Bytecode-centric Detection of Known-to-be-vulnerable Dependencies in Java Projects

Stefan Schott
Paderborn University
Paderborn, Germany
stefan.schott@upb.de

Serena Elisa Ponta
SAP Labs
Mougins, France
serena.ponta@sap.com

Wolfram Fischer
SAP SE
Stuttgart, Germany
wolfram.fischer@sap.com

Jonas Klauke
Paderborn University
Paderborn, Germany
jonas.klauke@upb.de

Eric Bodden
Paderborn University &
Fraunhofer IEM
Paderborn, Germany
eric.bodden@upb.de

Abstract

On average, 71% of the code in typical Java projects comes from open-source software (OSS) dependencies, making OSS dependencies the dominant component of modern software code bases. This high degree of OSS reliance comes with a considerable security risk of adding known security vulnerabilities to a code base. To remedy this risk, researchers and companies have developed various *dependency scanners*, which try to identify inclusions of known-to-be-vulnerable OSS dependencies. However, there are still challenges that modern dependency scanners do not overcome, especially when it comes to dependency modifications, such as re-compilations, re-bundlings or re-packagings, which are common in the Java ecosystem.

To overcome these challenges, we present Jaralyzer, a *bytecode-centric* dependency scanner for Java. Jaralyzer does not rely on the metadata or the source code of the included OSS dependencies being available but directly analyzes a dependency's bytecode.

Our evaluation across 56 popular OSS components demonstrates that Jaralyzer outperforms other popular dependency scanners in detecting vulnerabilities within modified dependencies. It is the only scanner capable of identifying vulnerabilities across all the above mentioned types of modifications. But even when applied to unmodified dependencies, Jaralyzer outperforms the current state-of-the-art code-centric scanner Eclipse Steady by detecting 28 more true vulnerabilities and yielding 29 fewer false warnings.

CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software maintenance tools**.

Keywords

Open-source software, Software composition analysis, Dependency scanner, Security vulnerabilities

1 Introduction

The use of open-source software (OSS) has become ubiquitous in modern software development. OSS is now so prevalent in software projects that third-party dependencies account for the largest portion of the overall code base. According to a 2023 report by Endor Labs on the state of dependency management [30], an average of 71% of the total code base in typical Java projects consists of OSS code. This also means that the use of vulnerable OSS dependencies poses a real threat to modern software projects as additionally highlighted by the OWASP Top Ten ranking of the most critical application security risks [11]. The infamous *log4shell* [10] and *equifax breach* [25] incidents, which had a major impact on the cybersecurity world, further emphasized the risk associated with vulnerable OSS. To this end, various open-source [18, 28, 34, 36] but also commercial *dependency scanners* [15, 22, 23, 26] have been developed, which seek to help minimize the posed threat by identifying known-to-be vulnerable OSS dependencies used within software projects.

Metadata-based scanners conduct dependency scans by leveraging the metadata associated with dependency inclusions. One common method involves generating a *Software-Bill-of-Materials* (SBOM) for the project under analysis. The SBOM lists all dependencies included in the software project along with their identifiers and versions. This information is then compared against a security advisory of known vulnerable dependency versions, often represented as Common Vulnerability Enumeration (CVE) entries [4]. In the case of a Java Maven project this means creating an SBOM from the dependency information obtained within the project's `pom.xml` files and comparing it to some advisory. However, as Ponta et al. [33, 34] have reported, this approach has multiple shortcomings. For one, security advisories often do not contain precise information about which *modules* of a specific OSS are affected by a certain CVE entry, causing metadata-based scanners to report false positives [34]. E.g., if one considers the popular Spring framework for Java, advisories often assign a specific CVE entry to the whole framework, even though only one of its 20 modules is actually affected. Even projects using only unaffected modules will nonetheless receive vulnerability alerts. On the other hand, vulnerabilities and patches will not be reported by dependency scanners until the used advisories have been updated. The median delay between public patch availability and advisory publication is 25 days [31]. In the Maven ecosystem, the average delay extends to 41 days. This timeframe does not even account for the period between a fix becoming available in the project's source code repository and its eventual release to a public artifact repository.

To combat these issues Ponta et al. [34] have proposed a *code-centric* approach, which does not rely on the information about known-to-be-vulnerable dependencies in advisories but directly

compares the *code* of included dependencies to a database of *fix commits*. These fix commits represent the real code changes performed within the OSS respective Git repositories that fix a specific CVE entry. Directly relying on the fix commits, one does not have to manually map the vulnerability to a list of affected OSS versions first. Furthermore, having access to the exact code changes comprising the fix, one exactly knows which module of the OSS is affected by a vulnerability. Moreover, it allows for a subsequent reachability analysis to check whether the vulnerable code segment is even executed within the context of the including software project. This enables the approach to reduce false positives in the detection of vulnerabitilties when compared to metadata-based approaches.

Ponta et al. have implemented the code-centric approach in the dependency scanner Eclipse Steady. A prerequisite of the code-centric approach is that both fix commits and scanned dependencies are available in the same code representation to be comparable, an assumption that generally holds for languages like Python or JavaScript. In contrast, Java poses a challenge, as dependencies are usually distributed in compiled *bytecode*, whereas fix commits modify the original *source code*. To circumvent the issue of mismatched code representations, Eclipse Steady attempts to retrieve the source code of dependencies from Maven Central, the most popular Java artifact repository. However, the source code may or may not be available alongside the compiled artifacts and, even when present, there is no guarantee that it actually matches the compiled artifact. If Eclipse Steady cannot retrieve the source code, it is unable to compare bytecode to source code [40], and thus requires costly and error-prone manual analysis to determine if a dependency is vulnerable.

While dependency scanners have gradually improved over the past years, important challenges remain, which considerably impair the detection performance of state-of-the-art tools. Dann et al. [7] conducted a study evaluating the performance of six state-of-the-art dependency scanners—both open-source and commercial—on Java projects where dependencies were included in *modified* forms. These modifications include re-compilations, re-bundlings and re-packagings, which as determined by Dann et al. are widespread modifications in the Java ecosystem. In their investigation of a sample of 254 vulnerable classes, they identified 67,196 artifacts on Maven Central that contained these classes in a modified form, highlighting the prevalence of such modifications. In these cases, the absence of precise metadata keeps metadata-based scanners from reliably detecting known-to-be-vulnerable dependencies, while Eclipse Steady is hindered by the lack of source code resulting from most types of modifications. A similar experiment conducted by Dietrich et al. [8] further confirmed these results and emphasized the prevalence of dependency modifications that can transparently be applied by the popular Maven Shade Plugin, unbeknown to developers. Such modifications can even be found in the Java standard library.

To overcome these challenges, we propose Jaralyzer, our approach to *bytecode-centric* Java dependency scanning, which extends the code-centric approach defined by Ponta et al. [34] to handle mismatched code representations. Jaralyzer does not rely on the availability of metadata or source code, but directly compares a dependency's *bytecode* against a fix-commit database. Due to this capability, Jaralyzer does not require any manual analysis to determine if a dependency is vulnerable and is able to reliably detect known-to-be-vulnerable dependencies in modified forms. To achieve this, Jaralyzer employs techniques to compile the code changes within a fix commit in isolation and normalizes the obtained bytecode to enable a comparison independent of the used compiler. Subsequently, it compares syntax, control and data flow to precisely identify whether known-to-be-vulnerable dependencies are included in the analyzed software project and whether corresponding fixes have been applied or not.

Our evaluation of Jaralyzer shows that, when it comes to detecting known-to-be-vulnerable dependencies included in modified form, it outperforms five other dependency scanners, four OSS and one commercial. With a miss rate of at most 6% for vulnerabilities in re-packaged dependencies compared to unmodified ones, it is the only scanner capable of handling all types of modifications identified by Dann et al [7]. But even when applied to unmodified dependencies, when directly compared to the state-of-the-art code-centric dependency scanner Eclipse Steady, Jaralyzer reported 28 more true and 29 fewer false vulnerabilities.

To summarize, the original contributions of the paper are:

- A bytecode-centric Java dependency scanner, which does not rely on metadata or source code being available and is able to detect known-to-be-vulnerable OSS dependencies, even in modified form.
- An implementation in an open-source tool[1].
- An evaluation across 56 popular Java OSS libraries using five different open-source and commercial dependency scanners.

2 Background

In the following we introduce the fundamental topics of how Java dependencies are included and modified.

2.1 Java Dependencies:

A Java project P can be defined as a tuple $P = (C, D)$, where C represents the application code written by the project's developers, and D denotes the set of *dependencies*, i.e., third-party components originating from external sources. Jaralyzer focuses exclusively on analyzing the set of dependencies D, and does not examine the application code C.

