Enhancing Security through Modularization: A Counterfactual Analysis of Vulnerability Propagation and Detection Precision

Mohammad Mahdi Abdollahpour*, Jens Dietrich†, Patrick Lam‡
*University of Waterloo; *mohammadmahdi.abdollahpour@uwaterloo.ca*
†Victoria University of Wellington; *jens.dietrich@vuw.ac.nz*
‡University of Waterloo; *patrick.lam@uwaterloo.ca*

***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).

***Index Terms*—Software Supply Chain, Vulnerability, Modularization, Simulation-based Counterfactual Analysis**

I. Introduction

In recent years there has been a remarkable shift towards the reuse of software components, particularly through the adoption of third-party libraries. These libraries are fundamental in the construction of modern software applications—they embody the collective advancements offered by shared infrastructure and functionalities. Libraries belong to large-scale ecosystems, with prominent examples Maven for Java [1], npm for JavaScript [2], and pypi for Python [3]. The Maven repository alone contains more than 13.5 million artifacts [4], serving as the foundation for most Java software systems.

Despite the benefits that these repositories bring, the reliance on third-party libraries is not without risk. The use of vulnerable components, for instance, has risen significantly in prominence as a security concern. For instance, on the OWASP Top 10 (security risks) list [5], the usage of vulnerable components climbed to rank 6 in 2021 from rank 9 in 2017. At the same time, the number of Common Vulnerabilities and Exposures (CVEs) reported each year continues to increase, per the MITRE CVE database [6]. This trend highlights a critical challenge in software development: while libraries can accelerate development and offer powerful, ready-made functionality, they also introduce potential security vulnerabilities that can be exploited by malicious entities. Developers have an unenviable choice [7] between constantly upgrading their libraries (and dealing with breaking changes); or being subject to newly-discovered vulnerabilities in their libraries.

Software Composition Analysis (SCA) tools [8, 9] aim to identify problems caused by library use, in particular the use of vulnerable libraries. Most extant SCA tools rely on project metadata, such as *pom.xml* files. They flag vulnerabilities in systems by finding depended-upon libraries that appear in security databases (NVD, GHSA, etc). In particular, they assume that if a library is vulnerable, all of its direct and indirect clients are vulnerable as well. This is easy to implement and scales well. However, the granularity of this analysis is extremely coarse, and this approach yields many false positives—it raises alerts for potential vulnerabilities, regardless of whether or not the vulnerable dependencies are ever used. Consequently, developers are liable to expend substantial resources addressing alerts that do not represent genuine threats to their system. GitHub's Dependabot [10] and Synopsys' Black Duck [11] are SCA tools that demonstrate this problem—they issue warnings based solely on dependency presence, without contextual usage analysis.

For instance, consider *CVE-2022-45688* [12]. This is a simple denial-of-service vulnerability in one of the most popular JSON parser libraries for Java—*org.json:json:20220924*. Although this is a JSON library, it also contains the utility `XML::toJSONObject` to convert XML to JSON. While this

utility is a marginal feature of this library, it contains a vulnerability: it is possible to craft XML input that triggers a stack overflow error. Not surprisingly, for most clients using this library to parse and write JSON, this vulnerability is not exploitable, as `XML::toJSONObject` is not reachable starting from client entry points. Therefore, a SCA notification is a false positive for most clients. For example, the July 2023 Oracle Critical Patch Update Advisory [1] contains the following line dismissing this false positive: *"Tools (JSON-java): CVE-2022-45688 [VEX Justification: vulnerable_code_not_in_execute_path]"*.

Precision issues of current SCAs are well-understood and have been comprehensively studied [8, 9, 13, 14]. A common solution is to move to more fine-grained analyses, often call-graph-based [14, 15, 16, 17, 18]. Commercial SCAs based on call-graph analysis are available [2]. However, such analyses carry much higher computation costs. Even the simplest call graph analysis, Class Hierarchy Analysis, needs to process a lot of data, and more sophisticated analyses can come with a high computational complexity [19]. What is more, call-graph-based analyses are unsound due to the presence of dynamic language features—they are *soundy* at best [20]. Empirical studies have shown significant gaps in call-graphs constructed for Java [21], JavaScript [22] and WebAssembly [23] programs.

Soundiness is not just a theoretical concern. Some critical vulnerabilities exploit dynamic language features, and the mere presence of vulnerable classes in the class path is sufficient to exploit a vulnerability, such that a soundy analysis might well report an exploitable class as, in principle, unreachable. Examples include the numerous critical vulnerabilities exploiting deserialization, discovered around 2015 (example: *CVE-2015-6420* [24], NVD severity base score: *9.8 CRITICAL*) and exploited in the wild in a ransomware attack on the San Francisco public transport system [25]. A related more recent example, not relying on deserialization but only reflection, is *CVE-2022-25845* [26] in the popular *fastjson* JSON parser, also with a NVD severity base score of *9.8 CRITICAL*. The activation of those vulnerabilities uses call chains containing dynamic language features like deserialization, reflection, dynamic proxies, methods in `Unsafe`, native methods, etc. Such call chains can be referred to as *gadget chains*.

The question is whether there is a way to improve precision but to avoid the performance overhead and the soundness issues of call-graph analysis. If the vulnerable dependencies are unused (to build and/or run the program), then a trivial alternative is simply to remove those dependencies. This approach (bloated dependency analysis) has been studied [27] but only addresses a small part of the issue. In particular, this would not solve the issue for *CVE-2022-45688*, discussed above, as the vast majority of clients will use *some* functionality from *org.json*.

On the other hand, we hypothesize that many extant libraries lack coherent modularizations. Indeed, library developers have been known to release new versions of libraries, refactored to include improved modularizations (some examples can be found in Section VII-B). Many libraries, of course, are already modularized, and we ignore those for the purpose of this study—there are more than enough still-unmodularized libraries in the wild. In this work, we explore the effect of breaking up libraries on isolating vulnerabilities. For instance, the XML-related functionality of *json.org* could be moved into a separate project (perhaps using Maven modules), and most clients would not depend on it, stopping the propagation of *CVE-2022-45688*. (The XML-related functionality does depend on the main JSON functionality, but not vice versa). Developers decide when to carry out modularizations by considering a range of software engineering issues, including maintainability, the cost of requiring clients to update, etc. A key contribution of this work is in showing an extra benefit of modularization: it can improve security as well.

