

The Role of Modularization in Minimizing Vulnerability Propagation and Enhancing SCA Precision

by
Mohammad Mahdi Abdollahpour

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Mohammad Mahdi Abdollahpour 2024

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis primarily comprises chapters derived from our research paper which has been published at SCAM 2024 [4], incorporating wording changes, stylistic updates, and other modifications.

Added to the original work are the following:

- Contrasting different SCA methods and describing the trade-offs in Chapter 1.
- Adding the description of *CVE-2022-45688* in Chapter 1.
- Description of Constant Pool and Java Archive (JAR) in Chapter 2.
- Further elaboration on the methodology, including the rationale behind each decision in Chapter 4.
- A more thorough examination of the results, supplemented by additional discussions in Chapter 5.
- Significant expansion of the discussion on threats and the corresponding mitigations, organized in distinct sections (Chapter 6).
- A concise summary of our methodology and key findings in Chapter 8.

Abstract

In today’s software development landscape, the use of third-party libraries is near-ubiquitous; leveraging third-party libraries can significantly accelerate development, allowing teams to implement complex functionalities without reinventing the wheel. However, one significant cost of reusing code is *security vulnerabilities*. Vulnerabilities in third-party libraries have allowed attackers to breach databases, conduct identity theft, steal sensitive user data, and launch mass phishing campaigns. Notorious examples of vulnerabilities in libraries from the past few years include *log4shell*, *solarwinds*, *event-stream*, *lodash*, and *equifax*.

Existing software composition analysis (SCA) tools track the propagation of vulnerabilities from libraries through dependencies to downstream clients and alert those clients. Due to their design, many existing tools are highly imprecise—they create alerts for clients even when the flagged vulnerabilities are not exploitable.

Library developers occasionally release new versions of their software with refactorings that improve modularity. In this work, we explore the impacts of modularity improvements on vulnerability detection. In addition to generally improving the nonfunctional properties of the code, refactoring also has several security-related beneficial side effects: (1) it improves the precision of existing (fast and stable) SCAs; and (2) it protects from vulnerabilities that are exploitable when the vulnerable code is present and not even reachable, as in gadget chain attacks.

Our primary contribution is thus to quantify, using a novel simulation-based counterfactual vulnerability analysis, two main ways that improved modularity can boost security. We propose a modularization method using a DAG partitioning algorithm, and statically measure properties of systems that we (synthetically) modularize. In our experiments, we find that modularization can improve precision of Software Composition Analysis (SCA) tools to 71%, up from 35%. Furthermore, migrating to modularized libraries results in 78% of clients no longer being vulnerable to attacks referencing inactive dependencies. We further verify that the results of our modularization reflect the structures that are already implicit in the projects (but for which no modularity boundaries are enforced).

Acknowledgements

First and foremost, I would like to express my profound gratitude to Dr. Patrick Lam for his unwavering support throughout every stage of my Master's journey. His constant positivity and encouragement helped me persevere through challenging times. He not only pushed me to excel but also provided the necessary support when I needed it most. His instrumental role in arranging my memorable trip to New Zealand for an in-person research collaboration with Victoria University of Wellington was invaluable. I am deeply grateful for his commitment and guidance.

I would like to extend my sincere thanks to Dr. Jens Dietrich for his invaluable support and dedication throughout this project. His invaluable ideas and expertise were crucial in driving our efforts to fruition. Dr. Dietrich's deep experience and wisdom in the field, combined with his pragmatism and constructive feedback, were instrumental in making this project successful. Collaborating with him in New Zealand has been very rewarding and enjoyable, and I have learned immensely from the experience.

Lastly, I would like to thank Dr. Weiyi Shang and Dr. Michael Godfrey for their time and thoughtful feedback in reviewing my thesis.

Table of Contents

Author's Declaration	ii
Statement of Contributions	iii
Abstract	iv
Acknowledgements	v
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Background	7
3 Data Collection	10
3.1 Collecting CVE records	10
3.2 Version Matching	11
3.3 Finding the Root of Vulnerabilities	12
3.4 Collecting Clients	12
3.5 Combining with existing datasets	13

4	Methodology	15
4.1	Vulnerability Analysis	16
4.2	Modularization	18
4.3	Modularization Algorithm	19
4.4	Impacts of Modularization	21
4.4.1	False Positives in Metadata-based SCA Alerts	21
4.4.2	Secure Deployment	25
4.5	Alignment of Proposed Modularization with Current Structure	26
5	Results	28
5.1	Preliminary Analysis	28
5.2	Security Improvements from Modularization	31
5.2.1	RQ1: Precision of Metadata-based SCA Alerts	32
5.2.2	RQ2: Secure Deployment	35
5.3	RQ3: Alignment of Modularization with Current Structure	38
6	Threats to Validity	43
6.1	Representativeness of the Vulnerability Dataset	43
6.2	Applicability to Different Types of SCA Tools	44
6.3	Alignment with Developer Modularization Practices	45
6.4	Usability Concerns of Modularized Code	46
6.5	Limitations Regarding Unknown Vulnerabilities	46
7	Related Work	47
7.1	Software Composition Analysis	47
7.2	Program Modularization	48
8	Conclusion	49
	References	51

List of Figures

4.1	Class-level dependency graph for the <i>org.json</i> library, color-coded based on modularization results. Nodes are classes, except that nodes with asterisked labels are condensed groups of multiple classes within a single SCC. Nodes marked in red indicate vulnerabilities. Edges represent dependencies (computed at constant pool level) between nodes.	16
4.2	Scenarios of relationships (as dotted lines) between a client and its (modularized) library. Library modules shown as nodes A–F. F (red hexagon) has a known vulnerability. E contains classes E1 directly or indirectly referencing vulnerable classes in F , in yellow to indicate potential vulnerability. E also contains classes E2 (green) that do not reference vulnerable classes in F . A also vulnerable (yellow) due to class-level references to classes in E1 . D shown in green, indicating safety: it lacks any class-level paths leading to specific vulnerable classes in E , despite module-level dependency. B and C in gray, signaling complete safety, with no references to any vulnerable modules.	24
5.1	Distribution of package splits post-modularization across libraries.	39
5.2	Distribution of module sizes normalized with respect to their parent library.	41

List of Tables

4.1	Security status of the client from Figure 4.2 pre- and post-modularization.	25
-----	---	----

Chapter 1

Introduction

In recent years, there has been a significant and noteworthy shift towards the reuse of software components, particularly through the widespread adoption of third-party libraries. These libraries have become fundamental building blocks in the construction of modern software. They embody the collective advancements and innovations offered by shared infrastructure and functionalities, which developers can leverage to accelerate development and enhance the capabilities of their software projects.

Libraries belong to large-scale ecosystems, with prominent examples being Maven for Java [57], npm for JavaScript [39], and pypi for Python [46]. These ecosystems provide a vast repository of reusable components that developers can easily integrate into their projects. For instance, the Maven repository alone contains more than 13.5 million artifacts [40], serving as the foundational backbone for the majority of Java software systems.

Despite the numerous advantages that these repositories offer, the reliance on third-party libraries is not without its own set of risks and challenges. One of the most significant concerns that has emerged in recent years is the use of vulnerable components within these libraries. For instance, the OWASP Top 10 list of security risks [41] has highlighted the increasing prominence of this issue, with the usage of vulnerable components climbing to rank 6 in 2021 from rank 9 in 2017. Simultaneously, the number of Common Vulnerabilities and Exposures (CVEs) reported each year has continued to rise, as documented by the MITRE CVE database [36]. This trend underscores a critical and ongoing challenge in the field of software development: while libraries can significantly accelerate development and provide powerful, ready-made functionality, they also introduce potential security vulnerabilities that can be exploited by malicious entities. As a result, developers face an unenviable choice [26] between constantly upgrading their libraries (and dealing with the

potential for breaking changes) or being exposed to newly-discovered vulnerabilities in their existing libraries.

Software Composition Analysis (SCA) tools [24, 62] are designed to identify and address issues that arise from the use of libraries in software projects, with a particular focus on the detection of vulnerable libraries. These tools typically rely on project metadata, such as the *pom.xml* files used in Maven projects, to perform their analysis. By examining this metadata, SCA tools can flag potential vulnerabilities in a system by identifying libraries that are listed in security databases like the National Vulnerability Database (NVD) and the GitHub Security Advisory (GHSA).

The underlying assumption of most existing SCA tools is that if a library is known to be vulnerable, then all projects that directly or indirectly depend on that library are also vulnerable. This approach is straightforward to implement and can be scaled to analyze large codebases efficiently. However, the granularity of this analysis is quite coarse, leading to a significant number of false positives. These false positives occur because the tools raise alerts for potential vulnerabilities without considering whether the vulnerable parts of the dependencies are actually used in the project.

As a result, developers often find themselves spending considerable time and resources addressing these alerts, many of which do not pose real threats to their systems. For example, GitHub’s Dependabot [17] and Synopsys’ Black Duck [55] are well-known SCA tools that generate warnings based solely on the presence of dependencies, without performing a contextual analysis of how those dependencies are used within the project.

For instance, let us consider *CVE-2022-45688* [3], which is a notable example of a vulnerability that has been identified in one of the most widely used JSON parser libraries for Java, specifically *org.json:json:20220924*. This particular vulnerability is classified as a denial-of-service (DoS) vulnerability. Although the primary purpose of this library is to parse and write JSON data, it also includes a utility function, `XML::toJSONObject`, which is designed to convert XML data into JSON format. While this function represents a relatively minor feature within the library, it contains a vulnerability that can be exploited by crafting specific XML input that triggers a stack overflow error. Not surprisingly, for the majority of clients who utilize this library solely for its JSON parsing and writing capabilities, this vulnerability is not exploitable because the `XML::toJSONObject` utility function is not reachable from the typical entry points used by these clients. Therefore, when Software Composition Analysis (SCA) tools generate notifications about this vulnerability, these notifications are false positives for many clients. For example, the July 2023 Oracle Critical Patch Update Advisory ¹ includes a specific line that dismisses

¹<https://www.oracle.com/security-alerts/cpujul2023.html>

this false positive by stating: “Tools (JSON-java): CVE-2022-45688 [VEX Justification: vulnerable_code_not_in_execute_path]“.

The precision issues associated with current Software Composition Analysis (SCA) tools are well-documented and have been extensively studied in the literature [24, 33, 60, 62]. One commonly proposed solution to address these precision issues is to adopt more fine-grained analysis techniques, which are often based on call-graph analysis [22, 27, 33, 37, 42]. There are several commercial SCA tools available that implement call-graph-based analysis methods².

However, it is important to note that such analyzes come with significantly higher computational costs. Even the most basic form of call graph analysis, known as Class Hierarchy Analysis, requires the processing of a substantial amount of data. When we consider more advanced and sophisticated forms of analysis, the computational complexity increases dramatically [21]. Furthermore, call-graph-based analyses are inherently unsound due to the presence of dynamic language features in modern programming languages. These analyses are often referred to as *soundy* at best [31], meaning they are not fully sound. Empirical studies have highlighted considerable gaps in the call-graphs that are constructed for various programming languages, including Java [54], JavaScript [11], and WebAssembly [28] programs.

Soundness is not merely a theoretical concern but a practical one that has real-world implications. Some critical vulnerabilities exploit dynamic language features, and the mere presence of vulnerable classes in the class path is sufficient to exploit a vulnerability. This means that a soundy analysis might well report an exploitable class as, in principle, unreachable, even though it is indeed exploitable.

Examples of such vulnerabilities include the numerous critical vulnerabilities exploiting deserialization, which were discovered around 2015. One notable example is *CVE-2015-6420* [1], which has an NVD severity base score of *9.8 CRITICAL*. This particular vulnerability was exploited in the wild in a ransomware attack on the San Francisco public transport system [49]. A more recent example, which does not rely on deserialization but instead on reflection, is *CVE-2022-25845* [2] in the popular *fastjson* JSON parser. This vulnerability also has an NVD severity base score of *9.8 CRITICAL*.

The activation of these vulnerabilities involves call chains that contain dynamic language features such as deserialization, reflection, dynamic proxies, methods in `Unsafe`, native methods, and so on. These complex call chains can be referred to as *gadget chains*.

²Examples of call-graph-based SCA tools include *sonatype* (<https://help.sonatype.com/en/call-flow-analysis.html>), *endorlabs* (<https://www.endorlabs.com/>), and *coana* (<https://www.coana.tech/>)

The question that arises is whether there exists a method to enhance the precision of vulnerability detection while simultaneously avoiding the significant performance overhead and inherent soundness issues associated with call-graph analysis. One straightforward alternative is to simply remove the vulnerable dependencies if they are not required for building or running the program. This approach, known as bloated dependency analysis, has been explored in previous studies [52], but it only addresses a small fraction of the overall problem. Specifically, this method would not resolve the issue for *CVE-2022-45688*, as discussed earlier, because the majority of clients will utilize some functionality provided by *org.json*.

