Software Composition Analysis for Vulnerability Detection: An Empirical Study on Java Projects

Lida Zhao
Nanyang Technological University
Singapore
LIDA001@e.ntu.edu.sg

Sen Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China
senchen@tju.edu.cn

Zhengzi Xu*
Nanyang Technological University
Singapore
zhengzi.xu@ntu.edu.sg

Chengwei Liu
Nanyang Technological University
Singapore
chengwei001@e.ntu.edu.sg

Lyuye Zhang
Nanyang Technological University
Singapore
zh0004ye@e.ntu.edu.sg

Jiahui Wu
Nanyang Technological University
Singapore
jiahui004@e.ntu.edu.sg

Jun Sun
Singapore Management University
Singapore
yangliu@ntu.edu.sg

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

ABSTRACT

Software composition analysis (SCA) tools are proposed to detect potential vulnerabilities introduced by open-source software (OSS) imported as third-party libraries (TPL). With the increasing complexity of software functionality, SCA tools may encounter various scenarios during the dependency resolution process, such as diverse formats of artifacts, diverse dependency imports, and diverse dependency specifications. However, there still lacks a comprehensive evaluation of SCA tools for Java that takes into account the above scenarios. This could lead to a confined interpretation of comparisons, improper use of tools, and hinder further improvements of the tools. To fill this gap, we proposed an *Evaluation Model* which consists of *Scan Modes*, *Scan Methods*, and *SCA Scope* for Maven (SSM), for comprehensive assessments of the dependency resolving capabilities and effectiveness of SCA tools. Based on the *Evaluation Model*, we first qualitatively examined 6 SCA tools' capabilities. Next, the accuracy of dependency and vulnerability is quantitatively evaluated with a large-scale dataset (21,130 Maven modules with 73,499 unique dependencies) under two *Scan Modes* (i.e., *build* scan and *pre-build* scan). The results show that most tools do not fully support SSM, which leads to compromised accuracy. For dependency detection, the average F1-score is 0.890 and 0.692 for *build* and *pre-build* respectively, and for vulnerability accuracy, the average F1-score is 0.475. However, proper support for SSM reduces dependency detection false positives by 34.24% and false negatives by 6.91%. This further leads to a reduction of 18.28% in false positives and 8.72% in false negatives in vulnerability reports.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*; • **Security and privacy** → **Software security engineering**; • **General and reference** → **Empirical studies**.

KEYWORDS

SCA, Package manager, Vulnerability detection

ACM Reference Format:
Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. 2023. 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 (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616299

1 INTRODUCTION

Open-source software (OSS) is getting increasing attention in software communities. Programmers import the OSS as third-party libraries (TPLs) to avoid redundant development and boost software development [65, 66]. However, importing OSS as dependencies also brings critical security threats [47, 64, 67, 68]. For example, in 2021, remote execution vulnerabilities [26, 40] were revealed in a critical OSS, named Apache Log4j2 [28]. More than 35,000 Java packages are affected [2], which accounts for roughly 8% of the packages in Maven Central Repository [30]. Therefore, individual developers and companies urge to determine whether their projects have introduced vulnerable versions of Log4j2 as dependencies.

To protect software from such security threats from TPL, Software Composition Analysis (SCA) tools [4, 6, 9–11, 16, 43, 46] have been proposed and widely adopted to identify TPL reuse and to expose the hidden vulnerability threats. With the growing complexity

*Sen Chen and Zhengzi Xu are the corresponding authors.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616299

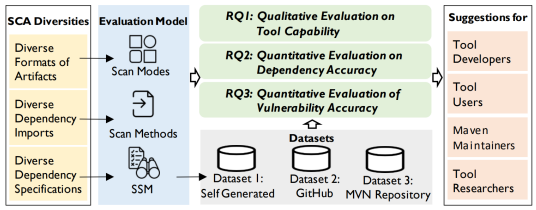


Figure 1: Overview of Our Work

of software functionality, SCA tools are facing various scenarios. Firstly, the working environment and target project format are diverse. Throughout the DevSecOps process, artifacts transition between different formats (e.g., source code, binaries) in different environments (e.g., code repositories, deployed runtime). Secondly, dependencies can be introduced by various approaches, such as through package managers or manual imports. Thirdly, different package managers maintain distinct dependency specifications for flexible dependency management such as version control. Additionally, some package managers have unique designs for specific features, such as peer dependencies in NPM [33], and dependency scopes in Maven [31]. In summary, there are three diversities that SCA tools need to consider, including *Diverse Formats of Artifacts*, *Diverse Dependency Imports*, and *Diverse Dependency Specifications*.

To get a deeper insight into the SCA tools, some research has been conducted. Nasif et al. [51] studied the differences among nine SCA tools by comparing the vulnerability results to provide insights into the adoption of SCA tools for both Java and JavaScript. Andreas et al. [49] evaluated the robustness of SCA tools in detecting vulnerabilities from TPL by four types of JAR (Java Archives) modifications (i.e., patched, Uber-JAR, bare Uber-JAR, and re-packaged Uber-JAR). However, the existing studies only peek into the performance of SCA tools by directly comparing the vulnerability results. They neglect the uneven capabilities of resolving dependencies and the various scenarios that SCA tools encountered. In fact, there is still a lack of comprehensive assessments of the SCA tool's dependency resolution capabilities that take into account the above. Such a lack could lead to a confined interpretation of comparisons, improper use of tools, and hinder further improvements of the tools.

To fill this gap, we proposed an *Evaluation Model*, which is a guideline for a comprehensive assessment of the dependency resolution capabilities and effectiveness of SCA tools for Java. In correspondence with the three diversities, the *Evaluation Model* evaluates SCA tools from three aspects: *Scan Modes*, *Scan Methods*, and *Scan Scopes*. *Scan Modes* consider the various working environments and formats of target projects; *Scan Methods* consider the TPL detection algorithms; *Scan Scopes* considered the range of dependencies to be included. By following the *Evaluation Model*, we can have a deep insight into the strength and weaknesses of the evaluated tools.

Our objective is to evaluate the performance of state-of-the-art SCA tools in their applicable scenarios based on the proposed *Evaluation Model* and to seek further improvements. Figure 1 gives an overview of our work. Based on the three diversities encountered by SCA tools, we proposed the *Evaluation Model* and introduce the three evaluation aspects in Section 3, including *Scan Methods*, *Scan Modes*, and *Scan Scopes*. In addition, we specified the *Scan Scopes* in the context of Maven to obtain the **S**can **S**cope for **M**aven (SSM). Subsequently, in RQ1, we first qualitatively investigated the tool's dependency resolving capabilities based on the Evaluation Model. In RQ2, we then quantitatively measured the dependency detection accuracy of the SCA tools in two scan modes (i.e., *build* scan & *pre-build* scan defined in Section 3.2) on a large set (21,130 Maven modules with 73,499 unique dependencies). In RQ3, we further investigated the vulnerability accuracy of all the tools and measured the improvements brought about by SSM.

The results show that current SCA tools do not support SSM well. OWASP has the best performance in dependency detection under *build* scan, while T1[1] performs the best in vulnerability reporting both *build* scan & *pre-build* scan. The average F1-score of dependency detection is 0.887 and 0.690 for *build* and *pre-build* respectively, and the average F1-score of vulnerability reporting is 0.474. Note that, proper support for SSM reduces dependency detection false positives by 34.24% and false negatives by 6.91%. This further leads to a reduction of 18.28% in false positives and 8.72% in false negatives in vulnerability reporting. Based on our comprehensive evaluation, insightful suggestions for improving SCA tools, user suggestions, and lessons learned are discussed.

In summary, we made the following main contributions:

- We proposed an *Evaluation Model* consisting of *Scan Methods*, *Scan Modes*, and *SCA Scope* for Maven (SSM) for SCA tools for comprehensive assessments of the dependency resolution capabilities and effectiveness of Java SCA tools.
- To investigate the situation of SSM support for SCA tools, we built and shared a test suite to evaluate tools' coverage of Maven dependency specifications.
- Based on the proposed *Evaluation Model*, we conducted a comprehensive study evaluating 6 state-of-the-art SCA tools for their capabilities of dependency and vulnerability detection on 13,708 real-world Maven projects. Useful findings are highlighted for further improving SCA tools.