A challenge to exploring the effects of library modularization is that the straightforward approach requires us to find libraries that have been modularized and that have known vulnerabilities. To overcome this challenge, we instead carry out a *simulation-based counter-factual analysis*: if libraries were modularized (by simulation), would this improve the precision of SCA tools and reduce vulnerability propagation? We study a particular modularization method based on DAG partitioning [28] to automatically segment a Java Archive (JAR) into smaller coherent artifacts. We verify how well the computed partitioning aligns with the existing structure of the libraries in our dataset. Better alignment between the partitioning and existing structure strengthens our belief in our conclusions.

Overall, our work examines the implications of third-party library vulnerabilities on software projects and, specifically, evaluates how smaller, modular libraries might enhance the effectiveness of SCA tools and improve client security. 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.

[1] https://www.oracle.com/security-alerts/cpujul2023.html

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

- **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 ⟨CVE, Library, Client⟩ 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.

II. Background

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

GAVs. Maven [29] is a build automation tool used primarily for Java projects. It simplifies build processes like compilation, documentation, packaging, testing, deployment, and distribution of software projects. Maven Central [1] is a widely used repository of Java and other JVM languages' libraries and artifacts. It acts as the central storage location for these projects, providing developers easy access to millions of libraries which they can include and use in their own software projects. In our context, a Maven "GAV" refers to Group ID, Artifact ID, and Version, which together uniquely identify a Maven project or dependency. The *pom.xml* file, or Project Object Model XML, is the main configuration file for a Maven project, detailing how the project is built, its dependencies, and its build profiles. This file serves as the blueprint for managing project builds and dependencies within the Maven ecosystem.

Vulnerability lists. CVE (Common Vulnerabilities and Exposures) [30] and CWE (Common Weakness Enumeration) [31] are key concepts in the field of cybersecurity. CVE is a list of publicly disclosed computer security flaws. When someone refers to a CVE, they mean a specific security vulnerability identified in this standardized list. Each entry provides a detailed (unstructured plaintext) explanation of a vulnerability, helping organizations to discuss and manage security issues. CWE, on the other hand, is a category system for software weaknesses and vulnerabilities. It provides a unified, measurable set of criteria for assessing the severity of software issues and aids in the prioritization of software security remediation efforts. The website *snyk.io* [32] maintains a CVE database that not only categorizes and details these vulnerabilities but also includes links to patches for some of the records, thereby helping developers and security professionals to quickly mitigate known threats.

Gadget chains and deserialization vulnerabilities. *CVE-2015-6420* in *commons-collections-3.2.1* is an example of a deserialization attack. The *commons-collections* library may remain dormant within the application, yet its mere presence in the classpath makes it exploitable. Attackers can exploit this CVE by crafting a stream of serialized objects that instantiate specific classes from *commons-collections*, known as gadgets. These include dynamic data structures such as *org.apache.commons.collections.map.LazyMap* and reflection-based utilities like *org.apache.commons.collections.functors.InvokerTransformer*. During deserialization, when an application processes incoming streams, these classes get loaded and instantiated, triggering a chain of method calls initiated by the JVM's deserialization process, and critically, not by the application code itself. This process can be manipulated to execute arbitrary code remotely, such as embedding scripts or making reflective calls to *Runtime::exec*. The widespread adoption of *commons-collections* at the time of the discovery made it a prime target for attackers, as evidenced by the 2016 cyberattacks on San Francisco's public transport system, as we mentioned earlier.

III. Data Collection

This section describes the systematic approach that we used to compile the comprehensive dataset underpinning our study on library vulnerabilities and their impacts. Essentially, we need a collection of vulnerabilities in libraries, along with clients that use the vulnerable libraries. We collected records of known vulnerabilities from the Snyk database, drawn from the Maven ecosystem, and gathered the corresponding patches and client dependencies. To obtain detailed information about the vulnerabilities, our methodology involves several key steps: identifying relevant CVE records, determining the latest vulnerable versions, analyzing patch commits, and compiling a list of client projects that depend on these vulnerable libraries. Each step is crucial for establishing a solid foundation for our subsequent analyses.

A. Collecting CVE records

Because our aim is to perform a fine-grained analysis of vulnerabilities and their propagation, we require a set of libraries that have disclosed vulnerabilities with publicly available patches. These patches will help us pinpoint the root causes of the vulnerabilities, which is essential for our analysis.

Our initial step utilizes the Snyk database, which includes a comprehensive collection of formatted CVE (Common Vulnerabilities and Exposures) records for various software repositories. Entries on the CVE list at https://www.cve.org include a plaintext summary of the vulnerability and may include a formatted range of versions; the Snyk database includes more fine-grained and curated information, e.g. a patch commit, and thus specific location information about the vulnerability. We focus our search on the Maven section of this database, starting with the most recent records. Each Snyk record specifies one or more version ranges for a specific artifact (GA), formatted as "[a.b.c, x.y.z)", where "a.b.c" is the initial vulnerable version and "x.y.z" is the version in which the vulnerability is resolved. We require libraries to use semantic versioning [33] and parse and compare version numbers using the SemVer library[3].

B. Version Matching

A minor challenge is identifying the latest vulnerable version of an artifact from the specified version range. This is not trivial

[3]https://github.com/maxhauser/semver, version 2.3.0, "Any" style.

because there is no straightforward way to infer the exact prior version of a fix, and developers' versioning strategies vary. Additionally, older versions of an artifact may be removed from Maven Central.

To overcome this, we acquire a complete list of all versions of an artifact from Maven Central by parsing its *maven-metadata.xml* file. We then extract the version immediately preceding the fixed version (as identified in the Snyk database) as the latest vulnerable version for that CVE. If the fixed version is not listed, we discard the CVE record.

We thus obtain a set of vulnerable GAVs by combining the Group and Artifact information for the libraries from the Snyk records and the vulnerable Versions that we have identified.

C. Finding the Root of Vulnerabilities

In this step, we analyze the patch commits provided by Snyk. We exclude repositories not using Git or not hosted on GitHub. We clone the repositories and programmatically inspect (using PyDriller [34]) the differences in the Java files modified in each commit. We discard records without Java file modifications or those modifying only test files. Using *javaparser* [35], we extract the name of the outer class in the modified Java file, then match this to the corresponding class file in the downloaded JAR from Maven Central. We also discard records for which we find no match.

