Does the Vulnerability Threaten Our Projects? Automated Vulnerable API Detection for Third-Party Libraries

Fangyuan Zhang, Lingling Fan*, Sen Chen, Miaoying Cai, Sihan Xu, and Lida Zhao

Abstract—Developers usually use third-party libraries (TPLs) to facilitate the development of their projects to avoid reinventing the wheels, however, the vulnerable TPLs indeed cause severe security threats. The majority of existing research only considered whether projects used vulnerable TPLs but neglected whether the vulnerable code of the TPLs was indeed used by the projects, which inevitably results in false positives and further requires additional patching efforts and maintenance costs (e.g., dependency conflict issues after version upgrades).

To mitigate such a problem, we propose VAScanner, which can effectively identify vulnerable root methods causing vulnerabilities in TPLs and further identify all vulnerable APIs of TPLs used by Java projects. Specifically, we first collect the initial patch methods from the patch commits and extract accurate patch methods by employing a patch-unrelated sifting mechanism, then we further identify the vulnerable root methods for each vulnerability by employing an augmentation mechanism. Based on them, we leverage backward call graph analysis to identify all vulnerable APIs for each vulnerable TPL version and construct a database consisting of 90,749 (2,410,779 with library versions) vulnerable APIs with 1.45% false positive proportion with a 95% confidence interval (CI) of [1.31%, 1.59%] from 362 TPLs with 14,775 versions. The database serves as a reference database to help developers detect vulnerable APIs of TPLs used by projects. Our experiments show VAScanner eliminates 5.78% false positives and 2.16% false negatives owing to the proposed sifting and augmentation mechanisms. Besides, it outperforms the state-of-the-art method-level vulnerability detection tool in analyzing direct dependencies, Eclipse Steady, achieving more effective detection of vulnerable APIs. Furthermore, to investigate the real impact of vulnerabilities on real open-source projects, we exploit VAScanner to conduct a large-scale analysis on 3,147 projects that depend on vulnerable TPLs, and find only 21.51% of projects (with 1.83% false positive proportion and a 95% CI of [0.71%, 4.61%]) were threatened through vulnerable APIs, demonstrating that VAScanner can potentially reduce false positives significantly.

Index Terms—Vulnerability Detection, Software Composition Analysis, Static Analysis

1 Introduction

Java developers frequently incorporate third-party libraries (TPLs) to speed up software development. However, the utilization of TPLs may introduce security threats [1], [2]. According to an open-source security and risk analysis report released by Synopsys [3], 97% of the 2,409 codebases contained open-source components, and 81% of them contained at least one known vulnerability. To mitigate such a severe problem, software composition analysis (SCA) [4]–[13] is typically used to identify vulnerable TPLs. A couple of SCA tools have been suggested including Eclipse Steady [14], Dependabot [7], OSSIndex [8], OWASP Dependency Check [5], etc.

However, from the detection side, nearly all SCA tools can only determine whether vulnerable TPLs are depended on by projects, but cannot tell whether vulnerable APIs are actually invoked, resulting in false positives introduced by analysis at the library level. From the patch side, vulnerabilities introduced by TPLs can have unpredictable effects on the developers' projects. Once the vulnerabilities are detected, updating to a new version is the most straightforward way. However, it may cause dependency conflict issues [15]–[19] and compatibility issues [20]–[23], which will require substantial maintenance costs. Consequently, it is imperative to precisely determine whether the project is threatened by known vulnerabilities. In other words, if the vulnerability has a real negative impact on the project in practice, developers can generate a patch immediately to avoid an exploit of the vulnerability. If the vulnerability has no effect on the project, the handling of vulnerable TPLs is not urgent and can be incorporated into the regular development cycle. Thus, the real impact analysis of vulnerable TPLs at the method level is urgently needed no matter from the perspective of detection or patching [24].

As far as we know, Eclipse Steady [6], [25], [26] is the only open-source work that provides a forward reachability analysis at the fine-grained method level for users. However, according to our analysis, we conclude the following deficiencies in Steady: *(1) The inaccuracy of patch method extraction.* Steady considers the methods whose abstract syntax trees have been changed in patch commits as patch methods, however, patch-unrelated methods may exist in patch commits, leading to false positives. *(2) The incompleteness of vulnerable root method identification.* Steady obtains vulnerable root methods directly from patch commits, however, some vulnerable root methods may exist in the commits

• *Fangyuan Zhang and Miaoying Cai are with DISSec, NDST, College of Computer Science, Nankai University, China. Emails: {fangyuanzhang, miaoyingcai}@mail.nankai.edu.cn. Lingling Fan (Corresponding author) and Sihan Xu are with DISSec, NDST, College of Cyber Science, Nankai University, China. Emails: {linglingfan, xusihan}@nankai.edu.cn. Sen Chen is with the College of Intelligence and Computing, Tianjin University, China. Email: senchen@tju.edu.cn. Lida Zhao is with School of Computer Science and Engineering, Nanyang Technological University. Email: LIDA001@e.ntu.edu.sg.*

that are not recognized or marked as patch commits. The incomplete identification would cause false negatives of vulnerable paths. *(3) Low efficiency of vulnerable path analysis.* Steady conducts forward reachability analysis for each TPL with low efficiency due to complex dependency analysis.

Therefore, in this paper, we aim to address the aforementioned problems to evaluate the real impact of vulnerable TPLs on projects. However, we are facing the following challenges: *(1)* How to extract accurate patch methods from patch commits? As we all know, not all modified methods in a patch commit are patch methods. Therefore, we need to sift patch-unrelated methods out on the patch commit, to extract precise patch methods. *(2)* How to obtain comprehensive and precise vulnerable root methods from patch commits? Due to the incompleteness of patch commits provided [27], it is not comprehensive to only handle the patch commits. *(3)* How to accurately scan the vulnerable code of libraries in the projects with less resource overhead? To ensure fewer resources spent during scanning, we need a comprehensive set of detected vulnerable APIs of known vulnerable TPLs.

To fill the gap, we propose VAScanner (**V**ulnerable **A**PI **Scanner**), an effective vulnerable API detection approach, to assess the impact of OSS vulnerabilities in Java projects. We first collect public patch commits based on the vulnerability knowledge database and map the changed source code files involved in patch commits with class files in TPLs. We collect diff methods from patch commits as initial patch methods and then sift out patch-unrelated methods to extract accurate patch methods. We propose an augmentation mechanism to identify vulnerable root methods based on these patch methods. Then we perform backward call-graph analysis on vulnerable root methods and construct a vulnerable API database mapping with the relation among the vulnerable library versions, CVEs, and vulnerable APIs, which includes 90,749 unique vulnerable APIs (2,410,779 with library versions) from 362 TPLs with 14,775 vulnerable versions involving 502 CVEs. Based on the results, developers can figure out whether vulnerable libraries need to be patched at this time and prioritize the patches, thereby reducing additional patching efforts and maintenance costs.

To demonstrate the effectiveness of VAScanner, we conducted comprehensive experiments. We took an in-depth analysis of the patch-unrelated methods sifted out by the patch-unrelated sifting mechanism, vulnerable root methods introduced by the augmentation mechanism, and vulnerable APIs in the vulnerable API database. Moreover, we summarized 5 patterns of added patch methods, to analyze the fixed intention of introducing them. Based on statistical results, we sifted out 1,352 patch-unrelated methods with 98.06% precision and augmented 249 vulnerable root methods which were absent in patch commits with 93.57% precision. And the vulnerable API database constructed by VAScanner contains a total of 90,749 unique vulnerable APIs with a false positive proportion of 1.45% and a 95% CI of [1.31%, 1.59%]. Furthermore, to demonstrate the effectiveness of our novel mechanisms, we conducted an ablation study on VAScanner and VAScanner- with different mechanisms, and the result shows VAScanner eliminates 5.78% false positives and 2.16% false negatives. Subsequently, we compared VAScanner with the state-of-the-art tool, Eclipse Steady. The experimental results have shown that VAScanner outperforms Steady in analyzing direct dependencies, achieving more comprehensive method-level detection (#Cases: 214 vs. 95). Specifically, Steady (Avg time: 769s) exists 61.71% false negatives, while VAScanner (Avg time: 353s) yielded 2.97% false positives and 20.45% false negatives. Besides, our large-scale analysis on 3,147 real-world projects shows that only 21.51% of projects (with 1.83% false positive proportion and a 95% CI of [0.71%, 4.61%]) were potentially threatened by vulnerable APIs of TPLs, indicating the effectiveness of VAScanner.

In summary, we make the following contributions:

- We proposed VAScanner, an effective and efficient tool that can detect vulnerable APIs from TPLs used by Java projects, reducing false positives of vulnerabilities.
- We proposed two mechanisms to achieve accurate and complete vulnerable API identification for vulnerable libraries, i.e., a sifting mechanism to sift out patch-unrelated methods and an augmentation mechanism to augment the vulnerable root methods, which eliminates 5.78% false positives and 2.16% false negatives.
- We constructed a reusable database including 90,749 vulnerable APIs (2,410,779 with library versions) with 1.45% false positive proportion with a 95% CI of [1.31%, 1.59%] based on the identification results of VAScanner, which assists in achieving more efficient vulnerability detection than forward reachability analysis.
- We compared VAScanner with the state-of-the-art tool, Eclipse Steady. The experimental result demonstrates that VAScanner achieves more effective method-level identification in analyzing direct dependencies.