The term *dependency* refers to a separately distributed software library or framework that is included in a software project [7]. In the case of Java, these software libraries are typically distributed as JAR archives containing the *bytecode* of the library. Bytecode is a machine-readable code representation Java source code is compiled to, allowing it to be executed by the Java Virtual Machine. To manage the inclusion of dependencies, developers usually use build-automation tools such as Maven or Gradle. In the case of Maven, the developer specifies the `groupId`, `artifactId` and `version` of the desired library within a `pom.xml` file, which contains the build information for the software project. These three properties, referred to as *GAV*, are used to uniquely identify a library. During build time, Maven automatically retrieves the specified dependencies from configured artifact repositories, typically Maven Central. In addition, Maven also resolves possible dependency version conflicts

[1] https://github.com/stschott/jaralyzer

and retrieves transitive dependencies, which are the dependencies of the project's direct dependencies.

2.2 Dependency Modifications:

Dependencies in Java projects are not always distributed and included in the straightforward way described in Section 2.1. Dann et al. [7] have identified four types of modifications developers frequently apply to OSS components, which are then in turn included by other developers as dependencies. In the remainder of this paper, 'modifications' refers to these four types.

Type 1 (patched): This type of modification frequently occurs when developers fork the source code of an OSS and modify or patch it. Dependencies modified this way do not include the original bytecode but the bytecode obtained when *re-compiling* the modified source code. Furthermore, one typically modifies the GAV by appending a suffix like `fix` or `patch` to it.

Type 2 (Uber-JAR): In this type of modification, developers *re-bundle* multiple OSS into a single JAR file, usually referred to as *Uber-JAR*. This type of modification is sometimes indicated by the JAR file containing `jar-with-dependencies` or `uber` in its file name. In contrast to type 1, the original bytecode of each individual OSS is preserved.

Type 3 (bare Uber-JAR): Type 3 modifications are similar to type 2 in the sense that multiple OSS are re-bundled into a single JAR file. However, in contrast to type 2, the metadata contained within the JAR file (`pom.xml`, `META-INF` folder and file timestamps) is removed. This type of modification can mostly be observed in legacy JAR files, which have been created before the advent of assembly plugins.

Type 4 (re-packaged Uber-JAR): Type 4 modifications are also similar to type 2. However, instead of just re-bundling multiple OSS into a single JAR file, the OSS are *re-packaged*. This involves modifying their original package names, either by prepending a string or by replacing them entirely with new names [13]. This is often done to avoid name clashes and version conflicts. In such cases the original bytecode of the OSS is changed substantially, because all references within the code need to be adjusted.

3 Bytecode-centric Dependency Scanning

In the following we present Jaralyzer, a *bytecode-centric* dependency scanning approach.

3.1 Overview

Figure 1 shows an overview of Jaralyzer's architecture. The approach consists of two major stages: *(1)* the knowledge-base creation and the *(2)* dependency scanning. The constructed knowledge base serves as the foundation for the dependency scanning. It contains all vulnerabilities that Jaralyzer can detect.

As a code-centric approach, Jaralyzer uses fix commits as input, which are commits fixing vulnerabilities in the source code repositories of OSS. In the first step of the knowledge-base creation stage, Jaralyzer compiles the fix commits to bytecode using two techniques: First it applies a compilation heuristic and, if this heuristic fails, it uses Jess [38] to compile commit changes within source code repositories. After compilation of the fix commits, Jaralyzer normalizes the resulting bytecode using jNorm [39] to enable a

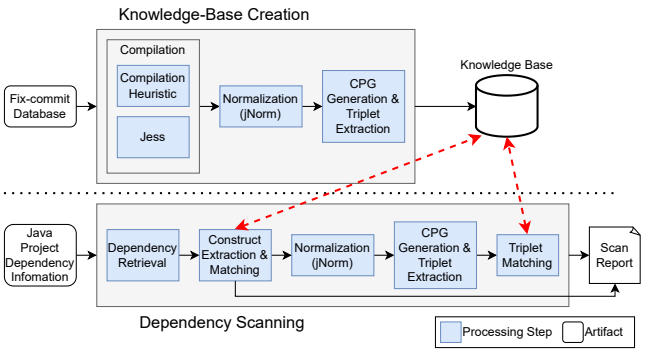

Figure 1: Overview of Jaralyzer

subsequent comparison. This step is necessary to remove differences that exist solely due to the use of different compilers. If not removed, those differences would interfere with matching the applied fix in the dependency scanning stage. Afterwards, for each changed method in a given fix commit, Jaralyzer generates a code property graph (CPG) [46], a representation that combines syntax, control flow, and data dependency information of a method. Finally, Jaralyzer extracts comparable string triplets [3], which encode edge information, from the generated CPGs and stores them in the knowledge base. Jaralyzer uses these triplets to identify the presence of the fix in the next stage.

To scan a software project for known-to-be-vulnerable dependencies, Jaralyzer performs the steps described in the dependency scanning stage (see Figure 1). First, Jaralyzer uses the underlying build-automation tool of the project being scanned to extract all its dependencies. Jaralyzer then scans each of the extracted JAR files individually for potential vulnerabilities. To do so, Jaralyzer matches all *constructs* (class, interface or method) within the JAR file to its knowledge base. If Jaralyzer identifies a matching construct, it indicates that an included dependency in the software project may be vulnerable. However, at this point, it remains unclear whether the version in use has already applied the fix. To determine the presence of the fix, all constructs added, removed or changed in the fix are considered. In particular, if the scanned construct is a method *changed* in the fix—which is the most common scenario—Jaralyzer normalizes it, generates a CPG for it and transforms it into comparable string triplets. Finally, Jaralyzer compares the triplets yielded from the scanned method's CPG to the triplets stored within its knowledge base. Based on the degree of matched triplets, Jaralyzer reports the method as fixed or vulnerable. This procedure is performed for each construct in each JAR file, resulting in a final report that contains all identified vulnerable dependencies.

In the following we will describe each processing step in detail.

3.2 Knowledge-Base Creation

The knowledge-base creation stage is essential to code-centric dependency scanning. In contrast to scanners relying on metadata, Jaralyzer requires the bytecode corresponding to vulnerability fixing commits. This stage only needs to be executed once to create Jaralyzer's knowledge base, which is then available for every

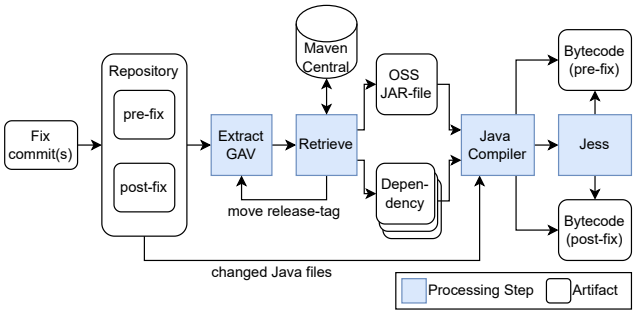

Figure 2: Detailed overview of Jaralyzer's compilation step

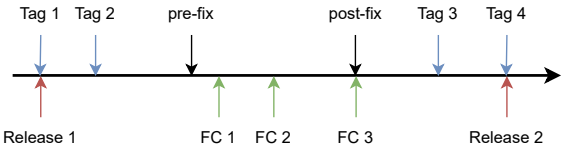

Figure 3: Exemplary version history of an OSS project with three fix commits. FC = Fix Commit

dependency scan. After construction, for every CVE entry, the knowledge base contains the *fully qualified name* (FQN) of each construct modified in the fix commits related to that CVE. Each CVE entry contains at least one modified construct. The FQN uniquely identifies a construct by specifying its full package and class name. Additionally, the knowledge base contains each construct's modification type (added, removed, or changed), and the precise bytecode additions, removals and changes, extracted from the CPG as string triplets.

3.2.1 Fix-commit Database. As a bytecode-centric approach, Jaralyzer relies on a fix-commit database as input to create its knowledge base. A fix-commit database maps CVE entries to the commits in an OSS project's source code repository that fix the respective CVE entries. Possible fix-commit database options include SAP's Project KB [35], CVEFixes [2] or MoreFixes [1].

3.2.2 Compilation. As dependencies are included in bytecode format but fix commits are in source code, to enable a comparison Jaralyzer needs to generate bytecode for each construct modified by the fix commits of each CVE. However, simply triggering a standard compilation with the configured build-automation tool is insufficient, as it frequently leads to failure [14, 38, 42, 44, 49]. As shown by Schott et al. [38], performing a standard compilation following the approach of Tufano et al. [44] successfully compiles only 14.9% of CVE entries in the Project KB fix-commit database. However, for bytecode-centric dependency scanning one does not need to compile the entire code base, but only the source code files modified within the respective fix commit. Leveraging this fact through a custom *compilation heuristic* significantly improves the compilation success rate (see Section 4.1). Instead of compiling the entire code base, Jaralyzer attempts to download all dependencies needed to compile the files modified in the fix commits and then compiles these files in isolation. Second, if the heuristic fails, it applies the commit compilation tool Jess [38].