On the other hand, we put forward the hypothesis that a significant number of existing libraries do not possess well-defined and coherent modularizations. It is worth noting that library developers have, on numerous occasions, released new versions of libraries that have been refactored to incorporate improved modularizations. Some illustrative examples of such modularizations can be found in Section 7.2. Of course, it is important to acknowledge that many libraries are already modularized, and for the purposes of this study, we will disregard those. Nevertheless, there remains a substantial number of libraries in the wild that have yet to undergo modularization.

In this study, we explore how breaking down libraries into smaller components enhances the isolation and mitigation of vulnerabilities. For example, the XML-related functionality provided by the *json.org* library could be segregated into a distinct project, potentially utilizing Maven modules. This separation would mean that most clients would not have to depend on the XML-related functionality, thereby halting the propagation of vulnerabilities such as *CVE-2022-45688*. It is important to note that while the XML-related functionality depends on the core JSON functionality, the reverse is not true. Developers typically decide when to undertake such modularizations by weighing various software engineering considerations, including maintainability, the cost implications for clients who need to update their dependencies, and other relevant factors. The significant contribution of this research is to demonstrate an additional advantage of modularization: it can significantly enhance the security of software systems.

A logical approach to understanding the effects of library modularization would be to compare the security state of libraries before and after they have been modularized, and then assess the relative impact on security, such as changes in vulnerability exposure. However, this straightforward method is often impractical because there are very few libraries that have already undergone modularization while also having well-documented vulnerabilities. To circumvent this limitation, we adopt an alternative strategy: a *simulation-based counterfactual analysis*. This method allows us to hypothesize and simulate the modularization of libraries to evaluate whether such modularization would enhance the precision of Software

Composition Analysis (SCA) tools and mitigate the propagation of vulnerabilities.

In software engineering, libraries can be modeled as graphs, where nodes represent classes and edges denote the dependencies between these classes. We utilized constant pool references to construct class-level dependency graphs for Java libraries. To meaningfully modularize a library, it is critical to ensure that the resulting module-level dependency graph is acyclic. To achieve this, we collapse the Strongly Connected Components (SCCs) within the class-level dependency graph to form a Directed Acyclic Graph (DAG). We then apply a DAG partitioning algorithm³ [23] to divide the graph. This algorithm ensures that the partitioned graph remains a DAG with minimized edge cuts. By treating a Java library as a DAG and employing a DAG-to-DAG partitioning algorithm, we can segment the library into smaller, more coherent units. We then evaluate how well the resulting partitions align with the existing structural organization of the libraries in our dataset. A high degree of alignment between the computed partitioning and the libraries' current structures strengthens our confidence in the validity and reliability of our findings.

Overall, our research delves into the impact of vulnerabilities in third-party libraries on software projects, with a particular focus on assessing the potential benefits of using smaller, more modular dependencies. This exploration aims to determine whether these streamlined libraries can augment the capabilities of Software Composition Analysis (SCA) tools, thereby enhancing the security measures available to clients. We systematically investigate how the adoption of these libraries might lead to more effective vulnerability management and risk mitigation strategies in software development.

³We used the original author's implementation of dagP, with a small patch to fix a parsing error of dot files.

We focus our explorations around the following research questions:

- **RQ1:** To what extent does the use of more modular libraries reduce the number of false positives flagged by metadata-based software composition analysis?
- **RQ2:** To what extent does the use of more modular libraries improve the security of the libraries' clients, by reducing vulnerability propagation (e.g., via gadget chains)?
- **RQ3:** How well does the proposed modularization method used in the counterfactual analysis reflect the current project structures?

By addressing these questions, we provide insights into the security challenges and opportunities associated with the use of third-party libraries in software development. The contributions of this research include:

- **Novel Modularization Method:** We introduce a method to divide a library into smaller, coherent modules while ensuring that the project remains compilable.
- **Counterfactual Analysis:** We assess the security benefits of modularization, offering insights into how it could potentially strengthen software security.
- **Empirical Findings:** We provide empirical data that can help library maintainers and software repository administrators develop better security policies.
- **Large-Scale Dataset:** With over 83,237 records of $\langle \text{CVE}, \text{Library}, \text{Client} \rangle$ tuples compiled, we provide a rich dataset that serves as a foundational resource for further research in this area.

Data availability statement.

We have made our tools and dataset available at <https://zenodo.org/doi/10.5281/zenodo.13381698>.

Chapter 2

Background

We define some terms that we use later, along with an example of a gadget chain attack.

GAVs. Maven [56] is a comprehensive build automation tool that is predominantly utilized for Java projects. It significantly streamlines various build processes, including but not limited to compilation, documentation, packaging, testing, deployment, and distribution of software projects. Maven Central [57] serves as a highly popular repository for libraries and artifacts pertaining to Java and other JVM languages. It functions as the primary storage hub for these projects, thereby providing developers with convenient access to a vast array of libraries, which they can seamlessly incorporate and utilize in their own software projects. In the context of our discussion, a Maven “GAV” refers to the Group ID, Artifact ID, and Version, which collectively serve to uniquely identify a Maven project or dependency. The *pom.xml* file, also known as the Project Object Model XML, is the principal configuration file for a Maven project. This file precisely details how the project is built, its dependencies, and its build profiles. Essentially, it serves as the blueprint for managing project builds and dependencies within the Maven ecosystem.

Vulnerability lists. CVE, which stands for Common Vulnerabilities and Exposures [34], and CWE, which stands for Common Weakness Enumeration [35], are fundamental concepts in the domain of cybersecurity. CVE is essentially a comprehensive list of publicly disclosed computer security flaws. When an individual refers to a CVE, they are specifically pointing to a particular security vulnerability that has been identified and documented in this standardized list. Each entry within the CVE list provides a detailed and often unstructured plaintext explanation of a vulnerability, thereby assisting organizations in discussing, managing, and addressing security issues. CWE, in contrast, is a categorization system for software weaknesses and vulnerabilities. It offers a unified and measurable set of criteria

for evaluating the severity of software issues, which in turn aids in the prioritization of efforts to remediate software security problems. The website *snyk.io* [51] maintains an extensive CVE database that not only categorizes and details these vulnerabilities but also includes links to patches for some of the records, thereby enabling developers and security professionals to swiftly mitigate known threats.

Gadget chains and deserialization vulnerabilities. The vulnerability identified as *CVE-2015-6420* in the *commons-collections-3.2.1* library serves as a prime example of a deserialization attack. The *commons-collections* library, although it may remain inactive within the application, poses a significant security risk due to its mere presence in the classpath, rendering it exploitable. Malicious actors can take advantage of this CVE by meticulously crafting a stream of serialized objects that instantiate specific classes from the *commons-collections* library, which are commonly referred to as gadgets. These gadgets encompass dynamic data structures such as *org.apache.commons.collections.map.LazyMap* and reflection-based utilities like *org.apache.commons.collections.functors.InvokerTransformer*. During the deserialization process, when an application processes incoming serialized data streams, these classes are loaded and instantiated, thereby triggering a sequence of method calls initiated by the JVM's deserialization mechanism, rather than by the application code itself. This deserialization process can be manipulated by attackers to execute arbitrary code remotely, which may include embedding malicious scripts or making reflective calls to the *Runtime::exec* method. The widespread adoption and integration of the *commons-collections* library at the time of the vulnerability's discovery made it an attractive target for attackers, as evidenced by the notable 2016 cyberattacks on San Francisco's public transport system, which we have previously mentioned.

Constant pool. The constant pool is a crucial component of the Java class file structure, utilized extensively in Java bytecode to handle various constant values and references which are used by the code. This structure acts as a central repository of constants, much like a lookup table, that Java bytecode instructions operate upon.

Each class file contains its own constant pool, which is indexed from 1 to $n - 1$, where n is the number of entries in the pool. The zeroth entry is invalid, as Java's indexing for this pool starts at 1. The entries in the constant pool can be of several types, such as numeric literals, string literals, class references, field references, method references, and name and type descriptors. The type of each entry is identified by a tag byte at the start.

For example, when a Java program uses a string literal, this literal is stored in the constant pool as a `const-string` type. Similarly, references to methods and fields that the class uses are also stored in the constant pool. When the bytecode instructions need to reference these constants, they use the index into the constant pool where the constant

is stored. This mechanism allows the bytecode to be more compact and efficient, as the instructions do not need to carry the full descriptors themselves. This compactness was particularly important in Java's early design, reflecting a time when minimizing the size of class files for network transmission was crucial.

The Java Virtual Machine (JVM) heavily relies on the constant pool during the runtime execution of the program. For instance, when a class is loaded, the JVM interprets the entries in the constant pool and resolves the actual references to other classes, methods, and fields as needed. This process is integral to Java's dynamic linking model, which allows classes to be loaded and linked at runtime.

Java Archive (JAR). The Java Archive (JAR) is a package file format used to aggregate many Java class files and associated metadata and resources (texts, images, etc.) into a single file for distribution. These files are built on the ZIP format and typically have a .jar file extension. JAR files are widely used because they can be easily downloaded as a single request, rather than making multiple downloads of many separate files. This makes distributing, managing, and deploying Java applications and libraries simpler and more efficient.

A typical JAR file includes:

- Java class files: Compiled Java code that can be executed on the Java Virtual Machine (JVM).
- A manifest file: Located at `META-INF/MANIFEST.MF`, this special file contains metadata about the files within the JAR, such as their entry point (main class) and other security and configuration data.

Developers use JAR files not only to distribute Java programs but also to deploy them on Java runtime environments and servers. By using JAR files, developers can easily package Java libraries, reuse code, and manage software components.

Chapter 3

Data Collection

This section provides a detailed description of the systematic approach that we employed to compile the comprehensive dataset that forms the foundation of our study on library vulnerabilities and their impacts. In essence, our objective is to gather a collection of vulnerabilities present in various libraries, along with the corresponding client projects that utilize these vulnerable libraries. To achieve this, we collected records of known vulnerabilities from the Snyk database, specifically focusing on those drawn from the Maven ecosystem. Additionally, we gathered the corresponding patches and client dependencies associated with these vulnerabilities. To obtain in-depth information about the vulnerabilities, our methodology encompasses several critical steps: identifying relevant CVE records, determining the latest versions that are still vulnerable, analyzing the patch commits, and compiling a comprehensive list of client projects that depend on these vulnerable libraries. Each of these steps is crucial for establishing a robust foundation for our subsequent analyses, ensuring that our dataset is both comprehensive and reliable.

3.1 Collecting CVE records

Given that our primary objective is to conduct a detailed and fine-grained analysis of vulnerabilities and their subsequent propagation, it is imperative that we compile a set of libraries that have publicly disclosed vulnerabilities along with the corresponding patches that are readily available. These publicly available patches are crucial as they will enable us to accurately identify and pinpoint the root causes of the vulnerabilities, which is an essential aspect of our comprehensive analysis.

Our initial step involves utilizing the Snyk database, which is an extensive repository that includes a comprehensive collection of formatted CVE (Common Vulnerabilities and Exposures) records for a wide array of software repositories. This database is particularly valuable because it provides detailed and curated information about each vulnerability, which goes beyond the basic plaintext summaries found on the CVE list at <https://www.cve.org>. The Snyk database includes additional fine-grained details such as patch commits and specific location information about the vulnerabilities, making it an indispensable resource for our study.

We specifically focus our search efforts on the Maven section of this database, beginning with the most recent records to ensure that our dataset is up-to-date and relevant. Each record in the Snyk database specifies one or more version ranges for a specific artifact, denoted as GA (Group and Artifact). These version ranges are formatted in a specific manner, such as “[a.b.c, x.y.z)”, where “a.b.c” represents the initial version that is known to be vulnerable, and “x.y.z” indicates the version in which the vulnerability has been resolved.

To ensure consistency and accuracy in our analysis, we require that the libraries adhere to semantic versioning [45]. We utilize the SemVer library¹ to parse and compare these version numbers effectively.

3.2 Version Matching

One of the challenges we face is identifying the latest version of an artifact that is still vulnerable within the specified version range. This task is not straightforward because there is no simple method to deduce the exact version that precedes the fix, and the versioning strategies employed by developers can vary significantly. Furthermore, it is possible that older versions of an artifact may not be present on Maven Central, adding another layer of complexity to this process.

To address this challenge, we first obtain a comprehensive list of all available versions of an artifact from the Maven Central repository by parsing its *maven-metadata.xml* file. This file contains detailed metadata about the artifact, including all its released versions. Once we have this exhaustive list, we proceed to identify the version that immediately precedes the fixed version, as specified in the Snyk database. This version is considered the latest vulnerable version for the corresponding CVE. In cases where the fixed version is not listed in the metadata, we choose to discard the CVE record to maintain the accuracy and reliability of our dataset.

¹<https://github.com/maxhauser/semver>, version 2.3.0, “Any” style.