2 BACKGROUND

2.1 Workflow of SCA

As shown in Figure 2, a typical workflow of SCA can be summarized into four steps [37]. In the first step *Scanning*, the target projects and associated artifacts are scanned to identify the reused TPLs. In this step, various scenarios should be taken into account for accurate TPL detection. Specifically, the target projects can exist in diverse environments and formats (Section 2.2), the TPLs can be introduced in different ways (Section 2.3), and different package managers may maintain distinct dependency specifications (Section 2.4). In the step of *Comparison*, the detected OSS components are compared to vulnerability databases, such as National Vulnerability Database (NVD) [34], for potential security risks. The quality of the results is affected by both the performance of dependency resolution and the completeness and correctness of the vulnerability databases. During the *Analysis* phase, guidance is provided for evaluating risks associated with detected vulnerable components, specifically security vulnerabilities. Finally, in *Reporting*, results from *Analysis* are reported to end-users in various digital formats, such as Software Bill of Materials (SBOM). In this study, we mainly focus on

[1]One of the commercial tools in our study. Details are in Section 4.2.

Figure 2: SCA Workflow

the first two steps because we aim to evaluate the dependency and vulnerability detection capabilities of the tools.

2.2 Diverse Formats of Artifacts

Software projects could be presented in different formats (i.e., source code and binary) in different environments (e.g., code repositories, deployed run-time) across their software lifecycle. Figure 3 demonstrate the main stages of DevSecOps [53]. We place emphasis on the stages where changes occur in the format or environment of the code. In the *Develop* stage, projects typically exist as source code within a development environment or code repositories that lack specific code execution environments (e.g., GitHub [23]). During the *Build* stage, the projects remain in the form of source code but are accompanied by the complete compilation environment. The building output may be a binary file, such as a JAR file. During the *Release & Deliver* stage, the projects typically take the form of binary files that are prepared for deployment in the runtime environment or to be released into artifact repositories.

2.3 Diverse Dependency Imports

TPLs can be introduced into projects in multiple ways and can reside in different locations or formats. Specifically, three ways are identified: ① by package managers, ② by external references, and ③ by source code cloning [5].

Firstly, dependencies can be imported via package managers. By explicitly claiming the dependencies in the manifest file (e.g., pom.xml), the package manager will collect the user-specified dependencies as well as the transitive dependencies from remote repositories, such as Maven central repository [30] and NPM Registry [39]. Secondly, dependencies can be introduced through external references, which may include individual components, services, or to the BOM itself [19]. In the context of Maven, users can manually pull a JAR into the project during run-time, etc. Thirdly, dependencies can be introduced by code clones [60] which refer to obtaining similar or identical code snippets by copying and pasting code in different locations [42].

2.4 Diverse Dependency Specifications

Dependency specifications refer to the features or settings used to perform dependency management. A mature package manager may maintain an extensive group of specifications for flexible dependency management, such as version control, settings inheritance, scope division, etc. Different package managers maintain distinct dependency specifications. For instance, in terms of scope division, NPM divides dependencies into two scopes, whereas Maven employs six scopes. Therefore, the study of each package manager should be conducted individually to understand its specific dependency management approach.

3 EVALUATION MODEL FOR SCA

Considering the various scenarios SCA tools are facing, we proposed an Evaluation Model for more comprehensive tool evaluation.

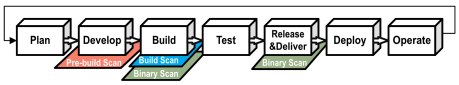


Figure 3: DevSecOps with Scan Modes

3.1 Overview of Evaluation Model

The Evaluation Model is a guideline that provides a structured approach for thoroughly and comprehensively assessing the dependency resolution capabilities and effectiveness of SCA tools. As shown in Figure 1, the Evaluation Model evaluates the SCA tools from three aspects: *Scan Modes*, *Scan Methods*, and *Scan Scopes*, which are corresponded with the three diversities: diverse format of artifacts, diverse dependency imports, and diverse dependency specifications. *Scan Modes* (Section 3.2) refers to the various environments or target project formats that the SCA tool has been designed for. Identifying the *Scan Modes* aims to determine the type of target project and working environment that the SCA tools are adapted to. *Scan Methods* (Section 3.3) refers to the various approaches employed by SCA tools to detect the dependencies. Evaluating the *Scan Methods* aims to identify the ways of introducing dependencies that the tools support. *Scan Scopes* (Section 3.4) refers to the specific range or scope of dependencies that are considered during scanning, typically those that may threaten security. Identifying the *Scan Scopes* aims to evaluate the extent to which the SCA tools handle the dependency specifications.

When using the *Evaluation Model* for tool comparison and result analysis, we followed the steps below. Firstly, we identified the *Scan Modes* adopted by the SCA tools based on their official guidance, considering the target project formats and working environments. For example, from [12], we learned that Dependabot [11] is specifically designed to scan source code within GitHub repositories. To ensure fairness and control variables, the tool comparison was conducted exclusively within the same *Scan Modes*. Once we obtained the scan results under each *Scan Mode*, we proceeded to analyze which *Scan Methods* are employed and whether the *Scan Methods* are properly used. For instance, a tool may exhibit low recall due to its limitation in parsing only the manifest file (e.g., pom.xml) while disregarding external references (e.g., manually included super JARs). Finally, we scrutinized each tool's *Scan Scope* to assess how effectively they handle dependency specifications. Some tools may exhibit inaccurate or redundant dependencies due to inadequate support for dependency specifications, for example with test-only dependencies included in the final results. By following this evaluation approach, we were able to conduct a comprehensive analysis of the dependency detection performance of SCA tools.

3.2 Scan Modes

In this section, we summarized three *Scan Modes* and their corresponding DevSecOps phases, namely *pre-build* scan, *build* scan, and *binary* scan.

In the *Develop* stage, *pre-build* scan, a low-cost scan for source code projects that works without run-time or building environment, should be applied. According to the concept of "Shift Left Security" [50], identification and correction of possible risks at an earlier stage can avoid the high costs of later repairs. Ideally, once new dependencies are added, they should be reported to avoid potential risks. *Pre-build* scan tools (e.g., [11, 36]) usually parse user-specified

dependencies directly from the manifest files (e.g., pom.xml, package.json), some of them also compute the transitive dependencies using a pre-built knowledge base. The fast and lightweight nature of *pre-build* scan also allows it to be used in multiple scenarios. *Pre-build* scan tools are usually integrated into IDEs (e.g., IDEA [25], Eclipse [21]) or code repositories (e.g., GitHub [23], Gitee [22]). Furthermore, for most industry SCA users, especially those from security sectors of traditional non-IT companies, *pre-build* scans are irreplaceable due to the following reasons: ① For many companies, establishing test environments is complex and heavy [45], *pre-build* scan relieves users from managing such complex environments. ② Training all developers in security and adding security testing processes into each CI/CD pipeline requires a tremendous workload that companies may be unwilling to undertake. While applying *pre-build* scan will not hinder the development progress nor jeopardize the stability of the CI/CD pipelines. ③ When companies need to quickly count and manage the assets of a large number of legacy projects, *pre-build* scans can avoid building various projects in intricate environments and finish the detection in a short period.

During the *Build* stage, *build* scan, the scan for source code projects within full compilation environments, should be applied. Tools support *build* scan [4, 7, 9] usually use built-in dependency resolvers (e.g., Maven Dependency Plugin [29]) or read run-time environment settings to collect dependencies. If a binary artifact is generated after building, it can also be applied to *binary* scan, which is a scan for binary targets.

During the *Release & Deliver* stage, *binary* scans are needed to avoid the further spread of vulnerabilities before being deployed or stored. Tools [4, 7, 16] could identify TPLs from binary files by multiple methods including parsing manifest files, comparing hashes, matching file names, etc.

3.3 Scan Methods

The first type of *Scan Method* is to detect dependencies imported with package managers. Two algorithms are commonly employed. The tools could either resolve dependencies using the built-in dependency resolvers [4, 7, 9] (e.g., Maven Dependency plugin [29]), or parse the manifest files directly [4, 11]. The second type of *Scan Method* is to detect dependencies introduced through external references. Tools could check the run-time environment (e.g., class-path for Maven) for all involved TPLs during the run-time. Some vendors, such as Snyk [43], claim to monitor the actual behavior of the open-source components at run-time [27]. The third type of *Scan Method* is to identify TPL reuse by cloning code snippets. Tools [63] could extract the code features and compare them against the features of all registered TPLs. An SCA tool may integrate multiple *Scan Methods*. Analyzing the types and completeness of *Scan Methods* helps explain the capability of the SCA tools.

3.4 Scan Scopes

Determining the scope of a package manager requires a detailed study of their dependency specifications. However, researchers have not reached a consensus on the Scan Scopes for Maven. Imtiaz et al. [51] and Ponta et al. [58] report all detected dependencies, while Dann et al. [49] only includes release dependencies (i.e., referable libraries that are shipped with the application) and excludes development-only dependencies by empirically excluding *test* and *provided* dependencies. Yet, they leave out other Maven dependency specifications (e.g., *type* and *classifier*) that could also help identify release dependencies. Thus, to facilitate the identification of release dependencies and confirm the Scan Scopes for Maven (SSM), a thorough study on Maven dependency specifications is required.