D. Collecting Clients

Given a set of vulnerable libraries, we need a set of client projects that use these libraries. We thus query the *libraries.io* database [36], focusing on Java projects hosted on GitHub that are not forks. We exclude projects that share an owner with the vulnerable library, under the belief that such projects are already somewhat modularized (they have at least one library/client separation). We also exclude clients not adhering to semantic versioning as evaluated by the SemVer library. We retain those client projects that depend on the same vulnerable GAV as identified earlier.

E. Combining with existing datasets

To augment our dataset, we integrated additional data sourced from prior studies [18, 37, 38, 39] and enriched our dataset with manually analyzed CVE patches detailed by another research group [13]. These sources used a granular classification of vulnerability roots, specifically at the function level. To align this data with our analysis, we performed post-processing to map each method to its respective outer class, normalizing issues with inner classes. We encountered issues with missing JAR files for some GAVs referenced in these datasets; we excluded from our study such records, which are also not present in Maven Central.

After completing these steps, we downloaded the JARs for all necessary upstream and downstream GAVs for future analysis, culminating in a dataset of 83,237 unique ⟨CVE, library GAV, client GAV⟩ records.

IV. Methodology

We describe our methodology for evaluating the positive impacts of modularization on vulnerability analysis and propagation. At its core is a simulation-based counterfactual analysis to systematically examine how modularization could influence key outcomes, particularly focusing on two main aspects: the reduction in false security alerts and the enhanced safety of software deployments that exclude vulnerable parts of libraries, thereby mitigating risks associated with attacks on unused code. However, we first describe our class-level dependency graph-based approach for evaluating whether a class is vulnerable to a dependency or not; it is more fine-grained than approaches relying solely on library-level metadata, but more efficient than full callgraph-based approaches. We then discuss our modularization technique. Our analysis of its effectiveness begins by assessing the baseline state—where no modularization has been applied—to establish initial metrics regarding security alerts and deployment safety. Subsequently, we modularize, and then measure these metrics again. This comparative analysis aims to quantify the analysis effectiveness and security enhancements attributable directly to modularization. Finally, we measure how well our modularization corresponds to the implicit modularization encoded in the Java package hierarchy of the libraries.

A. Vulnerability Analysis

Software Composition Analysis (SCA) is the principal methodology for identifying vulnerabilities within client systems. SCA tools are typically metadata-based. That is, they analyze dependency manifest files of a project—usually the *pom.xml* file in Maven-based Java projects. The tool cross-references each dependency against a database of library versions known to harbor vulnerabilities, facilitating the detection of potential security risks. However, this broad approach often results in a high rate of false positives, leading to the complications previously discussed.

In contrast, some studies [13, 15, 17], as well as some commercial tools, use call-graph construction for vulnerability analysis. This method provides a higher degree of accuracy by thoroughly examining the interactions between software components. However, it is challenging to scale, making it impractical for frequent use in large software systems.

To bridge the gap between these methods, we propose an intermediary solution that employs a class-level dependency graph. This graph is constructed using references from the constant pool of Java class files [40]. 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. The constant pool approach also works properly with inheritance and dynamic dispatch: subtype relations show up as dependencies between classes through information recorded in constant pools.

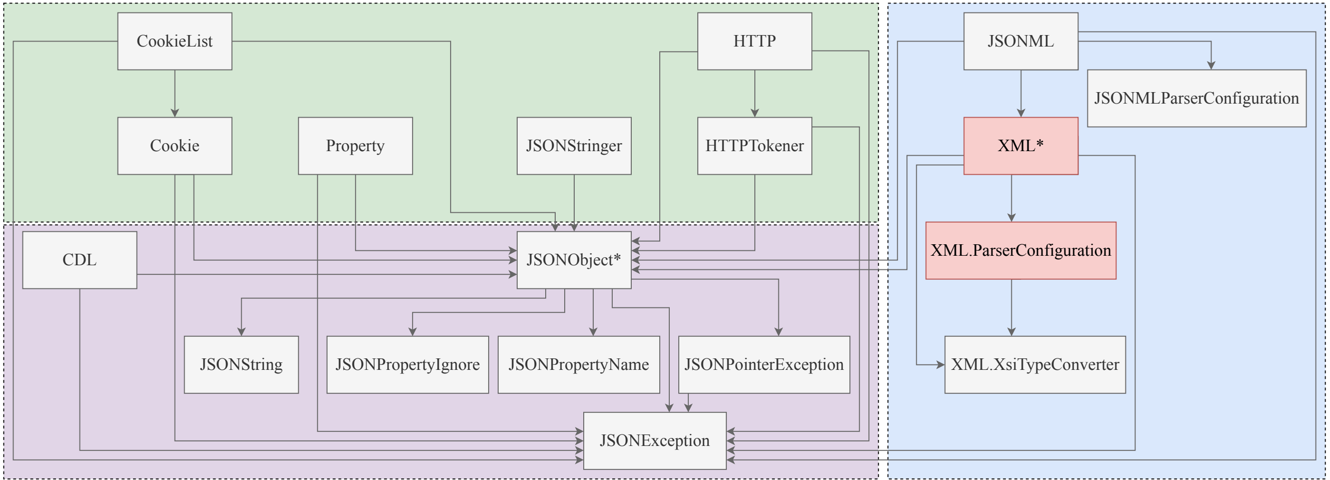

Fig. 1: Class-level dependency graph for the *org.json* library, color-coded based on modularization results. Nodes are classes, except that nodes with asterisked labels are condensed groups of multiple classes within a single SCC. Nodes marked in red indicate vulnerabilities. Edges represent dependencies (computed at constant pool level) between nodes.

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 simplify the analysis, we identify strongly connected components (SCCs) within the graph and collapse each SCC into a single node. This process transforms the graph into a directed acyclic graph (DAG), which is easier to analyze and manipulate.

For each pair of a vulnerable library and a downstream client in our dataset, we construct the class-level dependency graph by analyzing the classes contained in the JAR files of both entities. This allows us to identify the relationships and dependencies among classes across the two JARs. In the combined DAG, we mark the classes that are the root causes of vulnerabilities (as identified in our data collection) and employ graph traversal algorithms to identify all nodes that depend, either directly or indirectly, on these compromised classes.

B. Modularization

Our research seeks to explore the potential benefits associated with releasing software libraries in a modular format. Given the sheer volume of libraries available and the intricacies involved, manually refactoring each library into constituent modules is not feasible. The process of modularization itself is inherently subjective, and could vary significantly based on the developer's perspective. We use an off-the-shelf modularization algorithm and verify the reasonableness of its proposed modules (which we call "alignment" later).