We thus obtain a comprehensive set of vulnerable GAVs by carefully cross-referencing the Group and Artifact information for the libraries extracted from the Snyk records with the specific vulnerable Versions that we have identified through our metadata analysis.

3.3 Finding the Root of Vulnerabilities

In this step, we undertake a thorough analysis of the patch commits that are provided by the Snyk database. We begin by excluding any repositories that do not utilize Git as their version control system or are not hosted on GitHub.

Next, we proceed to clone the repositories that meet our criteria and programmatically inspect the differences in the Java files that have been modified in each commit, utilizing the PyDriller library [53] for this purpose. PyDriller is a Python library designed to mine information from software repositories using Git. It simplifies the process of retrieving detailed data about commits, developers, modified files, diffs, and source code. The library provides an intuitive interface for accessing the history and contents of repositories, making it easier to analyze the evolution of software projects, examine code changes over time, and extract useful metrics and insights.

We exclude any records that do not involve modifications to Java files or those that only include changes to test files, as these are irrelevant to our analysis. Using the *javaparser* library [59], we extract the name of the outer class from each modified Java file. By outer class, we refer to the top-level class definition within a file. As detailed in later sections, inner classes (including anonymous ones) are treated as part of their enclosing outer class. We then match the extracted class name to the corresponding class file in the JAR downloaded from Maven Central. If no match is found, the record is discarded to ensure the accuracy and reliability of the dataset.

3.4 Collecting Clients

Given a set of libraries that have been identified as vulnerable, it is essential to compile a corresponding set of client projects that utilize these libraries. To achieve this, we query the *libraries.io* database [25], specifically focusing on Java projects that are hosted on GitHub and are not forks. This ensures that we are examining original projects rather than duplicates or derivatives.

We deliberately exclude projects that share an owner with the vulnerable library. The rationale behind this exclusion is the belief that such projects are already somewhat modularized, meaning they have at least one level of separation between the library and the client project. This modularization could potentially skew our analysis.

Additionally, we exclude client projects that do not adhere to semantic versioning as determined by the SemVer library. The adherence to semantic versioning is essential for maintaining consistency and accuracy in our dataset. It is important to clarify that we do not verify whether a project truly abides by the “semantics” of semantic versioning, such as avoiding breaking changes in patch updates; instead, we focus solely on the format of the version numbers.

Finally, we retain those client projects that depend on the same vulnerable Group, Artifact, and Version (GAV) as identified in the earlier step.

3.5 Combining with existing datasets

To expand the scope of our dataset, we incorporated additional data derived from several previous studies [27, 38, 44, 61]. In particular, we enriched our dataset by including CVE patches that were manually analyzed and documented by another research group [60]. These studies utilized a fine-grained classification of vulnerabilities, with a specific focus on the function-level origins of security issues.

To ensure consistency between this external data and our own analysis, we undertook a detailed post-processing effort. This involved mapping individual methods to their respective outer classes, effectively normalizing any discrepancies involving inner classes. An outer class is the highest-level container of code (e.g., class `A` in `A.B.foo`), which may include multiple inner classes and methods. In our class-level dataset, we attribute vulnerabilities to this outer class, but the external datasets might attribute vulnerabilities to more specific components, such as methods (`B.foo`). Without aligning these differences, inconsistencies can occur, such as our dataset identifying class `A` as the source of vulnerability, while theirs points directly to `B.foo`, resulting in mismatches (e.g., `A != B`). By addressing these differences, we ensured that the structure of the vulnerability data aligned with our analytic framework.

However, we encountered challenges related to missing JAR files for certain GAVs (Group, Artifact, Version) referenced within these datasets. Many of these records could not be found in Maven Central or Jenkins Repository, the primary repositories for Java, and

were subsequently excluded from our analysis due to the unavailability of corresponding JAR files.

Following this exclusion, we proceeded with the retrieval of all necessary JARs for both upstream and downstream GAVs. As a result, we finalized a comprehensive dataset consisting of 83,237 unique $\langle \text{CVE}, \text{library GAV}, \text{client GAV} \rangle$ triplets.

Chapter 4

Methodology

We present a comprehensive methodology to evaluate the positive impacts of modularization on vulnerability analysis and its propagation across software systems. Central to our approach is a simulation-based counterfactual analysis, designed to systematically explore how modularization affects key outcomes. Specifically, we focus on two major areas: the reduction of false security alerts and the improved safety of software deployments that can exclude vulnerable sections of libraries, thereby mitigating risks associated with attacks on unused code.

Our methodology begins with a detailed, class-level dependency graph-based approach for assessing whether a given class is vulnerable due to a dependency. This approach provides a more granular level of analysis than traditional methods that rely on coarse, library-level metadata, while being computationally more efficient than full callgraph-based approaches. This refined technique enables a deeper understanding of the specific areas within a software system that are most susceptible to vulnerabilities, offering a more precise evaluation of potential security risks.

Next, we outline our modularization technique, which seeks to isolate and encapsulate code in a way that reduces the overall attack surface of a system. To determine the effectiveness of this modularization method, we first establish a baseline by assessing the system's state before modularization. In this initial phase, we collect key metrics related to the prevalence of security alerts and the safety of software deployments, serving as baseline for comparison.

After modularizing the system, we re-measure these metrics to conduct a comparative analysis. This post-modularization evaluation is critical to understanding the degree to which modularization contributes to reducing false security alerts and enhancing deployment

safety. By comparing the metrics from the baseline and modularized states, we aim to quantify the improvements in vulnerability detection and mitigation directly attributable to modularization.

Additionally, we evaluate how closely our modularization aligns with the implicit modular structures already present within the Java package hierarchy of the libraries. This analysis allows us to assess whether the modularization we apply reflects the natural decomposition of the software. Ultimately, our methodology offers a nuanced and data-driven approach to understanding how modularization can positively impact vulnerability analysis, reduce unnecessary security alerts, and increase the safety of software deployments.

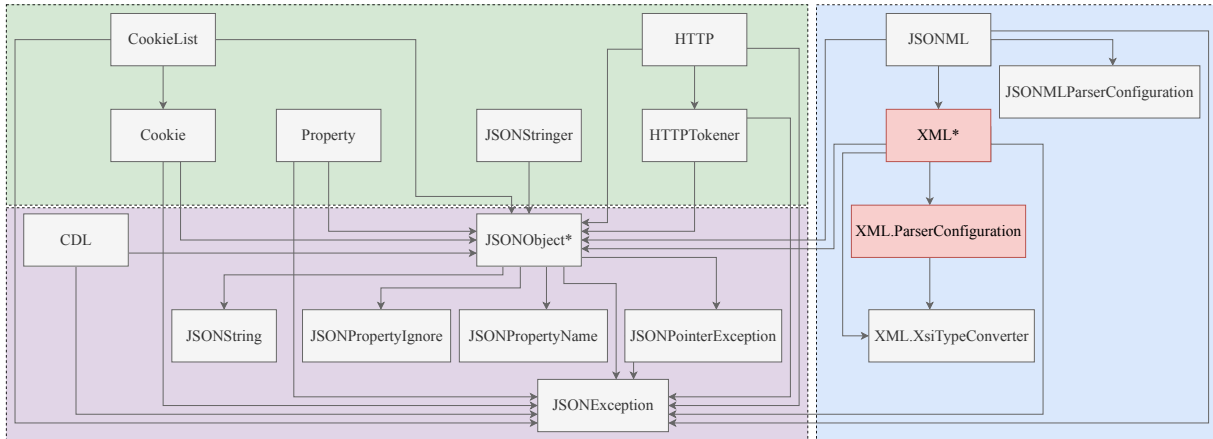


Figure 4.1: Class-level dependency graph for the *org.json* library, color-coded based on modularization results. Nodes are classes, except that nodes with asterisked labels are condensed groups of multiple classes within a single SCC. Nodes marked in red indicate vulnerabilities. Edges represent dependencies (computed at constant pool level) between nodes.

4.1 Vulnerability Analysis

Software Composition Analysis (SCA) serves as a key methodology for identifying vulnerabilities in client systems. SCA tools typically operate by analyzing metadata, focusing on dependency manifest files within a project—such as *pom.xml* file in Maven-based projects and *build.gradle* file in Gradle-based projects. These tools cross-reference the dependencies listed in these manifest files with an extensive database of known vulnerable library versions. This process is integral for detecting potential security risks within the project’s software

stack. However, the broad nature of this approach often leads to a high rate of false positives, causing challenges such as alert fatigue and reduced efficiency, as previously mentioned.

In contrast, more advanced techniques, as discussed in recent studies [37, 42, 60] and utilized by certain commercial tools, implement call-graph construction for vulnerability analysis. This method involves a detailed examination of the relationships and interactions between software components. By mapping out these call dependencies, it provides a much higher level of precision in identifying vulnerabilities, offering a more accurate risk assessment. However, this heightened precision comes at the cost of scalability. The complexity of building and analyzing call graphs makes this approach less feasible for large-scale software systems where regular vulnerability checks are necessary. The computational and time overheads make frequent use of this method impractical.

To address the limitations of both SCA and call-graph-based analysis, we propose an intermediary solution that utilizes a class-level dependency graph. This approach aims to strike a balance between the efficiency of metadata analysis and the accuracy of call-graph methodologies. The class-level dependency graph is constructed using references extracted directly from the constant pool of Java class files [30], allowing for a more granular view of dependencies without the overhead associated with full call-graph construction. This method offers a more refined approach to vulnerability detection by providing a detailed representation of the software’s dependency structure while maintaining scalability suitable for larger systems.

As described in Chapter 2, a constant pool is a collection of constants that Java compilers write and Java Virtual Machines (JVMs) read, including literals and symbolic references necessary for the class’s operations. In particular, a class must refer to other classes by name; unless the name is dynamically constructed, the name of the class being referred to will occur in the constant pool. In this context, if class A includes a reference to class B in its constant pool, we designate class A as “dependent” on class B and draw a directed edge from A to B.

To streamline the analysis, we first identify the strongly connected components (SCCs) within the graph. Each SCC is a subgraph where every node is reachable from every other node within the component. Once identified, we collapse each SCC into a single node, effectively reducing the overall complexity of the graph. This approach transforms the graph into a directed acyclic graph (DAG), as the cycles present within the SCCs are encapsulated within individual nodes. The resulting DAG is not only easier to analyze but also simplifies subsequent operations, such as traversal and optimization. However, it’s important to consider the potential trade-offs of this approach, such as the loss of internal structure within SCCs, which may have implications depending on the specific analysis

goals.

For each pair consisting of a vulnerable library and a downstream client in our dataset, we carefully construct a class-level dependency graph by analyzing the classes contained within the JAR files of both entities. This process allows us to map the relationships and dependencies among the various classes across the two JAR files, providing a clear view of how they interact.

By combining these class structures into a unified Directed Acyclic Graph (DAG), we gain a comprehensive understanding of the dependency relationships between the vulnerable library and the downstream client. Within this combined DAG, we mark the classes that are identified as the root causes of vulnerabilities, based on the data we have collected. These compromised classes represent critical points of potential risk within the system.

Once these vulnerable classes are marked, we use standard graph traversal algorithms to systematically explore the graph and identify all nodes that depend on these compromised classes, either directly or indirectly. This allows us to trace the potential spread of vulnerabilities through the system and assess their impact on related components.

4.2 Modularization

Our research aims to investigate the potential advantages of releasing software libraries in a modular format, focusing on how modularization can improve software security. With the vast number of software libraries available today, as well as the inherent complexity of their internal structures, manually refactoring each library into discrete, logical modules is an impractical and time-consuming task. Furthermore, the process of modularizing a library is inherently subjective, as the optimal decomposition into modules can vary significantly depending on the developer’s perspective, goals, and the specific use case of the library. However, our goal is not to achieve an optimal modular decomposition but rather to establish a reasonable approximation that is close to what developers might choose in practical scenarios.

To address these challenges, we employ an off-the-shelf modularization algorithm that automates the process of breaking down libraries into smaller, manageable modules. This algorithm provides a standardized approach to modularization, helping to ensure consistency across different libraries. However, since the modularization of software is not a one-size-fits-all solution, we carefully evaluate the results of the algorithm to verify the reasonableness and appropriateness of the proposed modules. This evaluation is what we refer to as “alignment” in our later discussion.

4.3 Modularization Algorithm

One of the primary challenges in this research is the development of a modularization technique that closely simulates the decision-making process of human developers when they refactor code. This involves creating a modular structure that is intuitive and logical, much like how a developer would naturally organize the code into modules.

Additionally, it is crucial that the interdependencies among the modules are structured in such a way that selecting a specific module does not necessitate bringing in dependencies on the entire set of modules. This ensures that each module can function independently to the greatest extent possible, reducing unnecessary complexity and potential for errors.

A final requirement is ensuring that when a library is split into smaller modules, each resulting module remains compilable and functionally coherent. This means that the modularization process must preserve the ability to compile the code without errors and maintain the functional integrity of the library. More formally: to ensure compilability, the class-level dependency graph must remain a directed acyclic graph (DAG) after modularization.

