Identifying Challenges for OSS Vulnerability Scanners - A Study & Test Suite

Andreas Dann*, Henrik Plate†, Ben Hermann‡, Serena Elisa Ponta†, Eric Bodden*§

Abstract—The use of vulnerable open-source dependencies is a known problem in today's software development. Several vulnerability scanners to detect known-vulnerable dependencies appeared in the last decade, however, there exists no case study investigating the impact of development practices, e.g., forking, patching, re-bundling, on their performance.

This paper studies *(i)* types of modifications that may affect vulnerable open-source dependencies and *(ii)* their impact on the performance of vulnerability scanners. Through an empirical study on 7,024 Java projects developed at *SAP*, we identified four types of modifications: re-compilation, re-bundling, metadata-removal and re-packaging. In particular, we found that more than 87% (56%, resp.) of the vulnerable Java classes considered occur in Maven Central in re-bundled (re-packaged, resp.) form. We assessed the impact of these modifications on the performance of the open-source vulnerability scanners OWASP Dependency-Check (OWASP) and Eclipse Steady, GitHub Security Alerts, and three commercial scanners. The results show that none of the scanners is able to handle all the types of modifications identified. Finally, we present *Achilles*, a novel test suite with 2,505 test cases that allow replicating the modifications on open-source dependencies.

Index Terms—Security maintenance, Open-Source Software, Tools, Security Vulnerabilities.

1 Introduction

The use of open-source software (OSS) is an established practice in software development, even for industrial applications as much as 75% of the code comes from OSS [1], [2], [3], [4]. At the same time, more than 67% of the applications include vulnerable OSS with on average 22 individual vulnerabilities [1].

Vulnerabilities in widely-used OSS, e.g., Jackson, Apache Commons, or Struts, already proved to have serious consequences. An (in)famous example is the Equifax breach [5], [6], which was caused by the vulnerability CVE-2017-5638 [7] in Apache Struts2.

To detect vulnerable OSS, research and industry have developed several open-source vulnerability scanners, e.g., the open-source tools OWASP Dependency-Check (OWASP) and Eclipse Steady, the free tool GitHub Security Alerts, and commercial tools such as Snyk, Black Duck, or WhiteSource.

Since vulnerabilities in open-source dependencies pose a high risk, scanners should detect them with high precision and recall. However, developers and distributors frequently fork, patch, re-compile, re-bundle, or re-package existing OSS [2], [3], [4], [8]. As a result, the same vulnerable code may occur in different, modified dependencies, thereby posing a challenge for the detection of known-vulnerable OSS [9], [10], [11].

Previous studies [1], [2], [3], [8], [12] investigate to which extent open-source or industrial applications include (vulnerable) OSS. However, they do not study modifications, like patching, re-compiling, or re-packaging, that may affect vulnerable open-source dependencies, nor their impact on the performance of vulnerability scanners. Furthermore, the studies do not present data sets or test suits that facilitate the comparison and evaluation of vulnerability scanners.

While existing benchmarks such as the Evaluation Framework for Dependency Analysis [13] allow one to evaluate single features of vulnerability scanners, e.g., dependency resolution, they do not provide a ground truth for assessing their performance. Moreover, they do not contain test cases for modifications such as re-compiled and re-packaged classes.

This paper studies the types of modifications that may affect open-source dependencies and investigates their impact on the performance of vulnerability scanners.

We conducted a two-folded case study on 7,024 Java projects developed at *SAP*, the world's third-largest software development company. First, we scanned the 7,024 Java projects with Eclipse Steady, OWASP Dependency Check, and a commercial vulnerability scanner to gain an in-depth understanding of the use of open-source dependencies at *SAP*. We applied three different vulnerability scanners that use different vulnerability databases to avoid being subject to the shortcomings of a particular scanner or database. Our study shows that the projects include about 79% of OSS transitively, supporting the studies by Pashchenko et al. [12], [14]. To investigate the prevalence of vulnerable OSS, we classify the vulnerabilities reported for