The SSM was determined with the following process. ***(1)*** We collected all official POM elements (Maven specifications) [3] and obtained 126 unique elements. ***(2)*** By reading the descriptions, we filtered out POM elements that obviously do not relate to dependency management, such as *organization*, *developers*, and *repository*. A POM element is considered dependency-related if any of its possible usages can fulfill any of the following two criteria: ① it changes the tree structures, including updating, removing, and adding dependencies to the dependency tree; Or ② it modifies any setting of dependencies, including *type*, *classifiers*, and *scope*. 11 out of 126 elements remained. ***(3)*** We created projects with each of the 11 POM elements and compared the dependency tree before and after adding them. If one of the two criteria is fulfilled, the element is dependency-related. 9 dependency-related POM elements (i.e., ⟨*dependencyManagement*⟩, ⟨*parent*⟩, ⟨*exclusion*⟩, ⟨*profiles*⟩, ⟨*optional*⟩, ⟨*version*⟩, ⟨*type*⟩, ⟨*scope*⟩, and ⟨*classifier*⟩) are obtained. Among them, ⟨*type*⟩, ⟨*classifier*⟩, and ⟨*scope*⟩ meet the second criterion. The settings managed by them are named *Maven dependency settings* (MDS). The functions of the rest six elements meet the first criterion and are named *Maven dependency features* (MDF). Each of the six elements corresponds to one MDF except ⟨*version*⟩. According to its possible usages, two MDF are derived from ⟨*version*⟩: *variable as version* and *version range*. *Variable as version* allows users to replace a specific version value with a variable defined in the ⟨*properties*⟩ section or other POM files, while *version range* specifies a version range to facilitate automatic version upgrades. ***(4)*** We investigated the functionalities of all official plugins [38]. By running the official examples, we found 2 plugins that can add dependencies to the project distributions. *Maven Shade Plugin* [14] includes shaded JARs and *Maven Assembly Plugin* [13] can include additional dependencies through assembly descriptor [15]. Since both plugins work similarly, they are collectively referred to as the same MDF, *plugin-packed dependencies* (PPD).

Finally, we got 8 MDF and 3 MDS in total. Table 1 summarizes the correspondence between the POM elements and MDF & MDS.

3.4.1 Maven Dependency Features. MDF are functions that change the tree structures, including updating, removing, and adding dependencies to the dependency tree. Table 1 provides a brief description of the features. A detailed explanation of PPD is provided here and further information about other features is on our website.

• *Plugin-packed Dependencies:* Distributions with PPD are *Uber JARs*. Some dependencies in the *Uber JAR* may conflict with other eponymous dependencies that have the same artifact and group Ids but in different versions. Then, *shading*, an algorithm that provides renaming methods for the JARs inside an *Uber JAR*, is introduced to avoid duplicated names and resolve conflicts. The JARs after shading are called *shaded JARs*. Except for the user-defined run-time dependencies, *shaded JARs* are one of the main forms of PPD and they may pose threats to the distribution. Taking *nacos-client-2.0.4.jar* as an example, *Google Gson 2.8.6* is one of its *shaded JARs*, which introduces CVE-2022-25647 [18].

Table 1: Overview of MDF & MDS

Category	Name	POM Element or Plugin	Description
Maven Dependency Features	Dependency Management	⟨*dependency Management*⟩	The Dependency Management is designed for batch management of component versions.
	Parent	⟨*parent*⟩	A child project can inherit all settings from the parent project, except the artifact Id.
	Exclusion	⟨*exclusion*⟩	Exclusion helps exclude artifacts from the project.
	Profile	⟨*profiles*⟩	The profiles modify the project-building process, including modifying and adding dependencies.
	Optional	⟨*optional*⟩	A dependency is not transitive if it is optional.
	Version range	⟨*version*⟩	Version ranges specify a range of versions to facilitate automatic version upgrades.
	Variable as version	⟨*version*⟩	Variable as version simplifies the batch version management by replacing versions with variables.
	Plugin-packed dependencies	*Shade Plugin Assembly Plugin*	PPD refers to the dependencies packed within a distribution, including run-time dependencies, shaded JARs, and non-open-source libraries.
Maven Dependency Settings	Type	⟨*type*⟩	Set the type of a dependency.
	Classifier	⟨*classifier*⟩	Set the classifier of a dependency.
	Scope	⟨*scope*⟩	Set the scope of a dependency.

3.4.2 Maven Dependency Settings. Some POM elements modify the settings of dependencies but keep the tree structure intact. The settings managed by such elements are MDS, including *type*, *classifier*, and *scope*. They can be further grouped into two categories. The settings that can be used to identify non-release dependencies are called *non-release dependency settings*, and others are called *release dependency settings*. Due to the page limitation, we only brief on their possible values and categories, a full discussion of reasons and usage examples will be placed on our website [20].

• *Type.* ⟨*type*⟩ has 11 predefined values, including *java-source*, *javadoc*, *pom*, *test-jar*, *jar*, *ejb*, *ejb-client*, *war*, *ear*, *rar*, and *maven-plugin*, where *jar* is the default value. In addition, developers can customize the *types*. Common examples are *zip*, *tar.gz*, and *nar*. The first four *types* are non-release dependency settings. With *java-source*, *javadoc*, and *pom*, the packing targets are the source code files, documentation, and POMs. *Test-jar* marks a dependency to be test-only and none of them are release dependency settings.

• *Classifier.* ⟨*classifier*⟩ is empty by default and has 4 predefined options, including *tests*, *javadoc*, *sources*, and *client*. Developers can customize *classifier* values such as *exec*, *bin*, and *runtime*. The first three *classifiers* are not release dependencies. *Test* marks test only dependencies, while *javadoc* and *sources* mark the package to be javadoc and source code only which are not release dependencies.

• *Scope.* ⟨*scope*⟩ has 6 values, i.e., *compile, run-time, provided, system, test*, and *import*, where *compile* is the default value. Only *compile* and *run-time* are release dependency settings. *Provided* and *system* dependencies are not shipped with distributions. Instead, projects look for valid alternatives in the running environment according to their artifact and group Ids, yet the versions can be different. *Test* dependencies are only used in the development and will not be shipped either. *Import* dependencies will not be actively added to the dependency list. Instead, all of its dependency management settings will be imported to the current ⟨*dependencyManagement*⟩, waiting to be referenced further.

3.5 Scope of Our Work

Our study specifically focused on evaluating Java SCA tools for scanning (1) source code projects within (2) the context of Maven, targeting (3) non-code-clone dependencies. Firstly, the reason for our emphasis on source code projects is their prevalence compared to binary releases. Out of the 13,709 GitHub projects we collected (details provided in Section 4.3), 9,152 projects (66.8%) are without available binary releases. Moreover, applying SCA during the source code stage enables early detection and prevention of potential security issues, aligning with the principles of "shift left security" [50]. Secondly, we chose to concentrate on Maven due to its intricate dependency management mechanism and more complex dependency specifications compared to other modern package managers. Additionally, a Java developer productivity report [8] revealed that 67% of the surveyed developers identified Maven as their primary build tool, while only 20% utilized Gradle and 11% used Ant. Thirdly, we did not evaluate the capability of detecting dependencies introduced by code clones, as to the best of our knowledge, no clone-based SCA tools are specifically designed for Java and it remains to be a challenge.

4 OVERVIEW

4.1 Research Questions (RQs)

The purpose of this paper is to evaluate the dependency and vulnerability detection performance of the SCA tools in depth based on the *Evaluation Model*. This study focuses on the following RQs.

RQ1: *How do the SCA tools cover the three aspects of the Evaluation Model?* This RQ qualitatively examines the tools' support for different *Scan Modes*, *Scan Methods*, and the SSM. Initially, we assessed the support of *Scan Modes* and *Scan Methods* of SCA tools. We then gave an overview of the prevalence of Maven dependency specifications in large datasets and examined the SCA tools' coverage of the specifications (i.e., SSM elements: MDF & MDS).

RQ2: *How is the dependency detection accuracy of SCA tools in build scan & pre-build scan?* To address this question, we conducted a quantitative analysis of the dependency resolution capabilities of SCA tools by comparing the results obtained from both the *build* scan and *pre-build* scan with the ground truth for each project.

RQ3: *What are the vulnerability accuracy of the tools and how can SSM help improve them?* After dependency detection, further risk assessments are required, where vulnerability mapping is the most common one [49, 51, 54]. In RQ3, vulnerability accuracy is evaluated. We then measured the improvement SSM brings to the vulnerability reports by removing non-release dependencies and adding missing release dependencies with their corresponding vulnerabilities.