To navigate these complexities, we utilize *dagP* [23], which is a specialized algorithm specifically designed for the purpose of partitioning directed acyclic graphs (DAGs). The primary objective of this algorithm is to minimize the number of connections (or edge cuts) between different parts of the graph, thereby promoting functional coherence, while simultaneously ensuring that the overall structure of the graph remains acyclic.

The *dagP* algorithm, designed for efficient graph partitioning while preserving acyclicity, operates in three distinct phases: coarsening, initial partitioning, and refinement. Each stage is carefully crafted to maintain both the acyclic property of the graph and a balance across partitions, all while striving to minimize the number of edge cuts. During the coarsening phase, the complexity of the graph is systematically reduced by merging vertices into larger clusters. This process is controlled to avoid the creation of cycles, continuing until the graph reaches a pre-defined complexity threshold. Following coarsening, the initial partitioning phase commences, where the simplified graph is divided into several parts. This step sets the stage for the final phase, refinement, which fine-tunes these divisions. As the graph gradually regains its original complexity, the refinement phase adjusts and optimizes the partitions, ensuring they remain balanced and that edge cuts are minimized, thus maintaining the overall efficiency and effectiveness of the partitioning process.

In our counterfactual analyses, we employ a dynamic sizing strategy that utilizes a pseudo-logarithmic function to determine the partitioning of libraries based on the size of their dependency graphs. Under this strategy, smaller libraries are subjected to minimal or no division, ensuring that their structure remains largely intact, while larger libraries are

segmented into multiple modules to better manage their complexity. To be more specific, we used the following formula for the partitioning:

$$\text{partitionCount}(n) = \begin{cases} 1 & \text{if } 1 \leq n \leq 16 \\ 2 & \text{if } 17 \leq n \leq 64 \\ 4 & \text{if } 65 \leq n \leq 256 \\ 8 & \text{if } 257 \leq n \leq 1024 \\ 16 & \text{if } 1025 \leq n \leq 4096 \\ 32 & \text{if } 4097 \leq n \leq 8192 \\ 64 & \text{otherwise} \end{cases} \quad (4.1)$$

In this formula, n is the number of SCC nodes.

We have conducted experiments with slight modifications to the size thresholds defined by this formula and have found that these adjustments do not significantly alter the outcomes of our experiments.

To obtain the proposed modularization, we employed the dagP authors' implementation of their algorithm. The algorithm was executed on our dataset of dependency graphs, which we had constructed from 3,298 Group-Artifact-Version (GAV) entries. The algorithm completed its run in 7 minutes. In our approach, we make the assumption that each GAV initially corresponds to a single module before the modularization process begins. Using our predefined formula, we split each GAV into a specified number of modules.

In our dataset, the median size of the libraries in terms of classes was 288. Based on our formula, we instructed dagP to split these libraries into a median of four modules. The modularization generated by dagP was notably balanced. For each library, we calculated the mean size of the generated modules as a ratio of the original library size. The resulting modularization produced a median module size ratio of 0.25, meaning that, on average, each module comprised 25% of the total library size. This indicates that the modules were evenly distributed, achieving a consistent and balanced partitioning of each library into smaller, more manageable units.

Figure 4.1 provides a visualization of the outcome of our modularization when applied to the *org.json* library. In the graph, certain nodes are marked with asterisks, such as *JSONObject** and *XML**, which indicate that these nodes represent all the classes contained within their respective strongly connected components (SCCs). This particular library includes a known vulnerability attributed to the *XML.ParserConfiguration* and *XML* classes.

For the purposes of this illustration, we selected the *XML* class to serve as the representative node for its SCC, encapsulating its related classes.

After applying our modularization approach, we identified three distinct and logically coherent modules within the library. The first module primarily handles XML-related functions, encapsulating the classes responsible for XML parsing and processing. The second module is focused on Web-related functionalities, likely involving communication protocols or integration with web-based systems. The third and final module is centered around the core JSON-related functionalities, isolating the essential components required for JSON parsing and data handling.

This modularization provides significant benefits, particularly in terms of security. By isolating the core JSON functionalities into a separate module, clients that rely on this library can avoid any dependencies on the XML module, thereby mitigating the risk of vulnerabilities related to the *XML.ParserConfiguration* and *XML* classes. As a result, this separation not only enhances security but also eliminates potential security alerts, offering a safer and more streamlined use of the *org.json* library for developers who only require JSON processing capabilities.

4.4 Impacts of Modularization

In this section, we present a counterfactual analysis aimed at assessing the potential security advantages of modularization in software development and maintenance processes. By investigating the role of modularity in software dependency management, we aim to better understand its effects on security outcomes, particularly in relation to software composition analysis (SCA) tools and their handling of security vulnerabilities.

4.4.1 False Positives in Metadata-based SCA Alerts

We specifically examine how modularization can influence the rate and impact of false positives generated by metadata-based SCA tools. These tools are designed to identify security vulnerabilities within the dependencies of client software by analyzing metadata such as version numbers, licenses, and known security flaws in third-party libraries. Typically, SCA tools generate alerts whenever a known vulnerability is present in any of the libraries used by client software.

While these alerts provide useful security information, they often lack the granularity to account for whether a client software actually uses the vulnerable functionality within the

compromised libraries. As a result, many security alerts are actually false positives because metadata-based SCA tools warn about vulnerabilities that are technically present in the library but are not reachable by the client.

This disconnect leads to several challenges for developers, who must spend time investigating and addressing these alerts, even if the underlying risk is minimal. While some vulnerabilities, though unreachable, can still be exploited via certain attack vectors, these situations often require a sophisticated threat model, which may not be relevant to all development teams.

Developers who work on creating and maintaining libraries (upstream) face a challenging decision when false security alerts arise (possibly reported by their clients). Refactoring code to address these alerts can be costly and time-consuming. This challenge is compounded when the perceived risk of the vulnerability is low or the probability of its exploitation is minimal. As a result, upstream developers might delay or avoid refactoring, as the immediate benefits do not seem to justify the significant investment required.

On the other side, developers who use these libraries in their own projects (downstream) also encounter dilemmas. When alerted to potential vulnerabilities in libraries they depend on, they must decide whether to invest in replacing these libraries or wait for upstream fixes. This process might be too costly and can divert resources from other critical development tasks, especially if the threat from the vulnerability is considered low. Consequently, downstream developers might opt to ignore such alerts. This raises the question: how can modularization help reduce the frequency of these false positives and alleviate the associated costs?

Modularization, by encouraging more fine-grained dependency management, can limit the scope of vulnerable code included in a software project. By breaking down large monolithic libraries into smaller, purpose-specific modules, developers can ensure that only the necessary portions of a library are integrated, reducing the surface area for potential vulnerabilities. This approach not only minimizes the likelihood of false positives but also helps developers isolate and address true vulnerabilities more efficiently.

Our analysis focuses on quantifying the extent to which increased modularity in software dependencies can reduce the number of false positives, thereby decreasing unnecessary remediation efforts and improving overall security posture. In the following sections, we will further explore the specific security threats mitigated by modularization and evaluate the cost-benefit trade-offs for development teams in adopting modularized dependency strategies.

To address research questions 1 and 2, we construct a module-level dependency graph to better understand the impact of modularization on vulnerability analysis. This graph

represents a condensed version of the more detailed class-level dependency directed acyclic graph (DAG) discussed in Section 4.1. The condensation process leverages the output of the modularization algorithm implemented by *dagP*, which transforms class-level relationships into module-level dependencies.

Figure 4.2 illustrates a module-level dependency graph, showcasing three distinct scenarios where a client application or library references a library that has been modularized into newly-created components. Table 4.1 further compares the security posture of the system across four states: one pre-modularization and three post-modularization scenarios.

In the pre-modularization state, traditional Software Composition Analysis (SCA) tools would flag any client that declares a dependency on the entire library as vulnerable, due to a security flaw in module F. Without modularization, metadata-based tools would provide a generalized report, failing to account for more granular relationships between modules and individual client references. For example, even if a client does not reference module F directly, the tool would still classify the client as vulnerable.

In the post-modularization analysis, we observe different outcomes across three scenarios. In Scenario 1, the client was initially secure, with no direct or transitive dependency on the vulnerable module F. However, pre-modularization, SCA tools falsely reported the client as vulnerable. Post-modularization, due to the refined dependency structure, the client’s *pom.xml* now lacks references to any module containing a vulnerability, resulting in the correct assessment of “not vulnerable.” This refinement demonstrates the benefits of modularization, as SCA tools can now more accurately identify whether a client is genuinely impacted by a vulnerability.

Scenario 2 presents a different outcome. Here, the client references module A, which transitively links to module F via an intermediary class in module E1. As a result, the client is indeed vulnerable to the security flaw in F, and post-modularization, SCA tools correctly report this true-positive vulnerability. This scenario shows how modularization enhances the precision of vulnerability tracking by identifying specific transitive relationships that would otherwise be obscured at the library level.

In Scenario 3, the client references module D, which has dependencies on parts of module E (specifically E2) but not on any paths that lead to the vulnerable code in F. At the class level, the client is secure, as no direct or transitive vulnerabilities are present. However, despite this, post-modularization, SCA tools still trigger a false-positive alert, flagging the client as vulnerable. This scenario underscores the limitations of modularization, as it cannot entirely eliminate false positives generated by SCA tools, particularly when module-level relationships do not fully account for the absence of class-level vulnerabilities.

While modularization offers significant improvements in reducing false positives and

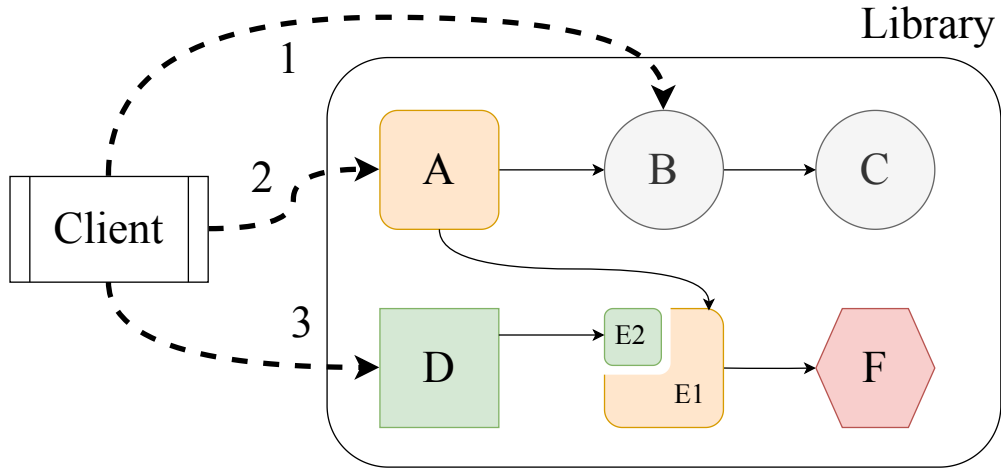


Figure 4.2: Scenarios of relationships (as dotted lines) between a client and its (modularized) library. Library modules shown as nodes A–F. **F** (red hexagon) has a known vulnerability. **E** contains classes **E1** directly or indirectly referencing vulnerable classes in **F**, in yellow to indicate potential vulnerability. **E** also contains classes **E2** (green) that do not reference vulnerable classes in **F**. **A** also vulnerable (yellow) due to class-level references to classes in **E1**. **D** shown in green, indicating safety: it lacks any class-level paths leading to specific vulnerable classes in **E**, despite module-level dependency. **B** and **C** in gray, signaling complete safety, with no references to any vulnerable modules.

refining vulnerability reports, certain cases remain where SCA tools may still overestimate the risk.

To address Research Question 1 (RQ1), we aim to quantify the false positives identified both before and after the process of modularization. In this context, a “false positive” occurs when security alerts are mistakenly issued to clients that are, in fact, secure¹. These alerts are generated by metadata-based Software Composition Analysis (SCA) tools, which rely on dependency structures to assess potential vulnerabilities within software packages.

More specifically, we focus on identifying clients similar to those depicted in Figure 4.2, particularly those that fall under Scenario 3 or exhibit characteristics of a hybrid between Scenario 1 and Scenario 3. In Scenario 3, clients are associated with modules that consist exclusively of classes which neither directly nor transitively reference any vulnerable classes. However, despite these modules being composed of secure classes, they may still depend on

¹This assessment does not account for attacks that utilize dynamic features or inactive vulnerabilities, such as gadget chains.

other classes within modules that have been flagged as vulnerable.

This creates a scenario in which the clients, though not directly at risk, are erroneously classified as vulnerable due to their indirect connections to flagged modules. This false identification arises because metadata-based SCA tools evaluate vulnerabilities by examining the structure of package dependencies, without necessarily distinguishing between actual and perceived risk based on the transitive relationships within the dependency graph. By calculating and comparing the occurrence of these false positives before and after modularization, we can gain insights into how modularization impacts the accuracy of vulnerability detection in SCA tools.