1) Modularization Algorithm: One of the primary challenges in this research is the development of a modularization technique that simulates the decision-making process of human refactoring as closely as possible. Additionally, it is essential that the interdependencies among the modules are structured such that selecting a specific module does not necessitate dependencies on the entire set. A final requirement is ensuring that when a library is split into smaller modules, each module remains compilable and functionally coherent. More formally: to ensure compilability, the class-level dependency graph must remain a directed acyclic graph after modularization.

To navigate these complexities, we utilize *dagP* [28], a specialized algorithm for partitioning directed acyclic graphs. In this algorithm, the main objective is to minimize the connections (or edge cuts) between different parts of the graph (promoting functional coherence), while ensuring that the overall structure remains acyclic.

The *dagP* algorithm comprises three phases: coarsening, initial partitioning, and refinement, each maintaining the graph's acyclic nature and balance while minimizing edge cuts. In coarsening, the graph is simplified by merging vertices into clusters without forming cycles, continuing until a complexity threshold is met. The initial partitioning phase then divides the simplified graph into parts, followed by a refinement phase that optimizes these partitions as the graph's original complexity is reintegrated, ensuring balanced and minimal edge cuts.

In our counterfactual analyses, we utilize a dynamic sizing strategy that employs a pseudo-logarithmic function to partition libraries according to the size of their dependency graphs. Under this strategy, smaller libraries undergo minimal or no division, while larger libraries are segmented into multiple modules. Detailed specifications of the formula can be found in the replication package. We have experimented with slight modifications to size thresholds, and found that they do not significantly alter the outcomes of our experiments.

To obtain a proposed modularization, we used the dagP authors' implementation of their algorithm, which ran in 7 minutes on the dependency graphs that we constructed from our dataset of 3,298 GAVs. We assume that each GAV corresponds to one module before modularization, and split it into the number of modules specified by our formula. The median library size is 288 classes, and our formula instructs dagP to split these libraries into a median of 4 modules. dagP generates a balanced modularization: taking each library, we computed

the mean size of the generated modules as a ratio of the library size, and found that the median ratio was 0.25.

Figure 1 illustrates the result of our modularization strategy applied to the *org.json* library. In this graph, asterisked nodes *JSONObject** and *XML** represent all classes in their respective SCCs. This library contains a vulnerability attributed to the *XML.ParserConfiguration* and *XML* classes. (For the illustration, we chose the *XML* class as the representative of its SCC.) Post-modularization, we can identify three distinct and coherent modules: one predominantly handling XML-related functions, another dealing with Web-related functionalities, and a third focused on the core JSON-related functionalities of the library. By isolating the JSON functionalities, clients can avoid references to the XML module, enhancing security and eliminating security alerts.

C. Impacts of Modularization

We continue with a counterfactual analysis designed to measure the potential security benefits of modularization.

1) False Positives in Metadata-based SCA Alerts: We evaluate how modularization affects the effectiveness of metadata-based SCA tools. We first identify vulnerable libraries within the dependencies of various client software. Metadata-based SCA tools generally issue security alerts for all clients using these libraries. However, the validity of these alerts varies; not all clients use the vulnerable segments of the compromised libraries. Despite this, there remains a risk of certain attack types exploiting such vulnerabilities, which we explore further in subsequent sections. Nevertheless, not all developers prioritize these risks highly enough to justify the cost of refactoring. Therefore, we aim to determine how much enhanced modularity in dependencies could mitigate unnecessary development costs associated with addressing these false positive alerts.

To address RQs 1 and 2, we construct a module-level dependency graph. This graph is a condensed version of the class-level dependency DAG detailed in Section IV-A. The condensation process is based on the output from the modularization algorithm provided by *dagP*.

Figure 2 illustrates a module-level dependency graph with three scenarios where a client references a library with newly-created modules. Table I compares security status across the pre-modularization scenario and three post-modularization scenarios. Pre-modularization, metadata-based SCA tools would report that any client that declares a dependency on the library (as a whole) would be vulnerable, due to the vulnerability in module F. In Scenario 1, the client was secure pre-modularization, but SCA tools would report a false positive. Post-modularization, the client remains not vulnerable, and SCA tools would find that there are no direct or transitive references to any module that links to a vulnerability. Tools would thus benefit from the modularization and now report the correct answer, “not vulnerable” for Scenario 1, since the client’s *pom.xml* would lack references to any vulnerable modules. In Scenario 2, the client references module A, which then transitively references (at the class level) a vulnerable class

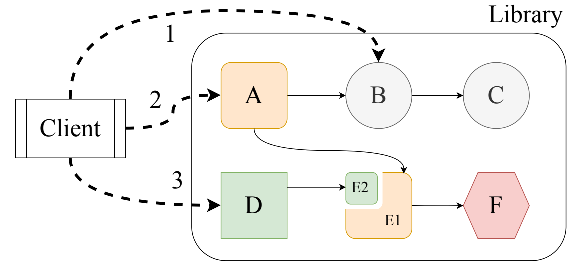

Fig. 2: Scenarios of relationships (as dotted lines) between a client and its (modularized) library. Library modules shown as nodes A–F. **F** (red hexagon) has a known vulnerability. **E** contains classes **E1** directly or indirectly referencing vulnerable classes in **F**, in yellow to indicate potential 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.

found in module F (via a class in E1). This makes the client vulnerable, and SCA tools report a true-positive vulnerability post-modularization. In Scenario 3, the client references module D, which does not have any class-level paths to the vulnerable class in F—only to parts of E (specifically E2) that do not reference vulnerable parts of F. This client is safe, at a class level, due to the absence of direct vulnerability, but still triggers security alerts from SCA tools post-modularization, which will be false positives. Modularization does not eliminate these.

To answer RQ1, we calculate the number of false positives before and after modularization. In this context, a “false positive” refers to a scenario where safe[6] clients receive security alerts from metadata-based Software Composition Analysis (SCA) tools. Specifically, we need to identify Figure 2-like clients that fall into Scenario 3 (or a mix of Scenarios 1 and 3). Clients are linked to modules composed exclusively of classes that neither directly nor transitively reference any vulnerable classes. However, these classes may depend on safe classes within modules that are otherwise (marked as) vulnerable. This assessment is how metadata-based SCA tools typically perform vulnerability checks based on package dependency structures.