4.2 Tool Selection

To select SCA tools for the empirical study, we performed a well-defined systematic literature review and picked 4 open-source tools from seven papers [48, 49, 51, 55–58] according to three criteria. First, the tool has to support scanning the Maven source code

projects and provides dependency and vulnerability reports separately. Second, it has to be mentioned in at least two papers. Third, it has to be available to us. The tools are OWASP Dependency Check (OWASP) [56] [58], Eclipse Steady (Steady) [49] [57], Dependabot Alerts (Dependabot) [49] [51], and OSSIndex [48] [55]. Moreover, we also selected two commercial tools T1, and T2 to evaluate the SCA accuracy of practical tools. To comply with their policies, they are anonymized. OWASP Dependency Check has 11 analyzers that provide full support for package managers such as Maven, NPM, and Nuget. Its vulnerability data comes from NVD. Eclipse Steady focuses on resolving projects of Maven, Gradle, and PIP. Its vulnerability data comes from NVD and some data works [59]. Dependabot Alerts is embedded in GitHub, which provides dependency graph detection and vulnerability alerts for 11 languages including Java. It maintains an advisory database based on NVD, public commits on GitHub, and Security advisories reported on GitHub. OSSIndex provides a free catalog of open-source components for 10 languages. All vulnerability data is derived from public sources.

4.3 Data Construction

4.3.1 Dataset 1 (DS1). *DS1* is designed to examine the coverage of Maven dependency specifications for SCA tools, where each project contains 1-to-8 examples MDF or 1 MDS. The test projects were generated from **an automated Maven project generator** developed by us according to the following steps. First, we determined 8 MDF and 3 MDS to be examined according to Section 3.4. Second, we generated all combinations of the eight MDF, a total of 256 (2^8) combinations. Third, since MDF & MDS work independently, projects are built separately. All possible options of each MDS are grouped in one project (12 options for $\langle type \rangle$, 5 options from $\langle classifier \rangle$, and 5 options from $\langle scope \rangle$), so there are three MDS-related projects. Finally, 259 projects were generated and a ground truth dependency list was attached to each of them.

4.3.2 Dataset 2 (DS2). *DS2* is a group of real-world Maven projects of all levels of popularity. The motivation for collecting it is to examine the performance of SCA tools on real-world source code projects. We searched for Java projects and sorted them according to the star number. But due to the GitHub limitation, we could only retrieve the first 1,000 projects of each star number. Then, we checked the project file structure and only kept the project with a *pom.xml* in its root path. In total, 13,708 projects were collected and the star numbers ranged from around 70,000 to 20.

4.3.3 Dataset 3 (DS3). *DS3* consists of POM files of all Maven libraries in all versions (8,364,337 versions when collected) from the Maven central [30]. This dataset is collected to reveal the distribution of dependency-related elements in all libraries.

5 EMPIRICAL STUDY

5.1 RQ1: Qualitative Evaluation on Tool Capability

5.1.1 Experimental Setup. Following the steps in Section 3.1, we first evaluate the tools' support of *Scan Mode*. Then we identifie the types of *Scan Methods* that the tools adopt based on *DS2*. The prevalence of all Maven dependency specifications (i.e., SSM elements: MDF & MDS) is investigated based on *DS2* and *DS3*. The GitHub

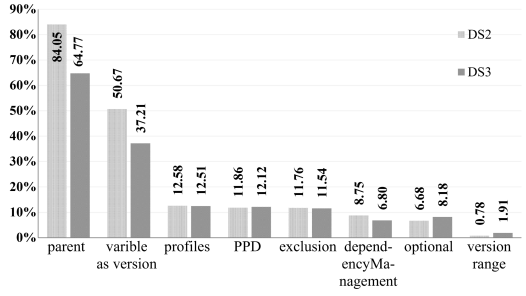
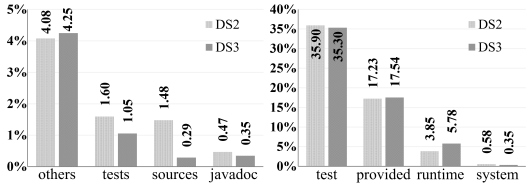
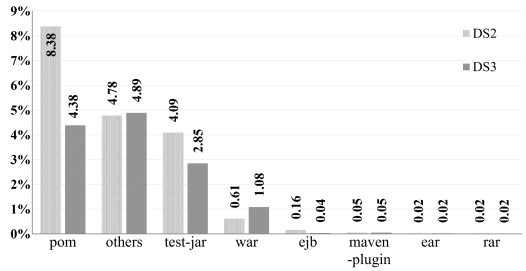


Figure 4: Prevalence of MDF



(a) Classifier Settings **(b) Scope Settings**



(c) Type Settings

Figure 5: Prevalence of MDS

Table 2: Support of Scan Methods and Scan Modes

Types of Scan Methods	*build* Scan				*pre-build* Scan		
	OSSIndex	OWASP	Steady	T1	Dependabot	T1	T2
package manager	√	√	√	√	√	√	√
external reference	×	×	×	×	×	×	×

projects from DS2 reflect the specification usage in daily coding practice, while DS3 reveals the usage in open-source libraries. We count the frequency according to the following rule. If a feature or a setting is used in a POM file, the frequency of the corresponding feature or setting is increased by one. Multiple occurrences in the same POM file are only counted once. The default values of MDS are not counted (i.e., *type-jar* and *scope-compile*) because almost all POM files have them. Then, the coverage of SSM is examined with *DS1* in both *build* scan & *pre-build* scan for each tools. For each project, by comparing the tool's detected dependencies with the ground truth, we can learn whether a specification is correctly handled. For example, when checking *dependency management*, *org.apache.logging.log4j:log4j-core:2.14.0* is placed in the $\langle dependencyManagement \rangle$ and referenced in the $\langle dependencies \rangle$ by its group and artifact name. Dependabot resolves it as *org.apache.logging.log4j:log4j-core* without version, so, *dependency management* is not supported by Dependabot in *pre-build* scans.

5.1.2 Result and Discussion. As shown in Table 2, all selected tools support detecting dependencies introduced by package managers while no one support detecting dependencies from external references, such as manually included super JARs.

Table 3: The Coverage on MDF.

Features	*build* Scan				*pre-build* Scan		
	OSSIndex	OWASP	Steady	T1	Dependabot	T1	T2
dependency -Management	√	√	√	√	×	×	√
exclusion	√	√	√	√	×	√	√
parent	√	√	√	√	×	×	×
profiles	√	√	√	√	√	√	√
optional	√	√	√	√	√	√	√
version range	√	√	√	√	√	√	√
variable as version	√	√	√	√	×	√	√
PPD	×	√	×	×	×	×	×

(1) A cross means the feature is not supported.

Table 4: The Coverage on MDS.

Category	Dependency Settings	*build* Scan				*pre-build* Scan		
		OSSIndex	OWASP	Steady	T1	Dependabot	T1	T2
Release Dependency Settings	type-jar	√	√	√	√	√	√	√
	type-ejb	√	√	√	√	√	√	√
	type-ejb-client	√	√	√	√	√	√	√
	type-war	√	√	√	√	√	√	√
	type-ear	√	√	×	√	√	√	√
	type-rar	√	×	×	√	√	√	√
	type-maven-plugin	√	√	√	√	√	√	√
	type-others	√	√	√	√	√	√	√
	classifier-client	√	√	√	√	√	√	√
	classifier-others	√	√	√	√	√	√	√
	scope-compile	√	√	√	√	√	√	√
	scope-runtime	√	√	√	√	√	√	√
Non-release Dependency Settings	type-pom	√	√	×	√	√	√	√
	type-test-jar	√	√	√	√	√	√	√
	type-javadoc	√	√	√	√	√	√	√
	type-java-source	√	√	√	√	√	√	√
	classifier-javadoc	√	√	√	√	√	√	√
	classifier-sources	√	√	√	√	√	√	√
	classifier-tests	√	√	√	√	√	√	√
	scope-provided	×	×	×	×	√	×	×
	scope-test	×	×	×	×	√	×	×
	scope-system	√	√	√	√	√	√	√

(1) For release dependency settings, a tick means correctly including release dependencies, while in non-release dependency settings, a tick means wrongly including redundant non-release dependencies.

The prevalence of MDF examined in *DS2* and *DS3* is shown in Figure 4. To keep it intuitive, columns with a frequency of less than 0.01% are not displayed (i.e., *type-java-source*, *type-javadoc*, *type-ejb-client*, and *classifier-client*). From the figure, ⟨*parent*⟩ is the most common MDF, with 84% usage in *DS2* and 64% in *DS3*. The *version range* is the least used MDF, with less than 2% occurrence.