- * *The authors are with the Chair for Secure Software Engineering, Heinz Nixdorf Institute, Paderborn University, Germany.*
E-mail: ⟨First Name⟩.⟨Last Name⟩@uni-paderborn.de
- † *The authors are with SAP Security Research Mougins, France.*
E-mail: ⟨First Name⟩.⟨Last Name⟩@sap.com
- ‡ *The author is professor for Secure Software Engineering, Technical University of Dortmund, Germany.*
E-mail: ben.hermann@cs.tu-dortmund.de
- § *The author is Director for Software Engineering and IT-Security at Fraunhofer IEM, Paderborn, Germany.*
E-mail: eric.bodden@iem.fraunhofer.de

Manuscript received July, 2020;

the most-used 20 dependencies into true- and false-positives in semi-manual reviews. A major finding is that the projects do not only include unmodified OSS distributed by the original OSS-project but also include code that has been altered in some way and is re-distributed in the context of other (downstream) projects. In our study, we observed four modification types: re-compilation, re-packaging, metadata-removal, e.g., MANIFEST files, Uber-JARs, or combinations of those.

Second, we investigated the prevalence of these modifications on the public OSS repository Maven Central, and their impact on vulnerability scanners. On Maven Central, we found cases of Uber-JARs (87%), re-compilation (56%), re-packaging, and metadata-removal (57%) for the vulnerabilities identified in the first part. To evaluate the modifications' impact on vulnerability scanners, we further chose a representative set of 16 vulnerabilities, applied the identified modifications, and evaluated the performance of three commercial vulnerability scanners[1], GitHub Security Alerts, and the open-source scanners OWASP and Eclipse Steady.

Our study shows that the identified modifications are a major challenge for the detection of vulnerable OSS as none of the scanners considered is able to handle all types of modification. Thereby, our work highlights the need for further research in the area to improve the detection algorithms and methods implemented by vulnerability scanners.

To facilitate a reproducible and comparative assessment of vulnerability scanners, this paper also introduces *Achilles*, a novel test suite to replicate modified dependencies. Together with *Achilles*, we also provide 2,505 test cases (labeled true- and false-positives) for 723 distinct open-source artifacts, all observed versions of the 20 most-used artifacts, and 249 distinct vulnerabilities. We derived the test cases directly from our case study.

To summarize, this work makes the following contributions:

- a case study on the use of OSS in industrial Java projects developed at *SAP* (Section 5),
- a case study on the prevalence of modifications of OSS dependencies and their impact on the performance of vulnerability scanners (Section 6),
- a novel test suite for assessing vulnerability scanners, *Achilles*, derived from our study (Section 8)

Achilles, along with our case study, is made available as an open-source project[2].

The paper is structured as follows. Section 2 explains the terminology. Section 3 discusses related work. Section 4 presents our research questions and methodology. Section 5 investigates the use of OSS at SAP. Section 6 studies the prevalence and impact of the found modifications. Section 7 summarizes and discusses our observations. Section 8 introduces *Achilles*. Section 9 discusses threats to validity and Section 10 concludes.

2 Background & Terminology

In this paper, we rely on the established terminology used by the well-known build-automation tool Maven.

We use the term *dependency* for a software library or framework, which is a separately distributed software artifact. In Java, a dependency is commonly distributed as a JAR file, which logically groups a set of classes and resources.

Build-automation tools like Maven, Gradle, and Ant+Ivy automate the process of including OSS as dependencies. All these tools use a similar syntax for declaring dependencies and pull JARs from private or public artifact repositories. The most popular public repository for Java is Maven Central with more than 4.7 million open-source artifacts and over 70 million downloads per week [15].

To declare a dependency, developers specify in a project's `pom.xml` file: the `groupId` identifying the vendor; the `artifactId` identifying the component; and the `version` of the OSS. This triple is referred to as *GAV*.

One can distinguish between release dependencies, which are shipped with the application, and development-only dependencies, which are only used during development, e.g., for testing. To declare a release dependency developers specify dependency with the scope: *compile, runtime, or system*. Whereas dependencies with the scope *test* or *provided* are development-only dependencies, e.g., the JUnit testing framework. Since only release dependencies are shipped with an application, vulnerabilities in development-only dependencies are not exploitable in production.