TABLE I: Comparing the security status of the client pre- and post-modularization. The scenarios are depicted in Figure 2

Pre/Post Modularization	Scenario	Class-level Safety	SCA Report
Pre	—	Mixed	Vuln
Post	1	Safe	Safe
Post	2	Vuln	Vuln
Post	3	Safe	Vuln (FP)

2) Secure Deployment: Vulnerable dependencies still pose security risks even if not actively used by an application. We

briefly present an example here; Section II presented another.

CVE-2022-25845, found in the Alibaba *FastJSON* parser library[4] (*fastjson-1.2.80*), translates JSON markup into Java objects. This vulnerability exploits classes within the classpath that have properties allowing script execution during object instantiation. These classes do not need to be used directly by the application; their presence in the classpath alone is enough for exploitation. Consequently, a bloated classpath can increase the risk of such vulnerabilities being exploited.

To address RQ2, we identify situations where the client interacts only with modules that contain no classes linked to vulnerabilities in other modules, like Scenario 1 from Figure 2. That client requires only Modules B and C at the deployment site. Both modules are secure. Because neither module references vulnerabilities from other modules, the client deployment is not vulnerable to exploits like gadget chain attacks that leverage inactive vulnerabilities in Module F.

3) Alignment of Proposed Modularization with Current Structure: We aim to explore how closely our synthetic modularization aligns with the existing structures, thereby gauging both the reliability of our counterfactual simulations and the potential refactoring effort required by developers to adopt these modular designs. This step is crucial for ensuring that the proposed modularization strategies are not only theoretically sound but also practically feasible.

Our evaluation compares the current packaging structure of the libraries to the post-modularization structure. Specifically, we examine the distribution of packages across the resulting modules. If most packages are consolidated into a few modules, this indicates a high-quality modularization algorithm and minimal refactoring costs. Conversely, packages consistently spread across multiple modules indicates that the synthetic modularization diverges significantly from the original developers' intentions, suggesting poor simulation quality and implying that substantial refactoring effort would be needed to realign the codebase with the algorithmic modularization.

V. Results

We first delve into the prevalence and impact of security vulnerabilities within the Java ecosystem. Our goal is to quantify the extent of these vulnerabilities across various libraries and assess the potential risks posed to client applications dependent on them. We begin by examining vulnerable Java Archive (JAR) files to understand the inherent vulnerabilities within the libraries themselves. We then explore the interactions between these libraries and client applications to gauge the broader implications of these vulnerabilities on application security.

A. Preliminary Analysis

Our analysis utilizes a comprehensive dataset which includes 3,812 unique dependency GroupId, ArtifactId, and Version (GAV) combinations. Along with 7,498 records that pair Common Vulnerabilities and Exposures (CVE) identifiers with dependency GAVs, when expanded to include client applications, our dataset consists of 83,237 unique records of ⟨CVE, dependency GAV, client GAV⟩, covering 40,450 unique client GAVs. Examining the 7,498 records of ⟨CVE, dependency GAV⟩ reveals a median library size of 288 classes, of which 275 are public. This indicates that nearly all classes in these libraries are accessible to external clients, which potentially increases the surface area for security exploitation.

> In Java libraries, 95% of classes are defined with access modifier "public". Public library classes can be seamlessly used by clients, and when they are, their vulnerabilities can be exploited by attacks on the clients.

Our vulnerability tracing (Section IV-A) uses a class-level dependency graph to identify transitively vulnerable classes. Tracing paths backward from vulnerability roots, we assess the contamination spread within each library by specific vulnerabilities. That is, we consider class A vulnerable if class C is named in a CVE, class B refers to C, and A refers to B.

Our findings show that typically only one class (median) is the root cause of each vulnerability, affecting 14 classes in total, including 13 public classes. If we calculate the impacted class-level API surface we observe that only 10.35% (median) of the public API is affected by a known CVE[5].

Additionally, the median number of steps to reach a vulnerable class in the library from the API surface is two. That is, in our dataset, we observed that client code potentially reaches vulnerabilities only indirectly, often requiring traversal through two other classes. This contrasts with our previous observation where we found that clients *could* potentially interact directly with vulnerabilities through the libraries' API; here, we measure how often they do interact. This finding focuses on scenarios where an attacker uses client code—as it currently exists—to access vulnerabilities in libraries. The requirement to start with existing client classes and to then pass through other classes to reach vulnerabilities mitigates immediate risks to some extent.

> Exploiting a vulnerability from an existing client class is often challenging—crafting the exploit usually requires finding a path through at least two other classes starting from any client class.

Turning our focus to the records of 83,237 ostensibly-vulnerable clients, our initial class-level dependency analysis finds that over 65% of client applications are in fact unaffected by vulnerabilities (a "soundy" number, modulo dynamic features and unused code attacks)—each such client transitively refers to 0 actually vulnerable classes, but may refer to other classes in the vulnerable dependency. A closer look reveals that about 35% of the 83,237 clients, and 55% of the unaffected clients, do not appear to use any classes at all from the vulnerable dependencies declared in their project object model (*pom.xml*), which may reflect over-declarations or outdated

[4]https://github.com/alibaba/fastjson

[5]Note that median does not have the distributive property over division.

maintenance of dependency records. When we exclude the 55% of clients that do not use any classes from vulnerable dependencies, our findings indicate that more than 54% of the remaining client applications are vulnerable to either direct or indirect exposure to vulnerable classes from dependencies—there is a path from the client to the vulnerability, and a user of these clients could potentially trigger the vulnerability. This emphasizes the need for more robust dependency management and proactive vulnerability assessments in the Java ecosystem.

> While a metadata-based approach may seem to suffer from a high false positive rate, removing unused dependencies from the denominator greatly increases the potential client vulnerability rate reported by metadata-based approaches (as measured using class-level dependencies) from 35% to 54%.

These observations underscore the complexity of Java's security landscape. Understanding both direct and indirect paths to vulnerabilities is crucial for security.

B. Security Improvements from Modularization

We next explore the security enhancements of modularization through a counterfactual approach. Our counterfactual approach evaluates the impact of a proposed modularization on properties of interest. We first establish a security baseline, analyzing outcomes prior to modularization using our data on vulnerabilities. We then apply the modularization from Section IV-B and compare pre- and post-modularization metrics. Our analysis includes two stages: we initially focus on vulnerabilities internal to the libraries, irrespective of their interactions with client applications; next, we examine the impact of modularization on the interplay between libraries and clients. We specifically assess the reduction in false security alerts and the percentage of clients eliminating (used and unused) vulnerable dependencies.

