnerable third-party software. To reach this conclusion, they analyzed 1,151 SBOMs obtained from third-party software repositories, examining how software vulnerabilities propagate across the analyzed SBOMs.

Conversely, Nocera et al. [3] published an industry paper presenting the results of a study conducted across six firms. Their findings indicate that, despite the well-known risks associated with software supply chain threats, SBOMs remain rarely adopted. This limited adoption is primarily attributed to an insufficient understanding of the regulations governing this area.

In parallel, a growing body of research has investigated the prevalence of vulnerabilities in third-party dependencies. For instance, with respect to **vulnerability identification**, Hejderup [36] examined the occurrence of vulnerable dependencies among JavaScript modules available in the NPM registry, using a custom-built database populated with security advisories from the Node Security Project (NSP)[1]. Ponta, Plate, and Sabetta [37] proposed a code-centric approach that combines static and dynamic analysis techniques to determine whether vulnerable code within libraries—used either directly or transitively—is actually reachable by an application. Similarly, Wang et al. [38] developed a bug crawler to extract security vulnerabilities from Java dependencies and to collect associated metadata from Veracode's[2] vulnerability database.

Gkortzis, Feitosa, and Spinellis [19] employed static analysis—specifically the SpotBugs analyzer and its FindSecBugs plugin—to detect security vulnerabilities in open-source projects. Their results suggest that the number of potential vulnerabilities in both original and reused code increases with project size, challenging the assumption that code reuse is either the primary cause of vulnerabilities or a foolproof strategy for their mitigation.

Similarly, Susheng et al. [39] proposed a weighted inter-procedural program dependency graph to identify which library versions are affected by reported vulnerabilities, comparing vulnerable and fixed dependency states. These and similar approaches typi-

[1]Node Security Project
[2]`https://www.veracode.com`

cally require substantial, ongoing effort to build and maintain comprehensive vulnerability databases.

Given that the **FinCompany** aims to avoid adding complexity to its existing processes, a commercial vulnerability database was considered the most straightforward and scalable solution for integrating vulnerability checks into our methodology. While this resource effectively supports large-scale vulnerability identification, it still lacks the holistic perspective necessary for the firm to manage and prioritize vulnerability remediation effectively.

Conversely, a substantial body of research has focused on identifying **strategies for targeting and prioritizing remediation efforts** for discovered vulnerabilities. Pashchenko et al. [5] argue that developers can better prioritize vulnerabilities by filtering out non-deployed dependencies, grouping dependencies by project, and identifying abandoned ones. In subsequent work, Pashchenko et al. [40] broaden this perspective, offering a more comprehensive view of how their state-of-the-art approach can be applied in practice. Dann, Hermann, and Bodden [6] introduce *UpCy*, a tool that determines an optimal sequence of update steps required to upgrade a dependency to a target version while maintaining compatibility with other libraries.

2.4 The Complexity of Dependency Updates

As previously established, a primary drawback inherent to code reuse is the resulting liability in long-term maintenance [17], stemming from the need to synchronize and update multiple versions across the codebase [41]. A failure to perform adequate maintenance procedures [42] can lead to the widely recognized predicament of "dependency hell" [43], characterized by a backlog of updates that demands significant technical investment to remediate.

This complexity is markedly amplified by the recurrent utilization of transitive dependencies — i.e., the dependencies of dependencies — since it is possible for multiple, potentially conflicting, versions of a single dependency to be declared for the same project, even though only one specific version will be executed at runtime [44].

This prevalent scenario generates dependency chains that frequently result in version incompatibilities, thus necessitating that the updating of libraries includes a prerequisite verification of the compatibility of the secured versions. Such incompatibilities can be either binary and source or semantic [6].

Binary and source incompatibility indicates scenarios in which changes within a dependency — such as the removal of types, changing fields to incompatible types, class alterations, method signature changes, or modifications to a function's visibility — hinder the compilation, linking, or execution of previously valid code[45].

Dietrich, Jezek, and Brada [45] describe a number of possible binary and source incompatibilities that can arise as libraries evolve. These can include both binary and source incompatibilities at the same time, or just one of them. A portion of these incompatibilities — primarily those of the source type — lead to project compilation issues, whereas binary-only incompatibilities might not manifest until runtime.

Accordingly, the verification of binary and source compatibility should be performed on a project-specific basis, given that certain errors only arise when a specific incompatible method is invoked [45]. Consequently, two projects with similar dependency graphs may exhibit different responses to a new version's compatibility, depending on which methods each project requires from the dependency.

One hypothetical method for verifying compatibility involves assessing Semantic Versioning (MAJOR.MINOR.PATCH) [46]. Per the specification, Major versions are allowed to introduce incompatibilities, whereas Minor and Patch versions are expected to be backward-compatible. Nevertheless, studies by Bogart et al. [47] and Xavier et al. [48] confirm that this is not always the case, given that versioning is also tied to the application developers' interpretation of backward compatibility.

After confirming binary and source compatibility, it becomes necessary to check for semantic compatibility, as a method's behavior might have been altered, leading the application to behave differently than intended [6].

Identifying semantic incompatibility is complex and difficult to calculate accurately and efficiently, which is why the primary approaches rely on the use of automatic test cases [49, 50]. Although failing unit tests are one of the main reasons for developers to refuse to update dependencies [51], this resource is dependent on the project's test quality (e.g., test coverage [48]).

In their study, Pashchenko et al. [5] introduced the perspective that dependencies from the same group can signify the use of a framework. Consequently, they suggested that when a particular dependency from this group is updated, all others should be updated simultaneously to guarantee higher compatibility among the framework's libraries. They named this update approach a blossom update.

Chapter 3

Ready-to-Update Approach

This chapter presents the *Ready-to-Update* approach, which builds upon and tailors design decisions proposed by Pashchenko et al. [5] and by Dann, Hermann, and Bodden [6]. As introduced earlier, we evaluate *Ready-to-Update* through two empirical studies that experiment with the two major *Ready-to-Update* components detailed in this chapter. Figure 3.1 provides an overview of our methodology, in which each step incrementally builds on the previous one to ensure a comprehensive and efficient analysis.

The first module (idVUDD) is designed to identify **Vulnerable, Updatable, Direct Dependencies** (VUDDs) (Section 3.1) and is responsible for executing the first three steps in our pipeline: (1) dependency resolution (Step 1 in Fig. 3.1), (2) vulnerability identification (Step 2 in Fig. 3.1), and (3) verification of the availability of new versions for vulnerable direct dependencies, as well as the identification of vulnerabilities in these new versions. Note that, in the third step, Steps 1 and 2 in Figure 3.1 must be executed again.

The second *Ready-to-Update* module (chkVUDD) (Section 3.2) assesses the compatibility of the projects with the "vulnerability-free" versions of the libraries identified as outcomes of the first module. To this end, it compiles and executes the unit tests defined for each project. The process is carried out twice: first, using the vulnerable dependency (Steps 4 and 4.1 in Fig. 3.1); and second, after applying the blossom update (see Section 2.4) (Steps 5 and 5.1 in Fig. 3.1).

The entire workflow is orchestrated by Python scripts specifically designed to coordinate all component modules through well-defined interfaces.

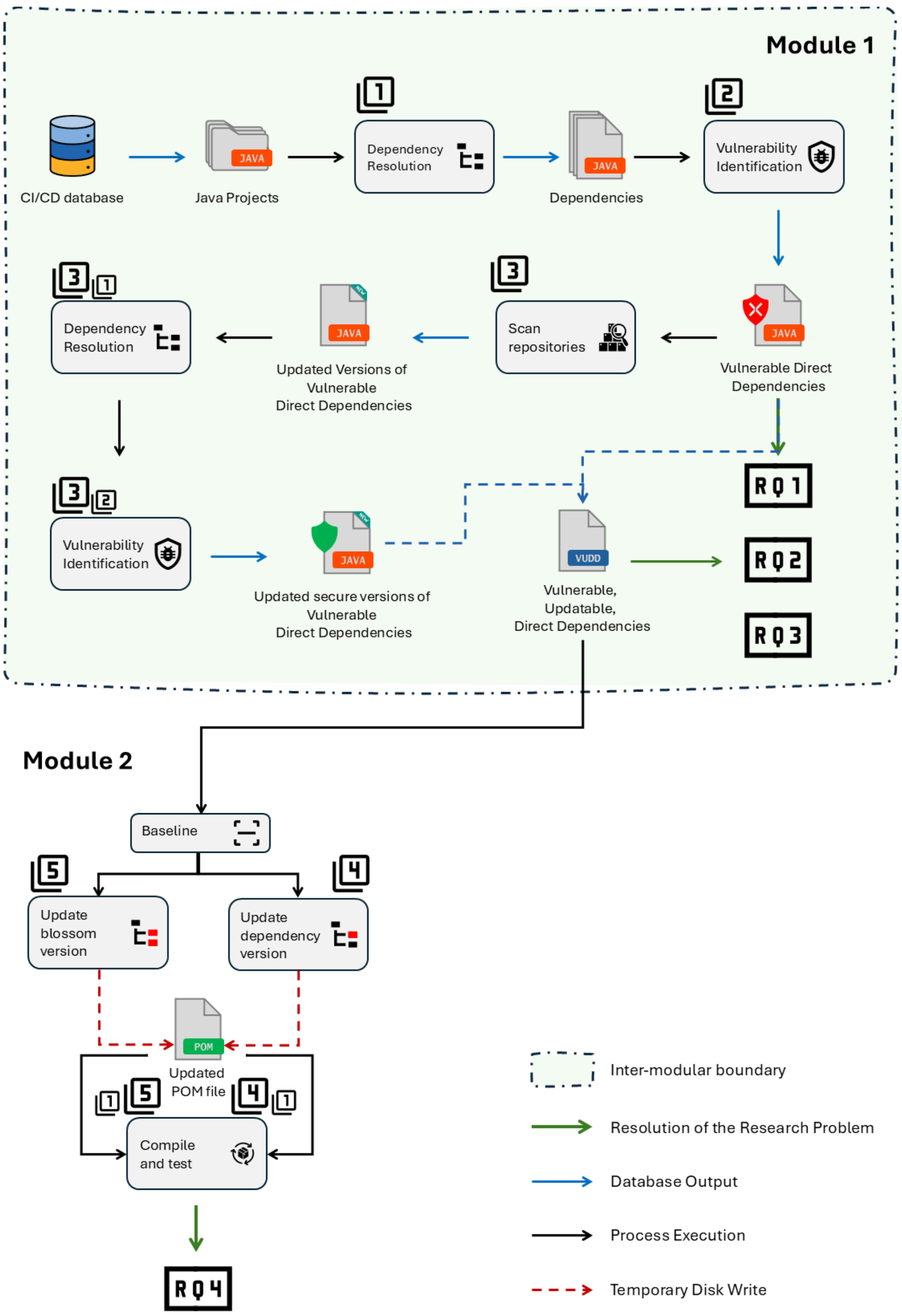

Module 1
CI/CD database
Java Projects
JAVA
1
Dependency Resolution
Dependencies
2
Vulnerability Identification
Vulnerable Direct Dependencies
3
Scan repositories
Updated Versions of Vulnerable Direct Dependencies
3 1
Dependency Resolution
3 2
Vulnerability Identification
Updated secure versions of Vulnerable Direct Dependencies
VUDD
Vulnerable, Updatable, Direct Dependencies
RQ1
RQ2
RQ3
Module 2
Baseline
5
Update blossom version
4
Update dependency version
POM
Updated POM file
1 5 4 1
Compile and test
RQ4
Inter-modular boundary
Resolution of the Research Problem
Database Output
Process Execution
Temporary Disk Write

Figure 3.1: *Ready-to-Update* workflow

3.1 Identifying VUDDs (idVUDD)

3.1.1 Dependency Resolution

In the first step, *Ready-to-Update* generates a library dependency graph for the projects using the `depgraph-maven-plugin` tool[1]. This is a Maven plugin that generates dependency graphs in various formats, such as DOT, GML, PlantUML, JSON, and Text. Given the documentation of Maven's scope declaration[2] and our empirical observation that all Maven dependency scopes, except for the `test` scope, are utilized in production within **FinCompany**, we decided to exclude only `test` dependencies from our dependency graph. This contrasts with previous work, which suggests removing dependencies marked with either the `test` or `provided` scopes [5].