2 Background & Concepts

2.1 Background

The Maven Ecosystem. The Maven ecosystem [28] plays a crucial role in the Java landscape. It contains nearly 2,000 repositories and over 37 million packages. Each maven package is distinctly identified by the combination of GroupId, ArtifactId, and Version (GAV). Maven provides a simple and consistent approach by utilizing the configuration file (pom.xml) to effectively manage project dependencies, streamline the build process, and facilitate release development. Furthermore, since a maven package can be utilized as a TPL by other projects, it can be considered a project as well as a Java TPL.

Vulnerable Libraries and the Associated Risks. Vulnerable libraries are TPLs that contain vulnerabilities. Using vulnerable libraries introduces potential security risks to the projects. For instance, the Log4Shell vulnerability [29] existed in Apache log4j, which is a widely used Java-based logging library, affecting numerous projects.

Software Composition Analysis. Software Composition Analysis (SCA) [30] involves analyzing the libraries and identifying their vulnerabilities. Vulnerable library identification is a subset of SCA, which typically relies on hash comparisons or configuration files (e.g., pom.xml) to identify TPLs, and detect vulnerable libraries based on vulnerability databases (e.g., NVD [31]). Vulnerability reachability analysis focuses on determining whether there is a path from the software to the vulnerable code in TPLs. This analysis often

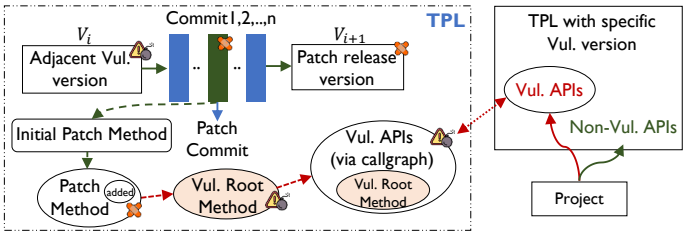


Fig. 1: Illustration for terms used in the paper

uses forward call analysis to ascertain whether the software can access the vulnerable code within the libraries.

2.2 Key Term Definition

We introduce some key concepts or terms used in the paper to make it easy to understand, as illustrated in Figure 1.

Adjacent Vul. version vs. Patch release version. When an open-source TPL is affected by a vulnerability (also known as CVE), the vulnerability knowledge base usually gives the vulnerable version range of the TPL. "Patch release version" means that it is the first release version to fix this vulnerability, i.e., V_{i+1}. "Adjacent vulnerable version" is the vulnerable version adjacent to the patch release version, i.e., V_i. Patch commits used by developers to fix this vulnerability exist between these two versions.

Initial Patch Method. Initial patch methods are the methods that have undergone code changes (i.e., added, deleted, or modified) in the patch commits.

Patch Method. Patch methods are methods that may be relevant to addressing vulnerabilities. Since not all initial patch methods play a role in patching, it is necessary to sift out patch-unrelated methods (Section 3.1.2) from the initial patch methods to generate precise patch methods. If a patch method is present only in the patch release version (i.e., V_{i+1}) and not in the adjacent vulnerable version (i.e., V_i), we consider it as an **added patch method**.

Vul. Root Method. Vulnerable root methods are those methods that are directly related to the vulnerability. Most of them are extracted from patch commits of vulnerabilities directly.

Vul. APIs. Vulnerable APIs are the methods that are directly or indirectly threatened by the vulnerability in the vulnerable TPL, including the vulnerable root methods and the methods that directly/indirectly invoke vulnerable root methods. For projects, APIs in TPLs are divided into 2 categories: vulnerable APIs and non-vulnerable APIs.

2.3 Problem Definition

As shown in Figure 1, our goal is to identify all vulnerable APIs for each vulnerable TPL version based on patch commits of CVEs and vulnerable root method identification and construct a database that maintains the mapping relation: *vulnerable library versions* $\leftrightarrow$ *CVEs* $\leftrightarrow$ *vulnerable APIs* (*libV-CVE-Vul.API*), based on which we aim to detect whether the project invokes vulnerable APIs of TPLs, to assess the real impact of OSS vulnerabilities on projects.

3 Approach

In this paper, we propose VAScanner to detect whether the projects are threatened by the vulnerable APIs in TPLs. Figure 2 shows the overview of our approach, consisting of 4 components: (1) Patch method extraction, which collects initial patch methods from patch commits and sifts out patch-unrelated methods to extract accurate patch methods. (2) Vulnerable root method identification, which identifies vulnerable root methods through locating the patch methods at the version level and employing an augmentation mechanism based on the extracted patch methods. (3) Vulnerable API identification, which utilizes call-graph analysis to identify vulnerable APIs for each library version, and constructs a database storing the mapping relations of vulnerable library versions (LibV), CVEs, and vulnerable APIs (Vul.API), presented as *libV-CVE-Vul.API*. (4) Used vulnerable API detection, which detects the vulnerable APIs in the libraries used by a given project.

3.1 Patch Method Extraction

This section describes the steps to extract the accurate *patch methods*. Specifically, we first collect methods that have undergone code changes in the patch commits (i.e., *initial patch methods*), and then sift out patch-unrelated methods.

3.1.1 Initial patch method collection

To collect the methods related to patching vulnerabilities, we first need to obtain patch commits of each CVE. Specifically, we collected vulnerabilities (identified by CVE ID) and their associated patch commits from Snyk Vulnerability DataBase [32] and GitHub Advisory Database [33]. We chose them as the vulnerability data collection sources for two reasons: (1) They maintain detailed information about CVEs and the corresponding patches, such as CVE ID, the vulnerable version ranges of TPLs, and patch-related links, which cover the CVE-related references provided by NVD. Besides, for most fixed CVEs, the two databases provide patch commit references on GitHub [34], which facilitates the collection and analysis of patch commits. (2) They map CVEs to vulnerable libraries, allowing us to identify libraries with vulnerable versions based on CVE IDs. Based on the two databases, we collected 2,640 CVEs and 1,551 affected libraries belonging to the Maven ecosystem. We filtered out CVEs without patch commits in patch-related links and those where the affected libraries did not have patch release versions. Finally, we gathered 1,116 CVEs and 957 affected libraries to collect initial patch methods.

For each patch commit, we extracted code differences by using the abstract syntax tree (AST), as it can accurately identify real code changes and filter out irrelevant modifications like adding or deleting identical code, changing the position of methods, or adding blank lines. This approach is more effective and accurate than traditional code-based change extraction. Specifically, we employed GumTree [35], a tool for generating code differences in AST, to obtain valid changed methods in patch commits. We first obtained the Java source code files before and after the commits based on the GitHub repository and used GumTree to generate the mappings between two ASTs. The identified code changes are divided into three types, i.e., insert, delete, and update.

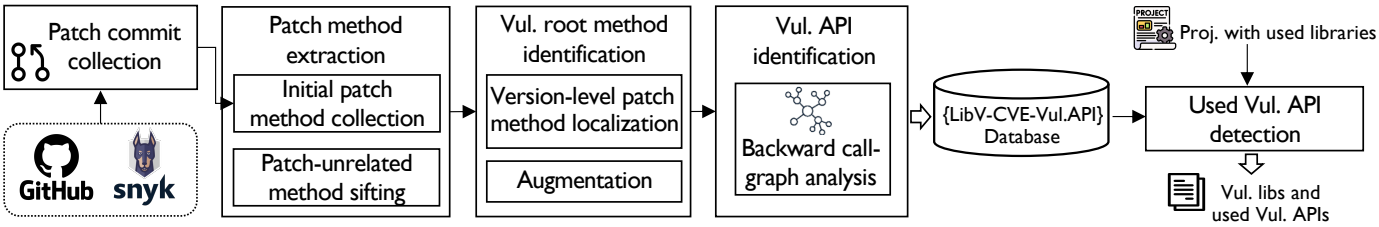


Fig. 2: Overview of VAScanner

According to the tree structure representing methods in the AST, we got the signature of methods where different nodes were located. Finally, we obtained methods with valid code changes in the patch commits (i.e., *initial patch methods*) with different change types (inserted, deleted, and modified). After filtering out CVEs whose patch commits involve languages other than Java (e.g., JavaScript), we consequently obtained the initial patch methods for 1,075 CVEs, 453 unique affected libraries and 1,350 patch commits.

3.1.2 Patch-unrelated method sifting

Since not all initial patch methods are related to vulnerability fixing, we aim to sift out patch-unrelated methods from initial patch methods, to obtain *patch methods*. To achieve this, we initially extracted the changed (i.e., inserted, deleted, or updated) statements within each initial patch method. We then assessed whether these changed statements were unrelated to the patch. If all the changed statements within an initial patch method are patch-unrelated, the method will be sifted out, otherwise, it is recognized as a patch method.

To achieve precise sifting of patch-unrelated methods, we adopted a conservative strategy for identifying irrelevant statements. Specifically, we summarized three patterns of patch-unrelated statements: (1) **Debugging code statements**, such as `System.out.println(..)`, log-related function calls (e.g., `log.warn(..)`), and error handling statements which only changed the exception messages, i.e., `throws new xxException(..)`; (2) **AST-equivalent statements after name normalization**. In detail, we initially collected the functions, class member variables, and formal parameters of functions that were solely renamed to generate a renaming set. We defined various renaming scenarios: when the function name changed but the function body remained unchanged, when a member variable merely altered its name but retained the same type and initialization, or when a formal parameter of a function only modified its name while maintaining the same type. In such instances, we categorized these functions, member variables, and formal parameters as being renamed. If a statement only includes modifications to the names of called functions, parameters of called functions, or the object of calling functions, we check whether the modified name exists in the renaming set. If it does, this statement is considered an AST-equivalent statement before and after the patch commit. Besides, if only the name of the assigned variable has been modified in an assignment statement (e.g., `A a = foo()`), the statement will also be regarded as AST-equivalent; (3) **Statements that solely compose the Getter/Setter functions**, such as `this.X = x`, `return X`, `return this` and `return this.X` (`X` is a class member variable). Note that we do not assert that the Getter/Setter functions are inherently patch-unrelated. Instead, our goal is to identify and sift out Getter/Setter functions that solely consist of those specific statements.