Figure 2 shows a detailed overview of the compilation process. It starts by cloning the source code repository corresponding to fix commits of a given CVE entry. The fix might be contained within a single commit or segmented across multiple commits. Figure 3 shows an exemplary version history of a project with the fix consisting of the three fix commits FC1–FC3. At first Jaralyzer collects all Java source code files (excluding test files) changed within the commits FC1–FC3. To be able to extract the precise bytecode changes performed in the fix, Jaralyzer needs to compile the changed files before and after the fix is applied. Thus, after collecting the changed files, Jaralyzer checks out the revision *before* the first fix commit (FC1) is applied (see pre-fix in Figure 3).

To compile the changed files, we need to retrieve all necessary dependencies. This includes a compiled release version of the OSS project, allowing us to avoid compiling the entire project and instead focus only on the changed files. We do not use the compiled release version to extract the fix, as the original fix might have been further changed before the actual release and only use it as compilation supplement. Directly using the release may interfere with the matching in modified cases, especially for type 1 modifications (see Section 2.2). Here developers sometimes copy the original fix and manually apply it to their forked project.

To retrieve the compiled release version, Jaralyzer first tries to extract the GAV identifier from the project (see Figure 2). This identifier is used to lookup if this specific version of the artifact exists on Maven Central. If Jaralyzer does not find an artifact on Maven Central corresponding to the extracted coordinates, it will checkout the next repository revision tagged with a release tag and repeat the process. Tags are typically used to mark release versions of the project, however sometimes developers tag a specific revision of the project that is not released on Maven Central. Within the example shown in Figure 3 one can see that Tag 3 is not pointing towards a release version of the project, but Tag 4 is. Thus, Jaralyzer needs to move the tag at least two times to find a corresponding release on Maven Central. We perform this step up to ten times, as our empirical testing revealed no cases where additional iterations resulted in a successful compilation. If Jaralyzer finds a release version on Maven Central, it retrieves the corresponding JAR file and all available dependencies configured within the `pom.xml` file, which is hosted alongside the JAR file. Extracting the GAV identifier typically works well for Maven projects, as one can simply invoke Maven to obtain this information. For Gradle projects this is often not as trivial. As Gradle build configurations are based on a Groovy/Kotlin-based language, Gradle first needs to execute the configuration scripts before the GAV can be extracted. However, this execution often fails for outdated builds. In such cases, we manually inspect the project's configuration scripts to extract the GAV and provide it to the compilation process. Finally, using the retrieved dependencies, as well as the pre-compiled JAR file from Maven Central, Jaralyzer will forward the collected changed source code files from the repository, to a JDK Java compiler. If the compilation is successful, Jaralyzer will obtain the bytecode of files changed in commits FC1–FC3 (see Figure 3), before the fix is applied. Now to obtain the bytecode of the files *after* the fix is applied, Jaralyzer will repeat the same process analogously for

the post-fix version, by checking out the revision that commit FC3 is pointing to, as depicted in Figure 3. If the compilation via the heuristic fails, Jaralyzer will invoke Jess [38], a tool that has been specifically designed for the purpose of compiling commit changes within Java source code repositories in isolation. Although, this is only a secondary option, as Jess might produce bytecode that is slightly different from the original bytecode and thus might cause interference for a subsequent comparison. If Jaralyzer is able to compile only one of the pre- or post-fix revisions with the Java compiler, it will instead attempt to compile both revisions with Jess to ensure a consistent bytecode representation.

3.2.3 Normalization. Obtaining the bytecode of fix commits alone is not sufficient for detecting the presence of vulnerable bytecode in included dependencies. Since we do not know what compilation environment the original dependencies have been compiled in, and different Java compilation environments (e.g. different compilers, versions and settings) produce different bytecode for equal source code [6, 39], the bytecode produced by Jaralyzer may differ from the one included in the scanned dependencies. Since vulnerability fixes are often small, involving only a few lines of code, compilation differences may well mask the original fix and make a comparison impossible. To make the bytecode comparable, we apply jNorm [39] right after compilation. jNorm is a tool to normalize bytecode by removing nearly all differences introduced by different compilation environments. After applying jNorm, compilation differences are mostly removed and the bytecode can be compared.

3.2.4 CPG Generation & Triplet Extraction. After normalization, Jaralyzer uses the SootUp [21] static analysis framework to generate Code Property Graphs (CPG) for each method changed in the fix commits. A CPG [46] is a graph-based representation of an individual method that combines syntax information in form of an abstract syntax tree, control flow information in form of a control flow graph and data, as well as control dependencies in form of a program dependence graph. To extract the precise code instructions that comprise the fix, one must compute the differences between the CPG of the method before the fix was applied and the CPG of the method after the fix was applied. This requires determining subgraph isomorphisms—a problem that is NP-complete—which makes a direct comparison of CPGs computationally expensive [3]. We thus resort to an approximation proposed by Bowman and Huang [3] and extract graph *triplets*. For every edge e within a given CPG g, a triplet t is defined as (n_s, e_l, n_t) where e_l denotes the label of edge e, n_s denotes the source node label and n_t the target node label of edge e within graph g. The set T is the set of all triplets for a given CPG g. Once we have created a CPG g_{vul} and a CPG g_{fix} for a method before and after the fix has been applied we extract the respective triplet sets T_{vul} and T_{fix}. We then define the *context triplets* CT, *positive triplets* PT, and *negative triplets* NT:

$$CT = T_{vul} \cap T_{fix} \qquad PT = T_{fix} \setminus T_{vul} \qquad NT = T_{vul} \setminus T_{fix}$$

CT contains all triplets that have not been changed by the fix, PT contains triplets that have been added by the fix and NT contains triplets that have been removed by the fix. Thus, PT and NT contain the fine-grained code changes performed within the fix. Finally, we store the generated triplet sets for each method in Jaralyzer's knowledge base, alongside the information about the FQN of the method and the affecting CVE entry.

3.3 Dependency Scanning

In the dependency scanning stage, Jaralyzer is applied to a Java software project containing dependency information and tries to identify whether known-to-be-vulnerable dependencies are included.

3.3.1 Dependency Retrieval. Jaralyzer starts with retrieving all dependencies included in the scanned project. Depending on the used build-automation tool, it invokes a command of the tool (e.g. the `copy-dependencies` goal of the maven-dependency-plugin [12]) to retrieve the JAR files corresponding to each dependency as defined in the project's dependency information (e.g. `pom.xml` for Maven projects).

3.3.2 Construct Extraction & Matching. In this step, Jaralyzer individually processes each previously retrieved dependency JAR file, first extracting the FQN of every construct within it. Then, Jaralyzer queries the knowledge base with each extracted FQN to verify if any of these FQNs is associated with a vulnerability. This query results in a set of CVE entries potentially affecting the JAR file. Jaralyzer proceeds to scan the JAR file for each potential CVE entry individually. Here it distinguishes between constructs removed, added or changed by the fix: *(1)* If, within the scanned JAR file, a construct is detected that is marked as *removed* by the fix of the respective CVE entry, Jaralyzer classifies the corresponding construct as vulnerable, otherwise as fixed. *(2)* If a method is marked as *added* by the fix but is missing from the scanned JAR file while its declaring class is present, Jaralyzer classifies the construct as vulnerable, otherwise as fixed. *(3)* If Jaralyzer detects a method in the scanned JAR file that has been *changed* by the fix, it extracts the method body and proceeds with normalization (see Section 3.2.3) and CPG generation (see Section 3.2.4), following the same process used in the knowledge-base creation stage.

3.3.3 Triplet Matching. After generating a CPG g_m and extracting the set of triplets T_m for the matched and normalized method (see Section 3.2.4), Jaralyzer performs a triplet matching to determine whether the method is in a vulnerable or fixed state. To do so, it compares the triplet set T_m to the set of positive triplets PT and negative triplets NT of the matched method within the knowledge base. If the equation $|NT \cap T_m| \geq |PT \cap T_m|$ holds, it indicates that the matched method is more or equally similar to the method *before* the fix has been applied. Jaralyzer thus classifies the matched method as vulnerable. If the fix did not remove or change any code, but just added code within a method, the set of negative triplets NT is always empty. In such cases, Jaralyzer verifies, whether $|PT \cap T_m| \geq \theta_{PT}$, for a configurable threshold θ_{PT}.

In the end, Jaralyzer counts these matched constructs that are associated with a specific vulnerability and have been classified as vulnerable. If the number of constructs classified as vulnerable is greater than or equal to those classified as fixed, Jaralyzer reports the included JAR file as vulnerable to this specific CVE entry. Especially when the initial fix is changed over time, it might be the case that some constructs are reverted to their pre-fix state and Jaralyzer thus classifies some constructs as fixed and others as

vulnerable. In cases involving discrepancies, tools such as Eclipse Steady require human intervention to reach a decision. In contrast, because Jaralyzer is designed to minimize human involvement, we decided on this majority criterion, which yielded the best performance in empirical testing. Finally, Jaralyzer continues to process the next CVE entry potentially affecting the JAR file.