Figure 5 shows the prevalence of MDS examined in *DS2* and *DS3*. The *scope* is the most used MDS. Over 35% of the POM files contain at least one *test* dependency. *System* dependencies are hardly used with less than 0.6% occurrence. Most values of *type* and *classifier* occur in less than 5% of POM files, and about half of the values appear in less than 0.1% of POM files. Generally, the prevalence of MDF & MDS is close in both of the datasets.

The tools' coverage of MDF & MDS examined by *DS1* is summarized in Table 3 and Table 4, respectively. By comparing the dependency output with the ground truth, if the dependency is in the report, then tick it, otherwise cross it. The results show that most of the MDF can be correctly handled by tools in the *build* scan, except the PPD. However, some high-frequency features are not supported in *pre-build* scans (i.e., *parent* and *dependencyManagement*). Dependencies with any non-release dependency settings are considered non-release dependencies. Most tools can distinguish between *scopes* and exclude *provided* and *test* dependencies. While no tool notices the differences between different kinds of *types* and *classifiers*, they include all kinds of *types* and *classifiers* indistinguishably, including the non-release dependencies settings.

Finding 1: In *build* scans, PPD is not supported by most SCA tools. Most SCA tools only rely on the result of the Maven dependency plugin, and they do not consider detecting components outside the scope of the dependency plugin. However, some PPD (e.g., shaded JARs and non-open-source libraries) are not handled by the Maven dependency plugin, which results in poor support for PPD. Figure 4 shows around 10% of projects contain PPD and thus ignoring it may entail false negatives for the detection results. There are two algorithms that help detect PPD. In *build* scans, the local library files can be obtained by reading the build-classpath. Each JAR file can then be reverse-engineered to recover the file structure to track down the PPD. In *pre-build* scans or any scenarios without local library files, tools can analyze the plugin settings and understand which files are to be packaged. OWASP uses an algorithm similar to the first solution, it collects pieces of *evidence* [4], i.e., pieces of information about dependencies, including POM files, class files, etc. The *evidence* is further used to infer the identifiers of dependencies. Previous works [49, 58] claimed Steady could analyze libraries by code-based methods, however, during the source code scans, such techniques are not applied to the library files.

Finding 2: In *pre-build* scans, SCA tools are weak in maintaining cross-project information. As shown in Table 3, *parent* and *dependencyManagement* are not well supported by most tools, though, they are used in around 80% and 8% of projects respectively. The low support for them is due to the poor maintenance of cross-project information, which is the information recorded in other related projects, usually the parent projects. Take *parent* as an example, all information in the POM file of parent projects (except for the artifact Id) is inherited by children. However, when only scanning the child project, no dependencies are inherited. The cases of *dependency management* are even worse. Users usually place dependency references in ⟨*dependencies*⟩ with only group Ids and artifact Ids, whose version will be further determined by the definition in ⟨*dependencyManagement*⟩. If the definition is placed in parent POM files, no version will be resolved for tools such as Dependabot. Reconstructing the relationships between projects is not straightforward. Some projects may have recursive parents, and settings or dependencies may conflict with their parents. Nevertheless, understanding and correctly implementing the resolving policy can decrease the false negatives significantly.

Finding 3: All SCA tools include the *system* dependencies, which are not release dependencies. Similar to *provided* dependencies, *system* dependencies are intentionally excluded from the project distributions since they may be platform specific and are usually provided by the JDK [32]. Thus, they are not release dependencies. From the perspective of security, importing dependencies with *system* scope is not suggested. Because such libraries are usually self-developed and not open-source libraries. Traditional name-based SCA for source code can hardly map them with known vulnerabilities. We recommend taking these tools out separately and testing them further with other *Scan Modes* (e.g., binary scans) or methods (e.g., code clone detection).

Finding 4: Most SCA tools pay little attention to *type* and *classifier*. In both *build* scan & *pre-build* scan, all SCA tools tend to include dependencies with all kinds of *types* and *classifiers* without displaying them in the report. Omitting the settings hinders the identification of non-release dependencies and leads to an increase

in false positive release dependencies. We recommend that these tools retain information about *type* and *classifier* in the reports and provide users with the option to manage them.

Answering RQ1: ***(1)*** In both *build* scan and *pre-build* scan, PPD (an MDF that occurs in around 12% of POM files) are not supported by most SCA tools. ***(2)*** Although cross-project managing MDF (i.e., *parent* and *dependency management*) have the highest occurrence, the tools of *pre-build* scan are less capable of supporting them compared to *build* scan. ***(3)*** In MDS, Dependencies with *Scope-system* are non-release dependencies but all tools include them. ***(4)*** Most tools do not distinguish *type* and *classifier* in MDS, and they tend to include all kinds of dependencies, including non-release dependencies.

5.2 RQ2: Quantitative Evaluation on Dependency Accuracy

5.2.1 Experiment Setup. This experiment evaluates the accuracy of SCA tools in both *build* scan and *pre-build* scan on *DS2*. We collected dependency ground truths for the projects that were buildable. Out of 13,708 projects, 3,955 are buildable which contain 21,130 Maven modules and 73,499 unique dependencies. The ground truths for dependencies were built up in the following steps. ***(1)*** First, we determined whether a project is buildable by running *mvn compile*. ***(2)*** Second, we retrieved the project dependency tree and the build-classpath by calling the Maven Dependency Plugin [29]. ***(3)*** Third, following the SSM, we removed the non-release dependencies from the ground truths according to their settings. ***(4)*** Fourth, we located all the library files according to the build-classpath and detected the PPD by parsing manifest files to add them to the ground truth. ***(5)*** Last, we searched for all JARs included in the projects based on their extension name (i.e., *.jar*, *.war*, *.rar*). Out of 3,955 projects, 340 projects were found to contain a total of 9,886 JARs. For each JAR, we first calculated the SHA1 value and searched it using Central Repository API [41]. 8,263 JARs were confirmed by SHA1 matching. If no match was found, we searched the JAR's name, which successfully identified only 1 JAR. Unfortunately, 1,622 JARs failed to match with any existing published JARs on Maven Central. In total, 8,264 TPLs were identified and added to the ground truth.

According to the functionality of the 6 tools, they were split into two groups and tested in *build* scan and *pre-build* scan respectively. Four tools, including OSSIndex, OWASP, Steady, and T1, support *build* scan, while three tools, i.e., T1, T2, and Dependabot, support *pre-build* scan. We triggered scans with all tools on the 3,955 projects, but some tools failed to scan part of the projects. T2 has scan limitations that we finally imported 1,509 Maven projects. Steady takes on average about 38 minutes to scan a project at 3 processes, and more than 3 processes will crash its service. Therefore, due to the time limitation, we scanned 467 projects with it. When analyzing the results, we only included the successful cases of each tool. All scanning records are available on our website [20]. We compared the scan results of SCA tools with the ground truth of the projects using precision ($\frac{\#TP}{\#TP+\#FP}$), recall ($\frac{\#TP}{\#TP+\#FN}$), and F1-score ($\frac{2\times recall\times precision}{recall+precision}$) as evaluation metrics. Here, $\#TP$ represents the number of correctly resolved dependencies by the tool, $\#FP$ represents the number of incorrectly resolved dependencies, and $\#FN$ represents the number of dependencies missed by the tools according to the ground truth.

Table 5: Dependency Accuracy for *build* Scan

Tool	Precision	Recall	F1-Score
OSSIndex	0.997	0.833	0.908
OWASP	0.997	0.885	0.937
Steady	0.996	0.730	0.843
T1	0.998	0.755	0.860
avg.	0.997	0.801	0.887

Table 6: Dependency Accuracy for *pre-build* Scan

Tool	Precision	Recall	F1-Score
Dependabot	0.525	0.287	0.371
T1	0.999	0.754	0.859
T2	0.838	0.840	0.839
avg.	0.787	0.627	0.690

5.2.2 Result and Discussion for build Scan . What stands out in Table 5 is that the precision of all tools is close and is generally much higher than the recall in *build* scan. Among all tools, OWASP has the highest recall, with 0.889. The recall of Steady and T1 are below the average, with 0.736 and 0.759, respectively.

Missing PPD and super JARs are the main reasons for low recall which counts for about 4% and 2.8% of all release dependencies, respectively. It is tricky that some of the dependencies are marked as *provided* dependencies in the POM file while the provider still packed them into the distribution as shaded JARs. Such a phenomenon reveals the truth that judgment by *scope* settings of dependencies alone is unreliable and must be supplemented by the detection of PPD. OWASP is the only tool that detects shaded JARs from local libraries. Another general cause that impairs the recall is the versions with *SNAPSHOT* which counts for about 4.8% of all release dependencies. *SNAPSHOT*, as a temporary version mark, is usually used in projects under development. Such *SNAPSHOT*-version packages are usually neither publicly released nor stable. OWASP, Steady, and T1 refuse to report versions with *SNAPSHOT*. Furthermore, the delay in scan time leads to a gap between the ground truth and the tools' reports. We found that about 2% of dependencies define their versions as *LATEST*, which will always retrieve the latest library version for not only themselves but also all their transitive dependencies. Some organizations, such as *software.amazon.awssdk*, publish dependencies every day, and if their reports are not generated on the same day, the results will be different.