Table 4.1: Security status of the client from Figure 4.2 pre- and post-modularization.

Pre/Post Modularization	Scenario	Class-level Safety	SCA Report
Pre	—	Mixed	Vulnerable
Post	1	Safe	Safe
Post	2	Vulnerable	Vulnerable
Post	3	Safe	Vulnerable (False Positive)

4.4.2 Secure Deployment

Vulnerable dependencies still pose significant security risks even if they are not actively used by an application. To illustrate this point, we present an example here; another example was discussed in Chapter 2.

Consider the vulnerability described in *CVE-2022-25845*, which was discovered in the Alibaba *FastJSON* parser library² (*fastjson-1.2.80*). This library is responsible for translating JSON markup into Java objects. The vulnerability exploits classes that are present within the classpath and have properties that allow script execution during the instantiation of objects. Importantly, these classes do not need to be directly used by the application; their mere presence in the classpath is sufficient for exploitation. Consequently, having a bloated classpath can significantly increase the risk of such vulnerabilities being exploited.

To address Research Question 2 (RQ2), we identify scenarios where the client interacts exclusively with modules that do not contain any classes associated with vulnerabilities

²<https://github.com/alibaba/fastjson>

present in other modules, similar to Scenario 1 depicted in Figure 4.2. In this particular scenario, the client requires only Modules B and C to be present at the deployment site. Both of these modules are deemed secure. Since neither of these modules contains references to vulnerabilities from other modules, the client deployment is secure against exploits such as gadget chain attacks that could potentially leverage inactive vulnerabilities residing in Module F.

4.5 Alignment of Proposed Modularization with Current Structure

In this section, we aim to provide a comprehensive investigation into how closely our proposed synthetic modularization aligns with the existing organizational structure of the libraries we are examining. This alignment is a critical factor in determining the effectiveness and relevance of the modularization technique we have developed. By thoroughly exploring this alignment, we seek to assess the degree to which our approach can accurately represent or improve upon the current configuration, ensuring that our counterfactual simulations are not only reliable but also grounded in practical reality.

The importance of this investigation lies in its ability to measure the quality and accuracy of the modularization approach when compared to the established design of the libraries. Furthermore, it offers insight into the potential amount of effort required by developers if they were to adopt the modularized design in practice.

Java code is already organized hierarchically into “packages,” which are reflected in the directory hierarchy of Java source files. These packages represent a form of modularization established by the library developers, grouping related classes and interfaces together to promote organization and maintainability. However, despite their organizational role, packages do not impact the functioning of SCA tools or contribute to deployment safety with regard to vulnerabilities. SCA tools typically operate at the library level rather than the package level, and vulnerabilities within packages can still affect the overall security of the deployment.

Given this context, we explored the alignment between our proposed modularizations and the existing library packages in our dataset. By examining how our automatic modularization interacts with the developers’ original package structures, we aimed to determine whether our approach preserves the logical groupings intended by the developers or significantly disrupts them.

If the analysis shows that the majority of packages (namespaces) are effectively consolidated into a relatively small number of modules, this would suggest that the modularization approach we have devised is performing well and its output is well-aligned with the existing structure of the libraries, which in turn means that the refactoring efforts required for upstream maintainers to adopt the modularized design would likely be minimal.

On the other hand, if the investigation reveals that Java packages are consistently spread across a large number of newly formed modules, it would signal that the synthetic modularization diverges considerably from the original structural design of the libraries. This would be an indication of poor simulation accuracy, as it would suggest that the modularization does not reflect the intent or logic behind the current organization. In such a case, the resulting structure would require substantial refactoring on the part of developers to align their codebase with the new modular framework, thereby increasing the cost and complexity of adopting modularization.

Chapter 5

Results

In this chapter, we present the findings of our study on the security enhancements achievable through modularization in Java libraries. We begin with a preliminary analysis of our dataset to understand the characteristics of the dependencies and vulnerabilities involved. We then examine how modularization impacts the precision of Software Composition Analysis (SCA) alerts and the security of client deployments. Finally, we assess the alignment of our modularization approach with existing program structures to evaluate its practicality and effectiveness.

5.1 Preliminary Analysis

In this section, we present the preliminary analysis of the dataset we utilized in our study. The dataset is extensive and contains 3,812 unique combinations of dependency GroupId, ArtifactId, and Version (commonly referred to as GAV). These GAVs represent the specific versions of software dependencies commonly used in various software projects.

Additionally, the dataset includes 7,498 records that link Common Vulnerabilities and Exposures (CVE) identifiers with the corresponding dependency GAVs. These pairings are crucial for understanding how vulnerabilities in specific libraries (represented by their GAVs) are reported and tracked via CVEs. Expanding our focus beyond just the dependencies themselves, we also incorporate information about clients that rely on these dependencies. This expansion brings the total number of records to 83,237, where each record represents a $\langle \text{CVE}, \text{dependency GAV}, \text{client GAV} \rangle$ tuple. This comprehensive dataset provides coverage over 40,450 unique client GAVs, allowing us to analyze how vulnerabilities in libraries propagate to clients.

Upon closer examination of the 7,498 (CVE, dependency GAV) records, we observe that the median size of the libraries involved in these vulnerabilities is 288 classes. Of these, a striking 275 classes are designated as public, meaning that the vast majority of classes within these libraries are available for use by external clients. This high percentage of public classes is significant, as it implies a larger attack surface that malicious actors can potentially exploit. When a library has many publicly accessible classes, it increases the number of entry points that attackers can target, especially when these libraries are integrated into clients.

Our analysis reveals that in Java libraries, 95% of classes are defined with the access modifier “public.” This means that nearly all classes in these libraries can be easily utilized by clients. However, this accessibility also presents a security risk. Vulnerabilities within these public classes can be exploited not only in the libraries themselves but also when they are invoked through clients, making the clients vulnerable to potential attacks.

This insight highlights the importance of understanding the structure and accessibility of library code when assessing the security implications of integrating third-party dependencies into software projects.

Our vulnerability tracing methodology, as discussed in Section 4.1, leverages a class-level dependency graph to identify classes that are transitively vulnerable. By tracing the dependency paths backward from known vulnerability roots, we are able to analyze how specific vulnerabilities propagate through each library. In this process, we consider a class (e.g., class A) to be vulnerable if it depends on another class (e.g., class B), which in turn depends on a class (e.g., class C) that is explicitly identified in a Common Vulnerabilities and Exposures (CVE) report. Essentially, if class A indirectly refers to class C, which is named in a CVE, class A is considered to be at risk.

Our analysis reveals that, in most cases, the root cause of a vulnerability can be traced back to a single class (median), affecting a total of 14 classes within the library, of which 13 are public classes. When examining the impact on the class-level API surface of these libraries, we find that only 10.35% (median) of the public API is affected by a known CVE. It is important to note, however, that the median does not distribute evenly over division when performing these calculations.

Moreover, we observed that, typically, a vulnerable class in a library is not directly accessible from the API surface. The median number of steps required to reach a vulnerable class from the API surface is two. In other words, in our dataset, client code tends to reach

vulnerabilities only indirectly, meaning that accessing vulnerabilities requires traversing through two intermediary classes. This is a notable contrast to our earlier observation, which suggested that client code could potentially interact directly with vulnerabilities through the library’s API. Here, we specifically measured how frequently such direct interactions occur in practice. This finding emphasizes the complexity involved in exploiting vulnerabilities, particularly in cases where attackers rely on existing client code to do so. In these scenarios, the exploit must navigate from a client class through a chain of other classes before reaching the vulnerable class within the library. The requirement for attackers to traverse these intermediary classes helps reduce the immediate risk, as it adds additional layers that must be navigated before a vulnerability can be exploited.

Exploiting a vulnerability starting from an existing client class is often a challenging process. Typically, it requires navigating through at least two intermediary classes before reaching a vulnerable class.

As we turn our focus to the detailed records of a substantial dataset comprising 83,237 clients that are ostensibly vulnerable, our initial class-level dependency analysis yields some intriguing and somewhat unexpected findings. Specifically, we discover that over 65% of these clients are, in fact, unaffected by the known vulnerabilities present in their declared dependencies. This percentage represents a “soundy” number—that is, it is reasonably accurate but may not be entirely precise due to certain limitations such as unaccounted-for dynamic features in the code or potential attacks that exploit unused code segments. For each of these unaffected clients, our analysis shows that they (transitively) refer to zero vulnerable classes within their dependencies. However, it is important to note that while they do not reference the vulnerable classes themselves, they may still refer to other classes within the same vulnerable dependency package.

Delving deeper into our examination, we take a closer look at the usage patterns of these dependencies within the clients. We find that approximately 35% of the total 83,237 clients—and an even higher proportion, about 55%, of the clients identified as unaffected by the vulnerabilities—do not appear to utilize any classes at all from the vulnerable dependencies declared in their Project Object Model files (pom.xml). This observation suggests that there may be a prevalent issue of over-declaration of dependencies, where developers include libraries in their project configurations that are not actually needed or used in the application code. Alternatively, it might indicate that the dependency records have not been adequately maintained or updated, possibly due to oversight or the complexities involved in managing extensive sets of dependencies in large projects.

When we adjust our analysis to exclude the 55% of clients that do not use any classes

from the vulnerable dependencies, we narrow our focus to the subset of clients that do make use of classes from these dependencies. Within this refined group, our findings indicate that more than 54% of these remaining clients are vulnerable to either direct or indirect exposure to vulnerable classes from their dependencies. This means there exists a dependency path from the client to the vulnerable code. In practical terms, a user interacting with these clients could potentially trigger the vulnerability during the application’s execution. This exposure could occur because the application directly invokes methods from the vulnerable classes or relies on other classes that, in turn, interact with the vulnerable code, thereby creating an indirect linkage to the vulnerability.

While a metadata-based approach might initially seem to suffer from a high false positive rate, the situation changes markedly when we adjust our analysis by removing unused dependencies from the denominator. This adjustment greatly increases the potential client vulnerability rate reported by metadata-based approaches (as measured using class-level dependencies) from 35% to 54%.

These findings emphasize the pressing need for more robust and diligent dependency management practices within the Java development ecosystem. They highlight the importance of developers and organizations engaging in proactive vulnerability assessments and regularly auditing their dependency configurations. However, it is important to recognize that implementing these measures requires additional effort that does not directly contribute to new features or immediate business objectives. By ensuring that only necessary dependencies are included, keeping dependency versions up to date, and monitoring for known vulnerabilities in their dependencies, the Java community can significantly reduce the risk of vulnerabilities being present in clients. This proactive approach is crucial for enhancing the overall security posture of Java programs and for protecting end-users from potential exploits that could compromise their data or the functionality of the software they rely on.

These insights highlight the complexity of Java’s security landscape, emphasizing that a thorough understanding of both direct and indirect paths to vulnerabilities is essential for improving security measures and reducing risks.

5.2 Security Improvements from Modularization

In this section, we proceed to explore the enhancements in security that can be achieved through the process of modularization, utilizing a counterfactual approach to guide our

analysis. This counterfactual methodology is designed to evaluate and assess the potential impact that a proposed modularization might have on specific security properties and metrics of interest.

We begin by establishing a security baseline, conducting an in-depth analysis of the outcomes observed prior to the implementation of modularization, using our collected data on known vulnerabilities. This baseline serves as a crucial point of reference against which we can measure any changes or improvements resulting from modularization.

Subsequently, we apply the modularization technique outlined in Section 4.2 and proceed to compare the security metrics obtained before and after modularization. This comparative analysis allows us to quantify the impact of modularization on the overall security posture of the system, highlighting any significant improvements or shifts in vulnerability patterns.

Our analysis is divided into two distinct stages. In the first stage, we focus exclusively on vulnerabilities that are internal to the libraries themselves, without considering their interactions with clients. This involves a detailed examination of the libraries to identify any inherent security weaknesses or flaws that could be exploited, thereby assessing the direct impact of modularization on library security.

In the second stage, we shift our attention to examine the impact that modularization has on the interactions and interdependencies between the libraries and their clients. This includes analyzing how modularization affects the propagation of vulnerabilities through dependencies and the overall security implications for the clients that utilize these libraries.

5.2.1 RQ1: Precision of Metadata-based SCA Alerts

In this section, we investigate the potential improvements to the precision of security alerts that can be achieved through the modularization of libraries, as elaborated in Section 4.4. Modularization involves reorganizing a library into smaller, more cohesive modules, each encapsulating a specific functionality or set of functionalities. We hypothesize that this restructuring can enhance the precision of security analyses¹ by reducing unnecessary coupling and making dependencies more explicit.