The resulting graph provides a comprehensive overview of all dependencies of a project, detailing their names, versions, and scopes. Note that, given Maven's dependency resolution process, it is possible for a single dependency to exhibit multiple scopes in the final graph. This arises because a transitive dependency can be associated with multiple dependencies, each having distinct scopes. The plugin, however, preserves data on the dependency resolution process, specifying, for example, which dependencies were incorporated and which were omitted because of conflicts or duplication.

The interaction with the Maven Dependency graph plugin is conducted via dedicated Python scripts, which are specifically engineered to automate the extraction of granular dependency data from each of the evaluated projects. For each dependency, we record detailed information such as `group`, `artifact`, `version`, `timestamp` of deployment, and an aggregated key (`GAV`[3]) for reference. Furthermore, *Ready-to-Update* introduces an attribute to differentiate between the primary projects under analysis (`main`) and their associated dependencies. For each dependency edge in the graph, we annotate the relationship with the primary project (`main`) being analyzed and the associated dependency `scopes`. This granularity is crucial, as a dependency can have different scopes in various projects due to Maven's dependency resolution mechanisms. This attribute allows us to precisely map the scope of each dependency within the context of its respective project, providing a more comprehensive understanding of the dependency relationships.

The pom.xml file, detailed below, was developed strictly to illustrate the *Ready-to-Update* process.

[1]`https://github.com/ferstl/depgraph-maven-plugin`

[2]`https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html`

[3]GAV is an acronym for the aggregate key that combines a project's group ID, artifact ID, and version, typically separated by colons. For instance, the GAV for version 1.2.17 of Log4j is `log4j:log4j:1.2.17)`

```xml
<?xml version="1.0"?>
<project
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.unb.rtu</groupId>
  <artifactId>rtu-toy</artifactId>
  <version>0.0.1</version>
  <name>rtu-toy</name>
  <properties>
    <compiler-plugin.version>3.8.1</compiler-plugin.version>
    <failsafe.useModulePath>false</failsafe.useModulePath>
    <maven.compiler.release>17</maven.compiler.release>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <version>2.6</version>
    </dependency>
    <dependency>
      <groupId>io.jsonwebtoken</groupId>
      <artifactId>jjwt-api</artifactId>
      <version>0.11.2</version>
    </dependency>
    <dependency>
      <groupId>io.jsonwebtoken</groupId>
      <artifactId>jjwt-impl</artifactId>
      <version>0.11.2</version>
    </dependency>
    <dependency>
      <groupId>io.jsonwebtoken</groupId>
      <artifactId>jjwt-jackson</artifactId>
      <version>0.11.2</version>
    </dependency>
    <dependency>
      <groupId>com.auth0</groupId>
      <artifactId>java-jwt</artifactId>
      <version>3.11.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>${compiler-plugin.version}</version>
        <configuration>
```

```xml
          <compilerArgs>
            <arg>-parameters</arg>
          </compilerArgs>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

Following the project clone the `depgraph-maven-plugin` tool is executable via Python scripts. The execution is performed from the directory containing the pom.xml file, using the following syntax and the resulting file is stored in the same path, named `dependency-graph.json`.

```
mvn com.github.ferstl:depgraph-maven-plugin:4.0.3:graph \
-DshowVersions -DshowGroupIds -DshowDuplicates -DshowConflicts
-DgraphFormat=json \
-DoutputDirectory=.
```

The outcome of applying this command to our specified file is presented in detail in the following section.

```json
{
  "graphName" : "rtu-toy",
  "artifacts" : [ {
    "id" : "br.unb.rtu:rtu-toy:jar",
    "numericId" : 1,
    "groupId" : "br.unb.rtu",
    "artifactId" : "rtu-toy",
    "version" : "0.0.1",
    "optional" : false,
    "scopes" : [ "compile" ],
    "types" : [ "jar" ]
  }, {
    "id" : "commons-lang:commons-lang:jar",
    "numericId" : 2,
    "groupId" : "commons-lang",
    "artifactId" : "commons-lang",
    "version" : "2.6",
    "optional" : false,
    "scopes" : [ "compile" ],
    "types" : [ "jar" ]
  }, {
    "id" : "io.jsonwebtoken:jjwt-api:jar",
    "numericId" : 3,
    "groupId" : "io.jsonwebtoken",
    "artifactId" : "jjwt-api",
    "version" : "0.11.2",
    "optional" : false,
  "scopes" : [ "compile" ],
  "types" : [ "jar" ]
}, {
  "id" : "io.jsonwebtoken:jjwt-impl:jar",
  "numericId" : 4,
  "groupId" : "io.jsonwebtoken",
  "artifactId" : "jjwt-impl",
  "version" : "0.11.2",
  "optional" : false,
  "scopes" : [ "compile" ],
  "types" : [ "jar" ]
}, {
  "id" : "io.jsonwebtoken:jjwt-jackson:jar",
  "numericId" : 5,
  "groupId" : "io.jsonwebtoken",
  "artifactId" : "jjwt-jackson",
  "version" : "0.11.2",
  "optional" : false,
  "scopes" : [ "compile" ],
  "types" : [ "jar" ]
}, {
  "id" : "com.fasterxml.jackson.core:jackson-databind:jar",
  "numericId" : 6,
  "groupId" : "com.fasterxml.jackson.core",
  "artifactId" : "jackson-databind",
  "version" : "2.9.10.4",
  "optional" : false,
  "scopes" : [ "compile", "runtime" ],
  "types" : [ "jar" ]
}, {
  "id" : "com.fasterxml.jackson.core:jackson-annotations:jar",
  "numericId" : 7,
  "groupId" : "com.fasterxml.jackson.core",
  "artifactId" : "jackson-annotations",
  "version" : "2.9.10",
  "optional" : false,
  "scopes" : [ "compile" ],
  "types" : [ "jar" ]
}, {
  "id" : "com.fasterxml.jackson.core:jackson-core:jar",
  "numericId" : 8,
  "groupId" : "com.fasterxml.jackson.core",
  "artifactId" : "jackson-core",
  "version" : "2.9.10",
  "optional" : false,
  "scopes" : [ "compile" ],
  "types" : [ "jar" ]
}, {
  "id" : "com.auth0:java-jwt:jar",
  "numericId" : 9,
  "groupId" : "com.auth0",
  "artifactId" : "java-jwt",
  "version" : "3.11.0",
  "optional" : false,
  "scopes" : [ "compile" ],
  "types" : [ "jar" ]
}, {
  "id" : "commons-codec:commons-codec:jar",
  "numericId" : 10,
  "groupId" : "commons-codec",
  "artifactId" : "commons-codec",
  "version" : "1.14",
  "optional" : false,
  "scopes" : [ "runtime" ],
  "types" : [ "jar" ]
}, {
  "id" : "junit:junit:jar",
  "numericId" : 11,
  "groupId" : "junit",
  "artifactId" : "junit",
  "version" : "4.13.2",
  "optional" : false,
  "scopes" : [ "test" ],
  "types" : [ "jar" ]
}, {
  "id" : "org.hamcrest:hamcrest-core:jar",
  "numericId" : 12,
  "groupId" : "org.hamcrest",
  "artifactId" : "hamcrest-core",
  "version" : "1.3",
  "optional" : false,
  "scopes" : [ "test" ],
  "types" : [ "jar" ]
} ],
"dependencies" : [ {
  "from" : "br.unb.rtu:rtu-toy:jar",
  "to" : "commons-lang:commons-lang:jar",
  "numericFrom" : 0,
  "numericTo" : 1,
  "resolution" : "INCLUDED"
}, {
  "from" : "br.unb.rtu:rtu-toy:jar",
  "to" : "io.jsonwebtoken:jjwt-api:jar",
  "numericFrom" : 0,
  "numericTo" : 2,
  "resolution" : "INCLUDED"
}, {
  "from" : "io.jsonwebtoken:jjwt-impl:jar",
  "to" : "io.jsonwebtoken:jjwt-api:jar",
  "numericFrom" : 3,
  "numericTo" : 2,
  "resolution" : "OMITTED_FOR_DUPLICATE"
}, {
  "from" : "br.unb.rtu:rtu-toy:jar",
  "to" : "io.jsonwebtoken:jjwt-impl:jar",
  "numericFrom" : 0,
  "numericTo" : 3,
  "resolution" : "INCLUDED"
}, {
  "from" : "io.jsonwebtoken:jjwt-jackson:jar",
  "to" : "io.jsonwebtoken:jjwt-api:jar",
  "numericFrom" : 4,
  "numericTo" : 2,
  "resolution" : "OMITTED_FOR_DUPLICATE"
```

```
}, {
  "from" : "com.fasterxml.jackson.core:jackson-databind:jar",
  "to" : "com.fasterxml.jackson.core:jackson-annotations:jar",
  "numericFrom" : 5,
  "numericTo" : 6,
  "resolution" : "INCLUDED"
}, {
  "from" : "com.fasterxml.jackson.core:jackson-databind:jar",
  "to" : "com.fasterxml.jackson.core:jackson-core:jar",
  "numericFrom" : 5,
  "numericTo" : 7,
  "resolution" : "INCLUDED"
}, {
  "from" : "io.jsonwebtoken:jjwt-jackson:jar",
  "to" : "com.fasterxml.jackson.core:jackson-databind:jar",
  "numericFrom" : 4,
  "numericTo" : 5,
  "resolution" : "INCLUDED"
}, {
  "from" : "br.unb.rtu:rtu-toy:jar",
  "to" : "io.jsonwebtoken:jjwt-jackson:jar",
  "numericFrom" : 0,
  "numericTo" : 4,
  "resolution" : "INCLUDED"
}, {
  "from" : "com.auth0:java-jwt:jar",
  "to" : "com.fasterxml.jackson.core:jackson-databind:jar",
  "numericFrom" : 8,
  "numericTo" : 5,
  "version" : "2.10.3",
  "resolution" : "OMITTED_FOR_CONFLICT"
}, {
  "from" : "com.auth0:java-jwt:jar",
  "to" : "commons-codec:commons-codec:jar",
  "numericFrom" : 8,
  "numericTo" : 9,
  "resolution" : "INCLUDED"
}, {
  "from" : "br.unb.rtu:rtu-toy:jar",
  "to" : "com.auth0:java-jwt:jar",
  "numericFrom" : 0,
  "numericTo" : 8,
  "resolution" : "INCLUDED"
}, {
  "from" : "junit:junit:jar",
  "to" : "org.hamcrest:hamcrest-core:jar",
  "numericFrom" : 10,
  "numericTo" : 11,
  "resolution" : "INCLUDED"
}, {
  "from" : "br.unb.rtu:rtu-toy:jar",
  "to" : "junit:junit:jar",
  "numericFrom" : 0,
  "numericTo" : 10,
  "resolution" : "INCLUDED"
} ]
```