5.2.3 Result and Discussion for pre-build Scans. Table 6 shows *pre-build* scan results on the three SCA tools. The average F1-score of these tools in *pre-build* scan is much lower compared to *build* scan, at 0.692. The performance of different tools varies greatly. T1 performs best in precision while T2 has the best recall which is about 3 times higher than the recall of the worst tool, Dependabot.

In *pre-build* scans, build-classpath and library files are not accessible, which makes the PPD inaccessible and decreases the recall by about 4%. Furthermore, compared to the tools in *build* scan, the tools in *pre-build* scan have worse support on MDF, especially in maintaining cross-project information, e.g., versions in $\langle dependencyManagement \rangle$ and variables in $\langle properties \rangle$. For instance, Dependabot does not resolve such information and many dependency versions remain empty or stay as variables that are unreadable. Another challenge of *pre-build* scan tools is to maintain the transitive dependency graph information for all libraries to extend the user-specified dependencies into integrated dependency

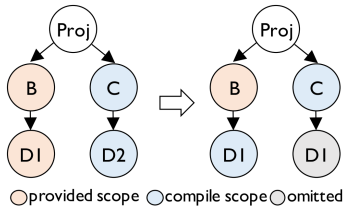


Figure 6: Maven Dependency Conflict Resolution

trees. However, with new library versions being released every day, the tools need to continuously update the graph to ensure its reliability. T1 and T2 partially resolve the transitive dependencies, while Dependabot only reports user-specified dependencies. None of the tools resolves the *LATEST* into specific versions. Furthermore, deficiencies in handling dependency conflicts contribute to lower recall. For example, T1 misses several release dependencies because the dependency conflict is not correctly handled. If there are two dependencies with the same name but in different versions, Maven will pick the "nearest definition". That is, Maven will always choose the version of the closest dependency on the dependency tree [31]. Therefore, *compile* dependencies are sometimes listed as children of the *test* or *provided* dependencies. Figure 6 explains the example. Component *D* is imported twice by different libraries (*B* and *C*) and they are in *scope-provided* and *scope-compile* respectively. Now that *D1* is the "nearest definition", then the *D2* will be updated to *D1* and omitted. Besides, *scope-compile* always has a higher priority over *scope-provided*. Thus, the scope of *D1* will be updated from *provided* to *compile*. Instead of taking this situation into account, T1 cuts off the whole branch of the *provided* dependency (*B*), causing the absence of the *compile* dependencies (D1). One possible solution is to cut the whole dependency branch after recovering the omits and reverse the scope update to restore the complete dependency tree.

Answering RQ2: ***(1)*** In *build* scan, OWASP performs the best with F1-score of 0.937, while in *pre-build* scan, T1 has the highest F1-score of 0.860. ***(2)*** The precision and recall of dependency detection for tools in *build* scan are higher than that in *pre-build* scan, with average F1-scores of 0.887 and 0.690, respectively. ***(3)*** For *pre-build* scan tools, the main reasons that reduce the detection performance are failing to maintain cross-project information (e.g., versions in ⟨*dependencyManagement*⟩ and variables in ⟨*properties*⟩), failing to resolve transitive dependency resolution, and errors in handling dependency conflicts, while for *build* scan tools, the main reasons are missing PPD. In both *Scan Modes*, tools do not translate "LATEST" to specific versions.

5.3 RQ3: Quantitative Evaluation of Vulnerability Accuracy

5.3.1 Experiment Setup. This experiment has two purposes: to evaluate the vulnerability detection accuracy and to quantify the impact SSM brought to the vulnerability reports.

For the first purpose, the scanning projects and methods are the same as RQ2, but the focus is on vulnerability reports. The vulnerability ground truth is derived from GitHub Advisory Database [24] (GAD) which is the biggest open-source vulnerability database available to the public. We only used the reviewed Maven advisories with 3,036 unique CVE Ids (Jan 11, 2023). The precision of all tools may be affected due to the incompleteness of GAD.

Table 7: Vulnerability Precision and Recall

Mode	Tool	Precision	Recall	F1
build	OSSIndex	0.470	0.363	0.410
build	OWASP	0.567	0.621	0.593
build	Steady	0.348	0.177	0.234
build	T1	0.636	0.629	0.633
pre-build	Dependabot	0.846	0.378	0.522
pre-build	T1	0.627	0.625	0.626
pre-build	T2	0.288	0.324	0.305

Table 8: Vulnerability Improvements Based on SSM

Mode	Tool	RFP	RFN
build	OSSIndex	0.000	0.120
build	OWASP	0.001	0.010
build	Steady	0.000	0.074
build	T1	0.000	0.051
pre-build	Dependabot	0.155	0.349
pre-build	T1	0.002	0.069
pre-build	T2	0.076	0.036
build	tree-plugin*	0.189	0.057

*tree-plugin refers to Maven Dependency Plugin

Table 9: Details of RFN and RFP for Tree-plugin Results

	Dependency		Vulnerability	
Item	**Number**	**Percentage**	**Number**	**Percentage**
Total Count	287,728	100%	113,111	100%
RFN	19,880	6.91%	9,864	8.72%
– PPD	11,617	4.16%	6,202	5.67%
– super JARs	8,263	2.87%	3,662	3.24%
RFP	98,516	34.24%	20,673	18.28%
– scope-provided	32,785	11.39%	11,196	9.90%
– scope-test	63,555	22.09%	9,382	8.29%
– classifier-tests	406	0.14%	117	0.10%
– type-test-jar	264	0.09%	74	0.07%
– classifier-sources	228	0.08%	60	0.05%
– type-pom	1,657	0.58%	11	0.01%
– scope-system	330	0.11%	7	0.01%
– type-java-source	0	0.00%	0	0.00%
– type-javadoc	4	0.00%	0	0.00%
– classifier-javadoc	8	0.00%	0	0.00%

To quantify the impact of SSM, the vulnerabilities related to non-release dependencies and missing release dependencies are referred to as the reduced false positives (RFP) and reduced false negatives (RFN), respectively. For each tool, RFP is the sum of vulnerability numbers related to non-release dependencies of all projects divided by the total vulnerability number of all projects; and RFN is the sum of vulnerability numbers related to missing release dependencies of all projects divided by the total vulnerability number. Vulnerability reports are obtained directly by mapping vulnerability data to detected dependencies, which quality is influenced by two factors. ① dependency detection accuracy and ② vulnerability mapping data quality. Improving vulnerability mapping data often requires extensive manual collection and validation by professionals. Some big organizations are working on it [24, 35] and their quality is not compared here. Besides, we aim to improve the accuracy of vulnerability reporting by improving the accuracy of dependency detection. Thus, to avoid the impact of vulnerability data quality, in the second experiment, RFP and RFN were only calculated based on each tool's own reported vulnerabilities. We also included the Maven Dependency Plugin [29] results as a reference (tree-plugin), whose dependency reports are unique dependencies parsed from the plugin's output, and the vulnerability data are mapped from GAD. We further decomposed the RFP and RFN of the tree-plugin according to SSM to detail the reasons for the improvements.

5.3.2 Result and Discussion. Table 7 shows the precision and recall in both *build* scan & *pre-build* scan. In *build* scan, T1 has the highest precision and recall in vulnerability detection while Steady has the worst. In *pre-build* scan, Dependabot has the best precision and T1 has the best recall. Overall, the precision and recall of all tools are generally not good enough, which can be improved by applying SSM, as dependency detection performance has a great influence on vulnerability detection performance. Take Dependabot as an example, it gets 0.287 recall in dependency detection and gets only 0.378 recall in vulnerability reporting.

Table 8 shows the vulnerability reporting improvements by removing non-release dependencies or adding missing release dependencies with their corresponding vulnerabilities according to the SSM. Some tools (i.e., OSSIndex, OWASP, T1, and Steady) have almost no reduction in FP because they excluded the *scope-provided* and *scope-test* dependencies by default, which are the two major contributors to non-release dependencies. As we analyzed in RQ1, Dependabot does not exclude such non-release dependencies and thus has more reduction in FP. Compare to RFP, RFN is more significant in all tools. Most RFN comes from PPD and only OWASP supports finding PPD; thus, it has the lowest RFN among all tools. PPD vulnerabilities count around 5.48% of all vulnerabilities.