C. RQ1: Precision of Metadata-based SCA Alerts

We investigate the potential improvements to precision from modularizing libraries as described in Section IV-C. After modularization, security alert precision improves significantly. Specifically, 71% of clients who trigger security alerts (28,913 of 40,720) are now correctly identified as interacting with vulnerable parts of their dependencies, up from 35% before modularization (the same 28,913, but out of 83,237). Furthermore, post-modularization, 94.5% of clients considered safe[6] do not trigger any false security alerts.

> Post-modularization, we observe that metadata-based SCA yields a precision of 71%, up from 35%.

These findings support our claim that organizing dependencies into smaller, more coherent modules leads to a substantial improvement in the performance of security analyses. By limiting the scope of each module, it becomes easier to audit and secure the code, reducing the likelihood of vulnerability propagation. Moreover, this modular approach can lead to quicker updates and patches, as each module can be updated independently. Note that, after modularization, clients are less likely to require updates, as they are now dependent on a smaller part of the library. Modularization not only enhances security but also improves the maintainability of the software.

We also investigated the impact of modularization on the distribution of vulnerabilities. Recall that the median number of vulnerabilities per library was 1 and each library was split into 4 modules (median). One can estimate that 1 in 4 of the new modules would contain a (known) vulnerability.

What is interesting, though, is investigating the proportion of newly-formed modules with classes that contain or depend on classes with vulnerabilities (perhaps transitively). Thus, the vulnerability may be in the same module, or it may be in another module. That proportion is 49%. Returning to Figure 2, we have one module containing a vulnerability (F), and 2 more modules with classes that depend on vulnerabilities (A, E); felicitously, the 3/6 proportion lines up with the 49%.

> Post-modularization analysis shows an enhancement in managing vulnerabilities: only 49% of the newly-created modules include classes that contain or depend on vulnerable classes, highlighting the positive impact of modular design on software security.

D. RQ2: Secure Deployment

We showed that modularization improves the performance of existing static SCA approaches by vastly reducing the false positive rate. We now investigate the potential of modularization in protecting against attacks that use dynamic Java features (e.g. gadget chain attacks). Modularization can thus enhance the security of client deployments: post-modularization, clients need to carry fewer unused classes in their classpath.

Pre-modularization, all clients in our dataset are vulnerable to attacks exploiting dormant vulnerabilities—those that client code does not reference, either directly or indirectly. Modularization implies that some of the library code can be dropped from the client's class path. Our findings demonstrate a significant shift post-modularization: 78.26% of clients that do not reference vulnerable code are no longer susceptible to attacks targeting inactive vulnerabilities during their deployment. (Conversely, 21.74% of clients that do not reference vulnerable classes still do reference modules containing vulnerable code.). Moreover, after removing unused modules, the number of public methods in the library modules that are referenced by the clients (the public attack surface) is reduced by 63.98%.

> 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%.

[6]Not considering attacks that utilize dynamic features or inactive vulnerabilities (e.g., gadget chains).

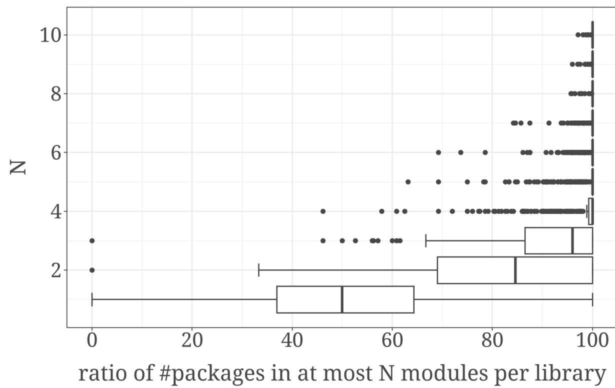

Fig. 3: Distribution of package splits post-modularization across libraries.

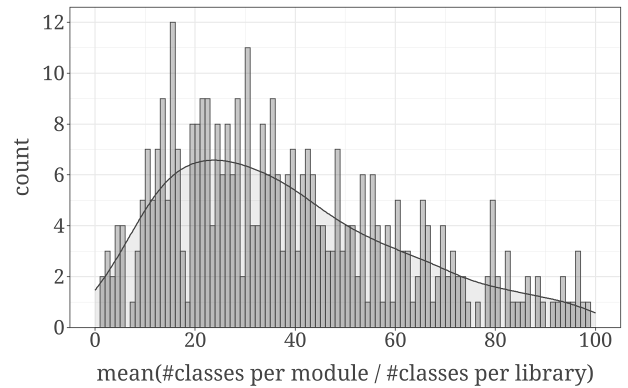

Fig. 4: Distribution of module sizes normalized with respect to their parent library.

Our results emphasize the importance of modularity in reducing the overall exposure to security risks. Modularization reduces the amount of code that clients need to deploy, thus decreasing the likelihood of including and being made to execute unnecessary, possibly-vulnerable code. Needing to deploy less code yields faster and more secure deployments.

E. RQ3: Alignment of Modularization with Current Structure

We have shown that our proposed modularization improves the performance of SCA tools and helps secure deployments against gadget chain attacks. Any form of refactoring, including modularization, implies developer effort. Central to our analysis is a novel modularization strategy designed to maintain the compilability of the library post-modularization. In this research question, we investigate whether our automatic modularization is similar to one that developers might propose, versus being an unrealistic and arbitrary division of the library.

Java code is already organized hierarchically into "packages", reflected in the directory hierarchy of Java sources. Packages are a form of modularization proposed by the library developers, but have no impact on SCA tools or deployment safety.

We thus explore the alignment between our proposed modularizations and existing library packages in our dataset. High alignment will support the validity of our counterfactual analysis and implies that less refactoring will be required to obtain modules similar to what our algorithm proposes.

The primary metric that we use for this assessment is the rate at which Java packages are distributed across multiple modules. Specifically, we measured the maximum number of modules into which each package is divided. The results, illustrated in Figure 3, reveal that half of the library developers' provided Java packages are contained within a single module in our modularization, with an average of 85% of packages not exceeding distribution across two modules. This figure demonstrates a rapid shift towards fewer modules as the module count increases, indicating that most packages are minimally divided across the libraries in our dataset.

> The output of our proposed modularization algorithm for Java libraries closely aligns with existing program structures—achieving it requires minimal refactoring—affirming the reliability of our simulation.

The findings suggest that our modularization closely mirrors existing program structures, affirming the reliability of our simulation. Moreover, the alignment implies minimal refactoring costs, as the proposed project structure largely corresponds with the pre-modularization setup. Our use of constant pool references to construct dependency graphs further ensures that projects remain compilable under our modularization.