```
198 }
```

The plugin tracks the Maven dependency resolution mechanism, making it feasible to pinpoint both the specific dependency version incorporated and those excluded — whether due to redundancy or conflict — as documented in lines 141 and 172.The graphical representation of these results is presented in Figure 3.2[4].

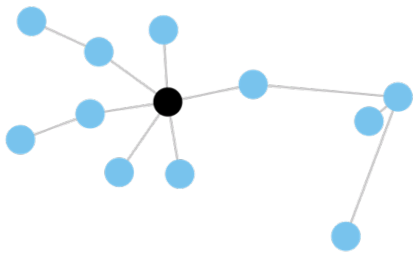


Figure 3.2: Dependency graph of the example project

3.1.2 Vulnerability Identification

In the second step, *Ready-to-Update* integrates a commercial tool that provides information about vulnerabilities in Java artifacts (such as programs and libraries) and data available from vulnerability databases and security advisories, including NVD and GitHub. The commercial tool calculates the severity of vulnerabilities based on data from vulnerability repositories and a comprehensive security posture derived from in-depth CVE findings and vulnerability data provided by a dedicated Security Research Team. The dependency graph is then augmented with comprehensive vulnerability data, such as identifier (e.g., CVE ID), severity level, and Common Weakness Enumeration (CWE) references. As already mentioned, we have a different view about the vulnerabilities that are truly important to the **FinCompany**. That is, *Ready-to-Update* focuses on identifying the most significant number of vulnerable dependencies with a newer and safer version; tagging those ones with *ready to apply* remediation. This approach establishes a clear distinction from the work of Pashchenko et al. [5], which emphasizes the identification of dependencies prone to abandonment and characterized by the absence of future releases necessary for effective vulnerability mitigation.

[4]The black node designates the project under analysis, while the blue nodes denote the project dependencies.

Furthermore, given the significant costs associated with assessing the exploitability of each identified vulnerability, we consider a project vulnerable if it depends on a vulnerable library, even when the specific conditions required for exploitation are not met. Determining exploitability demands mapping these conditions for every vulnerability and continuously monitoring them across numerous systems—an effort that is infeasible in the **FinCompany** environment. For instance, although a vulnerable version of Log4j poses a potential risk, exploitation depends on particular circumstances: the vulnerability can only be triggered if external input reaches a logging call and the default JNDI loading behavior is not disabled. To mitigate this risk, projects must ensure that external input cannot reach the log or that JNDI loading is disabled, and these safeguards must be continuously verified.

Given the high complexity of the **FinCompany** environment, however, it is not feasible to determine whether each vulnerability is actually exploitable. Consequently, we label a library as vulnerable if at least one vulnerability has been reported for it. This represents a different perspective from that of the existing literature and aligns with the guidelines established by the risk and security teams at **FinCompany**.

Ready-to-Update uses Python scripts that consume a web API to interface with the commercial tool's vulnerability database. The GAV identifier serves as the primary search criterion. A key methodological choice is to isolate vulnerabilities specifically associated with the target dependency. By excluding vulnerabilities originating from transitive dependencies, the analysis ensures a more precise attribution of vulnerabilities, thereby improving the overall accuracy of the results.

Furthermore, each vulnerability–dependency link includes an additional attribute — the search date — which enables future investigations into factors influencing the introduction of new vulnerabilities in dependencies. For example, it becomes possible to analyze cases where a project acquired new vulnerabilities between deployments despite no changes to its declared dependencies.

To keep track of dependencies that have been previously searched for vulnerabilities on a specific research date, an extra node labeled "`no vulns`" is introduced into the graph. This node is connected to each dependency that does not yield any vulnerabilities when queried against the commercial tool. By incorporating this mechanism,*Ready-to-Update* prevents redundant queries for the same dependency in future iterations.

The graph of the project, as presented in Figure 3.3, has been augmented with the discovered vulnerability information. Beyond the pre-existing nodes, vulnerable dependencies are rendered in orange, whereas the vulnerabilities are symbolized by red, star-form nodes. Dependencies designated for the test scope, along with their associated vulnerabilities, exhibit lower opacity.

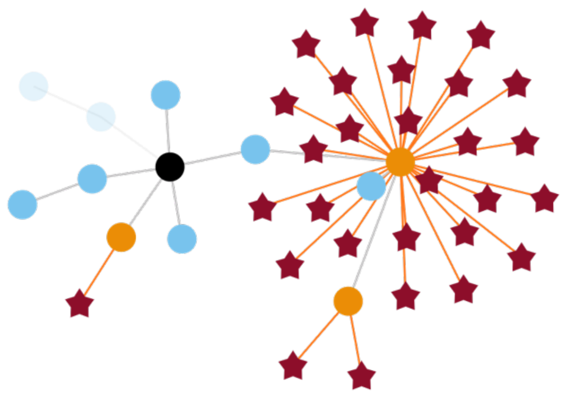


Figure 3.3: Dependency graph of the example project augmented with vulnerability information

3.1.3 Availability of updated secure versions identification

As a third step, *Ready-to-Update* maps all **direct dependencies** on the path to vulnerable dependencies, identifies the availability of newer versions, and then verifies whether those newer versions contain known vulnerabilities. In more detail, for every direct dependency exhibiting at least one vulnerability, whether directly or through transitive dependencies, we initiate a two-step iterative process. First, we search the company's repositories for newer versions of the dependency. Subsequently, for each newly identified version, we replicate the entire analysis, i.e., dependency resolution, filtering out non-deployed dependencies, and vulnerability identification. The results of all steps are persisted in an Oracle database at the **FinCompany**.

The diagram of the Oracle database employed in this investigation is displayed in the Figure 3.4. It is evident that all artifacts are uniformly stored in a single table (`artifacts`), which incorporates a Boolean flag to distinguish the primary projects under evaluation from their dependencies (`bool_main`). Furthermore, indicators track the completion of the source code cloning (`bool_clone`) and the dependency graph generation (`bool_dependency_graph`). Such flags are essential for ensuring that each step of the process is executed only once per project, particularly in the event of process interruptions, thereby enabling seamless resumption from the point of failure.

Ready-to-Update employs the Depth-First Search (DFS) [52] algorithm to traverse the dependency graph and identify direct dependencies whose transitive dependencies contain known vulnerabilities. Once these dependencies are identified, the scripts query Maven repositories to retrieve newer versions based on their group and artifact identifiers. These updated dependencies are then incorporated into the graph, appearing as isolated nodes — clearly distinguishable by the absence of connections to other dependencies or vulnerabilities.

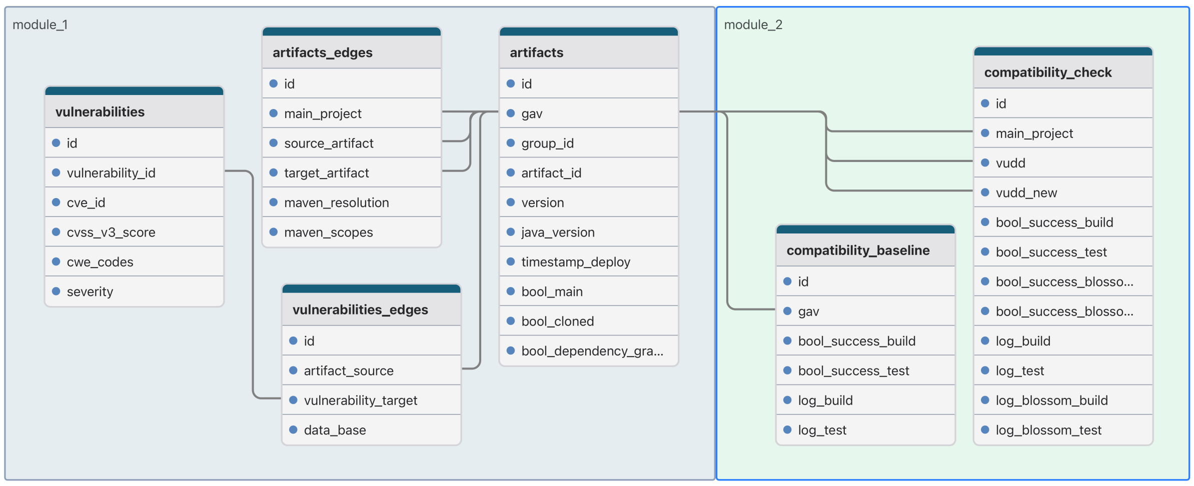


Figure 3.4: Oracle database diagram

An iterative process subsequently resolves these newly added dependencies and analyzes their latest versions for known vulnerabilities. After this iteration, a Breadth-First Search (BFS) [52] traversal is applied exclusively to the first level of the dependency graph, encompassing only direct dependencies. This methodological choice enables the profiling of the vulnerability landscape across all dependency versions.

The iterative nature of *Ready-to-Update* allows the identification of newer versions for each vulnerable direct dependency and the assessment of their relative security posture compared to the versions currently employed in the projects. Through this process, *Ready-to-Update* identifies **Vulnerable, Updatable, Direct Dependencies** (**VUDDs**). To avoid redundant analyses, *Ready-to-Update* verifies whether a dependency has been assessed in previous iterations, ensuring that each one is evaluated only once. Our analysis shows that this optimization reduces the number of executions by more than 97.8%.

3.2 Checking the VUDDs new versions compatibility (chkVUDD)

3.2.1 Baseline Definition

Building on the information obtained from the preceding module, the first step of the *Ready-to-Update* second module implements a set of Python scripts designed to assess whether the newer, secure versions of libraries remain compatible with the projects. To this end, *Ready-to-Update* performs checks to evaluate both the binary and semantic compatibility of the updated libraries, following the methodology proposed by UpCY [6].

To this end, *Ready-to-Update* fnitially performs a copy of the project from the source repository, guaranteeing the strict coupling of the retrieved code with the version deployed to the production environment. In our case, this version represents the most recent release deployed to production within the defined timeframe. The infrastructure employed by **FinCompany** utilizes Git[5] repositories and establishes version-to-commit mappings via tagging[6]. Consequently, the Python scripts interact with the Git command-line interface (CLI) to execute the cloning and the specific commit checkout operations.

Considering the **FinCompany** context, we designed *Ready-to-Update* to require the inclusion of additional steps to guarantee higher compatibility between the project and the Java and Maven versions used, steps necessary to guarantee higher compilation and test success rates. The determination of the requisite Java version for most **FinCompany** projects is largely derived from parsing the POM file and extracting attributes present in the properties section, including, but not limited to,"java.version", "maven.compiler.release", "maven.compiler.source", "maven.compiler.target", and other **FinCompany**-specific customizations. We observed that the projects in **FinCompany** predominantly rely on five different Java versions. As a result, five different execution environments were created to enhance compatibility between the various Java and Maven versions.

A specific annotation indicating the Java version is added to the database for each project to achieve two primary goals. This approach both contributes to improved efficiency — by removing the necessity of re-parsing the source file during each iteration — and makes the information accessible for utilization in subsequent investigations, such as examining projects that have undergone a temporal evolution of their Java version or correlating the adopted Java version with resulting compatibility metrics.

Next, *Ready-to-Update* executes the commands required to compile the project and run its automated tests, thereby establishing a baseline. This baseline allows us to verify that all build and test steps complete successfully before any modifications are introduced.

The Python scripts are used to query the underlying database, fetching the data required to establish the baseline, which includes: the project's identifier, its version, and the supported Java version.

Following data retrieval, *Ready-to-Update* proceeds to configure the most appropriate operational environment. Despite the best practice of utilizing containers (e.g., Docker) for environment isolation, specific restrictions dictated by **FinCompany** for the target execution platform necessitated that all processes be performed natively on the available Linux virtual server. Consequently, all necessary Java versions were pre-installed on the

[5]`https://git-scm.com`

[6]`https://git-scm.com/book/en/v2/Git-Basics-Tagging`

server. *Ready-to-Update* then employs Linux features like "update-alternatives" to specify the Python version and utilizes environment variables for the Maven version definition. For Java version 17, the execution involves the following commands:

Java version: `update-alternatives --set javac java-17-openjdk.x86_64`
Maven version: `JAVA_HOME=/usr/lib/jvm/java-17-openjdk-17.0.16.0.8-2.0.1.el9.x86_64 /opt/mvn/apache-maven-3.8.5/bin/mvn`

With the environment operational, *Ready-to-Update* invokes Maven commands to compile the project (excluding tests initially), followed by the execution of the designated project test cases. Maven is mandated to perform a clean operation on the build artifacts prior to each execution to establish test isolation. Compilation and testing are executed using the `mvn clean compile -DskipTests` and `mvn clean test` commands, respectively.

The resulting script outputs undergo evaluation via their generated logs and are subsequently designated as either “success” or “failure”. Given the constraints of the utilized Oracle database version, the approach persists the analyzed project’s metadata, the binary success/failure status for compilation and tests, and any logs associated with failures, prioritizing efficiency by discarding success logs.

3.2.2 Compatibility Checking for Isolated Dependency Updates

Once the baseline is established, projects that achieve success in the build and test phases are subjected to compatibility checking for the new versions, initially in an isolated manner. To this end, *Ready-to-Update* retrieves the necessary information from the database, encompassing the project under evaluation, the vulnerable dependency, and the most current, secure version that will be assessed.

The Python routines are tasked with removing any residual changes from preceding iterations and performing a commit checkout to the assessed project version. The ElementTree library is then employed to facilitate the automated adjustment of the POM file. Specifically, the script locates dependencies sharing the VUDD’s `group_id` and `artifact_id` among all declared dependencies and updates the corresponding version identifier. The altered file is subsequently persisted in the project folder. It must be highlighted that the namespace declaration is critical during the regeneration of the new POM to ensure the preservation of the initially defined document structure.

Following the POM file update, the build and test procedures used in the baseline creation phase are executed once more, with their outcomes and logs persisted in the database. Within the scope of this investigation, these evaluations are conducted for every available updated version. Nonetheless, *Ready-to-Update* can be configured to optimize the process by ceasing execution as soon as a compatible version is encountered.