After scanning all provided JAR files, Jaralyzer generates a final scan report that lists all discovered vulnerable dependencies along with their associated CVE entries.

3.4 Re-Packaging Detection

Jaralyzer has been designed to not require any metadata about included dependencies within its dependency scanning stage. This means that, by construction, it does not impact Jaralyzer's detection capabilities whether the included dependencies are re-compiled, re-bundled or their associated metadata is removed (type 1–3 modifications). As long as the bytecode is available, Jaralyzer will be able to scan them for known-to-be-vulnerable OSS. However, to uniquely identify constructs and to initially match them to its knowledge base, Jaralyzer does rely on their FQNs. The FQNs of constructs, however, change when the dependency is re-packaged (type 4 modification, see Section 2.2). Listing 1 shows a source code example (for illustration purposes) of such a re-packaging, which is typically applied to bytecode using tools like the Maven Shade Plugin [8]. Listing 1a shows the original class, while Listing 1b shows the same class, but re-packaged. The comments in lines 4 and 7, above class `C` and method `foo`, show their respective FQNs. Differences between both listings are highlighted. Due to re-packaging, which prepends the string "r" in front of each package name within the dependency, the FQNs of all constructs change and will never match with the knowledge base in case a vulnerability is associated with method `foo` in Listing 1a. To cover type 4 modifications, Jaralyzer uses a *re-packaging detection* mode that does not rely on FQNs for matching. The re-packaging detection follows the same process shown in Figure 1, yet instead of relying on FQNs during the construct-matching step, it uses the *unqualified* names of constructs (class names *without* package names). E.g., instead of querying the knowledge base for the fully qualified method signature `r.a.C: void foo(r.a.b.X)` (see Listing 1b), Jaralyzer will query for all methods in the knowledge base having the *unqualified* signature `C: void foo(X)`. As the unqualified signature does no longer uniquely identify methods, Jaralyzer additionally checks for the *class context* to avoid spurious matches: it checks whether all sibling methods and fields of the scanned method are also present in the method matched in the knowledge base. The information about the class context has also been collected during the knowledge-base creation stage. As an example, if the method name `foo` from Listing 1b) matches the knowledge base, the class context will consider the sibling field `baz` (line 6 in Listing 1b). If the class context exceeds a configurable threshold θ_{CC}, Jaralyzer considers the method as a correct match.

Finally, the triplet matching step also slightly differs from Jaralyzer's default process. As in Java bytecode type names are always fully resolved, Jaralyzer has to unqualify all triplets. E.g., consider the call of method `bar` in line 9 of Listing 1b. Within the bytecode, and thus within its corresponding triplet, the method call

```
1  package a;
2  import a.b.X;
3
4  // a.C
5  class C {
6    int baz;
7    // a.C: void foo(a.b.X)
8    void foo(X x) {
9      int i = x.bar();
10   }
11 }
```

(a) Original class

```
1  package r.a;
2  import r.a.b.X;
3
4  // r.a.C
5  class C {
6    int baz;
7    // r.a.C: void foo(r.a.b.X)
8    void foo(X x) {
9      int i = x.bar();
10   }
11 }
```

(b) Re-packaged class

Listing 1: Re-packaging example

will be expressed with the fully qualified name of `X` (i.e., `r.a.b.X`). Thus, before matching, we unqualify all triplets within the scanned JAR and the knowledge base. To avoid false-positive matches, before comparing the triplet set of the scanned method T_m to the set of positive and negative triplets, we verify that the equation $|CT \cap T_m| > \theta_{CT}$ holds for a configurable threshold θ_{CT}. Doing this, we verify whether the part of the method that has *not* been changed by the fix, expressed by the set of context triplets CT, is sufficiently similar to the scanned method.

These adjustments however, come with the cost of potentially missing vulnerable dependencies or erroneously reporting a dependency as vulnerable, often depending on the choice of the configurable thresholds. Thus, depending on the use case, the default mode and the re-packaging detection mode can be run in tandem or independent of each other. When only running the default mode, one will not identify any re-packaged known-to-be-vulnerable dependencies, while only running the re-packaging detection mode, one might receive more false-positive vulnerability reports.

Although re-packaging detection could be added as a step within Jaralyzer's main pipeline, we added it as a parallel mode to avoid repeated, costly executions of the full pipeline, including normalization and CPG-generation, caused by ambiguous method signatures. By providing two independent modes, developers can either run both or choose the one that best suits their specific needs.

4 Evaluation

In the following we evaluate the effectiveness of Jaralyzer. To do so we answer the following research questions.

- **RQ1:** How does Jaralyzer compare to state-of-the-art dependency scanners in identifying *modified* known-to-be-vulnerable dependencies?
- **RQ2:** How does Jaralyzer compare to the state-of-the-art code-centric dependency scanner in identifying *unmodified* known-to-be-vulnerable dependencies?
- **RQ3:** How runtime-efficient is Jaralyzer?

The first research question focuses on the challenge that current dependency scanners face, when dependencies are included in modified form. The second research question evaluates Jaralyzer's detection performance in the general case, when dependencies are

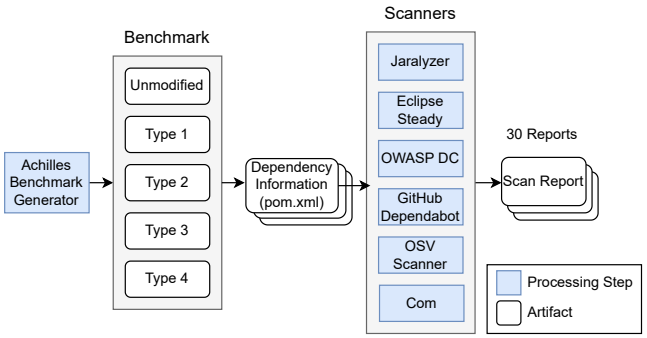

Figure 4: Overview of the experimental setup

not modified. To do so, we compare Jaralyzer against the state-of-the-art code-centric dependency scanner Eclipse Steady. The third research question evaluates Jaralyzer's runtime efficiency.

4.1 Experimental Setup

Figure 4 shows an overview of our experimental setup. The evaluation is based on the Achilles benchmark generator created by Dann et al. [7]. Achilles comes with a dataset of 534 distinct OSS artifacts and allows for an automatic creation of Maven projects having dependencies on a selected subset of these artifacts, either in unmodified or modified form (type 1–4). Most of the provided artifacts are different versions of the same OSS. We only included the latest provided version[2] of each distinct OSS (unique groupId and artifactId), as including multiple versions of the same OSS within a single project would lead to version conflicts. Using the Achilles benchmark generator we created five distinct Maven projects, each having 56 artifacts as dependencies, including popular OSS such as Guava, Spring and Jackson-Databind. According to Endor Labs [31], these are among the Java OSS accounting for most vulnerability findings. We kept the 56 artifacts together in a single project, as splitting them should not impact tool behavior and would require introducing arbitrary splitting criteria. Based on the definition presented in Section 2.1, we create five distinct projects $P = (C, D)$, with C being empty as no application code is provided and D containing the 56 artifacts from Achilles as dependencies. While one of the created projects contained the dependencies in unmodified form, the other four projects contained *modified* dependencies according to types 1–4 (see Section 2.2). We then provided the dependency information (`pom.xml` files) belonging to each of the generated projects to six different dependency scanners: Jaralyzer, Eclipse Steady [34], OWASP DC [28], GitHub Dependabot [16], OSV Scanner [18] and a commercial dependency scanner (Com). Applying each tool to the five different projects within our benchmark, we obtained a total of 30 scan reports containing the CVE entries that each tool detected for the individual projects.

As Jaralyzer's knowledge base, we use SAP's Project KB [35], as it is also the knowledge base utilized by Eclipse Steady. It contains a total of 1,297 manually-curated CVEs from 2005–2023, affecting popular and industry-relevant Java and Python OSS. We filtered out all entries that do not affect Java OSS or include no changes to Java source code files from Project KB. This left us with 733 relevant entries. Jaralyzer was able to compile 542 of these 733 entries using its compilation heuristic (see Section 3.2.2). For 170 of these compilations we had to supplement additional information about GAVs. Jess [38] was able to compile 122 of the remaining entries (54 full and 68 partial compilations). Thus, leveraging the fact that only files modified in the fix commits need to be compiled, Jaralyzer was able to compile 664/733 (90.6%) entries of Project KB for Java OSS. We also performed an investigation of the failed cases. Fix commits are applied in-between two releases (see Figure 3). Whenever there are code dependencies towards the release versions before *and* after the commits, the compilation fails, as we can only provide one of the versions for compilation. Furthermore, in few cases the compiled release version or the dependencies of the project to be compiled are not stored on Maven Central but on different repositories.