Dependencies are also distinguished into *direct* and *transitive*. A dependency is called *direct* if developers declare it explicitly in the `pom.xml`. A dependency is called *transitive* if it is not explicitly declared but automatically included by other dependencies.

The complete set of direct and transitive dependencies of a project is called Bill of Materials (BoM).

3 Related Work

3.1 Studies: Use of Vulnerable OSS

Most related is the empirical study on the use of vulnerable open-source dependencies in industrial Java applications developed at SAP by Pashchenko et al. [12], [14]. The authors applied the code-based vulnerability matching approach of Eclipse Steady to investigates the risk of vulnerable, transitive OSS from the perspective of the developers. In particular, the authors study how many vulnerable dependencies can be fixed by project developers themselves, whereas this paper investigates project metrics affecting vulnerability scanners. They found that even if the majority of vulnerabilities is located in transitive dependencies, developers can fix – in fact – 80% of the vulnerable release dependencies either by fixing a bug in a single OSS project or by updating a direct dependency [12]. The authors provide a new methodology for assessing the impact of vulnerable, transitive dependencies on a project by grouping (transitive) vulnerabilities that can be "easily" fixed by updating a direct one. In contrast to our study, they do not investigate modified OSS, their prevalence, and their impact on vulnerability scanners.

Further reports by Synopsis [16], and Williams and Dabirsiaghi [17] investigate the use of OSS in an industrial context. The reports show that more than 67% of the investigated applications include on average vulnerable

[1] Due to license restrictions, we cannot disclose their names.

[2] https://github.com/secure-software-engineering/achilles-benchmark-depscanners

dependencies with 22.5 vulnerabilities per dependency [16]. While the studies emphasize the need for open-source vulnerability scanners, they do not investigate modified OSS nor their impact on scanners' performance.

Kula et al. [8] and Bavota et al. [4] investigate vulnerable dependencies in the context of open-source projects and how developers update dependencies. Kula et al. [8] found that the majority (81.5%) of studied projects include outdated or vulnerable open-source dependencies. Both studies conclude that most developers are reluctant to update dependencies or unaware of new versions.

Analogous to our observation of modified OSS JARs, Lopez et al. [18] found that source-code clones and modified source-code occur in 80% of all studied open-source project.

None of the existing studies focuses on modifications that impact the performance of vulnerability scanners nor the prevalence of modified OSS. Our case study complements existing work and shows that – on top of source-code clones – vulnerable code clones are also introduced during the build process of downstream projects by re-bundling and re-packaging.

3.2 OSS Vulnerability Databases & Scanners

The most widespread vulnerability database is the National Vulnerability Database (NVD) [19]. The NVD links a vulnerability (CVE) to a set of operating systems, hardware, or software components using the Common Platform Enumeration (CPE) standard. Although the NVD is the main source for vulnerabilities, false-negatives easily arise because the NVD is not complete and the set of CPEs does not always contain all affected artifacts. False-positives also arise because the CPEs over-approximate the affected versions or specify a complete application instead of the affected library [20]. For instance, the CPEs of CVE-2018-1271 contain the complete Spring framework `pivotal_software:spring_framework`, whereas only a single library is vulnerable – `spring-webmvc` [21]. Additionally, the CPEs use a different granularity and schema than build-automation tools, like Maven or NPM.

To cope with the shortcomings of the NVD, additional (commercial) databases such as the Eclipse Steady database [10], the Exploit Database [22], or WhiteSource's vulnerability database [23] have been created. However, these databases differ w.r.t. the affected artifacts and versions, and thus no common ground truth exists.

To identify known-vulnerable dependencies, vulnerability scanners build a project's BoM and query a vulnerability database if known vulnerabilities for the found dependencies exist. For matching a dependency against the entries in a vulnerability database different strategies exist. The most popular techniques are name-based and code-based matching [9], [11].

OWASP, which relies on the NVD, and GitHub Security Alerts, for instance, apply name-based matching to identify known-vulnerable dependencies. They extract from a project's `pom.xml` (and JARs), for each dependency the vendor, product name, and version, and use fuzzy-matching to compare it against the CPEs in the NVD or WhiteSource's vulnerability database [23]. Such an approach fully relies on the information present in the database and the correctness of the metadata (vendor, name, version) in the `pom.xml` and JARs.