3.2.3 Compatibility Checking for Blossom Updates

Subsequently, *Ready-to-Update* leverages the blossom update technique to assess semantic compatibility. The goals of this approach are twofold: to preserve a higher degree of compatibility between project components and to ascertain whether significant disparities in success rates existed among the individual updates.

During this update, *Ready-to-Update* verifies whether any additional dependencies shared the same original group and version. We opted to restrict the scope to dependencies of the same group that exhibited the exact same version as the vulnerable dependency. This approach is intended to preserve the pre-existing state of any dependencies within the same group that were intentionally maintained at different versions, considering the unknown factors motivating those prior configuration decisions.

In instances where no such condition was found, a flag was set in the database, allowing us to distinguish between projects that received blossom updates and those that did not. This approach also enabled us to bypass redundant compilation and testing procedures.

In a manner consistent with the previous procedure, the script searches all declared dependencies for components meeting the cited criteria, namely matching `group_id` and `version`. For every located dependency, the version value is programmatically updated, the POM file is then saved to the project directory, the testing routines are subsequently executed, and both the outcomes and logs are persisted.

Chapter 4

Empirical Assessment

4.1 Study 1: Mapping and Prioritizing Vulnerability Remediation

The **objective** of this empirical assessment is twofold: first, to systematically identify the software dependencies that appear in production within the **FinCompany**, with a focus on those reported to contain vulnerabilities; and second, to determine which of these vulnerable dependencies “matter”—that is, which ones have updated, more secure versions readily available. To this end, we leverage the Module idVUDD of *Ready-to-Update* approach in the **FinCompany**, focusing on answering the following **research questions**:

RQ1 *What are the characteristics of the vulnerabilities discovered in the* **FinCompany***'s project dependencies?*

RQ2 *What is the impact of fixing VUDDs on the overall number of projects vulnerabilities?*

RQ3 *Given how vulnerabilities propagate, what strategies can be adopted to update libraries to secure versions?*

To build our dataset, we first queried a database containing information about the execution of the CI/CD pipeline for Java projects deployed to production. The database records metadata on project deployments (including project name, version, environment, and timestamp).

We primarily compute simple statistics of the **metrics** we collect, such as the total number of dependencies, vulnerable dependencies, and CWEs that propagate through the library dependencies. Our dataset contains Java projects that have been deployed at

FinCompany between January 1st and July 31st, 2024. Given the company's evolving infrastructure, it was determined that an analysis of projects implemented from the beginning of 2024 would provide the most accurate and up-to-date results. This decision was made to avoid including discontinued projects or those that had not been actively maintained in the dataset, which could skew the findings. Data collection was concluded at the end of July 2024 to allow for a comprehensive analysis. A total of 2,140 versions of 585 projects were deployed during this period.

4.1.1 Exploratory Analysis of Library Dependencies

Consistent with the discussion in Section 3.1, we employed data from the CI/CD pipeline database to support our research. The data for this assessment was sourced from projects that were deployed to production from January to December 2024.

To demonstrate the scale and complexity of these projects, we extracted their using SonarQube "ncloc" metric[1]. This analysis revealed that project sizes varied significantly, ranging from 99 to 2,776,150 lines of code (LOC), with a mean of 18,842.73 LOC and a median of 3,200 LOC.

We then applied the *Ready-to-Update* approach (Section 3) to the set of 585 projects in our dataset. This involved an iterative process of dependency resolution, vulnerability identification, and identification of new versions of vulnerable direct dependencies.

This application of *Ready-to-Update* within **FinCompany** reveals that the 2,140 artifacts are interconnected with more than 30,000 dependencies, and the dependency graph comprises more than 1.4 million edges representing dependencies.

The resulting graph database enables the visualization of all dependencies associated with each project version, helping us to identify those scoped for testing and those having reported vulnerabilities, for instance. As depicted in Figure 4.1, the graphical representation showcases the dependencies and vulnerabilities of some particular projects. In the figure, the project is represented as a black node at the center of the graph, while dependencies are indicated in blue and vulnerable dependencies in orange. Dependencies and vulnerabilities having test scope are visualized with reduced opacity.

Our analysis indicates a heterogeneous distribution of vulnerabilities across the examined projects. Project A presents a pattern of vulnerabilities concentrated in transitive dependencies, positioned at a relatively consistent distance from the project. Project B displays a more dispersed distribution of vulnerabilities, encompassing both direct and indirect dependencies. Contrasting, Project C highlights a significant number of vulner-

[1] `https://docs.sonarsource.com/sonarqube-server/latest/user-guide/code-metrics/metrics-definition/#size`

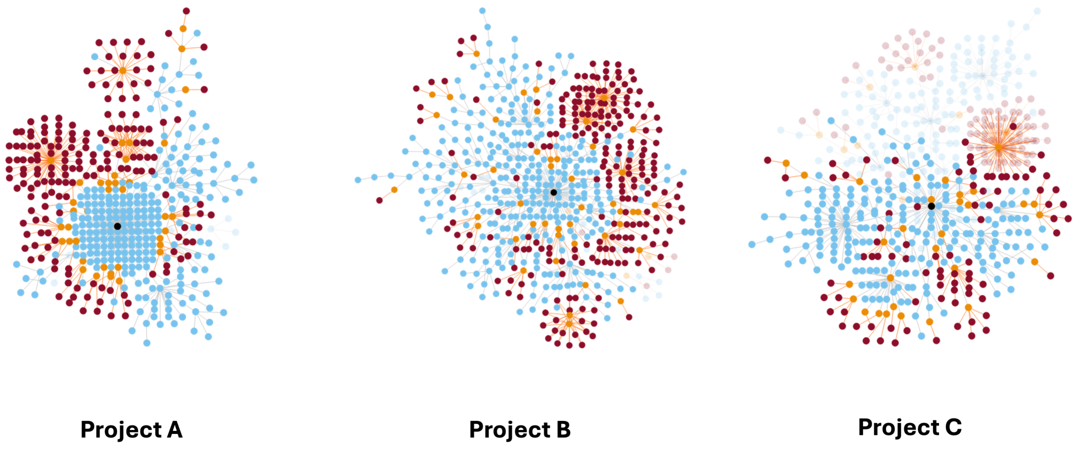


Figure 4.1: Dependency graph of versions of projects

abilities that will not be reported due to their association with dependencies with `test` scope.

The dependency resolution process revealed that the 2,140 project versions are linked to 30,856 unique deployed dependencies, of which 25,706 are internal, i.e., created by the **FinCompany** itself. Furthermore, 21,985 are direct dependencies, while 10,123 are transitive ones. The aggregation of unique direct and transitive dependencies does not equate to the overall count of unique dependencies due to the varying levels of transitivity that dependencies may exhibit across different projects.

Table 4.1 summarizes these findings, and reveals a substantial difference in the vulnerability distribution between direct and transitive dependencies. This finding emphasizes the importance of providing developers with the necessary tools and guidance to remediate vulnerabilities, given that most vulnerable components are hidden within transitive dependencies.

The total number of 1,435,525 dependencies highlights the complexity of the dependency graph. Furthermore, the analysis reveals a diverse range of dependency scopes within projects. Although most dependencies have a `compile` scope (74%), there are instances where a single dependency can assume multiple scopes (up to four), including `provided`, `runtime`, and `test`. Table 4.2 provides a detailed breakdown of these dependency scopes.

Two additional findings emerged during the exploratory data analysis, indicating potential weaknesses in the development processes used in **FinCompany**. The first finding,

Table 4.1: Descriptive statistics of the projects' dependencies

	Q1	**Q3**	**mean**	**stdev**	**max**
#dependencies	202	361	335.40	237,91	1,669
#direct dependencies	30	132	113.57	144,58	994
#transitive dependencies	139	235	221.99	190,49	1,613
#deployed dependencies	190	333	314.81	237,54	1,641

# - quantity of, Q1 - first quartile, Q3 - third quartile

Table 4.2: Dependency relationship scopes

Scopes	**# Relationships**
compile	1,061,893
provided	235,990
test	88,646
compile, provided	22,388
compile, test	13,596
compile, runtime	6,408
runtime	4,488
provided, test	1,056
compile, provided, test	476
compile, runtime, test	318
compile, provided, runtime	174
provided, runtime	48
runtime, test	38
compile, provided, runtime, test	6

# - quantity of

identified during the dependency resolution phase, suggests that allowing developers to reuse libraries from non-official repositories poses a risk. This practice enables the introduction of project-local dependencies, i.e., “.jar” files stored within a project’s repository and referenced in the “pom.xml”. In certain cases examined, these files were named similarly to open-source projects, hindering the verification of their integrity. This ambiguity raises concerns about potential internal threats of tampering with the code or the unintentional use of compromised third-party libraries. Malicious actors could exploit this by redirecting to unmapped or intentionally modified dependencies to carry out malicious activities.

The second finding, observed during the vulnerability identification phase, highlights a discrepancy in the software build process between projects and dependencies. This divergence exposes the company to risks and affects the accuracy of our analysis, as company-developed dependencies are not subjected to Static Application Security Testing (SAST). The **FinCompany** CI/CD pipeline is designed to perform security checks after artifacts have been published to internal repositories. This design results in a lack of security controls for projects that are not directly deployed but are merely published to the repositories, reducing the visibility of vulnerabilities within these dependencies. The need for assessing internally developed code was highlighted by Gkortzis, Feitosa, and Spinellis [19], whose research suggests that the median vulnerability density in both native and reused code is comparable, indicating that both codebases may pose similar security risks.

4.1.2 Assessment for RQ1: What are the characteristics of the vulnerabilities discovered in the FinCompany’s project dependencies?

Regarding our first research question, the analysis shows that vulnerabilities are widespread. Of the 2,140 versions across 585 projects in our dataset, 2,130 (99.53%) contain at least one identified vulnerability in their deployed dependencies. This trend holds even when focusing solely on critical vulnerabilities, with 2,125 versions affected.

By analyzing all versions, we can observe trends such as the evolution of the number of vulnerabilities over time for each project. To illustrate this analysis and underscore the absence of a systematic approach to addressing vulnerability remediation in dependencies, Figure 4.2 presents the number of vulnerabilities found in selected project versions. These versions were chosen from a subset of projects that underwent more than 25 deployments within the analysis period.

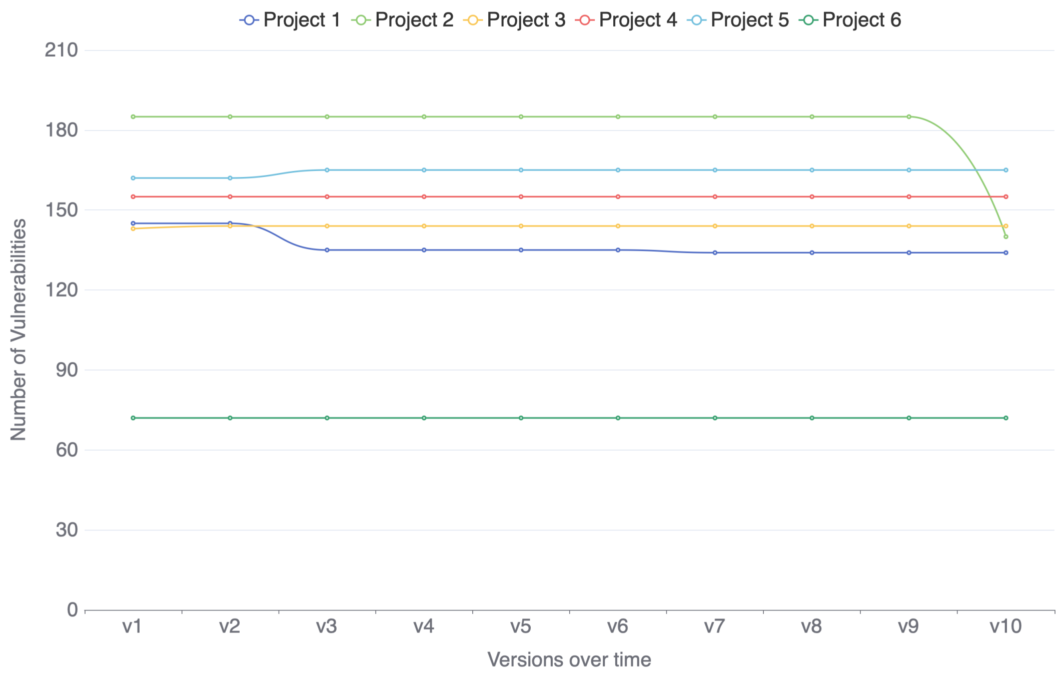


Figure 4.2: Vulnerability count in deployed projects over the observation windows

It is evident from the data that only a single project exhibited a decrease in the number of vulnerabilities. Two additional projects show minor fluctuations in the number of vulnerabilities, with some experiencing an increase and others a decrease. However, three projects maintained a consistent vulnerability count throughout the observation period.

The investigation revealed that 647 (12.56%) of the 5,150 external Java dependencies in production at the company are inherently vulnerable. Additionally, our analysis identified 555 unique vulnerabilities within the deployed dependencies, with a significant proportion classified as high severity (246) or critical (88). The pervasive nature of these vulnerabilities is underscored by the average of 63.51 high-severity and 28.46 critical vulnerabilities per project. These findings emphasize the need to address vulnerable dependencies, for instance, by upgrading library versions.

Table 4.3 highlights the CWE identifiers that appear the most and are associated with the vulnerabilities found in the systems and their dependencies. A comparative analysis of the presented list reveals that 7 out of the 10 most frequently identified CWEs within **FinCompany** align with those featured in The CWE Top 25 Most Dangerous Software Weaknesses List 2024[2], a comprehensive compilation curated by MITRE that underscores the most critical and widespread vulnerabilities reported in the current year.

[2] `https://cwe.mitre.org/top25/`

Table 4.3: CWE affecting dependencies

CWE ID	**CWE Description**	**# Vulnerabilities**
CWE-502	Deserialization of Untrusted Data	122
CWE-20	Improper Input Validation	47
CWE-400	Uncontrolled Resource Consumption	46
CWE-611	Improper Restriction of XML External Entity Reference	26
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	26
CWE-770	Allocation of Resources Without Limits or Throttling	24
CWE-79	Improper Neutralization of Input During Web Page Generation	20
CWE-787	Out-of-bounds Write	19
CWE-835	Loop with Unreachable Exit Condition	17
CWE-22	Improper Limitation of a Pathname to a Restricted Directory	16

# - quantity of

Findings from RQ1: We observed two key findings:
– An analysis of 2,140 projects revealed that more than 99% contain at least one critical vulnerability within their dependencies. The average number of critical vulnerabilities per project was 28.46.
– The analysis revealed that the vulnerabilities are associated with 647 (12.56%) dependencies, suggesting a prevalent practice of library reuse within the analyzed systems.

4.1.3 Assessment for RQ2: What is the impact of fixing VUDDs on the overall number of projects vulnerabilities?

For our second research question, a "more secure version" is defined as one with no reported vulnerabilities. As the remediation process matures, this definition could be refined to include parametrization, aiming to reduce risk exposure when complete elimination of vulnerabilities is not feasible. For instance, in the absence of dependencies with no reported vulnerabilities — based on the data provided by the commercial tool — a parameter configuration might prioritize dependencies with fewer critical or high vulnerabilities or those with a smaller number of exploitable vulnerabilities. Our analysis found that, among the 930 direct vulnerable dependencies, 852 (91.61%) have newer versions available. Of these, 529 (62.09%) meet our definition of a more secure version.