3.2 Vulnerable Root Method Identification

The patch methods are extracted based on patch commits, however, the patch release version or the adjacent vulnerable version of libraries (shown in Figure 1) may not contain the methods that were patched. Therefore, in this section, we aim to identify ***vulnerable root methods*** (denoted by $VulRoot$) by locating the patch methods at the version level instead of the commit level and augmenting them to obtain comprehensive vulnerable root methods.

3.2.1 Version-level patch method localization

Since a commit only records a timestamped change to the current code in the repository, the changed methods in a single patch commit may not appear in the release versions of the library. For example, a library has several release versions V_1, V_2, V_3, V_4..., V_n, where n is the number of versions, V_2 and V_3 are the vulnerable versions. There may be multiple commits between V_3 and V_4 aiming to patch the vulnerability in V_3, however, the changed methods in one commit might not be maintained in V_4 or exist in V_3, and should not be identified as a valid patch. Therefore, we need to locate the patch methods at the version level to ensure they exist in the release versions.

Specifically, We gathered all library versions from the Maven repository [28] and extracted patch releases and adjacent vulnerable versions based on vulnerable version ranges. If the patch release version or adjacent vulnerable version is not available in the repository, we filtered it out together with the associated CVEs from our database. Then we extracted the diff methods from pairwise class files between the adjacent vulnerable version and the patch release version and checked whether the methods that were patched exist in these diff methods. To obtain more accurate vulnerable root methods, we employ the following strategies to discard or retain patch methods for further augmentation: (1) Patch methods that exist in neither version (i.e., the patch release version and the adjacent vulnerable version) will be discarded; (2) Patch methods that exist in both versions are directly considered as vulnerable root methods. (3) Patch methods that only exist in the patch release version are newly added patch methods for the adjacent vulnerable version and will be retained for augmentation. During the process of patch method localization in library release versions, we observed the absence of all patch methods for

Fig. 3: Motivation for Augmentation

some CVEs, thus, we excluded these CVEs and obtained **362** libraries with **14,775** versions involved in **502** CVEs.

3.2.2 Augmentation mechanism

It is common to add a new class or method during vulnerability fixing. Then, the added patch methods are typically called by others, aiming to fix the vulnerability in those methods. Existing work (e.g., Eclipse Steady [6], [25], [26]) has overlooked the impact of added patch methods when identifying vulnerable root methods. They argued that these added patch methods are secure and can be ignored. However, our observation reveals that ignoring added patch methods can lead to overlooking vulnerable methods that invoked added patch methods but were absent in patch commits. For example, in Figure 3, if the patch method m_{patch} only exists in the patch release version but not in the adjacent vulnerable version, we regarded it as an added patch method. If a method m invoked the added patch method m_{patch} in the patch release version but did not invoke this patch in the adjacent vulnerable version, the method m from the adjacent vulnerable version is still considered vulnerable, even if it did not appear in the patch commit. Therefore, such methods should also be augmented as vulnerable root methods.

Listing 1: Patch Commit of CVE-2011-2730 [36]

```
boolean isJspExpressionActive(PageContext p) {
  ...
  if (sc.getMajorVersion() >= 3) {
-   if (sc.MajorVersion() > 2 || sc.MinorVersion() > 3) {
-    /* Application declares Servlet 2.4+:JSP 2.0 active.
-     * Skip our own expression support.*/
-     return false;
+   if (sc.MajorVersion() == 2 && sc.MinorVersion() < 4) {
+   /* Application declares Servlet 2.3-:JSP 2.0 not active.
+    * Activate our own expression support.*/
+     return true;
  }}
- return true;
+ return false;
}
```

Listing 2: Method Diff between 3.0.5 and 3.0.6 in org.springframework:spring-web

```
Object evaluate(Parameters) throws JspException {
  return isExpressionLanguage(attrValue)
+   && isJspExpressionActive(pageContext)
    ? doEvaluate(): attrValue;
}
```

Considering the situation that methods that invoked the added patch method in the patch release version may be due to the introduction of new functionalities rather than fixing the vulnerability; therefore, our augmentation mechanism is based on the following constraint: A method is considered a $VulRoot$ due to augmentation only if the method invoked the added patch method in the patch release version but not in the adjacent vulnerable version. In other words, there are no other changes in the augmented $VulRoot$ except for the call relationship to the added patch methods.

Algorithm 1: Vulnerable Root Method Augmentation

Input: m_0: an added patch method. P_{cg}: the call graph of patch release version, V_{cg}: the call graph of the adjacent vulnerable version.
Output: R: Vulnerable root methods based on m_0.

```
1  Visit ← ∅
2  Q ← Queue()
3  Q.push(m_0)
4  Visit ← Visit ∪ {m_0}
5  while Q ≠ ∅ do
6      m ← Q.pop()
7      S_m ← getCaller(m, P_cg) // Get direct callers of m.
8      if S_m = ∅ then
9          continue
10     foreach c ∈ S_m do
11         if isInGraph(c, V_cg) then
12             R ← R∪{c} // Incorporate it into the results.
13         else
14             if c ∉ Visit then
15                 Q.push(c)
16                 Visit ← Visit ∪ {c}
17     if R ≠ ∅ then
18         return R
```

For a real case, the TPL "org.springframework:spring-web" is affected by the CVE-2011-2730 [37], causing multiple versions (the versions before 2.5.6.SEC03, and 3.0.0∼3.0.6) to be vulnerable. CVE-2011-2730 is caused by evaluating Expression Language (EL) expressions in tags twice, which allows remote attackers to obtain sensitive information. As shown in Listing 1, the developers only activate their expression support when the application declares Servlet 2.3- (Lines 8-11) and set "springJspExpressionSupport" to false by default (Line 14), avoiding the potential double EL evaluation problem on pre-Servlet-3.0 containers, which indicates that this method acts as a bug fix. Although this patch method is shown as modified in the patch commit, however, we found that it only existed in patch versions (2.5.6.SEC03 and 3.0.6). Therefore, the method "`isJspExpressionActive()`" is an added patch method for vulnerable versions.

To further confirm the impact of the added patch method on fixing the vulnerability, we checked its call relationships in the patch release version (V3.0.6). We found that five methods directly called this added method and all of them existed in the adjacent vulnerable version (V3.0.5). For example, in Listing 2, the method "`evaluate()`" called the added patch method "`isJspExpressionActive()`" (Line 3) in V3.0.6 to fix CVE-2011-2730, and it still existed in V3.0.5 without invoking the added patch method. Therefore, this method located in V3.0.5 is vulnerable and should be augmented to the list of vulnerable root methods. Unfortunately, all of the patch commits did not record such call relationship, thus existing work only based on patch commits cannot identify the in-depth vulnerable root methods, while VAScanner augments the vulnerable root methods with such vulnerable methods via multi-version analysis.

Algorithm 1 details the augmentation procedure. Given an added patch method m_0, the call graph of the adjacent vulnerable version and patch release version (V_{cg} and P_{cg} respectively), VAScanner outputs the augmented vulnerable root methods R based on m_0. In detail, we leverage the function call relationship of the added patch methods in the patch release version, to mine the methods in the call chain that exist in the adjacent vulnerable version (Lines 5-18). In particular, for each added patch method m_0, if it is invoked by other methods in the patch release version, we will check whether these callers exist in the adjacent vulnerable version (Line 11). If exists, the caller will be augmented into the set of vulnerable root methods (Line 12), otherwise, it will be added into the queue for further mining vulnerable root methods (Lines 13-15). Note that, once we obtain the results of vulnerable root methods, we will exit the while loop directly (Lines 17-18), to avoid increasing the negative impact of the possible errors of the added patch methods. After the above process, the set of augmented vulnerable root methods is constructed.

3.3 Vulnerable API Identification

Based on the final vulnerable root methods identified in Section 3.2, in this section, we aim to mine the ***vulnerable APIs*** via call graph, which is defined in Section 2.2. We mine all the vulnerable APIs because if a project invokes an API of a library that eventually reaches or calls the vulnerable root method, then this API should also be regarded as vulnerable. In fact, according to our observation, the vulnerable root methods are hardly invoked by projects directly. Therefore, we also mine and maintain all the vulnerable APIs for each vulnerable library version for further analysis.

Specifically, for each vulnerable library, we mined for the vulnerable APIs affected by the vulnerable root methods based on backward call graph analysis. Firstly, we generated the call graph of the library by employing context-insensitive points-to analysis provided by the static framework Tai-e [38] and considered all the methods as the entry points to obtain a complete call graph. Subsequently, starting from the vulnerable root methods, we traversed their called traces in the call graph and recorded all the methods executed in the traces. In such a manner, we obtained all the vulnerable APIs for each vulnerable library version.

Database construction. Based on the identified vulnerable APIs, we constructed a vulnerable API database with the mapping relation: library version to CVEs to vulnerable APIs, denoted by *libV-CVE-Vul.API*. Specifically, we crawled all the vulnerability data and patch commits corresponding to the vulnerability from Snyk Vulnerability DB and GitHub Advisory (as of Feb. 2023) and downloaded the vulnerable libraries from Maven [28] to support our database. Since some versions are not available from Maven or some patch class files do not exist in the libraries, we filtered them out. We employ the approach above for each CVE in the vulnerable library, to obtain a set of vulnerable APIs and construct the vulnerable API database. Table 1 provides detailed information about the database. The column "#Vul. API (excl. root)" represents the number of vulnerable APIs obtained from the backward analysis of call graphs. "#Vul. root method" represents the number of vulnerable root methods, including ones directly obtained from patch commits ("#Commit") and the augmented ones ("#Augm.") mined by VAScanner. We used two counting methods for vulnerable APIs across different library versions: single counting ('API-once') and multiple counting ('API-multi.'). Identical APIs were determined by normalizing their function bodies and comparing hash values. The database contains 90,749 unique vulnerable APIs (2,410,779 across library versions) from 362 unique libraries with 14,775 library versions, involved in 502 CVEs. On average, our augmentation mechanism has supplemented 5.9 augmented vulnerable root methods per library and 2.5 per library version, related to 49 CVEs.