As we have previously mentioned, prior to the implementation of modularization, only 35% of clients directly or indirectly reference vulnerabilities present in their dependencies. This statistic highlights that a significant majority of clients include vulnerable packages in their projects without actually utilizing the vulnerable components within those packages.

¹This assessment does not account for attacks that utilize dynamic features or inactive vulnerabilities, such as gadget chains.

Consequently, these clients are unnecessarily exposed to security alerts, which could lead to alert fatigue and reduced trust in the security alert system.

In defining our scope, our denominator, or the total number of clients considered (100%), consists of all clients who declare vulnerable packages as dependencies in their software projects. It is important to note that under the existing, pre-modularization system, all of these clients would trigger security alerts. This occurs regardless of whether they actually use the specific functionalities where the vulnerabilities reside. Such a blanket approach to security alerts can be misleading and may not accurately reflect the true risk to each client.

When we consider the scenario where unused vulnerable packages are removed from the dependencies, the percentage of clients that reference vulnerabilities increases from 35% to 54%. This adjustment accounts for clients who have vulnerable packages declared as dependencies but do not utilize them in any capacity. By removing these unused packages, we get a clearer picture of the actual exposure to vulnerabilities.

After applying the process of modularization, we observe a major improvement in the precision of security alerts. Specifically, 71% of clients who trigger security alerts—amounting to 28,913 out of a total of 40,720 clients—are now correctly identified as interacting with the vulnerable parts of their dependencies. This is a substantial increase from the pre-modularization figure, where only 35% of clients (the same 28,913 clients, but out of a larger pool of 83,237 clients) were accurately identified. This improvement underscores the effectiveness of modularization in enhancing the precision of security analyses.

Furthermore, in the post-modularization context, an impressive 94.5% of clients considered safe¹ do not trigger any false security alerts. This indicates a substantial reduction in false positives within the security alert system, which is crucial for maintaining the trust of developers and encouraging prompt action on genuine security threats.

After implementing modularization, we observe that metadata-based Software Composition Analysis (SCA) yields a precision of 71%, which is a major improvement from the previous precision rate of 35%.

These empirical findings strongly support our claim that organizing dependencies into smaller, more coherent modules leads to a substantial enhancement in the performance of security analyses. By limiting the scope of each module, it becomes significantly easier to audit and secure the code contained within them. This modular approach reduces the connectedness of the codebase, thereby decreasing the likelihood of vulnerabilities propagating through the software ecosystem. Smaller modules are inherently more manageable and can be scrutinized more thoroughly, facilitating the identification and remediation of security

issues.

We also investigated the impact of modularization on the distribution of vulnerabilities within software libraries. Specifically, we aimed to understand how breaking down libraries into smaller modules affects the way vulnerabilities are spread and how this might improve overall software security.

It should be noted that the libraries included in our dataset initially had at least one identified vulnerability. Additionally, through the modularization process, each library was split into a median of four separate modules. This indicates that, on average, every library was divided into four more focused and coherent units of functionality.

Based on these medians, one might estimate that one out of every four of the newly created modules would contain a known vulnerability. This is a straightforward calculation derived from the ratio of one vulnerability per four modules. It suggests that vulnerabilities would be evenly distributed across the modules, purely based on chance.

What is particularly interesting, though, is investigating the actual proportion of these newly formed modules that include classes which either contain vulnerabilities or depend on classes with vulnerabilities, potentially through transitive dependencies. This means we are not only considering modules that directly include vulnerable code but also those that might be affected indirectly because they rely on other modules containing vulnerabilities.

In other words, the vulnerability associated with a module might reside within the same module itself, or it could be present in another module upon which the first module depends. These dependencies could be direct, where one module explicitly uses another, or transitive, where a module depends on another module that, in turn, depends on a third module containing the vulnerability.

Our analysis revealed that this proportion is actually 49%. This means that nearly half of the newly created modules either contain vulnerable classes themselves or depend on classes that are vulnerable. This finding is significant because it highlights that vulnerabilities are not just confined to the modules where they originate but can also impact other modules through dependencies.

Returning to Figure 4.2 for illustrative purposes, we observe a practical example of this distribution. In the figure, we have one module labeled F that directly contains a vulnerability. Additionally, there are two more modules, labeled A and E, which include classes that depend on the vulnerable classes in module F. Interestingly, this results in a proportion of 3 out of 6 modules being either directly or indirectly associated with vulnerabilities. This equates to 50%, which felicitously aligns with the 49% proportion observed in our broader analysis.

Our post-modularization analysis demonstrates an enhancement in managing vulnerabilities: only 49% of the newly created modules include classes that either contain or depend on vulnerable classes. This finding highlights the positive impact of modular design on software security by effectively reducing the spread of vulnerabilities.

These results underscore the beneficial effects of modularization on software security. By organizing libraries into smaller, more coherent modules, we can contain vulnerabilities within specific modules and prevent them from affecting the entire system. This modular approach limits the scope of potential damage, making it easier to audit, monitor, and address vulnerabilities as they are identified.

Furthermore, the fact that only about half of the modules are affected by vulnerabilities or their dependencies means that the other half remain unaffected. This separation enhances the security posture of the software by reducing the overall attack surface. Developers and security teams can focus their efforts on the modules that are actually at risk, rather than having to scrutinize the entire library.

In addition, modularization facilitates more effective dependency management in terms of security. Since dependencies are more clearly defined and confined within modules, it is easier to track how vulnerabilities might propagate through the system. This clarity allows for more targeted security interventions, such as updating or patching specific modules without impacting unrelated parts of the software.

Overall, the modular design not only improves the precision of security analyses, as previously discussed, but also positively influences the distribution and management of vulnerabilities within software libraries. By adopting a modular approach, organizations can enhance their ability to prevent, detect, and mitigate security risks, leading to more robust and secure software systems.

5.2.2 RQ2: Secure Deployment

We have demonstrated that modularization significantly improves the performance of existing static Software Composition Analysis (SCA) approaches by vastly reducing the false positive rate associated with security alerts. By organizing code into smaller, more manageable modules, we can enhance the precision of SCA tools, ensuring that they more accurately identify true vulnerabilities while minimizing incorrect alerts that can lead to unnecessary concern or wasted effort.

Building upon this finding, we now turn our attention to investigating the potential of modularization in protecting against attacks that utilize dynamic Java features, such as

gadget chain attacks. Gadget chain attacks are a form of exploit where attackers leverage existing, benign code sequences—referred to as “gadgets”—within an application to perform malicious actions. These attacks are particularly insidious because they do not rely on injecting new malicious code but instead manipulate the application’s own code to achieve harmful outcomes.

Modularization can play a crucial role in enhancing the security of client deployments against such sophisticated attacks. Post-modularization, clients are required to carry fewer unused classes in their classpath—the set of classes that the Java Virtual Machine (JVM) can load at runtime. By reducing the number of classes included, we limit the potential for attackers to find and exploit dormant vulnerabilities that exist in unused portions of the codebase.

Before modularization, all clients in our dataset were vulnerable to attacks exploiting dormant vulnerabilities. These are vulnerabilities present in the code that the client does not reference or utilize, either directly or indirectly. However, because the vulnerable code is included in the deployed application, it remains accessible and exploitable through dynamic features like reflection or serialization. This means that even if the client code does not invoke these vulnerable methods or classes, an attacker could potentially manipulate the application to execute them.

Modularization changes this scenario by implying that unnecessary parts of the library code can be excluded from the client’s classpath. By breaking the library into modules and only including the modules that the client actually needs, we can effectively remove unused and potentially vulnerable code from the deployment package.

Our findings demonstrate a significant shift in the security posture of clients post-modularization. Specifically, 78.26% of clients that do not reference vulnerable code are no longer susceptible to attacks targeting inactive vulnerabilities during their deployment. This means that over three-quarters of the clients have effectively eliminated their exposure to dormant vulnerabilities simply by adopting modularization.

Conversely, 21.74% of clients that do not reference vulnerable classes still include modules containing vulnerable code in their classpath. This residual risk highlights the importance of thorough modularization and dependency management. It suggests that while modularization substantially reduces exposure, it is not a complete solution on its own. Additional measures may be necessary to identify and remove unnecessary dependencies fully.

Moreover, after removing unused modules, we observed that the number of public methods in the library modules referenced by the clients—the public attack surface—was reduced by an impressive 63.98%. The public attack surface represents all the publicly

accessible methods that could potentially be exploited by an attacker. By significantly reducing this surface area, we limit the number of entry points available for malicious activities, thereby enhancing the overall security of the program.

Our analysis reveals that modularization can reduce the likelihood of a client including dormant vulnerabilities to less than 22% of the original value and can decrease the overall public attack surface by 64%. This underscores the substantial positive impact that modular design can have on software security.

These results emphasize the critical importance of modularity in reducing overall exposure to security risks within software applications. By adopting a modular design approach, developers can minimize the amount of code that clients need to deploy, which in turn decreases the likelihood of including unnecessary and possibly vulnerable code.

Deploying less code yields faster and more secure deployments for several reasons. From a performance standpoint, smaller programs have fewer classes to load and manage, which can improve startup times and reduce memory consumption. From a security perspective, with fewer components included, there are fewer potential vulnerabilities to monitor, patch, and protect against. This reduction in complexity can also make it easier for security teams to conduct thorough code reviews and for automated tools to analyze the codebase more effectively.

Furthermore, modularization enhances maintainability by allowing developers to update or replace individual modules without affecting the entire program. This modular flexibility can lead to quicker patching of vulnerabilities and more agile responses to emerging security threats. It also facilitates better collaboration among development teams, as different modules can be worked on independently, reducing bottlenecks and improving overall productivity.

Our findings highlight that modularization is a powerful strategy for enhancing software security. By significantly reducing both the inclusion of dormant vulnerabilities and the public attack surface, modularization contributes to creating more robust and secure software systems.

5.3 RQ3: Alignment of Modularization with Current Structure

In the preceding sections, we have demonstrated that modularization significantly improves the performance of existing static Software Composition Analysis (SCA) approaches by greatly reducing the false positive rate associated with security alerts. Additionally, we have shown that modularization enhances the security of client deployments by mitigating risks associated with attacks that exploit dynamic Java features, such as gadget chain attacks. These findings highlight the substantial benefits that modularization can bring in terms of both security and efficiency.

However, it is important to acknowledge that any form of refactoring, including modularization, inherently requires effort from developers. This effort involves reorganizing the codebase, updating project structures, and ensuring that the refactored code continues to function correctly. In our study, we employed a specific modularization algorithm to perform our analysis at scale. This algorithm was designed to maintain the compilability of the library after modularization, meaning that the refactored code remains capable of being compiled without errors.

Our central concern in this research question is to investigate whether the automatic modularization produced by our algorithm aligns with the modularization that developers might propose themselves, rather than being an unrealistic or arbitrary division of the library. By examining this alignment, we aim to assess the validity of our conclusions drawn from the analysis. If our automatic modularization closely mirrors the structure that developers would naturally create, it supports the idea that our findings are applicable and relevant to real-world scenarios.

To assess this alignment, we used a primary metric that measures the extent to which Java packages are distributed across multiple modules in our modularization scheme. Specifically, we calculated the maximum number of modules into which each package is divided. This metric provides insight into how fragmented the packages become after modularization, which in turn reflects how closely our automatic modularization aligns with the existing structure of the code.

The results of our analysis, illustrated in Figure 5.1, reveal several key findings. Firstly, we observed that half of the Java packages provided by the library developers are contained entirely within a single module in our modularization. This means that for 50% of the packages, all the classes and interfaces remain grouped together in one module, just as they were in the original package structure. This high level of containment indicates that

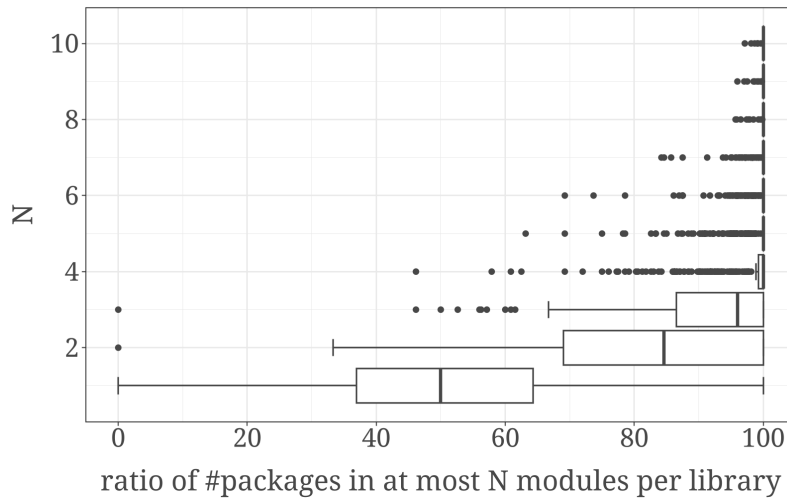


Figure 5.1: Distribution of package splits post-modularization across libraries.

our modularization approach largely preserves the developers’ intended organization for a significant portion of the codebase.