In total, leveraging the fact that only few files need to be compiled, Jaralyzer achieves a comparatively high compilation success rate on Java repository snapshots [14, 42, 49], bringing the 14.9% success rate reported by Schott et al. [38] when applying the procedure of Tufano et al. [44] on Project KB, up to 90%. Nevertheless, there remains room for improvement to enable the detection of all CVE entries within the Project KB fix-commit database.

All dependency scanners were executed within a Docker container (v26.1.4) running Alpine Linux v3.19 with Eclipse Temurin JDK 17. The system running the container, was configured to use four cores of an Intel Xeon E5-2695 v3 (2.3 GHz) CPU and 32GB of RAM. Furthermore, we used OWASP DC v11.1.0, OSV Scanner v1.4.3 and Eclipse Steady v3.2.5. GitHub Dependabot was executed in December 2024. We did not execute Eclipse Steady's reachability analysis, as our evaluation focuses on comparing detection performance. Eclipse Steady classifies certain reported vulnerabilities as "unknown", requiring manual analysis (57 in the unmodified benchmark, all of them in the modified benchmarks). In fact, for 4 out of the 56 artifacts in the unmodified benchmark, the source code is not available on Maven Central and for the modified benchmarks, no source code is available. We count these cases as reported vulnerabilities. It is worth noting that, even when source code is available, there is no guarantee that it is accurate: Maven Central does not verify if the source code matches the actual bytecode and even allows uploading placeholder files that contain no source code at all [41]. Another interesting observation we made is that Eclipse Steady performs significantly worse when Achilles packages the artifacts on Windows compared to Linux, so we let Achilles package all artifacts on a Linux machine. Finally, we executed Jaralyzer in default and re-packaging detection mode and configured the (default) threshold values: $\theta_{PT} = 0.5$, $\theta_{CC} = 0.3$, $\theta_{CT} = 0.3$. Through empirical testing, we found these values to provide the best trade off between detecting re-packaged known-to-be-vulnerable dependencies and minimizing wrong alerts.

[2]For OSS comprised of multiple modules, e.g. Spring and Jetty, we did not include the latest version provided in Achilles but the version with the most compatible modules

Table 1: Detected CVE entries by each scanner in comparison to Jaralyzer on modified dependency inclusions. Bold numbers indicate CVE entries missed by Jaralyzer.

	Agreed	Type 1	Type 2	Type 3	Type 4
Jaralyzer	61	61	**58**	**58**	**54**
Steady		59	61	61	-
Jaralyzer	78	78	78	78	**74**
OWASP DC		74	56	-	56
Jaralyzer	57	57	57	57	**56**
Dependabot		56	-	-	-
Jaralyzer	64	64	64	64	**62**
OSV Scanner		-	-	-	-
Jaralyzer	72	72	72	72	**71**
Com		72	-	-	-

4.2 RQ1: How does Jaralyzer compare to state-of-the-art dependency scanners in identifying *modified* known-to-be-vulnerable dependencies?

In this research question we evaluate Jaralyzer's performance on *modified* dependencies, which pose a major challenge for current dependency scanners [7, 8]. To measure the effectiveness of dependency scanners on modified dependencies, Dann et al. used a small benchmark of seven distinct artifacts created with the Achilles benchmark generator, applying the same modifications as we did in our experimental setup (see Section 4.1). To determine whether each tool reported a true or false positive, they rely on a ground truth, which they have semi-automatically uncovered within their [7] study and added to the Achilles benchmark generator. We manually inspected the ground truth used in their experiment and uncovered that it is not fully accurate. Even in their small benchmark containing only 7 distinct artifacts and 15 distinct CVE entries, two of the test cases are incorrect. They report httpclient 4.1.3 as being affected by CVE-2015-5262, whereas it only affects httpclient as of version 4.3[3]. Furthermore, they erroneously do not consider CVE-2018-11040 as affecting spring-webmvc 5.0.0.RELEASE[4]. Because of these inaccuracies, we decided not to rely on the ground truth provided by Achilles. Instead, we compare Jaralyzer head-to-head with each of the five other tools within our setup. We assume that if a tool is unaffected by a specific type of modification, it should report the same CVE entries for both the unmodified and modified benchmarks. First, we filter all the CVE entries where both tools *agree* upon the respective CVE entry affecting the unmodified benchmark. These agreed upon CVE entries are the only ones we consider for the modified benchmarks. Then, we investigate how many of the same CVE entries reported for the unmodified benchmark by both tools are also reported for the type 1–4 benchmarks.

Table 1 presents the results of this experiment. It shows the head-to-head comparison of Jaralyzer with each of the tools and the number of agreed upon CVE entries for the unmodified benchmark (column "CVEs"). Columns "Type 1" to "Type 4" show how many of agreed upon CVE entries are reported by each tool for the type 1–4 benchmarks. None of the tools Jaralyzer is compared against is able to handle all types of modifications. While some tools perform better on certain types of modifications, e.g., Eclipse Steady performing well on types 1–3, GitHub Dependabot, OSV Scanner and the commercial tool completely fail for type 2–4 modifications. Throughout type 1–3 modifications, Jaralyzer does not miss any CVE entry, except for three in its comparison to Eclipse Steady. A detailed investigation of these entries reveals this to be an interesting case, where the three missed CVE entries are actually *false positives* for the unmodified benchmark reported by both Jaralyzer and Eclipse Steady. These false positives are caused by the code-centric nature of the tools. The corresponding CVE entries (CVE-2013-6429, CVE-2014-0225, CVE-2015-2080) are affecting the multi-module projects Spring and Jetty. If the fix commits associated with a CVE entry modify code across multiple modules of a project, even though the vulnerability only affects a single module, this may interfere with detection in certain cases. For example, CVE-2015-2080 affects the *jetty-http* module of the Jetty project, however, its fix commits modify code in both the *jetty-http* and *jetty-util* modules. Now, if the JAR files corresponding to jetty-http and jetty-util are scanned individually, both tools erroneously report jetty-util as being vulnerable, since the majority of the fix code, which is applied to jetty-http, cannot be identified within jetty-util. However, in type 2–4 modifications, all dependencies are re-bundled into a single JAR-file. This then allows the tools to match all code changes from the respective fix commits at once. In contrast to Eclipse Steady, Jaralyzer is able to take advantage of this, and correctly determines that the re-bundled JAR file is unaffected by these CVE entries, reducing the signaled entries from 61 to 58 (54 for type 4). An important point to consider is that, due to the unavailability of source code, all CVE entries flagged by Eclipse Steady in the modified benchmarks, including those not agreed to by Jaralyzer, are classified as "unknown", meaning that Eclipse Steady requires manual analysis for each case.

None of the compared tools is able to handle type 4 modifications, except for OWASP DC, which still misses 22 CVE entries. For type 4 modifications, Jaralyzer, even though it does not find all CVE entries it detected in the other benchmarks, considerably outperforms the other tools. The missed CVE entries are caused by not being able to rely on the FQNs and thus Jaralyzer having to rely on its re-packaging detection mode with the configured thresholds. To further assess whether modifications affect Jaralyzer's *precision*, we examined whether it reported CVE entries for type 1—4 modifications that were not reported for the unmodified benchmark. This did not occur.

While our results show that no tool within our experimental setup, except Jaralyzer, is able to effectively handle modified dependencies, we are not fully able to confirm the results of Dann et al. Regarding the ability to handle different types of modifications, our results align with those of Dann et al., with e.g. OWASP DC being the only tool to partially support type 4 modifications. However, unlike their findings, we do not observe the same significant performance degradation for modifications that the tools can still partially handle (e.g., types 1–3 for Eclipse Steady), even when accounting for the inaccurate ground truth.

[3] https://issues.apache.org/jira/browse/HTTPCLIENT-1478
[4] https://security.snyk.io/vuln/SNYK-JAVA-ORGSPRINGFRAMEWORK-467268

Finally, we evaluated Jaralyzer on the small benchmark (with adjusted ground truth) used within Dann et al.'s study. Jaralyzer achieved 100% recall and precision across all modification types.

> Among the evaluated tools, Jaralyzer is the only one capable of handling all types of modified dependency inclusions, missing at most 6% of vulnerabilities in type 4 modifications.

4.3 RQ2: *Unmodified* Dependencies Evaluation

This research question considers the scenario where *no* modifications are applied to the dependency inclusions. To evaluate Jaralyzer's detection performance, we directly compare it to the state-of-the-art code-centric dependency scanner Eclipse Steady. We perform a study similar to the one conduced by Ponta et al. [34], where they compare the findings of Eclipse Steady to the findings of OWASP DC and manually review the cases where the tools disagree. In their conducted study they determine that Eclipse Steady significantly outperforms OWASP DC and detects considerably fewer false positives while also finding more true positives. Thus, we compare Jaralyzer to Eclipse Steady.