For instance, 92 projects directly depend on version 2.9.6 of library Dependency J[3], which contains 62 reported vulnerabilities. By upgrading to version 2.10.1, all these vulnerabilities are fixed. Further examples are Dependency K, reliant upon 139 projects and containing 61 known vulnerabilities, and Dependency L, dependent on 114 projects

[3] Hidden names of dependencies due to confidentiality restrictions

with 101 vulnerabilities. By upgrading Dependency L to versions devoid of these vulnerabilities, the overall vulnerability landscape of the environment can be significantly improved.

We found that updating the 529 identified VUDDs will enable us to mitigate vulnerabilities in a significant portion of our projects, specifically 568 projects (97.09%). This leads to a reduction of 66.13% in the total number of unique vulnerabilities, decreasing from 555 to 188. In particular, the reduction for critical vulnerabilities reaches 68.18%, as detailed in Table 4.4.

Table 4.4: Vulnerabilities remediable

Severity	**#Identified**	**#Remediable**	**Reduction (%)**
Critical	88	60	68.18%
High	246	168	67.47%
Medium	207	130	62.80%
Low	14	9	64.29%
Total	555	367	66,13%

# - quantity of

The impact of VUDD remediation on the benchmark projects presented in Figure 4.1 is depicted in Figure 4.3. The upper panels display the original dependency graphs, which serve as the baseline for comparison, whereas the lower panels illustrate the projected outcomes assuming the update of VUDDs and the corresponding remediation of known vulnerabilities. A consistent reduction in vulnerabilities is observed across all three projects. Project A shows a substantial decrease in vulnerability clusters, particularly in the upper-left quadrant. Project B also reveals a non-negligible reduction of vulnerabilities, while Project C exhibits a more diffuse pattern of vulnerability remediation. It is important to highlight that, although beyond the primary scope of this study, Project C also experienced indirect remediation of vulnerabilities associated with test-scoped dependencies.

Findings from RQ2:
– Over 60% of the direct dependencies identified as vulnerable have newer versions available that do not contain any reported security flaws.
– Our approach can lead to a reduction of 66.13% in the total number of unique vulnerabilities, decreasing from 555 to 188.

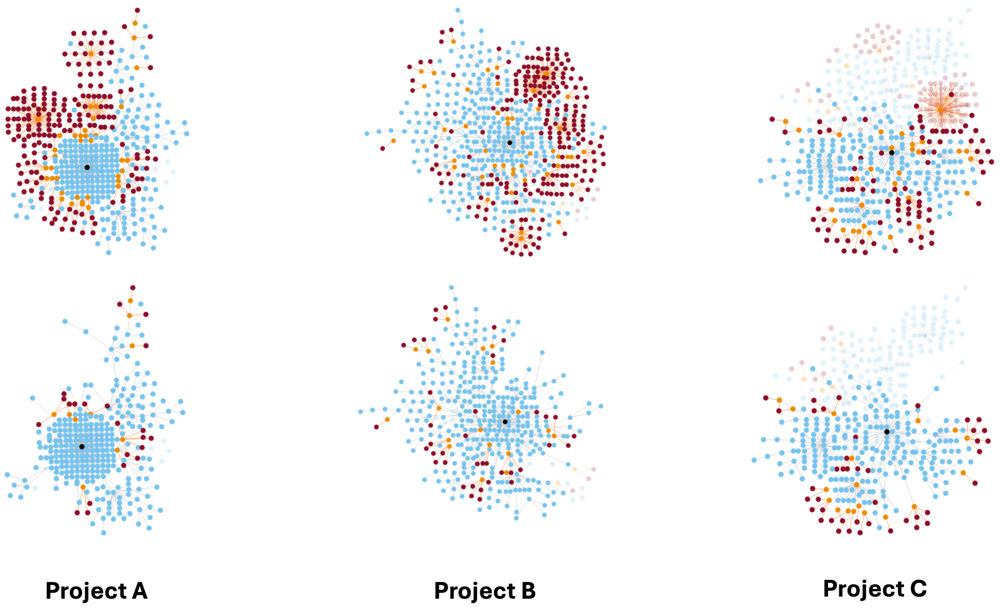


Figure 4.3: Dependency graph of versions of projects after fixing VUDDs

4.1.4 Assessment for RQ3: Given how vulnerabilities propagate, what strategies can be adopted to update libraries to secure versions?

Updating vulnerable dependencies is a non-trivial task, and implementing a company-wide strategy to do so poses additional challenges. A primary challenge is identifying vulnerable dependencies and determining appropriate replacement versions, a challenge that our approach addresses. Furthermore, studies [53, 54, 55] have shown that developers often hesitate to update dependencies due to the fear of introducing unintended consequences.

Given these findings, a key question arises: What steps should the teams take to upgrade the vulnerable dependencies? Reasoning about this question might provide actionable insights that could help development teams take more targeted actions to remediate vulnerabilities that spread with the reuse of internal and external libraries. In what follows, we outline the possible strategies we envisioned and their implications. In Section 5.1.1, we discuss the strategy that **FinCompany** considers most appropriate.

A first prioritization strategy might rely on the VUDDs that impact most projects. In the context of our study, the top three VUDDs affect 273, 260, and 256 projects, respectively. This insight enables the team responsible for defining remediation strategies to focus their efforts effectively. For example, they could provide detailed guidance on updating these dependencies and address potential issues related to the updates, thereby minimizing the effort required from developers.

A second strategy could focus on identifying the VUDDs with the most critical vulnerabilities. The top three VUDDs in this category have 51, 30, and 30 critical vulnerabilities, respectively. This perspective enables a company to prioritize the remediation process by addressing the VUDDs that pose the most significant risk, even if they are present in fewer projects. This approach can help reduce overall remediation efforts while maximizing risk mitigation.

Indeed, one can also combine the first and second perspectives by focusing on VUDDs that have, for instance, at least 15 critical vulnerabilities and impact at least 10 projects. This strategy would allow us to prioritize dependencies that pose significant risks while affecting many projects, ensuring a balanced and effective remediation effort. We have identified eight VUDDs that satisfy these conditions, as outlined in Table 4.5.

Table 4.5: VUDDs presenting a high-risk profile, with at least 15 critical vulnerabilities and affecting at least 10 projects.

VUDD[a]	**#Critical vulnerabilities**	**#Vulnerabilities**	**#Affected Projects**
Dependency G	26	73	122
Dependency U	26	108	24
Dependency V	25	56	20
Dependency W	25	56	20
Dependency X	26	73	18
Dependency Y	26	73	17
Dependency Z	26	73	15
Dependency AA	25	56	14

# - quantity of, [a]The dependencies G, X and Y are internal.

An alternative approach is to adopt a broader strategy that prioritizes updating dependencies based on vulnerability severity, starting with the most critical vulnerabilities and gradually expanding the scope. This method strikes a balance by incrementally addressing vulnerabilities across all projects while minimizing disruptions to development timelines and business objectives. While this approach may temporarily leave the company exposed to vulnerabilities beyond the initial focus, it provides a more sustainable and manageable long-term solution.

Findings from RQ3: Our approach enables balance in prioritizing vulnerability resolution by supporting the institution in selecting an equitable strategy that addresses vulnerabilities without significantly increasing the teams' remediation effort and compromising business requirements.

4.2 Study 2: Assessing the compatibility of the proposed new versions

The second study is aimed at assessing the compatibility of the new versions identified for the VUDDs. Our underlying premise guides this investigation: to pinpoint updates that concurrently mitigate the quantity of project vulnerabilities and avoid the utilization of an excessive amount of resources by the development teams. This dual focus ensures that the proposed remediation strategies are both effective from a security perspective and sustainable in terms of developer effort.

In this regard, the chkVUDD Module of the *Ready-to-Update* was deployed within the **FinCompany**. The execution of this step was directly aligned to respond to the subsequent **research questions**:

RQ4 *What are the costs associated with remediating VUDDs, taking into account potential compatibility issues between the updated secure versions and the versions presently deployed?*

After the execution of chkVUDD, the compatibility rates of the projects with the identified new versions were measured by observing: the percentage of projects achieving successful compilation, the percentage of projects successfully concluding the testing procedure, and the resultant ratio between them. The same analytical approach was also applied exclusively to the projects compatible with the blossom update, facilitating a direct comparison between the outcomes of performing individual updates and those yielded by the collective blossom update approach.

We first present the results of an additional exploratory data analysis, followed by a discussion addressing this research question.

4.2.1 Exploratory Data Analysis

To conduct this second study, we identified which of the 585 projects analyzed in Study 1 had new production deployments between December 2024 and July 2025, approximately one year after the data collection period of Study 1. Out of the 585 projects, 82 had no new deployments during this period. This absence of activity may be attributed to factors such as project inactivity. We decided to exclude these projects from the second study, as project maintainers were likely to reject updates to newer library versions.

According to the procedures detailed in Section 3.2, we cloned all remaining 503 projects to identify their Java versions. This step revealed that seven projects were running on Java 6. Given that **FinCompany**'s policy discourages new Java 6 deployments and mandates migration to more recent versions, these projects are slated for discontinuation and were therefore excluded from the second study. Two additional projects were removed from subsequent analyses because their source code could not be accessed due to permission restrictions.

Table 4.6: Project distribution by Java version

Java Version	Number of Projects
7	9
8	444
11	11
17	5
21	25

Our analysis reveals a strong concentration of projects using Java 8, indicating that a substantial portion of these systems began development several years ago, as Java 9 was released in 2017. Nevertheless, the presence of projects on Java 17 and Java 21 indicates a trend within the **FinCompany** toward codebase modernization.

The procedures outlined in the first study (Section 3.1) were applied to the 494 projects that remained after the exclusions described above. An examination of the latest deployed version of each project revealed that 493 of them (99.8%) still contained vulnerabilities, a result consistent with the findings of the initial study. However, the proportion of projects with critical vulnerabilities decreased to 73.5%. This reduction may be associated with a company-wide initiative launched in December 2024 to remediate vulnerabilities in project dependencies. During the period of the second study, only critical vulnerabilities with a CVSS score above a predefined threshold were prioritized for remediation. There were 596 unique vulnerabilities identified, whose severities are presented in Table 4.7.

Table 4.7: Vulnerability Distribution by Severity

Severity	Number of Vuilnerabilities
Critical	93
High	271
Medium	216
Low	16

These vulnerabilities are associated with 704 dependencies. Upon examining the direct dependencies connected to the vulnerable dependencies, we identified 243 direct dependencies, of which 146 (60.1%) have newer and more secure versions. In aggregate, these 243 dependencies have 1,304 available new versions, ranging from 1 to 109, with a mean of 12.49 and a median of 5.

These 243 dependencies, in turn, affect 352 projects (71,2%). These projects were then put through compilation and testing processes to create a baseline, in accordance with Section 3.2. For the baseline, all projects successfully compiled and passed their tests.

4.2.2 Assessment for RQ4: What are the costs associated with remediating VUDDs, taking into account potential compatibility issues between the updated secure versions and the versions presently deployed?

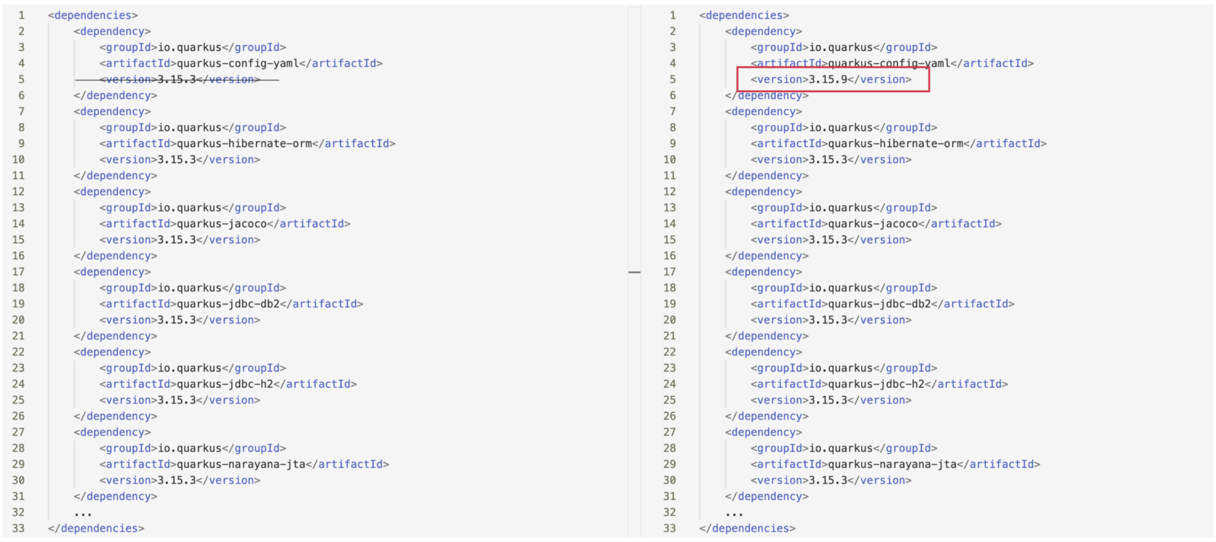
One of the challenges related to updating dependencies is the total number of available combinations. In our context, from the 392 projects affected by VUDDs, 243 dependencies, and their 1,304 available new versions collectively resulted in 11,571 combinations. Indeed, due to usage restrictions, 56 of the 1,304 identified dependencies were inaccessible to the organization's package manager. As a result, these dependencies were not included in the final analytical sample.

Every one of the remaining 11,079 combinations was tested by exclusively updating the vulnerable version (section 3.2.2), while also verifying and applying the update when feasible, using the blossom update method (4,690 cases). Within the scope of this investigation, these evaluations are conducted for every available updated version to determine the overall compatibility of the updated releases. However, in a real-world usage scenario, the directive is to pursue the most current available version that maintains compatibility with the project. Accordingly, the RTU system can be parameterized to optimize this process by commencing the search from the latest version and ceasing execution as soon as a compatible version is encountered.

When observing the application of dependency updates on a broad scale, we found that 75.7% of updates were compatible during the compilation process. This number decreased to 71,4% during the testing phase (or 94.2% when only considering the updates that successfully completed compilation). To the best of our knowledge, we are the first to conduct an exhaustive evaluation of the compatibility across all non-vulnerable versions available for dependency updates within a project. This distinguishes our research from previous efforts, as the identified related literature focuses on the application of a constrained group of specific fixes and compatibility tests, such as those presented by Dann, Hermann, and Bodden [6].

Focusing on the subset of projects compatible with the blossom update, we observed a compilation success rate of 95.5% for individual dependency updates, and 93.4% for group updates of all libraries. In the testing phase, 88.5% of the projects that compiled successfully also passed their tests with individual updates, compared to 85.3% for group updates. This finding challenges the hypothesis proposed by Pashchenko et al. [5] and Dann, Hermann, and Bodden [6]. Although group updates can enhance compatibility among dependencies within the updated set, they simultaneously require upgrading a larger number of transitive dependencies, thereby increasing the likelihood of conflicts with other project components. The concept is more clearly demonstrated by the subsequent example.

The Figure 4.4 represents the individual upgrade scenario for a library compatible with the blossom update. In the hypothetical case shown, the direct dependency upgrade incorporates **6 transitive dependencies**.

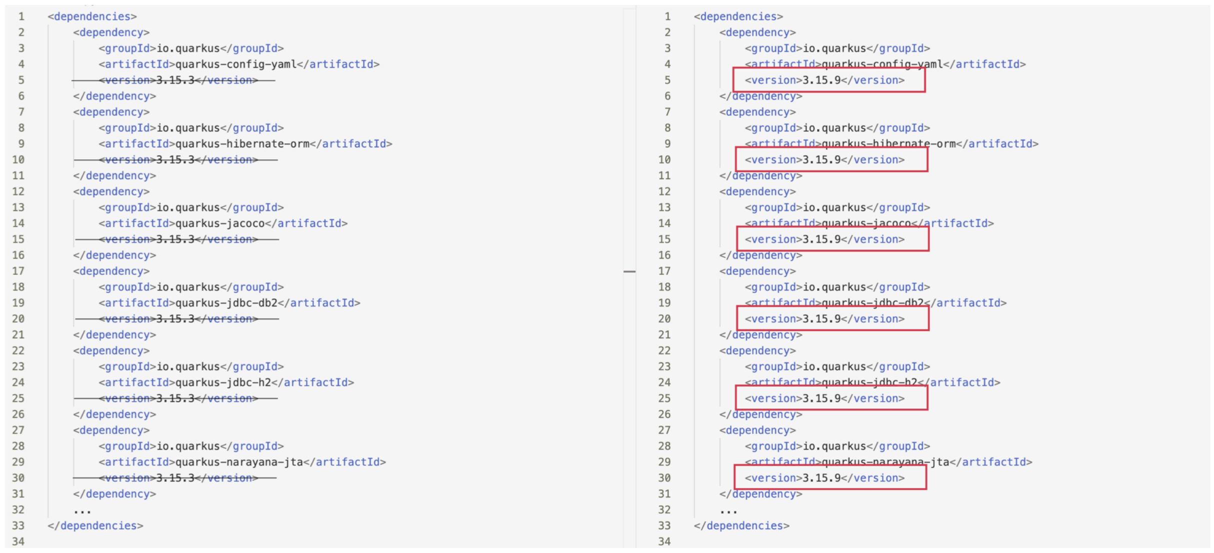