We also analyzed the structure of the modularized outputs. On average, module sizes are about 25% of the original library size, indicating a reasonable balance in granularity. Figure 4 displays the size distribution of modules across the surveyed libraries: we computed, for each library, the mean of its class count per module divided by total class count—its normalized module size ratio—and show a histogram of ratios.

Finally, we assessed the extent to which client applications utilize all dependency modules. We found that only 6% of client applications use functionalities from all proposed modules for their library. That is, 94% of clients can bring in less than the full library once it has been modularized. This further supports the case for developers to embrace modularization.

VI. Threats to Validity

We discuss a number of threats to the validity of our conclusion that modularization leads to better results for SCA tools and more secure software deployments.

We analyzed a set of vulnerabilities in libraries available in Maven, processed by Snyk, which have all of the information we need to perform further analysis—notably, links to the commits where the vulnerability was fixed, which allowed us to find the classes that are at the root of the vulnerability. A threat to external validity is that this set may be unrepresentative. However, it is hard to characterize the universe of even known vulnerabilities, leaving aside yet-unknown 0-days.

Our conclusion about SCA tools applies to tools that use metadata-based approaches. Some SCA tools use callgraphs; they are more computationally expensive than metadata-based tools, but report fewer false positives, so our approach would not eliminate as many false positives. However, it still contributes to callgraph-based approaches by shrinking the callgraph. Modularization additionally helps versus gadget chain attacks.

In this work, we proposed one specific modularization approach (as implemented by dagP). We do not consider unwillingness to modularize a threat to validity—we are saying that if a library is modularized, then it will have better security. We carefully chose our modularization to be feasible—it is scalable and automatic, and preserves compilability. However, our modularization might not match the modularization that developers would implement. We mitigated this threat by showing that our modularization matched the package-based modularization implicit in our subject libraries, and we believe that other modularizations will be similar to ours. We also believe that any reasonable modularization will put firewalls between vulnerable and non-vulnerable code, thus improving security. We hope that our work contributes evidence encouraging developers to increase the modularity of their libraries.

A software engineering concern that is not a threat to validity is usability of modularized code. Longer dependency lists (e.g. with 5 modules versus 1) can be more complicated for developers, and increase the risk of unnecessary or stale dependencies, but modern tools excel at managing dependencies.

VII. Related Work

A. 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 [41, 42]. In the context of SCA, Pashenko et al. found that "developers complain that dependency tools produce many false-positive and low-priority alerts" [43]. Imtiaz et al. found that differences in accuracy of SCA tools can often be attributed to the vulnerability database they are using [8]. 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 [16]. Mir et al. found that "less than 1% of packages have a reachable call path to vulnerable code in their dependencies" [14]. However, those results must be interpreted with caution. Their analysis uses a call graph constructed with OPAL [44], based on the less-accurate (but scalable) class hierarchy analysis (CHA, [45]). CHA is imprecise and suffers from recall issues, as it cannot model many dynamic language features [21] 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 [21]. 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 [46, 47, 48, 49] is not widely used by SCA tools, perhaps due to analysis complexity [21] or the need to have executable code [50].

In a similar approach to ours, Wu et al. [13] 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. [9] 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.

B. 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 [51], genetic algorithms [52], constraint-solving [53], clustering [52, 54] and cut-based debloating [55].

A purely graph-theoretical approach similar to our work has been proposed by Shah et al [56]. Their objective differs from ours: they aim to reduce instances of architectural anti-patterns [57] rather than security vulnerabilities.

Modularization is frequently done in practice. Several runtime module system for Java have been proposed and deployed, including OSGi [58] and Jigsaw (with several JSRs leading up to this) [59, 60, 61]. 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)[7], log4j[8] and OpenMRS[9].

VIII. Conclusion

In this work, we quantitatively demonstrated security benefits from improved modularization in library vulnerabilities on libraries' clients, with a novel large-scale dataset drawn from the Maven ecosystem. Using a simulation-based counterfactual analysis, we showed that smaller, modular libraries can enhance client security by improving SCA tool precision and helping to protect against gadget chain attacks. Our approach used the off-the-shelf *dagP* algorithm to find a proposed modularization; its output increased the precision of security alerts reported by existing metadata-based tools to 71% (from 35%), while decreasing the likelihood of including dormant vulnerabilities to 22% of the initial value. We also showed that our modularization aligned with the implicit package-based modularization in our dataset's libraries.

[7] https://junit.org/junit5/docs/5.9.0/user-guide/index.html
[8] https://logging.apache.org/log4j/2.x/manual/api-separation.html
[9] https://www.gregoryschmidt.ca/writing/openmrs-3-modularity-principles

References

[1] The Apache Software Foundation, "Maven Central Repository." [Online]. Available: https://repo.maven.apache.org/maven2/

[2] npm, Inc., "npm: Node Package Manager." [Online]. Available: https://www.npmjs.com/

[3] Python Software Foundation, "PyPI: The Python Package Index." [Online]. Available: https://pypi.org/

[4] F. R. Olivera, "MvnRepository." [Online]. Available: https://mvnrepository.com/repos/central

[5] OWASP, "OWASP Top Ten." [Online]. Available: https://owasp.org/www-project-top-ten/

[6] MITRE, "Published CVE Records." [Online]. Available: https://www.cve.org/About/Metrics

[7] P. Lam, J. Dietrich, and D. J. Pearce, "Putting the semantics into semantic versioning," in *Onward! Essays*, 2020.

[8] N. Imtiaz, S. Thorn, and L. 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)*, 2021, pp. 1–11.

[9] L. Zhao, S. Chen, Z. Xu, C. Liu, L. Zhang, J. Wu, J. Sun, and Y. 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*, 2023, pp. 960–972.

[10] GitHub, "Dependabot." [Online]. Available: https://docs.github.com/en/code-security/dependabot

[11] Synopsis, "Black Duck." [Online]. Available: https://www.synopsys.com/software-integrity/software-composition-analysis-tools/black-duck-sca.html

[12] "CVE-2022-45688," 2022, https://nvd.nist.gov/vuln/detail/CVE-2022-45688.

[13] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem," in *Proc. ICSE 2023*, 2023, pp. 1046–1058.

[14] A. M. Mir, M. Keshani, and S. 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)*. IEEE, 2023, pp. 201–211.