To provide a deeper analysis of the improvements, Table 9 shows the results of the decomposition of the RFP and RFN reported by the tree-plugin according to SSM. 34.24% of dependencies reported by the tree plugin are non-release dependencies (RFP) and 6.91% of release dependencies are missing (RFN). According to Section 3.4, 10 sub-classes contribute to RFP, (i.e., *scope-test*, *scope-provided*, *scope-system*, *type-pom*, *type-java-source*, *type-javadoc*, *type-test-jar*, *classifier-sources*, *classifier-tests*, and *classifier-javadoc*) and *scope-test*, *scope-provided* are two major contributors which count 22.09% and 11.39% dependencies respectively. The sum of the percentage of other contributors is less than 1%, and 5 of them are barely used with less than 0.05% prevalence. Since one dependency can match with several items and fit in multiple sub-classes, the sum of the breakdown numbers may be larger than the total counts. For instance, a dependency can be set with both *scope-test* and *type-test-jar*. While in the RFN, PPD and super JARs are related, which count 4.04% and 2.87% of the total dependency count respectively. The influence on dependency detection further influences vulnerability reporting. 18.28% of vulnerabilities from non-release dependencies are reduced and 8.72% of missing release dependencies vulnerabilities are identified. Items that have a big impact on dependency accuracy might have a small impact on vulnerability reporting. Although there are nearly twice as many *test* dependencies as *provided* dependencies, *provided* dependencies introduce more vulnerabilities than *test* dependencies. Removing *provided* dependencies reduces 9.90% of vulnerabilities while removing *test* dependencies reduces 8.29% of vulnerability reports. The remaining non-release dependency settings have almost no contribution to reducing false positive vulnerabilities. Detecting PPD and super JARs help reduce 5.48% and 3.24% of false negative vulnerabilities respectively.

Answering RQ3: ***(1)*** In both *build* scan and *pre-build* scan, T1 performs best in vulnerability reporting. ***(2)*** SSM shows the greatest improvement for Dependabot with an RFN of 0.349. For the result of the tree-plugin, applying SSM helps reduce 18.28% of false positive vulnerabilities and locate 8.72% of false negative vulnerabilities. *Scope-provided* and *scope-test* are the main contributors to the RFP while locating more PPD is the main contributor to the RFN. ***(3)*** To improve vulnerability detection performance, it is important to guarantee dependency detection performance. The flaws discussed in RQ2 should be mitigated (e.g., by excluding non-release dependencies (especially *scope-provided* and *scope-test*), paying more attention to detect PPD, and enhancing cross-project information maintenance).

6 DISCUSSION

Tool developers should improve the tools to fully handle Maven dependency specifications. ① Existing SCA tools rely heavily on the Maven Dependency Plugin [29] to retrieve the dependency trees, as this plugin resolves all user-specified dependencies and transitive dependencies. However, it sometimes includes non-release dependencies and misses the PPD (e.g., shaded JARs and non-open-source libraries). Existing SCA tools should enrich the *Scan Methods* and distinguish non-release dependencies with a better understanding of SSM. ② For *pre-build* scan tools, they do not have the assistance of the Maven Dependency Plugin and have to resolve the manifest files (e.g., pom.xml) from scratch. All MDF should be well supported, especially to maintain cross-project information and resolve transitive dependencies. ③ Performance and scalability need to be considered. In reality, large commercial companies or large open-source organizations often pay more attention to security reviews. These organizations often have massive projects that require rapid and scaled scans. During the experiment of RQ2, we recorded the execution time of *build* scans (*pre-build* scans are executed on remote servers, so no execution time is recorded.) The result shows that Steady is the slowest one with an average execution time of 2,266 seconds (about 38 minutes), while OSSIndex, OWASP, and T1 took 77, 105, and 209 seconds on average, respectively. OWASP, OSSIndex, and T1 provide reports in JSON or CSV format with full dependency lists, while Steady only provides the dependency list on its GUI web pages, which is not code-friendly and scalable. In terms of report acquisition and execution time, steady is not a suitable tool for scanning large-scale datasets.

Tool developers should focus on developing Java SCA tools that can detect TPLs imported by external reference and copy-and-paste. In terms of *Scan Methods*, most tools concentrate on detecting TPLs imported by package managers, with few or no tools supporting detection of TPLs imported by external reference and copy-and-paste. This is problematic as Lopes et al [1] discovered that approximately 26% of files are duplicated and 65% of functions are cloned in Java GitHub projects. Failure to include such capabilities results in incomplete TPL detection for Java and leaves potential vulnerabilities.

Users are recommended to choose different tools according to various scenarios. Tools perform differently in different scenarios, so when choosing a tool, users should never lose sight of the environmental context and purpose. To obtain vulnerability reports, for projects under development or large legacy projects that require rapid and scaled scanning, *pre-build* scan is appropriate. The best tool is T1 and the best free tool is Dependabot. For projects ready

for building and releasing, *build* scan is a superior option. The best tool is T1, and the best free tool is OWASP.

Package manager designers should simplify dependency specifications to facilitate dependency management. Maven is a legacy package manager with many dependency management problems [61, 62], and compared to newer package managers (e.g., cargo [17] and NPM [39]), it has too many dependency specifications. For example, NPM identifies release dependencies by two dependency scopes (i.e., *devDependencies* and *dependencies*), while Maven has 6 dependency *scopes* and other settings (MDS) or features (MDF). Complex dependency specifications can result in some settings rarely used (e.g., *type-ejb-client*, and *classifier-client*), increasing the barrier to use for both users and tool developers.

Tool testers or researchers should apply the *Evaluation Model* when measuring detection performance for all SCA tools. Applying the *Evaluation Model* helps users to understand the tools fairly and comprehensively. Although we only compared Maven SCA tools in this work, the three diversities of SCA scenarios are practical for most package managers of other languages. The *Evaluation Model* can be applied directly to them, except that each package manager has its own unique dependency specifications, which may require additional examination and modeling.

7 THREATS TO VALIDITY

The absence of TPLs imported by source code clones in the ground truth can introduce bias to the precision and recall measurements of the tools. Since no Java SCA tool can detect TPL reuse by code clone, we attempt to collect this ground truth data independently. However, we encounter challenges in reliably identifying TPL reuse through code clones. It is difficult to determine the extent to which clones indicate TPL reuse, as well as to identify the specific TPL when multiple TPLs contain the same functions. Ultimately, we decide not to include the TPLs imported by code clones in the ground truth, to avoid introducing greater uncertainty and reduce the reliability of our results.

Different versions of SCA tools threaten the validity of results. The tool versions we used are as follows: OWASP (v7.1.1), OSSIndex (v3.20), and Steady (v3.2.4). Other tools provide online services and the exact version can not be confirmed. These tools will progress over time, which may result in inconsistent results. To minimize the influence, we suggest using the same version of the tools if they are available.

Different scanning times for *build* scans can lead to different results. Most projects from RQ2 are under active development, therefore their contents may change over time. Possible contingencies include, but are not limited to ① the project updates to different dependency versions; ② the GitHub repository is switched to private or deleted. Even if the project contents are not changed, dependency trees could change if new versions satisfying dependency constraints are released [52]. To minimize the influence, we have tried our best to collect scan results at almost the same time and keep the building log for further analysis.

Excluding failed scans during data collection may compromise measurement validity. It is possible for tools to fail in scanning certain projects, and we opted to exclude these failed scans during the data collection process. However, this exclusion may introduce variations in the datasets used for evaluating different tools, which could undermine the validity of the results. To mitigate this issue, we made efforts to run as many projects as possible within the datasets.

8 RELATED WORK

Dann et al. [49] conducted an empirical study on the types of modifications on Java releasing libraries that could influence vulnerable dependencies detection. They summarized 4 types of modifications (i.e., re-compilation, re-bundling, metadata-removal, and re-packaging) and studied their prevalence on JARs. Then they triggered scans on 6 SCA tools to learn the impact of the modifications. Imtiaz et al. [51] evaluated nine SCA tools on a large web application in both NPM and Maven by comparing the differences in the vulnerability reports. The authors claimed that the key point for good SCA tools is the accuracy of the vulnerability database. We have identified significant differences between our work and the previous research works [51] and [49]. Both of these works directly compare vulnerability results, which we find insufficient to draw convincing conclusions. Vulnerability results can be influenced by the accuracy of dependency detection as well as the quality of vulnerability mapping data. In this study, we conducted separate evaluations on dependency accuracy and investigated the factors that impede the tools' ability to detect dependencies effectively. Additionally, unlike [49], we extended the evaluation to source code projects. Furthermore, in contrast to [51], we have studied the various scenarios encountered by SCA tools and proposed a comprehensive assessment of the dependency resolution capabilities for Java SCA tools. This approach has enabled us to gain new insights and make discoveries from a different perspective.