TABLE 1: Statistics of vulnerable APIs in libraries identified by VAScanner. (LibV.: library versions)

-		#Total	#Vul. API (excl. root)	#Vul. root method	
				#Commit	#Augm.
API	once	90,749	87,417	3,732	**249**
	mult.	2,410,779	2,348,684	58,736	**3,359**
Lib (LibV.)		362 (14,775)	304 (11,619)	358 (14,620)	**42 (1,365)**
CVE		502	405	493	**49**

*Note: **excl. root** - Vulnerable APIs that exclude vulnerable root methods; **once** - The same vulnerable API is counted once across versions; **mult.** - The same vulnerable API is counted multiple times across versions.*

3.4 Used Vulnerable API Detection

In this section, we describe how to detect whether the vulnerable APIs from TPLs are used in projects. For a given Java project with its used libraries, we generate its call graph by employing the context-insensitive points-to analysis of Tai-e [38], which is the bedrock to determine whether it invokes vulnerable APIs. If it depends on a library version in the vulnerable API database, we search out the used vulnerable APIs from the database for this library. Specifically, for each method in the call graph of the project, we analyze whether it invokes the vulnerable APIs in the library, if true, VAScanner marks the vulnerable APIs used by developers. Besides, it also reports the vulnerable dependency, the used vulnerable APIs in the library, the call frequency of vulnerable APIs, and the involved CVEs. Suppose all the methods in the project do not call the vulnerable APIs, in that case, the project uses the vulnerable library without using the vulnerable code, which should not be regarded as vulnerable usage.

4 Evaluation

In this section, we evaluated VAScanner on real-world projects to answer the following research questions:

RQ1: Can VAScanner effectively identify vulnerable root methods and vulnerable APIs?

RQ2: Can VAScanner outperform state-of-the-art tools in detecting vulnerable projects threatened by vulnerable third-party libraries?

RQ3: How do the sifting and augmentation mechanisms contribute to vulnerable API detection for VAScanner?

RQ4: How is the status quo of vulnerable libraries used in open-source projects?

4.1 RQ1: Effectiveness Evaluation

4.1.1 Setup

Given that vulnerable APIs are derived from the backward call graph analysis, it can be reasonably assumed that APIs directly or indirectly calling the vulnerable root methods may also contain vulnerabilities. As a result, the accuracy of vulnerable APIs depends on the accuracy of vulnerable root methods. This experiment aims to investigate the effectiveness of VAScanner in identifying vulnerable root methods and vulnerable APIs, and take an in-depth analysis of root causes. Specifically, this experiment is based on our database containing 362 vulnerable TPLs (14,775 library versions), involving 502 CVEs. Due to the lack of ground truth for vulnerable root methods correlated with CVEs, we manually analyze the sifted patch-unrelated methods and augmented vulnerable root methods to assess the effectiveness of sifting and augmentation mechanisms. Furthermore, we additionally provide the ground truth for vulnerable APIs to validate the vulnerable API database and also perform an error analysis to estimate the effectiveness of the vulnerable API database provided by VAScanner.

Listing 3: Patch patterns with examples

```
// Pattern 1: Checker
+ boolean checkPathSecurity(String path){
+   contain_ = path.contains("../");
+   end_ = path.endsWith(".log")
+   if (!StringUtils.isBlank(path)) {
+     if ( start_ && !contain_ && end_ ) {
+       return true; }}
+   return false; }
// Pattern 2: Filter
+ String filterSensitive(String url){
+   String resultUrl = url;
+   if (containsIgnoreCase(url, _SENSITIVE)) {
+     resultUrl = replaceIgnoreCase(url,_SENSITIVE,_FALSE);}
+   return resultUrl; }
// Pattern 3: Configuration
+ boolean isSupportActive(PageContext pc) {
+   ServletContext sc = pc.getServletContext();
+   String EXP_SUPPORT_CONTXT = "springJspExpressionSupport"
+   String Support = sc.getInitParam(EXP_SUPPORT_CONTXT);
+   if (Support != null) {
+     return Boolean.valueOf(Support);}
+   if (sc.getVersion() >= 3) {
+     Int maj_v = sc.getEffectiveMajorVersion()
+     Int min_v = sc.getEffectiveMinorVersion()
+     if (maj_v==2 && min_v<4) {
+       return true;}}
+   return false;}
// Pattern 4: Enhancer
+ String randomString(int byteLength) {
+   byte[] bytes = new byte[byteLength];
+   SECURE_RANDOM.nextBytes(bytes);
+   CharSet sc = StandardCharsets.ISO_8859_1;
+   return new String(bytes, sc);}
// Pattern 5: Assistance
+ ObjectMapper createVaadinConnectObjectMapper(
+   ApplicationContext c) {
+   ObjectMapper objMapper =
+     Jackson2ObjectMapperBuilder.json().build();
+   JacksonProperties jacksonProperties =
+     c.getBean(JacksonProperties.class);
+   if (jacksonProperties.getVisibility().isEmpty()) {
+     objMapper.setVisibility(PropertyAccessor.ALL,
+     JsonAutoDetect.Visibility.ANY);}
+   return objtMapper;}
```

4.1.2 Result

Table 1 shows that VAScanner can identify 90,749 unique vulnerable APIs (2,410,779 with library versions). Details are aforementioned in the dataset construction of Section 3.3. In the following, we aim to demonstrate the validity of the augmented vulnerable root methods, the sifted patch-unrelated methods and vulnerable APIs.

TABLE 2: Libraries, CVEs affected by unique added vulnerable root methods and projects invoking these libraries

-	#Augmented vulnerable root methods				
	0	1∼5	5∼10	10∼20	>=20
CVE	453	36	6	6	1
Lib (LibV.)	339 (13,925)	32 (1,566)	7 (6)	5 (30)	1 (0)

(1) Result of augmented vulnerable root methods. Table 2 shows the number of libraries (library versions) and CVEs affected by augmented vulnerable root methods. Columns 3-6 indicate that more vulnerable root methods are mined compared with those only extracted from patch commits. Since there is no single library version with more than 20 unique vulnerable root methods augmented, the value of "libV." is set to 0. We manually analyzed each method and summarized five patch patterns (Listing 3). These patterns highlight the scenarios in which developers address the vulnerabilities by introducing new patch methods.

P_1: Checker. To fix vulnerabilities reported in CVEs, developers sometimes add check mechanisms (e.g., add logic statements) to check the legitimacy of the input or improve the original check mechanism. For example, Listing 3 shows an added method "`checkPathSecurity(..)`" in CVE-2022-26884 [39] that checks whether the parameter "path" transferred conforms to security, e.g., whether it contains "../" which does not meet security requirements and may lead to security problems.

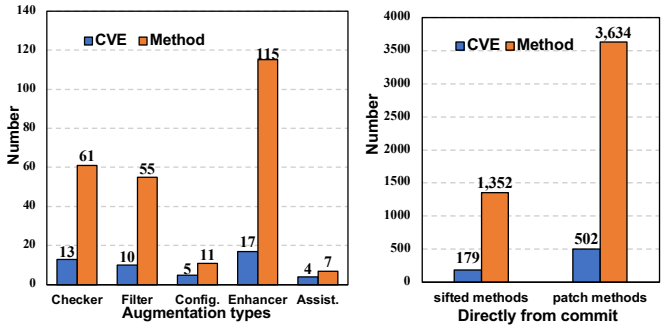
P_2: Filter. Some added methods aim to filter out unexpected input with specific conditions. In such a pattern, legitimate input will be retained, and illegitimate ones will be discarded. For example, in Listing 3, to fix CVE-2022-40955, developers added a new method "`filterSensitive(..)`" in the patch commit [40] to filter out invalid and sensitive cases and keep the url meeting security requirements.

P_3: Configuration. To avoid the vulnerabilities caused by the lack of default configuration or misuse of configuration, developers tend to standardize or improve existing configurations. As described in CVE-2011-2730, the springframework [41] suffered from Expression Language Injection. Developers addressed the potential Double EL Evaluation issue by defaulting the relevant parameter 'springJspExpressionSupport' to false in their patch commit [36].

P_4: Enhancer. Developers usually introduce a series of algorithms and operations to enhance existing programs for security, such as introducing more robust algorithms and safer authentications. Listing 3 shows an added method "`randomString()`" identified in the patch commit [42], which provides a randomly generated default value, enhancing the client-side session encryption secret after the update.

P_5: Assistance. Some added methods may not directly fix vulnerabilities, but their relevance can be assessed through correlation analysis of commit messages and methods. For example, the added method "`createVaadinConnectObjectMapper(..)`" in the patch commit [43], shown in Pattern 5 of Listing 3, creates a custom ObjectMapper to help address the vulnerability.