[15] H. Park, C. Park, S. Yoo, and K. Kim, "Detecting vulnerable Java classes based on the analysis of Java library call graph," in *iThings/GreenCom/CPSCom/SmartData*. IEEE, 2018, pp. 1872–1879.

[16] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *ICSE-NIER 2018*, 2018, pp. 101–104.

[17] B. B. Nielsen, M. T. Torp, and A. 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*, 2021, pp. 29–41.

[18] T. H. M. Le and M. A. 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*, 2022, pp. 621–633.

[19] N. Heintze and D. McAllester, "On the cubic bottleneck in subtyping and flow analysis," in *Proc. LICS 1997*. IEEE, 1997, pp. 342–351.

[20] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, p. 44–46, jan 2015. [Online]. Available: https://doi.org/10.1145/2644805

[21] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1049–1060. [Online]. Available: https://doi.org/10.1145/3377811.3380441

[22] M. Chakraborty, R. Olivares, M. Sridharan, and B. 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.

[23] D. Lehmann, M. Thalakottur, F. Tip, and M. 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*, 2023, pp. 892–903.

[24] "CVE-2015-6420," 2015, https://nvd.nist.gov/vuln/detail/CVE-2015-6420.

[25] I. Sayar, A. Bartel, E. Bodden, and Y. Le Traon, "An in-depth study of Java deserialization remote-code execution exploits and vulnerabilities," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–45, 2023.

[26] "CVE-2022-25845," 2022, https://nvd.nist.gov/vuln/detail/CVE-2022-25845.

[27] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the Maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, p. 45, 2021.

[28] J. Herrmann, M. Y. Ozkaya, B. Uçar, K. Kaya, and U. V. Catalyuurek, "Multilevel algorithms for acyclic partitioning of directed acyclic graphs," *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. A2117–A2145, 2019.

[29] The Apache Software Foundation, "Apache Maven." [Online]. Available: https://maven.apache.org/

[30] MITRE, "Common Vulnerabilities and Exposures." [Online]. Available: https://cve.mitre.org/

[31] ——, “Common Weakness Enumeration.” [Online]. Available: https://cwe.mitre.org/

[32] Snyk, “Snyk Security Database.” [Online]. Available: https://security.snyk.io/

[33] Preston-Werner, Tom, “Semantic Versioning.” [Online]. Available: https://semver.org/

[34] D. Spadini, M. Aniche, and A. Bacchelli, “Pydriller: Python framework for mining software repositories,” in *ESEC/FSE 2018*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 908–911. [Online]. Available: https://doi.org/10.1145/3236024.3264598

[35] Viswanadha, Sreenivasa and Gesser, Júlio Vilmar, “JavaParser.” [Online]. Available: https://javaparser.org/

[36] J. Katz, “Libraries.io Open Source Repository and Dependency Metadata,” Jan. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3626071

[37] C. Xu, B. Chen, C. Lu, K. Huang, X. Peng, and Y. Liu, “TRACER: finding patches for open source software vulnerabilities,” *arXiv preprint arXiv:2112.02240*, 2021.

[38] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, “CrossVul: a cross-language vulnerability dataset with commit data,” in *ESEC/FSE*, 2021, pp. 1565–1569.

[39] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, “A manually-curated dataset of fixes to vulnerabilities of open-source software,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[40] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java Virtual Machine Specification: Java SE 22 Edition*. Oracle, February 2024, https://docs.oracle.com/javase/specs/jvms/se22/html/index.html.

[41] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at Google,” *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[42] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O’Hearn, “Scaling static analyses at Facebook,” *Communications of the ACM*, vol. 62, no. 8, pp. 62–70, 2019.

[43] I. Pashchenko, D.-L. Vu, and F. Massacci, “A qualitative study of dependency management and its security implications,” in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1513–1531.

[44] M. Eichberg and B. 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*, 2014, pp. 1–6.

[45] D. Grove, G. DeFouw, J. Dean, and C. Chambers, “Call graph construction in object-oriented languages,” in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1997, pp. 108–124.

[46] B. Livshits, J. Whaley, and M. S. Lam, “Reflection analysis for Java,” in *Proc. APLAS 2005*. Springer, 2005, pp. 139–160.

[47] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis, “Static analysis of Java dynamic proxies,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 209–220.

[48] Y. Li, T. Tan, and J. Xue, “Understanding and analyzing Java reflection,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 2, pp. 1–50, 2019.

[49] G. Fourtounis and Y. Smaragdakis, “Deep static modeling of invokedynamic,” in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[50] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, “Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders,” in *Proc. ICSE 2011*, ser. ICSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 241–250. [Online]. Available: https://doi.org/10.1145/1985793.1985827

[51] P. Tonella, “Concept analysis for module restructuring,” *IEEE Transactions on software engineering*, vol. 27, no. 4, pp. 351–363, 2001.

[52] G. Antoniol, M. Di Penta, and M. Neteler, “Moving to smaller libraries via clustering and genetic algorithms,” in *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.* IEEE, 2003, pp. 307–316.

[53] M. Hall, N. Walkinshaw, and P. McMinn, “Supervised software modularisation,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 472–481.

[54] N. Anquetil, C. Fourrier, and T. Lethbridge, “Experiments with hierarchical clustering algorithms as software modularization methods,” in *Proceedings of the Working Conference on Reverse Engineering*, 1999.

[55] C. Blumschein, F. Niephaus, C. Stancu, C. Wimmer, J. Lincke, and R. Hirschfeld, “Finding cuts in static analysis graphs to debloat software,” in *Proc. ISSTA 2024*, ser. ISSTA 2024. Association for Computing Machinery, 2024.

[56] S. M. A. Shah, J. Dietrich, and C. McCartin, “On the automation of dependency-breaking refactorings in Java,” in *Proc. ICSM 2013*. IEEE, 2013, pp. 160–169.

[57] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, “On the existence of high-impact refactoring opportunities in programs,” in *Proc. ACSC 2012*, 2012, pp. 37–48.

[58] R. S. Hall and H. Cervantes, “An OSGi implementation and experience report,” in *Proc. CCNC 2004*. IEEE, 2004, pp. 394–399.

[59] A. Buckley, “JSR 277: JavaTM Module System,” 2006, https://jcp.org/en/jsr/detail?id=277.

[60] ——, “JSR 294: Improved Modularity Support in the JavaTM Programming Language,” 2007, https://jcp.org/en/jsr/detail?id=294.

[61] M. Reinhold, “JSR 376: JavaTM Platform Module System,” 2017, https://www.jcp.org/en/jsr/detail?id=376.