9 CONCLUSION

In this work, we proposed an *Evaluation Model* for SCA tools consists of *Scan Methods*, *Scan Modes*, and *Scan Scopes* for Maven. Then, we qualitatively examined the capabilities of the tools based on the *Evaluation Model* and quantitatively evaluated the dependency and vulnerability detection performance of 6 SCA tools on 21,130 Maven modules in two *Scan Modes* (i.e., *build* scan & *pre-build* scan). We also measured the improvement brought about by SSM and provided some recommendations for further tool improvements.

10 DATA AVAILABILITY

More discussions and examples of MDF & MDS, the details of the datasets, and the evaluation data can be publicly accessed at [44].

ACKNOWLEDGEMENTS

This work was supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004), the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore, National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1 (10 2017), 28. Issue OOPSLA. https://doi.org/10.1145/3133908

[2] 2021. Google Online Security Blog: Understanding the Impact of Apache Log4j Vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html.

[3] 2021. Maven Pom Descriptor Reference documentation. https://maven.apache.org/ref/3.8.5/maven-model/maven.html.

[4] 2021. OWASP Dependency-Check Project - OWASP. https://owasp.org/www-project-dependency-check/.

[5] 2021. Software dependencies: How to manage dependencies at scale | Why you should manage open source dependencies. https://snyk.io/series/open-source-security/software-dependencies/#managing-open-source-dependencies.

[6] 2022. Component Analysis OWASP Foundation. https://owasp.org/www-community/Component_Analysis.

[7] 2022. Eclipse Steady. https://github.com/eclipse/steady.

[8] 2022. Java build tools comparison. https://www.jrebel.com/blog/java-build-tools-comparison.

[9] 2022. Sonatype OSS Index. https://ossindex.sonatype.org/.

[10] 2022. WhiteSource - Open Source Security and License Management. https://www.whitesourcesoftware.com/.

[11] 2023. About alerts for vulnerable dependencies - GitHub Docs. https://docs.github.com/en/code-security/supply-chain-security/managing-vulnerabilities-in-your-projects-dependencies/about-alerts-for-vulnerable-dependencies.

[12] 2023. About code scanning - GitHub Docs. https://docs.github.com/en/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/about-code-scanning#about-code-scanning.

[13] 2023. Apache Maven Assembly Plugin. https://maven.apache.org/plugins/maven-assembly-plugin/examples/multimodule/module-binary-inclusion-simple.html.

[14] 2023. Apache Maven Shade Plugin. https://maven.apache.org/plugins/maven-shade-plugin/examples/attached-artifact.html.

[15] 2023. Assembly Descriptor. https://maven.apache.org/plugins/maven-assembly-plugin/assembly.html.

[16] 2023. Black Duck Software Composition Analysis (SCA) - Synopsys. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html.

[17] 2023. Cargo. https://cargo.site/.

[18] 2023. CVE 2022 25647. https://www.cve.org/CVERecord?id=CVE-2022-25647.

[19] 2023. CycloneDX Use Cases. https://cyclonedx.org/use-cases/#external-references.

[20] 2023. Data Website. https://sites.google.com/view/fse2023scastudy.

[21] 2023. eclipse IDE for Java Developers. https://www.eclipse.org/downloads/packages/release/kepler/sr1/eclipse-ide-java-developers.

[22] 2023. Gitee. https://gitee.com/.

[23] 2023. GitHub. https://github.com/.

[24] 2023. github/advisory-database - GitHub. https://github.com/github/advisory-database.

[25] 2023. IDEA Jetbrain. https://www.jetbrains.com/idea/.

[26] 2023. Incomplete fix for Apache Log4j vulnerability. https://deps.dev/advisory/GHSA/GHSA-7rjr-3q55-vv33.

[27] 2023. Introducing open source security runtime monitoring. https://snyk.io/blog/introducing-open-source-security-runtime-monitoring/.

[28] 2023. Log4j – Apache Log4j 2. https://logging.apache.org/log4j/2.x/.

[29] 2023. Maven Dependency Tree Plugin. https://maven.apache.org/plugins/maven-dependency-plugin/tree-mojo.html.

[30] 2023. Maven Repository: Search/Browse/Explore. https://mvnrepository.com/.

[31] 2023. Maven – Introduction to the Dependency Mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[32] 2023. Maven – Introduction to the Dependency Mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[33] 2023. NPM - Peer Dependencies. https://nodejs.org/es/blog/npm/peer-dependencies/.

[34] 2023. NVD - Vulnerabilities. https://nvd.nist.gov/vuln.

[35] 2023. OSV - A distributed vulnerability database for Open Source. https://osv.dev/.

[36] 2023. OSV-Scanner. https://github.com/google/osv-scanner.

[37] 2023. Overview | Software composition analysis. https://en.wikipedia.org/wiki/Software_composition_analysis.

[38] 2023. Plugins Supported By The Maven Project. https://maven.apache.org/plugins/index.html.

[39] 2023. Registry for Node Package Manager. https://www.npmjs.com/.

[40] 2023. Remote code injection in Log4j. https://deps.dev/advisory/GHSA/GHSA-jfh8-c2jp-5v3q.

[41] 2023. REST API - The Central Repository Documentation. https://central.sonatype.org/search/rest-api-guide/.

[42] 2023. Snippet Information - specification v2.2.2. https://spdx.github.io/spdx-spec/v2.2.2/snippet-information/.

[43] 2023. Snyk - Developer security - Develop fast. Stay secure. https://snyk.io/.

[44] 2023. Software composition analysis for vulnerability detection: An empirical study on Java projects. https://sites.google.com/view/fse2023scastudy.

[45] 2023. What Is a Test Environment? A Guide to Managing Your Testing. https://www.testim.io/blog/test-environment-guide/.

[46] 2023. Your Partner in Open Source - Debricked. https://debricked.com/.

[47] Sultan S Alqahtani, Ellis E Eghan, and Juergen Rilling. 2017. Recovering semantic traceability links between APIs and security vulnerabilities: An ontological modeling approach. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 80–91.

[48] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. 2021. Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2021), 4255–4269.

[49] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. 2021. Identifying Challenges for OSS Vulnerability Scanners - A Study & Test Suite. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2021.3101739

[50] Steven J Hutchison. 2013. Shift Left!-Test Earlier in the Life Cycle. *Defense AT&L Magazine*, 35–39. Issue September–October. http://www.gao.gov/

[51] Nasif Imtiaz, Seaver Thorne, and Laurie Williams. 2021. A Comparative Study of Vulnerability Reporting by Software Composition Analysis Tools. *International Symposium on Empirical Software Engineering and Measurement* (8 2021). https://doi.org/10.1145/3475716.3475769

[52] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. *arXiv preprint arXiv:2201.03981* (2022).

[53] Department of Defense (DoD) Chief Information Officer. 2019. DoD Enterprise DevSecOps Reference Design.

[54] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. *IEEE Transactions on Software Engineering* (9 2020), 1–1. Issue 01. https://doi.org/10.1109/TSE.2020.3025443

[55] Christina Paule, Thomas Düllmann, and Andreas Falk. 2018. Securing DevOps-Detection of vulnerabilities in CD pipelines. (2018), 77–78.

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

[57] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond Metadata: Code-centric and Usage-based Analysis of Known Vulnerabilities in Open-source Software. *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018* (6 2018), 449–460. https://doi.org/10.48550/arxiv.1806.05893

[58] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (2020), 3175–3215.

[59] Serena E Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source softwareCCF CMining Software Repositories (MSR): CCF C. *ieeexplore.ieee.org* (2019). https://ieeexplore.ieee.org/abstract/document/8816802/

[60] G. Shobha, Ajay Rana, Vineet Kansal, and Sarvesh Tanwar. 2021. Code Clone Detection—A Systematic Review. (2021), 645–655. https://doi.org/10.1007/978-981-33-4367-2_61

[61] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing Chi Cheung. 2018. Do the dependency conflicts in my project matter? *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (10 2018), 319–330. https://doi.org/10.1145/3236024.3236056

[62] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, and Chang Xu. 2020. Will Dependency Conflicts Affect My Program's Semantics? *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* 6 (2020). Issue 6. https://sensordc.github.io/.

[63] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 860–872.

[64] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1695–1707.

[65] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, Yang Liu, and Yang 2020 Liu. 2020. Automated Third-Party Library Detection for Android Applications: Are We There Yet? (2020). https://doi.org/10.1145/3324884.3416582

[66] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2021. Research on third-party libraries in Android apps: A taxonomy and

systematic literature review. *IEEE Transactions on Software Engineering* (2021).

[67] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Linlin Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating Persistence of Open-Source Vulnerabilities in Maven Ecosystem. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering*.

[68] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu. 2023. Compatible Remediation on Vulnerabilities from Third-Party Libraries for Java Projects. *arXiv preprint arXiv:2301.08434* (2023).

Received 2023-02-02; accepted 2023-07-27