For the 5 types of added patch patterns, we further investigated the number of vulnerable root methods that



(a) Augmentation mechanism (b) Sifting mechanism

Fig. 4: Unique augmented methods, sifted patch-unrelated methods and corresponding CVEs

are augmented due to each type as well as the CVEs involved. Figure 4a shows the result. Among the 49 CVEs supplemented with vulnerable root methods, 13 CVEs and 17 CVEs are fixed by adding a **Checker** and **Enhancer** in patch commits, respectively, which shows that they are the common fix solutions. Moreover, we augment 249 unique methods into vulnerable root methods in total, and 115 methods (the most) are augmented by **Enhancer**. As for the analysis of augmented vulnerable root methods in patch commits, our validation strategy unfolds in two steps: first, we validate whether the added patch method associated with augmented root methods achieves the patching effect; second, we check whether the augmented root method was defective before invoking the added patch method. If the added patch method is patch-unrelated, or if the augmented root method was secure in the adjacent vulnerable version of the library, we determine that this augmented root method was an FP. We validate the above steps for 49 CVEs affected by the augmentation mechanism. Out of the 249 augmented vulnerable root methods, 16 functions (involving 6 CVEs) were confirmed as FPs, achieving **93.57%** precision.

(2) Result of sifted patch-unrelated methods. Figure 4b displays the number of involved CVEs and sifted patch-unrelated methods, including 1,352 sifted methods associated with 179 CVEs. Since the sifting mechanism involves a large number of methods, we conducted manual analysis on 50 randomly selected CVEs to evaluate the effectiveness (i.e., precision and recall) of the sifting mechanism. The sample set consisted of 807 initial patch methods, after manual analysis, 298 methods were identified as patch-unrelated and served as ground truth. Note that since we are evaluating the effectiveness of the patch-unrelated sifting mechanism, we consider correctly sifting out patch-unrelated methods as a true positive. Therefore, incorrectly sifting out the patch-related method is considered a false positive, and incorrectly identifying and retaining a patch-unrelated method is considered a false negative.

Overall, our sifting mechanism identifies 258 methods patch-unrelated, achieving an impressive precision of **98.06%**, with only 5 methods mistakenly considered as invalid patches. The reason is that developers move code snippets from one place to another (e.g., into an if clause), which changes the code semantics and causes false positives of VAScanner. As for the false negatives,

Algorithm 2: Construction of Ground Truth for Vulnerable APIs

Input: A_{db}: vulnerable API database, R_{sam}: sampled vulnerable root methods, R_{err}: the false positives of vulnerable root methods in R_{sam}.
Output: A_{sam}: sampled vulnerable APIs, A_{err}: the false positives of vulnerable APIs in A_{sam}.

```
A_sam ← ∅
A_err ← ∅
foreach vulAPI ∈ A_db do
    // Get associated vul. root methods of vulAPI.
    vulRoots ← getSourceRoots(vulAPI)
    if R_sam ∩ vulRoots == ∅ then
        continue
    A_sam ← A_sam ∪ {vulAPI}
    isErrAPIFlag ← True
    foreach vulRoot ∈ vulRoots do
        if vulRoot ∉ R_err then
            isErrAPIFlag ← False
            break
    if isErrAPIFlag then
        A_err ← A_err ∪ {vulAPI}
return A_sam, A_err
```

45 patch-unrelated methods were not recognized successfully, resulting in a recall rate of 84.90%. The reasons are as follows: (1) Certain methods have undergone intricate modifications, limiting the sifting mechanism. (2) Method changes before and after patch commits are semantically equivalent. As VAScanner employs ASTs to extract the changed code, it cannot recognize semantic equivalence. For example, in the patch commit [44], the function "`protocolViolation(ChannelHandlerContext, String)`" was split into two functions, with one calling the other. However, VAScanner fails to recognize it as patch-unrelated, leading to a false negative.

Furthermore, we employed Wilson's score confidence interval [45] to calculate the real false positive rate (FPR) and false negative rate (FNR) of the sifting mechanism, which requires solving for p in the following formula:

$$|p - \hat{p}| = z \cdot \sqrt{\hat{p} \cdot (1 - \hat{p}) / n} \tag{1}$$

where p is the real FPR or FNR, representing the probability of FPs or FNs in the overall population; $\hat{p}$ is the estimated FPR or FNR, representing the proportion of FPs or FNs calculated from the sample n; and $z = 1.96$ is the critical coefficient for a 95% confidence interval. Thus, the FPR of the sifting mechanism is 0.98% with a 95% confidence interval (CI) of [0.42%, 2.28%], and the FNR is 15.10% with a 95% CI of [11.48%, 19.61%].

(3) Result of vulnerable APIs. In light of the need to validate the experiments' results, including the effectiveness of vulnerable APIs (RQ1), the comparison experiment (RQ2), the ablation study (RQ3), and the large-scale analysis (RQ4), we have established a common ground truth for evaluating these experiments. Specifically, given the large number of CVEs associated with the vulnerable APIs in RQ1, the overlapping APIs detected in RQ3, and the detection results in RQ4, it is impractical to analyze each vulnerable API individually. Therefore, we chose to conduct a sampling

analysis on them and selected CVEs relevant to the detection results of RQ2 and the non-overlapping vulnerable APIs detected in RQ3. Finally, the *ground truth*'s data sources were from 58 CVEs. These 58 CVEs involved 26,720 unique vulnerable APIs, which were directly or transitively reached from 270 unique vulnerable root methods.

The vulnerable APIs are generated based on vulnerable root methods and function call relationships. Since we have utilized the advanced tool for generating call graphs as the foundation of our research, and validating the accuracy of these call graphs is beyond the scope of our research, we assume that the function call relationships are accurate and determine the validity of vulnerable APIs based on the validity of vulnerable root methods. Our validation strategy for these large number of vulnerable APIs in the *ground truth* is as follows: (1) we first manually analyze vulnerable root methods based on patch commits and vulnerability descriptions, following the method described in references [46], [47], to obtain the labels of the 270 vulnerable root methods. (2) As shown in Algorithm 2, we automatically extract the associated vulnerable root methods for each vulnerable API. If all of these vulnerable root methods are not vulnerable, we consider that vulnerable API to be a false positive. This process results in obtaining the labels for the 26,720 unique vulnerable APIs as *ground truth*. Ultimately, we identified 386 of 26,720 unique vulnerable APIs (including 25 of 270 unique vulnerable root methods) as false positives, and the vulnerable API database has a false positive proportion of 1.45% with a 95% CI of [1.31%, 1.59%].

Answer to RQ1: VAScanner can effectively augment vulnerable root methods which are absent in patch commits with 93.57% precision and sift out patch-unrelated methods with 98.06% precision. Eventually, we construct a database consisting of 90,749 vulnerable APIs (2.4M with library versions) with 1.45% false positive proportion with a 95% CI of [1.31%, 1.59%] from 362 TPLs.

4.2 RQ2: Comparison with existing work

In this section, we demonstrate the effectiveness of VAScanner by comparing it with the state-of-the-art tool, Eclipse Steady [6], [25], [26], which is the only open source tool providing a forward reachability analysis at the method level so far.

4.2.1 Dataset Collection

We collected Java projects from GitHub with different numbers of stars. In total, we crawled 13,708 real-world projects with stars ranging from 70,000 to 0, among which 6,416 can be successfully compiled (using "mvn compile"), while others failed to be compiled due to the use of private libraries or some unpassed plugins. We further filtered projects that did not depend on the vulnerable library versions in our database, and eventually obtained 3,147 real-world potentially vulnerable projects.

Steady manages its vulnerability data within Project KB, which includes CVE-related information, including vulnerability descriptions, affected libraries, affected library versions, patch library versions, patch links, and more. To ensure a fair comparison with Steady, we selected the CVEs that are both maintained by Steady and VAScanner as the comparison dataset, i.e., 213 CVEs in total. We obtained the vulnerable libraries versions affected by these CVEs on GitHub Advisory Database [33], 171 libraries with 6,153 library versions in total, and finally located 1,045 projects which depended on them.

TABLE 3: Comparison with Steady in terms of the detected cases, involved vulnerable projects, libraries, CVEs, and the time cost.

-	#Cases	#Projs	#Libs	#CVEs	Avg Time (s)
VAScanner	214	177	32	42	353
Steady	95	66	12	13	769
Overlapped	40	44	9	11	N.A.

4.2.2 Setup

Steady supports static analysis and dynamic-based analysis to analyze the vulnerable code reachability, while the dynamic-based methods require JUnit or application-specific tests, which are often unavailable or insufficient in public Maven projects. Therefore, we compare VAScanner with Steady in terms of static analysis. Steady takes a project as input and initially identifies TPLs directly or transitively dependent on the project using Project KB [48], and employs either Soot [49] or WALA [50] to facilitate static analysis. To eliminate the side-effect caused by different static analysis frameworks between VAScanner and Steady, we choose Soot as the call graph construction framework of Steady and set up the same configuration as we have done with using Soot. As for recording the detection time for Steady and VAScanner, since only the vulnerability reachability analysis part is focused on, we exclude the time spent on identifying vulnerable libraries and directly record the time spent on reachability analysis.

We run Steady and VAScanner on the aforementioned 1,045 projects, and compare the effectiveness of detecting vulnerable projects. One project identified as vulnerable means that there exists at least one execution path from the project to the vulnerable API of the vulnerable library.