As reported in Section 4.2, there are 61 CVE entries that both tools report and agree upon for the unmodified benchmark. In this RQ we primarily focus on the cases where the tools *disagree*. We only considered the 664 CVE entries shared across the respective knowledge bases of Jaralyzer and Eclipse Steady.

Tables 2a and 2b present the CVE entries only reported by one of the tools. There are 43 CVE entries, which Jaralyzer reports but Eclipse Steady does not (Table 2a). Conversely, Eclipse Steady reports 44 CVE entries that Jaralyzer does not (Table 2b). As previously noted, we consider CVE entries that are part of both knowledge bases.

For each of the reported CVE entries we performed a manual investigation to determine whether it is a true or false report. To do so we considered four different OSS vulnerability advisories. In particular we considered the GitHub Advisory Database (GA) [17], the National Vulnerability Database (NVD) [27], the Snyk Vulnerability Database (SV) [24], and Project KB (KB) [35]. Although we included the OSV Scanner in our experiments, we do not consider the OSV database in this experiment, as it primarily aggregates existing vulnerability data (e.g. NVD and GA) and does not provide independent classification of new vulnerabilities [19]. In fact, Project KB contains not only fix commits that we used as described in Section 4.1, but also manual assessments indicating whether an artifact is affected by a specific CVE entry. Eclipse Steady uses these manual assessments to enhance its detection performance and cope with its inability to compare source code with bytecode. To not interfere with our experiments, we removed these manual assessments from Eclipse Steady's and Jaralyzer's respective knowledge bases. However we use them for our manual investigation. We consider a CVE report to be a true positive (false positive, resp.) for an artifact, if all advisories that contain the CVE entry agree on the artifact being affected (not affected, resp.) by the given CVE entry. Two authors independently verified whether the advisories listed the respective CVE entry to be affecting the specific artifact and version as reported by either tool. Furthermore, to analyze the causes of disagreement between the tools, we reviewed their individual scan reports and manually inspected the corresponding fix commits to identify potential sources of misclassification. Unlike Jaralyzer, Eclipse Steady produces scan reports with limited details, which makes it difficult to pinpoint the exact reasons behind erroneous results. The reports only list the CVE entries assigned to the scanned artifacts and classify them as either "vulnerable" or "unknown". In

Table 2: Disjunct CVE reports of Jaralyzer and Eclipse Steady for the unmodified benchmark
✓ = true positive; ✗ = false positive

(a) Jaralyzer reports

CVE	KB	GA	NVD	SV
commons-fileupload-1.3.2				
CVE-2016-6793[a]	-	✗	✗	✗
dom4j-1.6.1				
CVE-2020-10683	-	✓	✓	✓
itextpdf-5.5.0				
CVE-2017-9096	-	✓	✓	✓
jackson-databind-2.9.7				
CVE-2019-12086	✓	✓	✓	✓
CVE-2019-12384	✓	✓	✓	✓
CVE-2019-12814	✓	✓	✓	✓
CVE-2019-14379	✓	✓	✓	✓
CVE-2019-14439	✓	✓	✓	✓
CVE-2019-14892	✓	✓	✓	✓
CVE-2019-14893	✓	✓	✓	✓
CVE-2019-16942	✓	✓	✓	✓
CVE-2019-16943	✓	✓	✓	✓
CVE-2019-17267	✓	✓	✓	✓
CVE-2019-17531	✓	✓	✓	✓
CVE-2019-20330	✓	✓	✓	✓
CVE-2020-8840	✓	✓	✓	✓
CVE-2020-9546	✓	✓	✓	✓
CVE-2020-9547	✓	✓	✓	✓
CVE-2020-9548	✓	✓	✓	✓
CVE-2020-10650	✓	✓	✓	✓
CVE-2020-10672	✓	✓	✓	✓
CVE-2020-10968	✓	✓	✓	✓
CVE-2020-10969	✓	✓	✓	✓
CVE-2020-11112	✓	✓	✓	✓
CVE-2020-11113	✓	✓	✓	✓
CVE-2020-11619	✓	✓	✓	✓
CVE-2020-11620	✓	✓	✓	✓
CVE-2020-14060	-	✓	✓	✓
CVE-2020-14061	-	✓	✓	✓
CVE-2020-14062	-	✓	✓	✓
CVE-2020-14195	-	✓	✓	✓
CVE-2020-24616	✓	✓	✓	✓
CVE-2020-24750	✓	✓	✓	✓
jetty-server-9.4.10.v20180503				
CVE-2019-10241	✓	✓	✓	✗
CVE-2019-10247	-	✓	✓	✓
CVE-2019-17632	-	✗	✗	✗
jetty-servlet-9.4.10.v20180503				
CVE-2019-10241	✓	✗	✗	✗
jetty-util-9.4.10.v20180503				
CVE-2019-10241	-	✗	-[b]	✗
CVE-2019-10246	-	✗	-[b]	✗
CVE-2021-44832	-	✗	✗	✗
netty-all-4.0.36.Final				
CVE-2021-21290	-	✓	✓	✓
poi-ooxml-3.14				
CVE-2019-12415	✓	✓	✓	✓
undertow-core-1.4.23.Final				
CVE-2017-2670	-	✗	✗	✓

[a]CVE actually affects Apache Wicket, however, affected file is copied from commons-fileupload

[b]The NVD entry did not contain any version indications

(b) Steady reports

CVE	KB	GA	NVD	SV
bcprov-jdk15on-1.58				
CVE-2015-6644	-	✗	✗	-
CVE-2016-1000338	-	✗	✗	✗
CVE-2016-1000339	-	✗	✗	✗
CVE-2016-1000340	-	✗	✗	✗
CVE-2016-1000341	-	✗	✗	✗
CVE-2016-1000342	-	✗	✗	✗
CVE-2016-1000343	-	✗	✗	✗
CVE-2016-1000344	-	✗	✗	✗
CVE-2016-1000345	-	✗	✗	✗
CVE-2016-1000346	-	✗	✗	✗
CVE-2016-1000352	✗	✗	✗	✗
commons-compress-1.9				
CVE-2019-12402	-	✗	✗	✗
groovy-all-2.4.7				
CVE-2015-3253	✗	✗	✗	✗
hibernate-validator-5.4.1.Final				
CVE-2014-3558	✗	✗	✗	✗
httpclient-4.5.2				
CVE-2013-4366	✗	✗	✗	✗
jackson-databind-2.9.7				
CVE-2017-17485	-	✗	✗	✗
CVE-2018-5968	✗	✗	✗	✗
CVE-2018-11307	✗	✗	✗	✗
CVE-2018-12022	✗	✗	✗	✗
CVE-2018-12023	✗	✗	✗	✗
jackson-dataformat-xml-2.9.3				
CVE-2016-3720	-	✗	✗	✗
jetty-http-9.4.10.v20180503				
CVE-2015-2080	✗	✗	✗	✗
CVE-2017-7657	✓	✗	✓	✗
jetty-server-9.4.10.v20180503				
CVE-2016-4800	-	✗	✗	✗
CVE-2017-7656	✓	✓	✓	✓
CVE-2017-7657	✓	✗	✓	✗
CVE-2017-7658	✓	✓	✓	✓
CVE-2018-12538	-	✓	✗	✗
jetty-util-9.4.10.v20180503				
CVE-2016-4800	-	✗	✗	✗
CVE-2017-9735	-	✗	✗	✗
CVE-2018-12536	-	✗	✗	✓
okhttp-2.7.0				
CVE-2016-2402	✓	✓	✓	✓
spring-core-5.0.4.RELEASE				
CVE-2015-0201	✗	✗	✗	✗
spring-oxm-5.0.4.RELEASE				
CVE-2014-0054	✗	✗	✗	✗
CVE-2014-0225	-	✗	✗	✗
CVE-2014-3578	-	✗	✗	✗
spring-web-5.0.4.RELEASE				
CVE-2013-6429	✗	✗	✗	✗
CVE-2013-6430	-	✗	✗	✗
CVE-2014-0054	-	✗	✗	✗
CVE-2014-3578	-	✗	✗	✗
tomcat-embed-core-8.5.33				
CVE-2020-13934	-	✓	✓	✓
undertow-core-1.4.23.Final				
CVE-2018-14642	-	✓	-[b]	✓
CVE-2019-3888	-	✓	✓	✓
CVE-2020-10705	-	✓	✓	✓

37 of 44 cases, Eclipse Steady marked the entries as "unknown" and requested manual analysis.

Table 2a shows the CVE entries only reported by Jaralyzer, with corresponding artifacts in gray boxes. According to our manual investigation, 35 out of 43 reports are true positives for Jaralyzer and false negatives for Eclipse Steady. Note that 30 of them affect jackson-databind. Eclipse Steady misses those, even though source code is available, because it cannot account for code transformations outside of methods, such as field initializations or initializer blocks, due to its inability to directly compare bytecode with source code. These jackson-databind vulnerability fixes consist of either changing a regular expression stored in a field or adding entries to serialization blocklists in static initializers. Since Jaralyzer operates on bytecode rather than source code, and because the Java compiler inlines field initializations and static initializer code into methods, in contrast to Eclipse Steady, it natively supports such cases.