Eclipse Steady, for instance, applies code-based matching [9], [11]. The scanner checks a JARs' bytecode for known vulnerabilities. To do so, Eclipse Steady computes the digest of each JAR and the fully-qualified name (FQN) of all classes and methods. The scanner then uses Steady's Database to check for vulnerabilities that affect the found FQNs. Such an approach requires the creation of a separate database that contains all the vulnerable software constructs (e.g., constructors, methods, initializers, and their FQNs) for each vulnerability. To this end, for each disclosed vulnerability the commits fixing that vulnerability must be identified to derive the vulnerable constructs.

Note that the matching approaches and databases used by commercial vulnerability scanners are not made publicly available. Commercial scanners may use a combination of those or rely on additional features for matching, e.g., file digests, timestamps, bytecode, etc.

In our study, we apply OWASP, Steady, and a commercial scanner C3 to balance the shortcomings of one particular matching strategy or a single database – as the used scanners rely on different databases.

3.3 Vulnerability Benchmarks

Most related to *Achilles* is the SourceClear benchmark [13]. The benchmark provides test cases invoking the vulnerable code of an open-source dependency for Java, Scala, Ruby, Python, C, C#, JavaScript, PHP, and Go. However, the benchmark does not replicate modifications, such as re-compiled class-files, re-packaging, deleted or lost metadata. The benchmark's ground truth does *not* specify the vulnerable dependency nor the published vulnerabilities. Thus, it cannot be utilized to evaluate a vulnerability scanner's precision and recall.

Vulnerability collections and benchmarks for assessing the performance of security scanners are BugBox [24] for PHP, the SAMTE reference data set [25], SecuriBench [26], and the Juliet TestSuite [27]. The benchmarks provide test cases with known security flaws for evaluating application security testing tools but do not provide test cases for detecting known-vulnerable OSS.

Wide-spread benchmarks like the DaCapo [28] benchmark suite, the Qualitas Corpus [29], or the XCorpus [30] provide test corpora for evaluating static analyses but do not provide test cases for detecting known vulnerabilities in OSS.

Existing benchmarks and vulnerability databases like the NVD [19] and ExploitDB [22] do not provide a curated collection of vulnerable OSS, do not specify the vulnerable classes and methods, nor provide ground truth. Thus, they are insufficient to assess vulnerability scanners. *Achilles* and its ground truth extend and complement existing databases and benchmarks by aggregating vulnerable OSS and specifying the vulnerable classes.

4 Study Design

4.1 Research Questions

The purpose of this paper is to gain an in-depth understanding of the settings in which vulnerability scanners have to operate and the impact of modified OSS on their performance. To gain such an in-depth understanding of real-world situations and processes case studies are a suitable mean [31]. The observations that we conclude from our study can be seen as hypotheses or be the basis on which new hypotheses, studies, and experiments can be formulated and conducted.

Our case study is two-folded. In the first part, we investigated the settings regarding the use of OSS in an industrial context, in particular, *SAP*. We investigated **RQ1: What are the practices in using OSS at *SAP*?** We evaluated project metrics that influence the construction of a complete BoM, which is a necessary first step before checking for vulnerable dependencies. In particular, we computed the number of (direct and transitive) dependencies a project includes on average. This is the number of dependencies a vulnerability scanner has to analyze. Further, we compared the ratio of direct to transitive dependencies. As developers only include direct dependencies explicitly, vulnerabilities are less likely to be discovered in manual reviews if they are located in transitive dependencies [8], [14]. Next, we checked the ratio of release to development-only dependencies. Since only release dependencies would be available in production, only vulnerabilities in those are exploitable. Finally, we determined the ratio of open-source to proprietary dependencies. Since vulnerability scanners check for known-vulnerable OSS, investigating the extent of OSS helps understanding how critical the performance of vulnerability scanners is.

To investigate the prevalence of vulnerable OSS, we investigated the following questions. We selected a representative set of the 20 most-used dependencies at *SAP* first, and then checked **RQ2: What vulnerabilities affect the most-used dependencies?** Therefore, we semi-manually evaluated for 723 distinct open-source artifacts, all observed versions of the 20 most-used dependencies, what vulnerabilities affect them. This semi-manual classification also serves as the basis for *Achilles*.