4.2.3 Result

Table 3 shows the comparison results between VAScanner and Steady. The "Overlapped" row represents the results identified by both VAScanner and Steady. Considering the overall performance, both VAScanner and Steady can identify vulnerable projects in a finer-grained manner, sharply reducing the vulnerable projects from 1,045 to 177 and 66 respectively. Specifically, VAScanner identified more vulnerable cases than Steady (214 vs. 95), with VAScanner averaging 353s per project for detection, and Steady averaged 769s. Besides, 40 cases are both identified by two tools. To validate the precision of identified cases scanned by VAScanner and Steady, we used the *ground truth* for vulnerable APIs proposed in RQ1 to check whether the vulnerable APIs used by projects were false positives. If there is at least one vulnerable API that is confirmed to be the true positive, the detected case is considered a true positive. Moreover, if one tool identifies a case as a true positive, while another tool does not detect this case, then this case is considered a false negative for the latter. Consequently, we identified 166 (61.71%) FNs in the scanning results of Steady, while

VAScanner yielded 8 (2.97%) FPs and 55 (20.45%) FNs. We thus further take an in-depth analysis to investigate the reasons and insights, which are summarized as follows.

• **Identified by both tools (40 cases)**. For vulnerable projects identified by both tools, these detected cases are all true positives. Furthermore, we found that these projects all directly invoked vulnerable libraries, i.e., directly invoked the vulnerable APIs or other APIs of the library which finally reached the vulnerable root methods via call graph. Besides, the vulnerable root methods of these used libraries were all extracted from patch commits, and this is the simplest case that existing patch commit analysis focused on. Therefore, Steady and VAScanner both can identify them.

• **Only identified by Steady (55 cases)**. For projects that were only identified as vulnerable by Steady, some projects invoked vulnerable libraries indirectly. Since Steady started analysis from the project and further analyzed the direct- and transitive-invoked libraries to detect whether the project became vulnerable through the dependencies, it thus can identify such cases. While VAScanner focused on distilling the vulnerable APIs of each vulnerable library, i.e., only considering the vulnerable libraries directly depend on projects, it cannot identify whether the project can reach vulnerable APIs/code from such transitive dependency.

• **Only identified by VAScanner (174 cases)**. For the projects only identified by VAScanner, we found these projects invoked vulnerable library APIs that are not marked as vulnerable by Steady. There are four possible reasons: (1) Due to the missing information of vulnerable libraries affected by the same CVE, Steady exhibits false negatives in the identification of vulnerable libraries, where such vulnerable libraries are mistakenly classified as safe. For example, the library "dom4j-2.0.0" is suffered by CVE-2020-10683, but Steady fails to identify it as a vulnerable TPL. (2) Steady identified vulnerabilities based on all the modified and deleted methods in the patch commits. However, if the patch commit added a patch method that is not directly invoked in any other patch commits but is later invoked by methods in the vulnerable library version in another commit, Steady may not recognize it as vulnerable. In contrast, VAScanner can mark it as vulnerable owing to its augmentation mechanism. For example, the project "jbufu/openid4java" directly depends on the library "xercesImpl-2.8.1" which is affected by CVE-2012-0881. VAScanner reported that it invoked the vulnerable API from "xercesImpl-2.8.1", however, Steady showed that it did not reach the vulnerable code related to CVE-2012-0881. After our investigation, we found it indirectly invoked the vulnerable root method augmented by VAScanner. Since Steady only extracts the diff methods from patch commits as vulnerable methods, it cannot cope with such a situation, resulting in false negatives. (3) When the libraries contain both vulnerable structures and patch structures, Steady is uncertain about whether they include vulnerable code, resulting in missing some identified results. Steady stored the AST associated with vulnerability to determine whether the current library version contains vulnerable code. Due to some internal errors, the version that is vulnerable is not recognized by Steady. (4) The depth of call analysis in forward vulnerability reachability analysis is shallow compared to backward call graph analysis. Forward reachability analysis traces paths from external code to the vulnerability point, emphasizing breadth, while backward call graph analysis starts from the vulnerability point and traces its calling paths outward, focusing more on depth. Consequently, forward reachability analysis lacks the comprehensiveness of backward analysis, as achieving the same depth would require a significant resource investment.

As for 8 false positives generated by VAScanner, they involved 4 CVEs and 4 libraries. The misidentification of these cases stems from the fact that the root methods associated with reported APIs are unrelated to the vulnerabilities. Since the patch involves the addition of member variables with the result of necessitating complex modifications in the initial methods, VAScanner erroneously determined these root methods were vulnerable before patching.

TABLE 4: Ablation study results on different mechanisms. (✓: Enabled; ✗: Disabled; prop.: proportion)

Mechanisms		#Vul. APIs	FP prop.(%)	FN prop.(%)
Sifting	Augm.			
✗	✗	1,229	11.89% ∼ 16.52%	2.16%
✓	✗	1,158	6.11% ∼ 10.74%	2.16%
✓	✓	1,183	6.11% ∼ 10.74%	0

Answer to RQ2: VAScanner can enhance the current tool chains by detecting security threats more effectively through deep call chains at the price of potentially missing some cases due to transitive dependencies.

4.3 RQ3: Ablation Study on different mechanisms

To showcase the contribution of the proposed sifting and augmentation mechanisms, we set up an ablation study on them. Specifically, we execute VAScanner and VAScanner- with different mechanisms enabled on the same projects respectively, shown in Table 4. The contribution of the augmentation mechanism is not separately studied because it is based on the sifting mechanism. We then compare the results of the individual scans against each other.

4.3.1 Dataset Collection

Our proposed sifting and augmentation mechanisms affected 179 and 49 CVEs, respectively, involving 183 libraries with 6,529 library versions. To evaluate the impact of these two mechanisms, we selected 1,191 projects that are dependent on the 183 libraries from the 3,147 potential vulnerable projects mentioned in Section 4.2, which enables us to assess the contribution of these mechanisms.

4.3.2 Result

After scanning these potentially vulnerable projects, VAScanner identified 284 projects that utilized vulnerable APIs. However, VAScanner- without any mechanisms, and with only the sifting mechanism, detected 293 and 272 projects calling vulnerable APIs, respectively. Table 4 shows the vulnerable API detection result of the ablation study. The "#Vul. APIs" column displays the number of detected vulnerable APIs. We assessed the accuracy of the detected APIs by analyzing the precision of the corresponding vulnerable

root methods. VAScanner decreased 71 (5.78%) false positives by employing the sifting mechanism, and 25 (2.16%) false negatives by utilizing the augmentation mechanism. Besides, VAScanner and VAScanner- (both without any and augmentation mechanism) identified 1,158 overlapping APIs, with an 8.13% false positive proportion with a 95% CI of [6.11%, 10.74%]. Next, we first provide detailed explanations for how VAScanner achieves the reduction in FPs and FNs through these two mechanisms, and subsequently validate the overlapping detected APIs.

• **FP reduction analysis.** Since VAScanner- without any mechanisms identified the diff methods before and after the patch commits as patch methods directly, its vulnerable API database may include many non-vulnerable APIs. However, our proposed sifting mechanism can sift out patch-unrelated methods with high precision, reducing the generation of some non-vulnerable APIs for both VAScanner and VAScanner- with the sifting mechanism. Therefore, the sifting mechanism can eliminate 71 (5.78%) FPs detected by VAScanner- without any mechanisms. For example, the project "gavincook/githubOfflineInstaller" depended on the TPL "dom4j-2.0.0-RC1" influenced by CVE-2020-10683. VAScanner- without any mechanisms shows that it invoked 3 vulnerable APIs which indirectly called the root method "`SAXReader:configureReader(XMLReader,DefaultHandler)`". However, through meticulous manual verification, we found that it was not vulnerable in the patch commit [51], causing FPs of VAScanner- (without any mechanisms).

• **FN reduction analysis.** Table 4 reveals that VAScanner detected 25 additional vulnerable APIs compared to VAScanner- without augmentation mechanism, indicating that augmentation mechanism can eliminate 25 (2.16%) FNs. The augmentation mechanism enables VAScanner to generate more accurate vulnerable APIs. For example, the project "fabric8io/shootout-docker-maven" utilized the TPL "tomcat-embed-core-7.0.91" affected by CVE-2021-30640. In the patch commit [52], developers introduced a patch method named "`JNDIRealm:doAttributeValueEscaping(String)`" to implement the necessary escaping. Through our augmentation mechanism, two methods invoking this newly added patch method in the patch release version (V7.0.109) were absent in any other patch commits. This absence resulted in VAScanner- failing to identify vulnerable APIs related to these augmented vulnerable root methods, leading to FNs in scanning projects.

• **Validation for overlapping APIs.** We used the *ground truth* to validate the overlapped vulnerable APIs detected by both tools. Among the 58 CVEs in *ground truth*, 36 were involved in the ablation study, covering 541 vulnerable APIs out of 1,158 overlapping APIs. We found that 44 out of 541 APIs from *ground truth* were false positives, and then performed an error analysis using Wilson's score confidence interval [45] to estimate the false positive proportion. Thus, the detected overlapping APIs have an 8.13% false positive proportion, with a 95% CI ranging from 6.11% to 10.74%.

Answer to RQ3: VAScanner effectively reduces FPs by 5.78% through sifting mechanism and FNs by 2.16% through augmentation mechanism, leading to more accurate and comprehensive vulnerable API detection.

TABLE 5: Overall status of the real-world projects invoking vulnerable libraries and potentially vulnerable APIs.

	Not calling vul libs	Calling libs but not vul APIs	Calling libs and vul APIs			
			1 lib	2 libs	3 libs	4+ libs
#Proj	1,753	717	596	69	11	1
#CVEs	N.A.	N.A.	73	47	22	6

4.4 RQ4: Large-scale Analysis

Based on the 3,147 projects mentioned in Section 4.2, we further conducted a large-scale study by leveraging VAScanner, to reveal the fact of using potentially vulnerable APIs from the vulnerable libraries in real-world projects.