However, we also uncovered five cases where Jaralyzer reported false positives, according to the advisories. All can be attributed to erroneously reporting a CVE for multiple modules of the same project, due to spurious fix commits that contain changes unrelated to the actual fix as described in Section 4.2, or because of incorrect re-packaging matches due to the threshold configurations. For CVE-2016-6793, which Jaralyzer reported as affecting commons-fileupload-1.3.2, we made an interesting observation. While this CVE entry actually affects Apache Wicket[5], Jaralyzer reports it because it detected a re-packaging of the class affected by the vulnerability. Further investigation revealed that the developers of Apache Wicket copied vulnerable code from commons-fileupload into the Wicket project. The vulnerability within commons-fileupload received its own CVE entry[6], even though they describe the same vulnerability. The CVE entry for commons-fileupload is not part of Project KB and thus not reported by either tool. Therefore, we do not consider this as a clear false positive, but rather as an unclear case. We identified three other unclear cases, where the four advisories reported conflicting information.

To summarize, of the 43 CVE entries reported only by Jaralyzer, we classified 35 as true positives, 4 as false positives and 4 as unclear.

Table 2b contains the CVE entries only reported by Eclipse Steady. According to our investigation, 33 out of 44 are false positives. Only in 7 cases Eclipse Steady reported a true positive that Jaralyzer missed. Furthermore, 4 reports are unclear, since the advisories reported conflicting information. According to Eclipse Steady's scan report, it was unable to automatically classify the false positive cases and instead deferred their evaluation to manual analysis. This might be due to an inability to retrieve the corresponding source code, inaccuracies in the retrieved code or due to ambiguous fix commit matches [40]. Due to the limited details in the scan reports the exact cause cannot be determined. A similar issue appears for the five Jaralyzer reports classified as true positives, which Eclipse Steady missed, excluding the ones related to jackson-databind. The corresponding fix commits involve changes to method bodies and should be thus detectable by Eclipse Steady. However, the scan report does not provide enough details to determine the exact reason for the missed detections.

Using the same methodology, we also validated the 61 CVE reports where both tools agree. According to the advisories, both tools reported 28 true positives and 16 false positives. The 17 remaining cases are unclear due to advisory disagreements.

As noted in Section 4.2, the Achilles benchmark includes a ground truth that we initially excluded from our experiments due to identified inaccuracies. Based on the findings from this research question, we conducted an additional validation of the provided ground truth. Overall, it contains classifications only for 35 of the 148 CVE reports (23%) generated by either Jaralyzer or Eclipse Steady, covering 9 of the 87 cases (9%) where the tools disagreed and 26 of the 61 cases (42%) where they agreed. We further examined the 10 out of 35 ground truth classifications that contradicted the findings of Jaralyzer. This involved a manual review of the corresponding reports, including an analysis of advisory data, inspection of code changes in the relevant fix commits, and investigation of related issue tracker entries. Our analysis indicated that 9 out of the 10 disagreeing classifications were likely incorrect within the ground truth. We reported these 9 discrepancies, as well as the 52 cases absent from Achilles, where Jaralyzer and all considered advisories agreed, to the authors of Achilles, who subsequently incorporated our feedback into the benchmark[7].

> Jaralyzer outperforms Eclipse Steady on unmodified dependencies. It identifies 35 unique true positives while reporting only 4 false positives. In contrast, Eclipse Steady detects only 7 unique true positives but produces 33 false positives.

4.4 RQ3: Runtime Evaluation

We investigated how much time each of the scanners in our experimental setup (see Section 4.1) required to perform a vulnerability scan on the unmodified benchmark. On average, an industry-grade Java project contains 36 dependencies [7]. In comparison, our benchmark incorporates moderately more dependencies than a typical industry-grade project.

Code-centric scanners, in contrast to metadata-based scanners, typically come with significant overhead, which naturally increases the scan time. Jaralyzer, e.g., needs to extract methods from the scanned JAR file, normalize and generate code property graphs for them. These are all complex tasks, which take a non-trivial amount of time. In total, Jaralyzer takes 131 seconds to scan the 56 OSS dependencies in our benchmark. As stated previously, we execute Jaralyzer in its default mode *and* re-packaging detection mode. Executing Jaralyzer's re-packaging mode is only required to detect type-4 modified dependencies. Running only its default mode, Jaralyzer does not miss any vulnerabilities in the unmodified benchmark and only takes 63 seconds. Meanwhile, Eclipse Steady takes 1,387 seconds, requiring more than ten times the scanning time for the benchmark. Due to the lightweight approach behind metadata-based scanners, they require lower runtimes. Specifically, OWASP DC required 9 seconds, OSV Scanner required 5 seconds,

[5] https://security.snyk.io/vuln/SNYK-JAVA-ORGAPACHEWICKET-31022
[6] https://security.snyk.io/vuln/SNYK-JAVA-COMMONSFILEUPLOAD-30401
[7] https://github.com/secure-software-engineering/achilles-benchmark-depscanners/pull/ 44 & 45

and the commercial scanner required 30 seconds. GitHub Dependabot performs the analysis automatically, when a commit containing dependency metadata is pushed to GitHub, where analysis results can then be obtained via a request to the GitHub API. In contrast to the other scanners, this analysis does not run locally, but on the GitHub servers. We thus were unable to measure its precise runtime and do not include it here.

> For projects of above-average size, Jaralyzer takes between 63 and 131 seconds to complete the scan. Overall, Jaralyzer is more than ten times faster than Eclipse Steady.

5 Threats to Validity

A few potential threats might impact the validity of our experiments with Jaralyzer. We used the Achilles benchmark generator to create the individual benchmarks serving as a baseline for our evaluation. Achilles only provides versions of OSS artifacts released by 2021, thus for many artifacts we did not use the latest releases. Furthermore, while we did not use Achilles inaccurate ground truth, we still relied on Achilles correctly modifying the dependency inclusions. Although, our manual investigation revealed these to be most likely correct. The number of CVE entries detected by Jaralyzer within type 4 modifications depends on the configured threshold parameters. Lowering these thresholds may enable Jaralyzer to detect more CVE entries in the type 4 benchmark but could also reduce its precision in the unmodified benchmark. We selected the default values based on empirical testing with our benchmark. However, this approach may lead to overfitting, meaning other datasets might achieve better performance with different values. Jaralyzer was unable to compile 61 CVE entries from Project KB, which we therefore excluded from RQ2. However, in cases where compilation fails or vulnerability fixes lack source code changes, the detection can be augmented by using metadata, as demonstrated by Eclipse Steady. Finally, the effectiveness of code-centric dependency scanning depends on the quality of fix commits. When fix commits include spurious changes unrelated to the actual fix, the dependency scanning performance may be affected.

6 Related Work

Current legislation like the EU's Cyber Resilience Act and the US's Cybersecurity Executive Order require software development companies to focus on the secure usage of OSS components. Therefore many commercial and open-source tools have been developed to detect known-to-be-vulnerable OSS dependencies. Within our study we used the popular metadata-based tools OWASP DC [28], OSV Scanner [18] and GitHub Dependabot [16], as well as the code-centric tool Eclipse Steady [32–34]. Over the years, the term *software composition analysis* (SCA) has emerged to describe dependency scanners, particularly in commercial contexts. Popular commercial SCA tools include Endor Labs SCA [22], Snyk Open Source SCA [23], Mend SCA [26] and Black Duck SCA [15].

Some approaches under the name of patch presence testing attempt to decide whether a patch has been applied to a specific binary. FIBER [50] generates signatures from the security patch's C/C++ source code and searches for its presence in the target binary. Osprey [43] lifts the C/C++ binary into a platform-agnostic intermediate representation (IR) and performs the patch presence test based on this IR. PDiff [20] performs patch presence testing within downstream kernel images by generating semantic summaries of the patch and checking whether the target image is closer to the image before or after the patch is applied. PS^3 [47] uses semantics-level symbolic execution to extract signatures that are stable under different compiler options. It then uses those signatures to test the presence of the patch within the target C/C++ binary. BScout [5] uses feature-extraction and leverages line number information to match lines of Java source code, belonging to the patch, to lines of bytecode within the target file. Doing so, Dai et al. can determine whether the patch has been applied or not. PPT4J [29] employs a similar feature-extraction based approach, which extracts features that are stable through the compilation process and uses these features to determine whether the patch is applied or not. FIBER, PDiff, and PS^3 target C/C++ binaries, BScout and PPT4J address Java applications. In the future the above mentioned approaches could complimentary be integrated into Jaralyzer's vulnerability scanning stage to further aid with the CPG-based patch presence testing. Since they do not perform the initial library matching nor address modifications, we did not include them into our evaluation, since a comparison with dependency scanners aiming for similar goals is more relevant and adequate.