```
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-config-yaml</artifactId>
        ~~<version>3.15.3</version>~~
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-hibernate-orm</artifactId>
        <version>3.15.3</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jacoco</artifactId>
        <version>3.15.3</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-db2</artifactId>
        <version>3.15.3</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-h2</artifactId>
        <version>3.15.3</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-narayana-jta</artifactId>
        <version>3.15.3</version>
    </dependency>
    ...
</dependencies>
```

```
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-config-yaml</artifactId>
        <version>3.15.9</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-hibernate-orm</artifactId>
        <version>3.15.3</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jacoco</artifactId>
        <version>3.15.3</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-db2</artifactId>
        <version>3.15.3</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-h2</artifactId>
        <version>3.15.3</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-narayana-jta</artifactId>
        <version>3.15.3</version>
    </dependency>
    ...
</dependencies>
```

Figure 4.4: Individual update of a dependency

When employing the blossom update, however, as visualized in Figure 4.5, the collective update of all direct dependencies involves **51 transitive dependencies**, which leads to an increased risk of inter-library conflict.



```
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-config-yaml</artifactId>
        ~~<version>3.15.3</version>~~
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-hibernate-orm</artifactId>
        ~~<version>3.15.3</version>~~
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jacoco</artifactId>
        ~~<version>3.15.3</version>~~
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-db2</artifactId>
        ~~<version>3.15.3</version>~~
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-h2</artifactId>
        ~~<version>3.15.3</version>~~
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-narayana-jta</artifactId>
        ~~<version>3.15.3</version>~~
    </dependency>
    ...
</dependencies>
```