Furthermore, on average, 85% of packages are not distributed across more than two modules. This suggests that the vast majority of packages experience minimal division, with their contents largely preserved within one or two modules. In further analysis, we see a rapid decrease in the number of packages spread across multiple modules as the module count increases, which indicates that only a small fraction of packages are fragmented into several modules. This trend demonstrates that our modularization tends to keep related classes and interfaces together, respecting the logical groupings established by the developers.

The minimal fragmentation of packages suggests that our modularization approach does not arbitrarily split packages but instead maintains their integrity wherever possible. By preserving the existing package structures, we minimize disruption to the codebase, and this suggests that our automatic modularization is close to what developers might naturally produce if they were to modularize the code themselves.

The output of our automatic modularization algorithm for Java libraries closely aligns with existing program structures—achieving it requires minimal refactoring—affirming the reliability of our simulation and supporting the validity of our conclusions.

These findings indicate that our modularization closely mirrors existing program struc-

tures, affirming the reliability of our analysis. The alignment between our automatic modularization and the developers’ original package organization demonstrates that our approach is grounded in the natural structure of the code rather than being arbitrary. This supports the validity of our conclusions, suggesting that the benefits observed in our analysis are likely achievable in real-world scenarios where developers manually modularize their code.

Furthermore, this close alignment implies that developers would need to undertake minimal refactoring to achieve a modularization method similar to what our algorithm proposes. Since the proposed project structure largely corresponds with the pre-modularization setup, developers can implement modularization without significant disruption to their codebase. This suggests that the improvements in SCA performance and deployment security identified in our analysis are attainable with reasonable effort.

By utilizing constant pool references to construct dependency graphs, we ensure that the modularization reflects the actual dependencies within the code. This method allows us to accurately capture the relationships between classes and modules, maintaining the compilability of the library as all necessary dependencies are preserved.

Demonstrating that our automatic modularization aligns closely with existing program structures and maintains compilability shows that our analysis is based on a realistic representation of how modularization would occur in practice. This reinforces the validity of our conclusions regarding the benefits of modularization in improving SCA performance and enhancing deployment security.

We also conducted an in-depth analysis of the structure of the modularized outputs generated by our approach. This examination aimed to understand how modularization affects the size and distribution of modules within the libraries. On average, we found that the sizes of the modules are approximately 25% of the size of the original library. This means that each module, on average, contains about a quarter of the total classes found in the entire library. This proportion indicates a reasonable balance in granularity, suggesting that the modules are neither too large to defeat the purpose of modularization nor too small to become unwieldy or impractical for developers to manage.

To visualize the distribution of module sizes across the surveyed libraries, we present Figure 5.2. In this figure, we display a histogram that illustrates the normalized module size ratios. Specifically, for each library in our dataset, we computed the mean of its class count per module and then divided this value by the total class count of the library. This calculation provides a normalized measure of module size relative to the entire library, allowing for meaningful comparisons across libraries of different sizes. The histogram showcases the frequency distribution of these ratios, giving us insights into how module

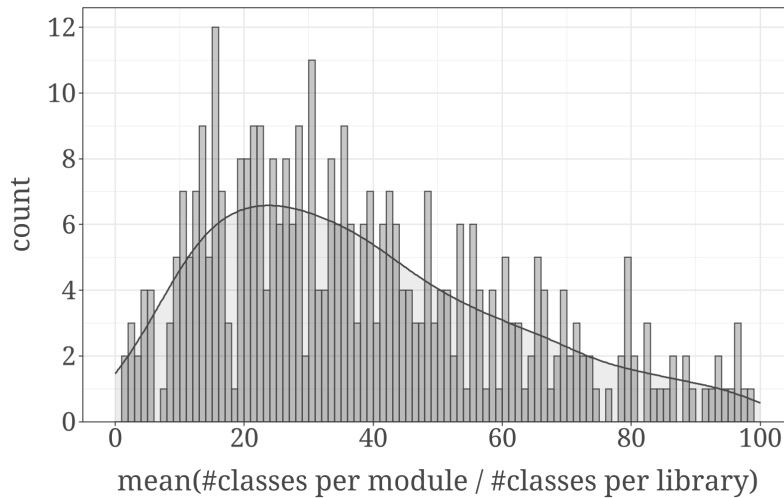


Figure 5.2: Distribution of module sizes normalized with respect to their parent library.

sizes vary among the libraries analyzed.

Our analysis of the module size distribution reveals that most modules are reasonably sized, avoiding extremes of being too large or too small. This balance is crucial because it ensures that the benefits of modularization—such as improved maintainability, easier code navigation, and enhanced security—are realized without imposing excessive overhead on developers in terms of managing a large number of tiny modules or dealing with monolithic modules that offer little advantage over the original library.

Finally, we assessed the extent to which clients utilize all the dependency modules provided by the modularized libraries. This assessment aimed to understand how modularization affects the actual usage patterns of clients and whether it leads to more efficient dependency management.

Our findings were quite revealing: only 6% of clients use functionalities from all the proposed modules of their respective libraries. In other words, a mere fraction of clients require the entire functionality offered by the full library after modularization.

Conversely, this means that a substantial 94% of clients can include and rely on less than the full library once it has been modularized. These clients can selectively include only the modules that contain the specific functionalities they need, reducing unnecessary bloat and potential security risks associated with including unused code.

This significant finding further supports the case for developers to embrace modularization in their projects. By modularizing libraries, developers enable clients to include only

the necessary components, leading to more efficient and secure software systems. The high percentage of clients that do not require the full library underscores the practical benefits of modularization in real-world scenarios.

Chapter 6

Threats to Validity

We discuss several potential threats to the validity of our conclusion that modularization leads to better results for Software Composition Analysis (SCA) tools and contributes to more secure software deployments. By critically examining these threats, we aim to provide a comprehensive understanding of the limitations and scope of our study, ensuring that our findings are interpreted accurately and that the implications for practice are well-founded.

6.1 Representativeness of the Vulnerability Dataset

We analyzed a set of vulnerabilities present in libraries available in the Maven repository, which were processed and documented by Snyk. These vulnerabilities provided us with all the necessary information to conduct a thorough analysis—including details such as links to the commits where each vulnerability was fixed. This information enabled us to identify the specific classes that were at the root of each vulnerability, allowing for precise mapping between vulnerabilities and their corresponding code segments.

A potential threat to external validity arises from the possibility that our selected set of vulnerabilities may not be representative of the broader landscape of library vulnerabilities. This concern stems from the fact that our dataset is limited to the vulnerabilities identified and documented by Snyk within the Maven ecosystem, which may not capture all possible vulnerabilities in all libraries. Additionally, our analysis does not account for unknown vulnerabilities, such as zero-day exploits, which by their nature remain undiscovered until exploited.

However, it is inherently challenging to characterize the entire universe of known vulnerabilities comprehensively, let alone account for unknown or future vulnerabilities. The field of software security is constantly evolving, with new vulnerabilities being discovered and disclosed regularly. Therefore, while our dataset may have limitations in terms of representativeness, it nonetheless provides a substantial and relevant sample for evaluating the impact of modularization on SCA tool performance and software deployment security.

6.2 Applicability to Different Types of SCA Tools

Our conclusions regarding the effectiveness of SCA tools specifically apply to those that utilize metadata-based approaches. These tools typically rely on metadata such as package names, versions, and dependency information to identify potential vulnerabilities in software components. While metadata-based SCA tools are widely used due to their efficiency and ease of integration into development workflows, they can suffer from higher false positive rates because they may flag vulnerabilities in dependencies that are included but not actually used by the client.

It is important to note that some SCA tools employ more sophisticated techniques, such as call-graph analysis, to determine whether vulnerable code is actually invoked by the client. Call-graph-based tools analyze the code’s execution paths to identify which methods and classes are reachable from the application’s entry points. These tools tend to report fewer false positives because they can more accurately assess whether a vulnerability is exploitable in the context of the client. However, call-graph analysis is more computationally expensive and may not scale well for large codebases or complex dependency graphs.

Our analysis primarily focused on enhancing the effectiveness of metadata-based SCA tools by reducing the inclusion of unused code, thereby decreasing the false positive rate. While call-graph-based tools generally exhibit lower false positive rates due to their ability to analyze actual execution paths, modularization can still contribute significantly to their performance. By breaking down libraries into smaller, more focused modules, the overall size and complexity of the call graph are reduced. This reduction can improve the efficiency of call-graph analysis, potentially allowing for the use of more precise algorithms that might otherwise be too computationally intensive for larger codebases. Although we have not empirically verified or quantified the extent of these benefits, the theoretical implications suggest that modularization can enhance both the performance and accuracy of call-graph-based SCA tools. This remains an area for future investigation to determine the practical impact of modularization on such tools.

Furthermore, modularization offers additional security benefits beyond improving SCA tool performance. Specifically, it helps protect against gadget chain attacks, which exploit dormant code paths or unused functionalities within a library. By minimizing the amount of code included in the deployment, modularization reduces the potential attack surface that could be leveraged in such attacks. This means that even in scenarios where sophisticated SCA tools are employed, modularization remains a valuable strategy for enhancing software security.

6.3 Alignment with Developer Modularization Practices

In this work, we utilized a specific modularization approach as implemented by the *dagP* algorithm. This algorithm was chosen for its ability to efficiently partition the dependency graph of a library into coherent modules while preserving the overall functionality and compilability of the code. Our assertion is conditional: if a library is modularized, then it will exhibit improved security characteristics.

However, we acknowledge that our modularization may not precisely match the modularization that developers would implement if they were to modularize their libraries themselves. Developers may have specific domain knowledge, design considerations, or architectural preferences that influence how they partition their code into modules. This discrepancy poses a threat to the internal validity of our study, as it raises the question of whether our findings would hold true for other modularization techniques.

To mitigate this threat, we analyzed the extent to which our automatic modularization aligns with the existing package-based modularization implicit in our subject libraries. We found that our modularization closely matches the logical groupings established by developers, suggesting that our approach is not arbitrary but reflects natural divisions within the codebase. This alignment increases our confidence that other reasonable modularizations would yield similar security benefits.

Furthermore, we posit that any sensible modularization is likely to create boundaries—or “firewalls”—between vulnerable and non-vulnerable code. By isolating vulnerable components within specific modules, modularization limits the propagation of vulnerabilities and reduces the potential impact on the overall system. Therefore, even if developers adopt different modularization approaches, we believe that the general principle of modularization contributing to improved security remains valid.

6.4 Usability Concerns of Modularized Code

An important software engineering concern related to modularization is the usability of the modularized code, particularly from the perspective of developers who rely on these libraries. While this concern does not pose a threat to the validity of our conclusions, it is nonetheless a practical consideration that may influence the adoption of modularization.

One potential issue is that modularization can lead to longer dependency lists for clients. For example, instead of depending on a single monolithic library, a developer might need to include dependencies on multiple separate modules that together provide the required functionality. Managing a larger number of dependencies can be more complicated, as it increases the overhead associated with specifying, updating, and resolving dependencies. There is also a heightened risk of including unnecessary or stale dependencies, which could introduce their own vulnerabilities or compatibility issues.

However, it is important to recognize that modern build tools and dependency management systems are well-equipped to handle these complexities. Tools such as Maven, Gradle, and others provide sophisticated mechanisms for managing dependencies, including automatic resolution of transitive dependencies, version conflict resolution, and support for modular project structures. These tools can simplify the process of including multiple modules, making it relatively straightforward for developers to adopt modularized libraries without incurring significant additional effort.

6.5 Limitations Regarding Unknown Vulnerabilities

Our analysis does not account for unknown vulnerabilities, such as zero-day exploits, which by their nature remain undiscovered until exploited. This limitation poses a threat to the completeness of our security assessment, as we cannot evaluate the impact of modularization on vulnerabilities that have not yet been identified. While modularization can reduce the amount of code included in deployments and potentially limit exposure to unknown vulnerabilities, we cannot quantify this effect without knowledge of the vulnerabilities themselves.

It is important to acknowledge that no security strategy can entirely eliminate the risk posed by unknown vulnerabilities. However, modularization contributes to a defense-in-depth approach by reducing the overall codebase included in deployments, thereby limiting potential exposure. While we cannot measure the impact on unknown vulnerabilities directly, the general principles of reducing code complexity and minimizing the attack surface are widely recognized as beneficial in the field of software security.

Chapter 7

Related Work

7.1 Software Composition Analysis

Software composition analysis (SCA) is a static analysis that models vulnerability propagation from upstream to downstream projects via dependencies. Its underlying assumption is that any vulnerability in a dependency can be propagated to any part of a downstream dependent package, without considering that the vulnerable code may be unreachable. Because of this, SCA is prone to false positives. Precision has been widely identified as a crucial aspect for the acceptance of program analyses in industry [13, 48]. In the context of SCA, Pashenko et al. found that “developers complain that dependency tools produce many false-positive and low-priority alerts” [43]. Intiaz et al. found that differences in accuracy of SCA tools can often be attributed to the vulnerability database they are using [24]. The use of vulnerability databases is orthogonal to our approach.