Some approaches explicitly target third-party library (TPL) identification for Android to find licensing issues or vulnerable components usages. OSSPolice [9] finds license violations and security risks within Android apps by maintaining a database of unique app features and comparing the target app to this database to determine the exact TPL used. ATVHunter [48] applies control-flow graph based feature-extraction to more reliably identify the correct TPL and version. PHunter [45] relies on special obfuscation-resilient features to determine whether patches are applied to an Android TPL, even with code obfuscations applied.

In recent years, researchers have published different approaches to identify security-relevant commits and resulting fix-commit databases. Ponta et al. [35] created a manually-curated set of fix commits, called Project KB, focusing on industry-relevant OSS projects. Bhandari et al. [2] propose an approach that automatically extracts fix commits from CVE entries within the NVD. They create the CVEFixes database, containing fix commits for 5,365 CVE entries affecting OSS projects of various programming languages. Sabetta et al. [37] propose the rule-based tool Prospector, which tries to automatically find and rank fix commits for specified vulnerability identifiers. Akhoundali et al. [1] propose an approach that uses Prospector and create the MoreFixes database, containing fix commits for 26,617 CVEs affecting a diverse set of OSS projects.

7 Conclusion

This paper presents Jaralyzer, a novel *bytecode-centric* dependency scanner for Java. While there are many open-source and commercial dependency scanners available, none of the existing tools is able to effectively deal with modified dependencies. Jaralyzer is able to largely overcome these challenges. By adopting an approach independent of the dependency's metadata or source code availability and instead directly analyzing its bytecode, Jaralyzer

effectively detects known-to-be-vulnerable dependencies in Java projects, even across all common types of modifications.

An evaluation performed on 56 popular OSS artifacts, including five state-of-the-art dependency scanners, showcased that, while all other tools suffer performance deterioration when dependencies are modified, Jaralyzer's performance remains stable. In few cases, it was even able to exploit the modifications and report fewer false positives. But even when dependencies are unmodified, Jaralyzer outperforms the state-of-the-art code-centric dependency scanner Eclipse Steady, by detecting more vulnerabilities and producing fewer false alerts. These results position Jaralyzer to be the code-centric dependency scanner of choice for Java projects, especially when it comes to identifying modified dependencies.

Acknowledgement

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (GZ: SFB 901/3) under the project number 160364472.

References

[1] Jafar Akhoundali, Sajad Rahim Nouri, Kristian Rietveld, and Olga Gadyatskaya. Morefixes: A large-scale dataset of cve fix commits mined through enhanced repository discovery. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 42–51, 2024.

[2] Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39, 2021.

[3] Benjamin Bowman and H Howie Huang. Vgraph: A robust vulnerable code clone detection system using code property triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69. IEEE, 2020.

[4] The MITRE Corporation. About the CVE Program, 2024. Accessed 2024-12-13. URL: https://www.cve.org/About/Overview.

[5] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. {BScout}: Direct whole patch presence test for java executables. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1147–1164, 2020.

[6] Andreas Dann, Ben Hermann, and Eric Bodden. Sootdiff: Bytecode comparison across different java compilers. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, pages 14–19, 2019.

[7] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for oss vulnerability scanners-a study & test suite. *IEEE Transactions on Software Engineering*, 48(9):3613–3625, 2021.

[8] Jens Dietrich, Shawn Rasheed, Alexander Jordan, and Tim White. On the security blind spots of software composition analysis. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 77–87, 2024.

[9] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke 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*, pages 2169–2185, 2017.

[10] Douglas Everson, Long Cheng, and Zhenkai Zhang. Log4shell: Redefining the web attack surface. In *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*, pages 1–8, 2022.

[11] OWASP Foundation. OWASP Top Ten, 2021. Accessed 2024-12-13. URL: https://owasp.org/www-project-top-ten/.

[12] The Apache Software Foundation. Apache Maven Dependency Plugin, 2024. Accessed 2024-12-16. URL: https://maven.apache.org/plugins/maven-dependency-plugin.

[13] The Apache Software Foundation. Apache Maven Shade Plugin - Relocating Classes, 2025. Accessed 2025-06-23. URL: https://maven.apache.org/plugins/maven-shade-plugin/examples/class-relocation.html.

[14] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 38–47. IEEE, 2017.

[15] Black Duck Software Inc. Black Duck SCA, 2024. Accessed 2024-12-13. URL: https://www.blackduck.com/software-composition-analysis-tools/black-duck-sca.html.

[16] GitHub Inc. GitHub Dependabot Alerts, 2024. Accessed 2024-12-13. URL: https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts.

[17] GitHub Inc. GitHub Advisory Database, 2025. Accessed 2025-06-01. URL: https://github.com/advisories.

[18] Google Inc. OSV Scanner, 2024. Accessed 2024-12-13. URL: https://google.github.io/osv-scanner/.

[19] Google Inc. OSV - Frequently Asked Questions, 2025. Accessed 2025-06-23. URL: https://google.github.io/osv.dev/faq/#ive-found-something-wrong-with-the-data.

[20] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1149–1163, 2020.

[21] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. Sootup: A redesign of the soot static analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–247. Springer, 2024.

[22] Endor Labs. Endor Labs SCA, 2024. Accessed 2024-12-13. URL: https://www.endorlabs.com/use-cases/reachability-based-sca.

[23] Snyk Limited. Snyk Open Source SCA, 2024. Accessed 2024-12-13. URL: https://snyk.io/product/open-source-security-management/.

[24] Snyk Limited. Snyk Vulnerability Database, 2025. Accessed 2025-01-10. URL: https://security.snyk.io/.

[25] Jeff Luszcz. Apache struts 2: how technical and development gaps caused the equifax breach. *Network Security*, 2018(1):5–8, 2018.

[26] Mend.io. Mend SCA, 2024. Accessed 2024-12-13. URL: https://www.mend.io/sca/.

[27] National Institute of Standarads and Technology. National Vulnerability Database, 2025. Accessed 2025-01-10. URL: https://nvd.nist.gov/.

[28] OWASP. OWASP Dependency-Check, 2024. Accessed 2024-12-13. URL: https://owasp.org/www-project-dependency-check/.

[29] Zhiyuan Pan, Xing Hu, Xin Xia, Xian Zhan, David Lo, and Xiaohu Yang. Ppt4j: Patch presence test for java binaries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[30] Henrik Plate. State of Dependency Management 2023, 2023. Accessed 2024-12-13. URL: https://www.endorlabs.com/learn/state-of-dependency-management-2023.

[31] Henrik Plate and Darren Meyer. 2024 Dependency Management Report, 2024. Accessed 2024-12-13. URL: https://www.endorlabs.com/lp/2024-dependency-management-report.

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

[33] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–460. IEEE, 2018.

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

[35] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 383–387. IEEE, 2019.

[36] RertireJS. Retire.js, 2024. Accessed 2024-12-16. URL: https://retirejs.github.io/retire.js/.

[37] Antonino Sabetta, Serena Elisa Ponta, Rocio Cabrera Lozoya, Michele Bezzi, Tommaso Sacchetti, Matteo Greco, Gergő Balogh, Péter Hegedűs, Rudolf Ferenc, Ranindya Paramitha, et al. Known vulnerabilities of open source projects: Where are the fixes? *IEEE Security & Privacy*, 2024.

[38] Stefan Schott, Wolfram Fischer, Serena Elisa Ponta, Jonas Klauke, and Eric Bodden. Compilation of commit changes within java source code repositories. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 325–336. IEEE, 2024.

[39] Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. Java bytecode normalization for code similarity analysis. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[40] SAP SE. Eclipse Steady - Library Assessment, 2021. Accessed 2025-06-25. URL: https://eclipse-steady.github.io/steady/user/manuals/library_assessment/.

[41] Inc. Sonatype. Maven Central Repository - Requirements, 2025. Accessed 2025-07-01. URL: https://central.sonatype.org/publish/requirements/#supply-javadoc-and-sources.

[42] Matúš Sulír and Jaroslav Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25, 2016.

[43] Peiyuan Sun, Qiben Yan, Haoyi Zhou, and Jianxin Li. Osprey: A fast and accurate patch presence test framework for binaries. *Computer Communications*, 173:95–106, 2021.

[44] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, 2017.

[45] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. Precise and efficient patch presence test for android applications against code obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 347–359, 2023.

[46] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.

[47] Qi Zhan, Xing Hu, Zhiyang Li, Xin Xia, David Lo, and Shanping Li. Ps3: Precise patch presence test based on semantic symbolic signature. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[48] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang 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)*, pages 1695–1707. IEEE, 2021.

[49] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 176–187, 2019.

[50] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, 2018.