```
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-config-yaml</artifactId>
        <version>3.15.9</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-hibernate-orm</artifactId>
        <version>3.15.9</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jacoco</artifactId>
        <version>3.15.9</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-db2</artifactId>
        <version>3.15.9</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-h2</artifactId>
        <version>3.15.9</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-narayana-jta</artifactId>
        <version>3.15.9</version>
    </dependency>
    ...
</dependencies>
```

Figure 4.5: Blossom update of a dependency

Table 4.8 shows how the compilation and testing success rates vary across Java versions.

Table 4.8: Compilation Distribution by Java Version

Java Version	**#Projects**	**#Success Compilation**	**%**
7	713	475	66.6%
8	8,523	6,075	71.3%
11	913	912	99.9%
17	193	193	100.0%
21	737	737	100.0%

Based on the table, it is evident that the lowest success rates are associated with older Java versions (7 and 8). While the low success rate for Java 8 could be attributed to its high prevalence in the projects, the similarly low percentage for Java 7 suggests a broader trend. It is plausible that, over time, dependencies have dropped support for older Java versions. This presents an additional motivation for source code rejuvenation [56]: to ensure compatibility with newer and more secure versions of dependencies.

The results discussed previously, however, take a broad view, evaluating all 11,079 possible combinations. To adequately address our research question, we might instead identify just one more up-to-date and secure version that is compatible with the project in which it is being used. From this perspective, our data reveals that 96.4% of the VUDDs have at least one compatible version at the compilation stage, with this compatibility rate being 86.3% during test execution.

Analogous to the previous findings, we verified the impact on the dependencies where the blossom update mechanism was applicable. Analysis of the individual dependency update within this sample indicated that 97.24% had at least one compatible version when examining the build process, and 92.63% when considering the test execution process (96.26% of the successfully compiled instances). Furthermore, the performance resulting from the blossom update proved to be inferior: 94.01% of the projects successfully compiled, whereas 90.78% concluded the tests (representing 96.57% of the successfully compiled subset).

Table 4.9 highlights that, despite the trend of dependencies losing compatibility with previous Java versions over time, it remains possible to identify compatible versions for the projects that can resolve the identified vulnerabilities.

Table 4.9: Compilation Distribution by Java Version by VUDD

Java Version	**#Projects**	**#Success Compilation**	**%**
7	30	29	96.7%
8	991	951	96.0%
11	78	77	98.7%
17	18	18	100.0%
21	55	55	100.0%

Findings from RQ4: We observed three key findings:
– The success rate for compatible updates was higher for single-vulnerable-dependency updates than for the blossom update approach.
– More modern Java versions exhibited greater compatibility with newer dependency versions.
– The *Ready-to-Update* approach successfully identified updated, more secure, and compatible versions for 86.3% of vulnerable direct dependencies.

Chapter 5

Final Remarks

Software reuse is a common and efficient practice in modern development, but it may introduce risks by propagating vulnerabilities from reused open- and closed-source libraries into both new and legacy systems [19]. To mitigate this issue, this paper presented an approach tailored to the context of a large financial company (**FinCompany**), building on previous research [5, 6] to identify and characterize vulnerable libraries.

After analyzing over 2,000 revisions of Java projects in the **FinCompany**, we found that more than 90% of the revisions deployed between January and July 2024 and December 2024 and July 2025 contained high-severity or critical vulnerabilities. Nevertheless, a significant proportion of these vulnerabilities are amenable to correction through the simple measure of updating direct dependencies to newer, secure versions. Our analysis confirms that more than 60% of the dependencies possess such updated versions, with around 86% of those demonstrating project compatibility, thus requiring developers only to implement the available version upgrades.

The results provided several insights into alternative approaches for prioritizing which dependencies should be updated first, which emerged after sharing our findings with the risk, cybersecurity, and development teams within the company. We hope that the approach presented here, along with the findings and insights shared, could help other companies address software that depends on vulnerable dependencies.

5.1 Discussion

In this section, we discuss the implications of our research after applying the *Ready-to-Update* approach at **FinCompany** and presenting the results to the risk, cyber security, and technology teams (Section 5.1.1). Additionally, we outline potential threats to the validity of our work, which are inherent to this type of research (Section 5.1.2).

5.1.1 Research Implications to the FinCompany

Our approach, which incorporates a web portal, is designed to meet the requirements outlined in collaborative meetings between risk, cybersecurity, and technology teams, attended by a subgroup of the authors. The discussions highlighted the need for increased visibility into vulnerabilities across the organization and the prioritization of remediation efforts, addressed by the portal.

The results obtained in RQ1 and RQ2 were presented to two of the **FinCompany**'s stakeholders: the Risk Management Department and the Cyber Security Department. The Risk Management Department decided to use the approach presented in cyber risk management, creating an exposure indicator related to vulnerable dependencies. The indicator is designed to provide a monthly assessment of the percentage of VUDDs present in deployed projects, thereby tracking the trend and effectiveness of the implemented process within **FinCompany**.

The **FinCompany**'s Board of Directors decided that, after a period of validation of the results presented, the indicator will become part of the corporate indicator of exposure to cyber risks, linked to its Risk Appetite Statement (RAS). The Cyber Security Department is evaluating the feasibility of including the process within the CI/CD pipeline, considering the gains in visibility and the support developers can receive in identifying remediation options.

With respect to the approaches for prioritizing vulnerability fixes outlined in RQ3, they were introduced and discussed with practitioners in three meetings. The goal was to identify the most appropriate approach for **FinCompany**. Discussions indicated that while approaches prioritizing certain groups, such as projects with more vulnerabilities or vulnerabilities found in multiple projects, offer a greater potential for remediating a larger number of vulnerabilities in a shorter time, these approaches may overload teams or areas responsible for specific project groups. This could potentially compromise the timely delivery of functionalities and essential services necessary to support the corporate strategy.

Conversely, updating dependencies based on the severity of identified vulnerabilities offers a more effective means of distributing the workload among teams, mitigating impact and enabling a more staggered approach to dependency updates. The CI/CD team noted that this approach streamlines the implementation of process rules, as only one variable—the severity of the vulnerability—requires evaluation. Consequently, this was the approach adopted by **FinCompany** to address the challenge of updating vulnerable dependencies.

The process was implemented in December 2024, and mandates that projects with vulnerable dependencies meeting the established criteria must adhere to a defined timeline for dependency updates. This aligns with the company's broader vulnerability remediation processes and its defined cyber risk appetite.

5.1.2 Threats to Validity

Concerning replicability, to ensure the confidentiality of the organization's sensitive information, the specific dependencies and vulnerabilities could not be disclosed in a replication package. While this decision limits the potential for replication (**introducing a reproducibility and replication threat**), protecting the organization from potential security threats was deemed necessary. Balancing the need for research with the imperative of safeguarding sensitive information was a key consideration in designing this study.

Regarding **construct validity**, due to the absence of SAST and similar security analyses for internal dependencies, the vulnerability assessment was limited to transitive dependencies. As a result, the existence of vulnerabilities within internal dependencies cannot be ruled out, potentially leading to a lower number of identified vulnerabilities.

Furthermore, vulnerability identification is contingent upon a commercial solution, which employs proprietary methods to associate vulnerabilities with dependencies. While the literature presents a variety of methods for identifying vulnerabilities in dependencies [37, 6, 57, 5, 19, 36, 53, 38, 58, 40, 39], the company has opted for a commercial tool as a strategic choice for establishing a specific vulnerability management process, given the reduced implementation and maintenance efforts.

Regarding the generalization of our results (**threats to external validity**), although this study focused on the projects of a specific company, the more than 5,000 dependencies analyzed were related to the use of open-source libraries available in central repositories and addressed known vulnerabilities, primarily accessible in public repositories. This suggests that the findings may apply to a broader range of software systems that rely on open-source components.

Also, despite being initially applied to Java projects, the proposed approach exhibits the flexibility to accommodate projects written in other programming languages, such as JavaScript and Python, or even containerized applications. For JavaScript-related analyses, for example, it is necessary to consider the common practice of using semantic versioning to specify acceptable update types[1]. By performing analyses as close to the build as possible, ideally during the build itself, we can ensure that the analyzed versions accurately reflect the deployed versions. This is necessary to account for the possibility of automatic updates that may have been accepted by developers.

[1]About semantic versioning. `https://docs.npmjs.com/about-semantic-versioning`

The selection of the most suitable methodology to address vulnerability remediation is dependent upon the organization's risk tolerance, the degree to which software development supports its global business strategy, its readiness to identify potential exploits of existing vulnerabilities, and its acceptance of potential delays in business initiatives to address identified vulnerabilities, reflecting a strategic balancing of risks and rewards. Our approach, applicable to any prioritization strategy for addressing identified vulnerabilities, enables the identification of vulnerabilities that can be remediated throughout the dependency graph.

5.2 Positioning This Thesis in the Context of Related Work

While our work builds upon prior research—reusing essential steps such as dependency resolution and vulnerability identification—**we focus on the *practical implications of identifying, characterizing, and upgrading vulnerable dependencies within a real-world financial setting***. Specifically, we assess the availability of updated and more secure versions of vulnerable dependencies using a ready-to-update strategy, which distinguishes our work from previous studies. Nonetheless, the foundational contributions of prior research remain central to our approach.

The primary contribution of Pashchenko et al. [5] lies in their observation that a one-size-fits-all solution is often unsuitable for specific organizational contexts. This insight motivated us to analyze the **FinCompany** and recognize that, in the absence of an established approach for managing dependency vulnerabilities, a lightweight and low-risk solution—one that provides rapid results with minimal disruption to developers—would be more effective in the short to medium term than a resource-intensive approach aiming for complete vulnerability remediation. Pashchenko et al. [5] also provided the initial high-level design, later refined by Dann, Hermann, and Bodden [6], outlining the key steps in our approach: dependency resolution and vulnerability identification.

Dann, Hermann, and Bodden [6] advanced Pashchenko's proposal in two main ways. First, they introduced an improved dependency-resolution tool that streamlined the overall process and enhanced efficiency. Their most significant contribution, however, was the emphasis on verifying the compatibility—binary, source, and semantic—of newer dependency versions. The rationale is that if code refactoring becomes necessary to accommodate a dependency update, the resulting cost may offset the benefits of adopting a more secure version. The method proposed by Dann, Hermann, and Bodden [6] aims to determine the minimal sequence of update steps needed to migrate a vulnerable dependency from its current to a target version, relying on a comprehensive database of dependencies from the Maven repository. Their primary goal was to specify these update paths, while compatibility tests served mainly to benchmark their approach against alternative methods.

Therefore, our approach surpasses the limitations of prior studies, whereas Pashchenko et al. [5] emphasized identifying dependencies that are potentially abandoned and lack future releases to address known vulnerabilities, our approach focuses on highlighting direct dependencies that already have more secure versions available for immediate update by developers.

In contrast with Dann, Hermann, and Bodden [6], our approach identifies which versions of direct dependencies simultaneously satisfy two conditions: improved security and compatibility with the project and its remaining dependencies, given a specific vulnerable version.

5.3 Future works

Future work involving industry analyses on this topic could assess the extent to which SAST analysis on internal dependencies affects the total number of institutional vulnerabilities. This would enable an analysis of whether this situation — where internal dependencies are not subjected to SAST tool analysis — is prevalent across the industry and the degree to which companies failing to implement this practice are susceptible to unmapped risks.