Others have tried to quantify SCA tools’ precision. Most of this work is based on call graph analysis, first proposed by Hejderup et al., for this purpose [22]. Mir et al. found that “less than 1% of packages have a reachable call path to vulnerable code in their dependencies” [33]. However, those results must be interpreted with caution. Their analysis uses a call graph constructed with OPAL [14], based on the less-accurate (but scalable) class hierarchy analysis (CHA [18]). CHA is imprecise and suffers from recall issues, as it cannot model many dynamic language features [54] which are exploited in vulnerabilities. More precise methods (e.g., VTA, context-sensitive methods) retain similar limitations: they typically aim to reduce call graph size, not to handle dynamic features [54]. Vulnerability CVE-2015-6420 can be exploited by deserializing objects from an incoming stream. The call graph path from application classes to vulnerable classes in an exploit is

highly obfuscated, and unlikely to be detected by fully-static techniques. Dynamic language feature modelling [15, 16, 29, 32] is not widely used by SCA tools, perhaps due to analysis complexity [54] or the need to have executable code [8].

In a similar approach to ours, Wu et al. [60] used call-graph analysis to investigate the reachability of vulnerable functions in Java projects. They found that most of the vulnerable functions (i.e., 86.1%) cannot be accessed by the corresponding downstream projects. Zhao et al. [62] evaluated multiple Software Composition Analysis (SCA) methods, finding that most existing tools perform poorly. The reported precisions for standard tools ranged between 0.363 and 0.621.

7.2 Program Modularization

A large body of research aims to modularize software, aiming at splitting larger programs into smaller units to improve quality. Existing approaches are based on formal concept analysis [58], genetic algorithms [6], constraint-solving [19], clustering [5, 6] and cut-based debloating [7].

A purely graph-theoretical approach similar to our work has been proposed by Shah et al. [50]. Their objective differs from ours: they aim to reduce instances of architectural anti-patterns [12] rather than security vulnerabilities.

Modularization is frequently done in practice. Several runtime module system for Java have been proposed and deployed, including OSGi [20] and Jigsaw (with several JSRs leading up to this) [9, 10, 47]. Jigsaw is part of Java 9, requiring heavy refactoring of the Java standard library. Popular open source projects have been modularized, often aiming for build time modularity using Maven modules. Examples include JUnit (between versions 4 and 5)¹, log4j² and OpenMRS³.

¹<https://junit.org/junit5/docs/5.9.0/user-guide/index.html>

²<https://logging.apache.org/log4j/2.x/manual/api-separation.html>

³<https://www.gregoryschmidt.ca/writing/openmrs-3-modularity-principles>

Chapter 8

Conclusion

In this work, we have quantitatively demonstrated the security benefits that can be achieved through improved modularization of libraries, specifically focusing on how library vulnerabilities impact their clients. Our study was conducted using a novel, large-scale dataset drawn from the Maven ecosystem, which is one of the largest repositories of 3rd-party software libraries. This comprehensive dataset allowed us to analyze a vast array of libraries and their clients, providing a robust foundation for our findings.

By employing a simulation-based counterfactual analysis, we showed that restructuring libraries into smaller, more modular components can significantly enhance client security. This enhancement is achieved by improving the precision of Software Composition Analysis (SCA) tools and by providing better protection against sophisticated attacks, such as gadget chain attacks that exploit dynamic Java features. Our analysis simulated scenarios where libraries were modularized, allowing us to observe the potential improvements in security metrics without requiring actual refactoring of the codebases.

Our approach utilized the off-the-shelf *dagP* algorithm to determine a proposed modularization of the libraries. The *dagP* algorithm is designed to partition directed acyclic graphs (DAGs) efficiently, making it suitable for decomposing the dependency graphs of Java libraries into meaningful modules. By applying this algorithm, we were able to create a modularization technique that increased the precision of security alerts reported by existing metadata-based SCA tools from 35% to 71%. This substantial improvement means that more than twice as many security alerts are accurate post-modularization, significantly reducing the number of false positives that can overwhelm developers and security teams.

In addition to improving the precision of security alerts, our modularization approach decreased the likelihood of clients including dormant vulnerabilities to 22% of the initial

value. Dormant vulnerabilities refer to vulnerabilities present in code that the client does not actively use but still includes in their deployment. By modularizing the libraries and allowing clients to include only the modules they actually need, we reduce the amount of unused code—and therefore unused vulnerabilities—that are packaged with the client. This reduction greatly diminishes the attack surface and the potential for exploitation through methods like gadget chain attacks, where attackers leverage unused or obscure code paths.

Furthermore, we demonstrated that our modularization aligns closely with the implicit package-based structure already present in the libraries within our dataset. Java developers often organize their code into packages to group related classes and interfaces logically. Our findings showed that our proposed modularization maintained these logical groupings to a significant extent, suggesting that our approach is not arbitrary but instead respects the natural structure of the code. This alignment implies that adopting such modularization in practice would require minimal refactoring effort, as it corresponds well with the developers' original organization of the codebase.

By validating that our modularization is consistent with existing program structures, we reinforce the reliability and practicality of our simulation. Developers looking to enhance the security of their libraries can adopt a modularization technique similar to ours without substantial disruption to their existing workflows. The benefits we observed—improved SCA tool precision and reduced inclusion of dormant vulnerabilities—are thus attainable in real-world scenarios.

In summary, our study highlights the significant security advantages that can be achieved through improved modularization of libraries. By breaking down libraries into smaller, coherent modules, we enable clients to include only the code they need, reducing unnecessary bloat and potential vulnerabilities. This approach not only enhances the effectiveness of SCA tools by improving the accuracy of security alerts but also helps protect client applications or libraries against advanced attacks that exploit unused code. Our simulation-based counterfactual analysis provides strong evidence for the efficacy of modularization as a strategy for enhancing software security in the Java ecosystem.

References

- [1] CVE-2015-6420, 2015. <https://nvd.nist.gov/vuln/detail/CVE-2015-6420>.
- [2] CVE-2022-25845, 2022. <https://nvd.nist.gov/vuln/detail/CVE-2022-25845>.
- [3] CVE-2022-45688, 2022. <https://nvd.nist.gov/vuln/detail/CVE-2022-45688>.
- [4] Mohammad Mahdi Abdollahpour, Patrick Lam, and Jens Dietrich. Enhancing Security through Modularization: A Counterfactual Analysis of Vulnerability Propagation and Detection Precision. In *Proc. of SCAM 24*, 2024.
- [5] N Anquetil, C Fourier, and T Lethbridge. Experiments with hierarchical clustering algorithms as software modularization methods. In *Proceedings of the Working Conference on Reverse Engineering*, 1999.
- [6] Giuliano Antoniol, Massimiliano Di Penta, and Markus Neteler. Moving to smaller libraries via clustering and genetic algorithms. In *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.*, pages 307–316. IEEE, 2003.
- [7] Christoph Blumschein, Fabio Niephaus, Codruț Stancu, Christian Wimmer, Jens Lincke, and Robert Hirschfeld. Finding cuts in static analysis graphs to debloat software. In *Proc. ISSTA 2024*, ISSTA 2024. Association for Computing Machinery, 2024.
- [8] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. ICSE 2011*, ICSE '11, page 241–250, New York, NY, USA, 2011. Association for Computing Machinery.
- [9] Alex Buckley. JSR 277: Java™ Module System, 2006. <https://jcp.org/en/jsr/detail?id=277>.

- [10] Alex Buckley. JSR 294: Improved Modularity Support in the Java™ Programming Language, 2007. <https://jcp.org/en/jsr/detail?id=294>.
- [11] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic root cause quantification for missing edges in JavaScript call graphs. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.
- [12] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M Ali Shah. On the existence of high-impact refactoring opportunities in programs. In *Proc. ACSC 2012*, pages 37–48, 2012.
- [13] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O’Hearn. Scaling static analyses at Facebook. *Communications of the ACM*, 62(8):62–70, 2019.
- [14] Michael Eichberg and Ben Hermann. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.
- [15] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static analysis of Java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 209–220, 2018.
- [16] George Fourtounis and Yannis Smaragdakis. Deep static modeling of invokedynamic. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- [17] GitHub. Dependabot.
- [18] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 108–124, 1997.
- [19] Mathew Hall, Neil Walkinshaw, and Phil McMinn. Supervised software modularisation. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 472–481. IEEE, 2012.
- [20] Richard S Hall and Humberto Cervantes. An OSGi implementation and experience report. In *Proc. CCNC 2004*, pages 394–399. IEEE, 2004.

- [21] Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proc. LICS 1997*, pages 342–351. IEEE, 1997.
- [22] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. In *ICSE-NIER 2018*, pages 101–104, 2018.
- [23] Julien Herrmann, M Yusuf Ozkaya, Bora Uçar, Kamer Kaya, and Umit V Catalyuurek. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM Journal on Scientific Computing*, 41(4):A2117–A2145, 2019.
- [24] Nasif Imtiaz, 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)*, pages 1–11, 2021.
- [25] Jeremy Katz. Libraries.io Open Source Repository and Dependency Metadata, January 2020.
- [26] Patrick Lam, Jens Dietrich, and David J. Pearce. Putting the semantics into semantic versioning. In *Onward! Essays*, 2020.
- [27] Triet Huynh Minh Le and M Ali Babar. On the use of fine-grained vulnerable code statements for software vulnerability assessment models. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 621–633, 2022.
- [28] Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. That’s a tough call: Studying the challenges of call graph construction for WebAssembly. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 892–903, 2023.
- [29] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing Java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(2):1–50, 2019.
- [30] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Virtual Machine Specification: Java SE 22 Edition*. Oracle, February 2024. <https://docs.oracle.com/javase/specs/jvms/se22/html/index.html>.
- [31] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: a manifesto. *Commun. ACM*, 58(2):44–46, jan 2015.

- [32] Benjamin Livshits, John Whaley, and Monica S Lam. Reflection analysis for Java. In *Proc. APLAS 2005*, pages 139–160. Springer, 2005.
- [33] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the Maven ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 201–211. IEEE, 2023.
- [34] MITRE. Common Vulnerabilities and Exposures.
- [35] MITRE. Common Weakness Enumeration.
- [36] MITRE. Published CVE Records.
- [37] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of Node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 29–41, 2021.
- [38] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. CrossVul: a cross-language vulnerability dataset with commit data. In *ESEC/FSE*, pages 1565–1569, 2021.
- [39] npm, Inc. npm: Node Package Manager.
- [40] Fernando Rodriguez Olivera. MvnRepository.
- [41] OWASP. OWASP Top Ten.
- [42] Hyosung Park, Chulwoo Park, SangBong Yoo, and Kichang Kim. Detecting vulnerable Java classes based on the analysis of Java library call graph. In *iThings/GreenCom/CP-SCom/SmartData*, pages 1872–1879. IEEE, 2018.
- [43] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1513–1531, 2020.
- [44] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 383–387. IEEE, 2019.

- [45] Preston-Werner, Tom. Semantic Versioning.
- [46] Python Software Foundation. PyPI: The Python Package Index.
- [47] Mark Reinhold. JSR 376: Java™ Platform Module System, 2017. <https://www.jcp.org/en/jsr/detail?id=376>.
- [48] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspán. Lessons from building static analysis tools at Google. *Communications of the ACM*, 61(4):58–66, 2018.
- [49] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. An in-depth study of Java deserialization remote-code execution exploits and vulnerabilities. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–45, 2023.
- [50] Syed Muhammad Ali Shah, Jens Dietrich, and Catherine McCartin. On the automation of dependency-breaking refactorings in Java. In *Proc. ICSM 2013*, pages 160–169. IEEE, 2013.
- [51] Snyk. Snyk Security Database.
- [52] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering*, 26(3):45, 2021.
- [53] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *ESEC/FSE 2018*, ESEC/FSE 2018, page 908–911, New York, NY, USA, 2018. Association for Computing Machinery.
- [54] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1049–1060, New York, NY, USA, 2020. Association for Computing Machinery.
- [55] Synopsis. Black Duck.
- [56] The Apache Software Foundation. Apache Maven.
- [57] The Apache Software Foundation. Maven Central Repository.
- [58] Paolo Tonella. Concept analysis for module restructuring. *IEEE Transactions on software engineering*, 27(4):351–363, 2001.

- [59] Viswanadha, Sreenivasa and Gesser, Júlio Vilmar. JavaParser.
- [60] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem. In *Proc. ICSE 2023*, pages 1046–1058, 2023.
- [61] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. TRACER: finding patches for open source software vulnerabilities. *arXiv preprint arXiv:2112.02240*, 2021.
- [62] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. Software composition analysis for vulnerability detection: An empirical study on Java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 960–972, 2023.