4.4.1 Impact analysis of potentially vulnerable APIs

Based on the collected dataset, we aim to investigate the impact of potentially vulnerable APIs on real-world projects. The results are shown in Table 5. We found that 1,753 projects did not use any of the modules in the vulnerable libraries in our database, 717 projects only used the non-vulnerable modules in the vulnerable libraries, and 677 projects were potentially affected by vulnerable libraries. Moreover, we used the *ground truth* for vulnerable APIs proposed by RQ1, to validate the scanning results for conducting a sampling analysis. These CVEs involve 35 libraries, 134 library versions, and 219 projects using vulnerable modules. Among these 219 projects, TP=215 and FP=4. Furthermore, we conducted an error analysis using Wilson's score confidence interval and found that approximately 21.51% of all projects have utilized potentially vulnerable modules in the vulnerable libraries. The false positive proportion is 1.83% with a 95% CI of [0.71%, 4.61%]. This means that for most projects, even if calling the vulnerable TPL, they are still not affected by the vulnerable library. For example, the project "elibom/jogger" directly relies on two vulnerable dependencies: jetty-server-8.1.15 and httpclient-4.5.2, and it invoked 9 APIs from jetty-server-8.1.15, but none of these APIs were deemed vulnerable. Thus, it can suspend the processing of these three vulnerable libraries. Our analysis indicates that vulnerable TPLs may not have a substantial impact on most projects. We explore the reasons from the following points: (1) For the vulnerability itself, the vast majority of vulnerabilities threaten only one or specific modules of the software. We attempt to maximize the impact range of vulnerabilities in the TPL through backward call graph analysis, to ensure that all the modules potentially affected by vulnerabilities are identified. (2) For the project itself, it often uses only specific modules from a TPL, not the entire library, meaning it may not invoke potentially vulnerable APIs and thus avoid certain vulnerabilities. In large projects that rely on multiple vulnerable libraries, it is crucial to identify if any vulnerable modules are used. This can help developers plan patches and prioritize vulnerability mitigation.

4.4.2 Top vulnerable libraries and vulnerable APIs

We further investigate the most frequently vulnerable libraries and potentially vulnerable APIs invoked by projects based on the collected dataset. Table 6 shows the result. The library "com.alibaba:fastjson:1.2.47" , a JSON processor, tops with the list with a maximum frequency of 170

TABLE 6: Top 5 vulnerable libraries and potentially vulnerable APIs being invoked by projects in the dataset.

ID	Library and version	Frequency	Top invoked potentially vulnerable APIs (Frequency)
1	com.alibaba:fastjson:1.2.47	44	1. `JSON:toString()` (171) 2. `JSON:toJSONString(Object)` (165)
2	org.apache.httpcomponents:httpclient:4.5.2	36	1. `CloseableHttpClient:execute(HttpUriRequest)` (72) 2. `CloseableHttpClient:execute(HttpUriRequest,HttpContext)` (49)
3	org.apache.httpcomponents:httpclient:4.5.3	26	1. `CloseableHttpClient:execute(HttpUriRequest)` (61) 2. `CloseableHttpClient:execute(HttpUriRequest,ResponseHandler)` (10)
4	com.alibaba:fastjson:1.2.62	23	1. `JSON:toJSONString(Object)` (63) 2. `JSONPObject:toString()` (52)
5	org.apache.activemq:activemq-all:5.13.2	21	1. `ClassPathXmlApplicationContext:<init>(String)` (27) 2. `AbstractApplicationContext:getBean(String)` (26)

invocations of vulnerable APIs. This is primarily due to the widespread usage of "`JSON:toString()`", which serves as a fundamental functional component of the library. As TPLs such as "com.alibaba:fastjson" are commonly used by numerous developers, the impact of vulnerabilities in TPLs can be highly unpredictable. Furthermore, as the frequency of calling potentially vulnerable APIs increases, the risks within projects escalate accordingly. Take the project "luanqiu/java8_demo" as an example. This project directly relies on "com.alibaba:fastjson:1.2.47" affected by CVE-2022-25845 and has invoked the potentially vulnerable API "`JSON:toJSONString(Object)`" 45 times, indicating that resolving this vulnerable TPL is crucial to mitigate its impact. This example highlights the importance of promptly addressing vulnerability risks in TPLs when fundamental functional APIs are potentially vulnerable.

Answer to RQ4: By leveraging VAScanner, we found that only 21.51% of projects (with 1.83% false positive proportion and a 95% CI of [0.71%, 4.61%]) were potentially affected by vulnerable TPLs, which indicates that most coarse-grained detection tools produce many false positives, highlighting the need for more precise analysis.

5 Threats to Validity

The threats to our work come from the following aspects: (1) Possible bias of project dataset selection. Since we crawled projects in GitHub according to star numbers, there may be some project deviations. To alleviate it, we tried our best to crawl a large number of real-world projects whose star numbers range from about 70,000 to 0, to make the experiments more representative. (2) Possible inaccuracy of vulnerable versions of libraries. There may be inaccuracies in the vulnerable version ranges provided by Snyk Vulnerability DB and GitHub Advisory Database, based on NVD. This can lead to mistakenly identifying a safe version as vulnerable [53]. To address this, we determined vulnerable root methods by examining adjacent vulnerable versions for patch commits. If vulnerable root methods is not found in earlier vulnerable versions, it indicates that the version is not actually affected, thus minimizing threats and ensuring the validity of our results. (3) Not consider other semantically equivalent refactoring in the sifting mechanism. Since we implement the sifting mechanism based on the AST, which focuses on syntax and structure, it cannot comprehensively capture the context of the code. We will consider detecting all semantically equivalent refactoring in our future work. (4) Possible bias of the ground truth acquisition strategy. We avoided dynamic testing due to its complex setup and high costs, especially for large codebases. Although vulnerabilities were demonstrated in previous work [54], the provided repositories didn't fully support our validation needs for ablation and comparison experiments. Instead, we used a manual validation approach similar to VERJava [47] and Nguyen et al. [46]. Besides, since we assume the call graph generation is accurate and determine the validity of vulnerable APIs based on the validity of vulnerable root methods, this strategy may affect the validity of our results. (5) Limitation of the static analyzer. Although we used the state-of-the-art call graph generation tool Tai-e, it still has certain limitations because Tai-e [38] is a static analysis framework, which inherently struggles to accurately handle dynamic features, polymorphism, and runtime dependencies, which prevents Tai-e from generating completely precise call graphs. (6) Possible bias arising from different vulnerability data utilized by VAScanner and Eclipse Steady. Steady manages its vulnerability data within Project KB [48], which does not completely match the data we collected. This discrepancy may introduce bias in the comparison experiment results. (7) The vulnerable root methods we have augmented are not always vulnerable, which may affect the accuracy of vulnerable root methods.

6 Related Work

The most related work to our paper is software composition analysis (SCA) [30] of Java projects. Plate et al. [25] proposed a dynamic analysis to determine if the project could reach vulnerable methods in TPLs. It was implemented by the dynamic and static instrumentation techniques for unit tests and integration tests, respectively. Ponta et al. [6], [26] advanced this approach and presented a code-centric and usage-based tool, named Eclipse Steady, to identify the reachability of vulnerable methods or code. Specifically, they first conducted a dynamic analysis to assess the reachability of vulnerable constructs. Then, they used the set of constructs that have actually been executed as the starting point for static analysis. Combining dynamic and static analysis, they found all constructs potentially reachable for vulnerability analysis. Despite the progress, their dynamic analysis required unit tests or integration tests as the input for vulnerability analysis, which limited its scalability and effectiveness due to the availability and quality of test code. Wang et.al [55] proposed a bug-driven alerting system that focuses on security bugs. In their approach, they directly considered the methods modified in patches as buggy library methods. INSIGHT [56] explores the cross-ecosystem impact of vulnerabilities, specifically

determining whether a Python or Java project utilizes a vulnerable C library based on the forward cross-language vulnerability reachability analysis. Wu et.al [57] conducted an empirical study aiming to explore the impact of vulnerabilities in upstream libraries on downstream projects. They considered all modified functions in the vulnerability patch as vulnerable functions in libraries. By constructing call graphs for downstream projects and upstream vulnerable libraries, they investigated whether there exists paths in the projects that can invoke the vulnerable functions from the libraries. Relying on dependency management tools such as Apache Maven and Apache Ivy, Pashchenko et al. [24], [58] identified dependencies with known vulnerabilities. They built the paths from projects to their vulnerable dependencies, to address the over-inflation problem when reporting vulnerable dependencies. In addition, both commercial SCA services (e.g., Snyk [9], SourceClear [11]) and open-source SCA tools (e.g., GitHub Dependabot [7], OWASP Dependency Check [5]) detected vulnerable TPLs based on vulnerability information from NVD [31]. Although some SCA commercial tools (e.g., SourceClear [11], and BlackDuck [10]) support vulnerability reachability analysis, they do not provide open source alternatives, posing a hindrance to executing them. Moreover, their methodology for vulnerability reachability analysis like Steady's, uses call graph analysis to check if the project invokes vulnerable APIs. Therefore, we only compared VAScanner with Steady.

There are also other researches that focused on SCA of Android apps, usually known as TPL identification [1], [59]–[68]. Most of them focused on identifying the libraries or library versions used by Android apps via similarity-based or clustering-based methods. Some studies investigated vulnerable TPLs used by projects by detecting whether the projects contained vulnerable TPLs or vulnerable TPL versions [1], [66], [69]. Specifically, OSSPolice [66] maintained a feature database of TPLs, and utilized a similarity-based method to identify whether the used library version was vulnerable by comparing it with the vulnerable libraries affected by CVE. Yasumatsu et al. [69] conducted a similar work by using LibScout [64] to extract the library versions used by APK and comparing them with vulnerable versions. Based on TPLs' feature generation and vulnerability collection, Zhan et al. [1] built a vulnerable TPL database to identify the vulnerable TPL versions used by Android apps. These studies identified vulnerable TPLs but did not analyze whether the apps accessed the vulnerable code. In summary, these studies would cause false positives through analysis only at the library level.