The verification of outcomes from the strategies discussed under RQ3 is also a suitable topic for new studies. This would deliver valuable feedback to both academia and industry concerning the adoption of these strategies and the subsequent evolution of vulnerability profiles and organizational exposure throughout the study's timeline.

A further research avenue includes the formulation of strategies aimed at minimizing the correction effort for vulnerabilities whose dependencies do not possess safer versions (e. g., in the case of legacy or discontinued projects).

Ultimately, the impact of work pertaining to code rejuvenation can be observed, particularly the extent to which this modernization effort facilitates the updating of project dependencies to more secure versions by enhancing compatibility.

References

[1] Standards, National Institute of and Technology (U.S.): *CVE-2021-44228 Detail*. https://nvd.nist.gov/vuln/detail/cve-2021-44228, 2021. Accessed: 2024-12-05. 1

[2] Stalnaker, Trevor, Nathan Wintersgill, Oscar Chaparro, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk: *BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems*. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, page 1–13. ACM, February 2024. http://dx.doi.org/10.1145/3597503.3623347. 1, 8

[3] Nocera, Sabato, Massimiliano Di Penta, Rita Francese, Simone Romano, and Giuseppe Scanniello: *If it's not SBOM, then what? How Italian Practitioners Manage the Software Supply Chain*. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 730–740, 2024. 1, 7, 9

[4] Massacci, Fabio and Laurie Williams: *Software Supply Chain Security [Guest Editors' Introduction]*. IEEE Security & Privacy, 21(6):8–10, 2023. 2, 8, 9

[5] Pashchenko, Ivan, 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, New York, NY, USA, 2018. Association for Computing Machinery, ISBN 9781450358231. https://doi.org/10.1145/3239235.3268920. 2, 3, 10, 11, 12, 14, 20, 39, 43, 45, 46, 47

[6] Dann, Andreas, Ben Hermann, and Eric Bodden: *UPCY: Safely Updating Outdated Dependencies*. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 233–244, 2023. 3, 10, 11, 12, 23, 39, 43, 45, 46, 47

[7] Naur, Peter and Brian Randell (editors): *Proceedings, NATO Conference on Software Engineering*, Garmisch, Germany, 1968. 6

[8] Krueger, Charles W.: *Software reuse*. ACM Comput. Surv., 24(2):131–183, June 1992, ISSN 0360-0300. https://doi.org/10.1145/130844.130856. 6

[9] Biggerstaff, Ted J. and Charles Richter: *Reusability Framework, Assessment, and Directions*. IEEE Software, 4:41–49, 1987. https://api.semanticscholar.org/CorpusID:17640010. 6

[10] Capilla, Rafael, Barbara Gallina, Carlos Cetina, and John Favaro: *Opportunities for software reuse in an uncertain world: From past to emerging trends.* Journal of Software: Evolution and Process, 31(8):e2217, 2019. `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2217`, e2217 smr.2217. 6

[11] Arvanitou, Elvira Maria, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Jeffrey C. Carver: *Software engineering practices for scientific software development: A systematic mapping study.* Journal of Systems and Software, 172:110848, 2021, ISSN 0164-1212. `https://www.sciencedirect.com/science/article/pii/S0164121220302387`. 6

[12] Zaimi, Asimina, Apostolos Ampatzoglou, Noni Triantafyllidou, Alexander Chatzigeorgiou, Androklis Mavridis, Theodore Chaikalis, Ignatios Deligiannis, Panagiotis Sfetsos, and Ioannis Stamelos: *An Empirical Study on the Reuse of Third-Party Libraries in Open-Source Software Development.* In *Proceedings of the 7th Balkan Conference on Informatics Conference*, BCI '15, New York, NY, USA, 2015. Association for Computing Machinery, ISBN 9781450333351. `https://doi.org/10.1145/2801081.2801087`. 6

[13] Haefliger, Stefan, Georg Von Krogh, and Sebastian Spaeth: *Code Reuse in Open Source Software.* Management Science, 54(1):180–193, January 2008, ISSN 0025-1909, 1526-5501. `https://pubsonline.informs.org/doi/10.1287/mnsc.1070.0748`. 6

[14] McIlroy, M. D.: *Mass-produced software components.* Proc. NATO Conf. on Software Engineering, Garmisch, Germany, 1968. 7

[15] Vale, Tassio, Ivica Crnkovic, Eduardo Santana de Almeida, Paulo Anselmo da Mota Silveira Neto, Yguaratã Cerqueira Cavalcanti, and Silvio Romero de Lemos Meira: *Twenty-eight years of component-based software engineering.* Journal of Systems and Software, 111:128–148, 2016, ISSN 0164-1212. `https://www.sciencedirect.com/science/article/pii/S0164121215002095`. 7

[16] Szyperski, C., D. Gruntz, and S. Murer: *Component Software: Beyond Object-oriented Programming.* ACM Press Series. ACM Press, 2002, ISBN 9780201745726. `https://books.google.com.br/books?id=U896iwmtiagC`. 7

[17] Chen, Xingru, Muhammad Usman, and Deepika Badampudi: *Understanding and evaluating software reuse costs and benefits from industrial cases—A systematic literature review.* Information and Software Technology, 171:107451, 2024, ISSN 0950-5849. `https://www.sciencedirect.com/science/article/pii/S0950584924000569`. 7, 10

[18] Gupta, Anita, Jingyue Li, Reidar Conradi, Harald Rønneberg, and Einar Landre: *A case study comparing defect profiles of a reused framework and of applications reusing it.* Empirical Software Engineering, 14(2):227–255, Apr 2009, ISSN 1573-7616. `https://doi.org/10.1007/s10664-008-9081-9`. 7

[19] Gkortzis, Antonios, Daniel Feitosa, and Diomidis Spinellis: *A Double-Edged Sword? Software Reuse and Potential Security Vulnerabilities.* In Peng, Xin, Apostolos Ampatzoglou, and Tanmay Bhowmik (editors): *Reuse in the Big Data Era*, pages 187–203, Cham, 2019. Springer International Publishing, ISBN 978-3-030-22888-0. 7, 9, 31, 43, 45

[20] Cox, Russ: *Surviving software dependencies.* Commun. ACM, 62(9):36–43, August 2019, ISSN 0001-0782. `https://doi.org/10.1145/3347446`. 7

[21] Crossley, C.: *Software Supply Chain Security: Securing the End-to-end Supply Chain for Software, Firmware, and Hardware.* O'Reilly Media, 2024, ISBN 9781098133702. 7, 8

[22] Balliu, Musard, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero, and Martin Wittlinger: *Challenges of Producing Software Bill of Materials for Java.* IEEE Security & Privacy, 21(6):12–23, 2023. 7, 8

[23] National Telecommunications and Information Administration (NTIA): *Minimum Elements For a Software Bill of Materials (SBOM)*, 2021. `https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom`, visited on 2025-04-24. 8

[24] House, The White: *Executive Order 14028 on Improving the Nation's Cybersecurity.* `https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/`, 2021. [Accessed 12-08-2024]. 8, 9

[25] Torres-Arias, Santiago, Dan Geer, and John Speed Meyers: *A Viewpoint on Knowing Software: Bill of Materials Quality When You See It.* IEEE Security & Privacy, 21(6):50–54, 2023. 8

[26] Wang, Ying, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu: *Will Dependency Conflicts Affect My Program's Semantics?* IEEE Transactions on Software Engineering, 48(7):2295–2316, 2022. 8

[27] Synopsys: *2024 Open source security and risk analysis report (OSSRA)*, Feb 2024. `https://www.synopsys.com/blogs/software-security/open-source-trends-ossra-report.html`, [Accessed 12-08-2024]. 8

[28] Foundation, OWASP: *OWASP Top 10 - 2021.* Publication, OWASP Foundation, 2021. `https://owasp.org/Top10/`, [Accessed 12-08-2024]. 8

[29] Souppaya, Murugiah, Karen Scarfone, and Donna Dodson: *Secure Software Development Framework (SSDF) version 1.1: recommendations for mitigating the risk of software vulnerabilities.* NIST, February 2022. `http://dx.doi.org/10.6028/NIST.SP.800-218`. 8

[30] Boyens, Jon, Angela Smith, Nadya Bartol, Kris Winkler, Alex Holbrook, and Matthew Fallon: *Cybersecurity supply chain risk management for systems and organizations.* NIST, May 2022. http://dx.doi.org/10.6028/NIST.SP.800-161r1, [Accessed 12-08-2024]. 8

[31] Rice, Tony, Josh Brown-White, Tania Skinner, Nick Ozmore, Nazira Carlage, Wendy Poland, Eric Heitzman, and Danny Dhillon: *Fundamental Practices for Secure Software Development: Essential Elements of a Secure Development Lifecycle Program. Third Edition.* Publication, Software Assurance Forum for Excellence in Code (SAFECode), 2018. https://safecode.org/uncategorized/fundamental-practices-secure-software-development/, [Accessed 12-08-2024]. 9

[32] Bisht, Prithvi, Mike Heim, Manuel Ifland, Michael Scovetta, and Tania Skinner: *Managing Security Risks Inherent in the Use of Third-party Components.* Publication, Software Assurance Forum for Excellence in Code (SAFECode), 2017. https://safecode.org/wp-content/uploads/2017/05/SAFECode_TPC_Whitepaper.pdf, [Accessed 12-08-2024]. 9

[33] Parliament, European and of the Council: *Regulation (EU) 2019/881 of the European Parliament and of the Council of 17 April 2019 on ENISA (the European Union Agency for Cybersecurity) and on information and communications technology cybersecurity certification and repealing Regulation (EU) No 526/2013 (Cybersecurity Act).* https://eur-lex.europa.eu/eli/reg/2019/881/oj, 2019. [Accessed 12-08-2024]. 9

[34] Hendrick, Stephen: *Software Bill of Materials (SBOM) and Cybersecurity Readiness.* Technical report, The Linux Foundation, January 2022. https://www.linuxfoundation.org/research/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readiness, visited on 2025-04-25. 9

[35] O'Donoghue, Eric, Ann Marie Reinhold, and Clemente Izurieta: *Assessing Security Risks of Software Supply Chains Using Software Bill of Materials.* In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, pages 134–140, 2024. 9

[36] Hejderup, Joseph: *In Dependencies We Trust: How vulnerable are dependencies in software modules?* PhD thesis, TU Delft, May 2015. 9, 45

[37] Ponta, Serena Elisa, 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)*, pages 449–460, 2018. 9, 45

[38] Wang, Ying, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu: *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)*, pages 35–45, 2020. 9, 45

[39] Wu, Susheng, Ruisi Wang, Kaifeng Huang, Yiheng Cao, Wenyan Song, Zhuotong Zhou, Yiheng Huang, Bihuan Chen, and Xin Peng: *Vision: Identifying Affected Library Versions for Open Source Software Vulnerabilities.* In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1447–1459, New York, NY, USA, 2024. Association for Computing Machinery, ISBN 9798400712487. `https://doi.org/10.1145/3691620.3695516`. 9, 45

[40] Pashchenko, Ivan, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci: *Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies.* IEEE Transactions on Software Engineering, 48(5):1592–1609, 2022. 10, 45

[41] Di Cosmo, Roberto, Stefano Zacchiroli, and Paulo Trezentos: *Package upgrades in FOSS distributions: details and challenges.* In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, New York, NY, USA, 2008. Association for Computing Machinery, ISBN 9781605583044. `https://doi.org/10.1145/1490283.1490292`. 10

[42] Kerzazi, Noureddine, Foutse Khomh, and Bram Adams: *Why Do Automated Builds Break? An Empirical Study.* In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50, 2014. 10

[43] Fan, Gang, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang: *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*, ISSTA 2020, page 463–474, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450380089. `https://doi.org/10.1145/3395363.3397388`. 10

[44] Liang, Sheng and Gilad Bracha: *Dynamic class loading in the Java virtual machine.* In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, page 36–44, New York, NY, USA, 1998. Association for Computing Machinery, ISBN 1581130058. `https://doi.org/10.1145/286936.286945`. 10

[45] Dietrich, Jens, 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)*, pages 64–73, 2014. 11

[46] Preston-Werner, Tom: *Semantic Versioning.* `https://semver.org`, 2023. [Accessed 15-09-2025]. 11

[47] Bogart, Christopher, 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, page 109–120, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450342186. `https://doi.org/10.1145/2950290.2950325`. 11

[48] Xavier, Laerte, Aline Brito, Andre Hora, and Marco Tulio Valente: *Historical and impact analysis of API breaking changes: A large-scale study.* In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147, 2017. 11

[49] Hejderup, Joseph and Georgios Gousios: *Can we trust tests to automate dependency updates? A case study of Java Projects.* Journal of Systems and Software, 183:111097, 2022, ISSN 0164-1212. `https://www.sciencedirect.com/science/article/pii/S0164121221001941`. 11

[50] Wang, Ying, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu: *Will Dependency Conflicts Affect My Program's Semantics?* IEEE Transactions on Software Engineering, 48(7):2295–2316, 2022. 11

[51] Mostafa, Shaikh, Rodney Rodriguez, and Xiaoyin Wang: *Experience paper: a study on behavioral backward incompatibilities of Java software libraries.* In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 215–225, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450350761. `https://doi.org/10.1145/3092703.3092721`. 11

[52] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009, ISBN 0262033844. 22, 23

[53] Düsing, Johannes and Ben Hermann: *Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories.* Digital Threats, 3(4), February 2022. `https://doi.org/10.1145/3472811`. 35, 45

[54] Hejderup, Joseph and Georgios Gousios: *Can we trust tests to automate dependency updates? A case study of Java Projects.* Journal of Systems and Software, 183:111097, 2022, ISSN 0164-1212. `https://www.sciencedirect.com/science/article/pii/S0164121221001941`. 35

[55] Kula, Raula Gaikovina, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue: *Do developers update their library dependencies?* Empirical Software Engineering, 23(1):384–417, Feb 2018, ISSN 1573-7616. `https://doi.org/10.1007/s10664-017-9521-5`. 35

[56] Lucas, Walter, Rodrigo Bonifácio, and João Saraiva: *Understanding the Motivations, Challenges, and Practices of Software Rejuvenation.* In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2023, Bogotá, Colombia, October 1-6, 2023*, pages 611–616. IEEE, 2023. `https://doi.org/10.1109/ICSME58846.2023.00082`. 41

[57] Imtiaz, Nasif, Seaver Thorn, and Laurie Williams: *A comparative study of vulnerability reporting by software composition analysis tools.* In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*

(ESEM), ESEM '21, New York, NY, USA, 2021. Association for Computing Machinery, ISBN 9781450386654. `https://doi.org/10.1145/3475716.3475769`. 45

[58] Dennig, Frederik L., Eren Cakmak, Henrik Plate, and Daniel A. Keim: *VulnEx: Exploring Open-Source Software Vulnerabilities in Large Development Organizations to Understand Risk Exposure*. In *2021 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 79–83, 2021. 45