As for VAScanner, we maintain all vulnerable APIs for each vulnerable TPL version. Once projects used a specific library version, VAScanner can effectively determine whether the used library version could threaten the projects by analyzing if the projects used vulnerable APIs.

7 Conclusion

In this paper, we proposed VAScanner, a vulnerable API detection system for TPLs, which can precisely find vulnerable APIs used by Java projects. VAScanner can sift out patch-unrelated methods with high precision, and augment vulnerable root methods which are absent in patch commits, to identify relatively precise and complete vulnerable root methods. Evaluation results show that VAScanner can effectively detect vulnerable APIs based on the constructed vulnerable API database and can find the vulnerable APIs and real impact on real projects.

Acknowledgement

We thank the reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China (No. 62102197, and 62202245), and the Natural Science Foundation of Tianjin (No. 22JCYBJC01010).

References

[1] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "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, 2021, pp. 1695–1707.

[2] X. Zhan, T. Liu, L. Fan, L. Li, S. Chen, X. Luo, and Y. Liu, "Research on third-party libraries in android apps: A taxonomy and systematic literature review," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4181–4213, 2021.

[3] (2022) The 2022 "Open Source Security and Risk Analysis" (OSSRA) report. [Online]. Available: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

[4] (2022) Component Analysis OWASP Foundation. [Online]. Available: https://owasp.org/www-community/Component_Analysis

[5] (2022) OWASP. [Online]. Available: https://owasp.org/

[6] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[7] (2022) Dependabot. [Online]. Available: https://github.com/dependabot/dependabot-core

[8] (2022) OSS Index. [Online]. Available: https://ossindex.sonatype.org/

[9] (2022) Snyk. [Online]. Available: https://snyk.io/

[10] (2022) blackduck. [Online]. Available: https://community.synopsys.com/s/black-duck

[11] (2022) SourceClear: Software composition analysis for devsecops. [Online]. Available: https://www.sourceclear.com/

[12] (2022) WhiteSource. [Online]. Available: https://www.whitesourcesoftware.com/

[13] 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.

[14] (2022) Eclipse Steady. [Online]. Available: https://eclipse.github.io/steady/

[15] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 319–330.

[16] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 125–135.

[17] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, and S.-C. Cheung, "Could i have a stack trace to examine the dependency conflict issue?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 572–583.

[18] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S.-C. Cheung, H. Yu, C. Xu, and Z. Zhu, "Will dependency conflicts affect my program's semantics?" *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2295–2316, 2021.

[19] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 672–684.

[20] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, L. Zhao, J. Wu, and Y. Liu, "Compatible remediation on vulnerabilities from third-party libraries for java projects," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*. IEEE, 2023.

[21] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, and Y. Liu, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," in *the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023.

[22] L. Zhang, C. Liu, S. Chen, Z. Xu, L. Fan, L. Zhao, Y. Zhang, and Y. Liu, "Mitigating persistence of open-source vulnerabilities in maven ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 191–203.

[23] S. Yang, S. Chen, L. Fan, S. Xu, Z. Hui, and S. Huang, "Compatibility issue detection for android apps based on path-sensitive semantic analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 257–269.

[24] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[25] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 411–420.

[26] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[27] K. Li, J. Zhang, S. Chen, H. Liu, Y. Liu, and Y. Chen, "Patchfinder: A two-phase approach to security patch tracing for disclosed vulnerabilities in open-source software," 2024.

[28] (2022) Maven Central Repository. [Online]. Available: https://repo1.maven.org/maven2/

[29] (2024) What is Log4Shell?— IBM. [Online]. Available: https://www.ibm.com/topics/log4shell

[30] (2020) Software composition analysis (SCA): what is it and does your company need it? https://snyk.io/blog/what-is-software-composition-analysis-sca-and-does-my-company-need-it/.

[31] (2022) NATIONAL VULNERABILITY DATABASE. [Online]. Available: https://nvd.nist.gov/

[32] (2022) Snyk Vulnerability DB. [Online]. Available: https://security.snyk.io/

[33] (2022) Github Advisory Database. [Online]. Available: https://github.com/advisories

[34] (2022) Github. [Online]. Available: https://github.com/

[35] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[36] (2012) Patch Commit of CVE-2011-2730. [Online]. Available: https://github.com/spring-projects/spring-framework/commit/9772eb8

[37] (2012) CVE-2011-2730. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2011-2730

[38] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2023, p. 1093–1105.

[39] (2022) Patch Commit of CVE-2022-26884. [Online]. Available: https://github.com/apache/dolphinscheduler/commit/9717da6

[40] (2022) Patch Commit of CVE-2022-40955. [Online]. Available: https://github.com/apache/inlong/commit/0c2e9fe

[41] (2023) spring-framework. [Online]. Available: https://github.com/spring-projects/spring-framework/

[42] (2021) Patch Commit of CVE-2021-29480. [Online]. Available: https://github.com/ratpack/ratpack/commit/603e0c5

[43] (2020) Patch Commit of CVE-2020-36319. [Online]. Available: https://github.com/vaadin/flow/commit/3c089c6

[44] (2024) Patch Commit of CVE-2014-0193. [Online]. Available: https://github.com/netty/netty/commit/93fab1d

[45] A. Agresti and B. A. Coull, "Approximate is better than "exact" for interval estimation of binomial proportions," *The American Statistician*, vol. 52, no. 2, pp. 119–126, 1998.

[46] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Empirical Software Engineering*, vol. 21, pp. 2268–2297, 2016.

[47] Q. Sun, L. Xu, Y. Xiao, F. Li, H. Su, Y. Liu, H. Huang, and W. Huo, "Verjava: Vulnerable version identification for java oss with a two-stage analysis," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 329–339.

[48] (2023) Home page of project "KB". [Online]. Available: https://github.com/sap/project-kb

[49] (2022) Soot. [Online]. Available: http://soot-oss.github.io/soot/

[50] (2022) Wala. [Online]. Available: https://github.com/wala/WALA

[51] (2020) Patch Commit of CVE-2020-10683. [Online]. Available: https://github.com/dom4j/dom4j/commit/a822852

[52] (2021) Patch Commit of CVE-2021-30640. [Online]. Available: https://github.com/apache/tomcat/commit/f4d9bde

[53] S. Dashevskyi, A. D. Brucker, and F. Massacci, "A screening test for disclosed vulnerabilities in foss components," *IEEE Transactions on Software Engineering*, vol. 45, no. 10, pp. 945–966, 2018.

[54] Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, "Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 464–468.

[55] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 35–45.

[56] M. Xu, Y. Wang, S.-C. Cheung, H. Yu, and Z. Zhu, "Insight: Exploring cross-ecosystem vulnerability impacts," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[57] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1046–1058.

[58] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, 2020.

[59] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in Android markets," in *Proc. ICSE*, 2017.

[60] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *ASE*, 2020.

[61] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: Reliable identification of obfuscated third-party Android libraries," in *Proc. ISSTA*, 2019.

[62] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in Android applications with high precision and recall," in *SANER*, 2018.

[63] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in Android apps," in *Proc. ICSE-C*, 2016.

[64] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *CCS*, 2016.

[65] Y. Wang, H. Wu, H. Zhang, and A. Rountev, "Orlis: Obfuscation-resilient library detection for Android," in *Proc. MOBILESoft*, 2018.

[66] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, 2017, pp. 2169–2185.

[67] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 659–674.

[68] L. Li, T. Bissyandé, J. Klein, and Y. L. Traon, "An investigation into the use of common libraries in Android apps," in *SANER*, 2016.

[69] T. Yasumatsu, T. Watanabe, F. Kanei, E. Shioji, M. Akiyama, and T. Mori, "Understanding the responsiveness of mobile app developers to software library updates," in *the 9th ACM Conference on Data and Application Security and Privacy*, 2019, pp. 13–24.

Fangyuan Zhang is currently a Ph.D. candidate in the College of Computer Science at Nankai University (NKU). She received her BSc degree in computer science from Jilin University in 2021. Her research focuses on software supply chain security.

Lingling Fan is an Associate Professor at the College of Cyber Science, Nankai University, China. In 2017, she joined Nanyang Technological University (NTU), Singapore as a Research Assistant and then had been a Research Fellow of NTU since 2019. Her research focuses on program analysis and testing, and software security. She got 4 ACM SIGSOFT Distinguished Paper Awards at ICSE 2018, ICSE 2021, ASE 2022, ICSE 2023.

Sen Chen (Member, IEEE) is an Associate Professor at the College of Intelligence and Computing, Tianjin University, China. Before that, he was a Research Assistant Professor in the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His research focuses on software security. He got 6 ACM SIGSOFT Distinguished Paper Awards. More information is available on https://sen-chen.github.io/.

Miaoying Cai received her BEng degree in Information Security from Nanjing University of Aeronautics and Astronautics in 2023. She is currently pursuing a Ph.D. degree with the Nankai University. Her research interests lie in the area of mobile security and web security.

Sihan Xu received the B.Sc. and Ph.D. degrees in computer science from Nankai University in 2013 and 2018, respectively. For her research, she spent a year with the National University of Singapore. She is currently an associate professor at the College of Cyber Science, Nankai University. Her research interests include intelligent software engineering and AI security.

Lida Zhao is a Ph.D. candidate at Nanyang Technological University (NTU). His research focuses on software security and software engineering, with a particular emphasis on open-source supply chain security and software composition analysis.