As already stated by Ponta et al. [9], [11], the metadata of a dependency is often invalid or may be missing as developers fork, patch, re-compile, re-bundle, or re-package existing open-source artifacts [2], [3], [4], [8]. To elaborate this observation, we investigated **RQ3: How do developers include OSS?** We classified the observed modifications into four different types.

In the second part, we further investigated the identified modification types. First, we checked **RQ4: How prominent are the modifications outside *SAP*?** In particular, we evaluated how often the modifications that we identified in RQ3 occur on Maven Central.

To evaluate **RQ5: What is the impact of the modifications on vulnerability scanners**, we compared the performance of six open-source and commercial vulnerability scanners w.r.t. the identified modifications. To do so, we computed for each scanner its precision, recall, and F1-score when analyzing modified OSS.

4.2 Study Objects & Methodology

We conducted an industrial case study [31] on 7,024 different Java projects developed at *SAP* to answer the research questions.

Table 1: 20 most-used dependencies, grouped by GA

#Projects	GA (groupId, artifactId)	Rank [32]
3211	commons-codec:commons-codec	25
3026	org.slf4j:slf4j-api	2
2899	com.fasterxml.jackson.core:jackson-annotations	50
2854	com.fasterxml.jackson.core:jackson-core	45
2851	com.fasterxml.jackson.core:jackson-databind	15
2831	org.apache.httpcomponents:httpcore	79
2781	commons-logging:commons-logging	20
2774	org.apache.httpcomponents:httpclient	21
2662	com.google.code.gson:gson	19
2617	org.springframework:spring-core	47
2574	org.springframework:spring-beans	54
2533	org.springframework:spring-context	23
2518	org.springframework:spring-aop	-
2503	org.springframework:spring-expression	-
2495	commons-io:commons-io	6
2371	org.apache.commons:commons-lang3	13
2133	org.springframework:spring-web	55
2105	com.google.guava:guava	4
2046	javax.validation:validation-api	73
1895	org.springframework:spring-webmvc	70

4.2.1 Studied Projects & Project Metric Extraction

The investigated 7,024 different Java projects cover a wide range of industrial enterprise-applications, platforms, in-house tools, microservices, and monoliths. In particular, we investigated the BoMs of each project created by the vulnerability scanner Eclipse Steady, which uses the API of the build-automation system Maven [11]. The generated BoMs contain the project's GAV (uniq. identifier) and a complete description of the used dependencies. In particular, the BoM states for each dependency (and its JAR file) the GAV, the scope, whether it is direct or transitive, the filename, and SHA1. Further, we investigated the use of modified OSS and classified the observed modifications into four different types.

The initial data set consisted of 49,752 different BoMs generated by Eclipse Steady. Since the BoMs were generated during the build process, the data set included a separate BoM for each version of a project. To balance our data set, we only included the latest BoM of each project. As a result, the filtered data set consisted of the BoMs of 7,024 distinct projects (projects with distinct GAs).

4.2.2 Dependency Selection

For answering the research questions 2, 4, and 5, we investigated in semi-manual reviews what vulnerabilities affect a given dependency and what Java classes contain the vulnerable code. To keep the workload for these semi-manual reviews manageable, we decided to limit ourselves to 20 OSS library projects. To ensure industrial relevance, we selected the sample set from the data set as follows : we only considered release dependencies, dependencies with scope *compile* and *runtime*, grouped them by their GA, counted how many projects include a dependency with the given GA, and, finally, selected the 20 most-used ones, shown in Table 1.

To study the relevance of our sample set within the open-source community, we checked if the selected 20 most-used dependencies are also popular on Maven Central. Table 1 shows their ranking among the 100 most-used dependencies, using the statistics provided by MvnRepository [32] in column *Rank*. The table shows that 18 out of the 20 most-used dependencies are within the 79 most-used dependencies on Maven Central. Only two dependencies, `spring-expression` and `spring-aop`, are not within the 100 most-used.

The table also shows that the libraries' popularity within *SAP* and Maven Central slightly differs. One reason for this deviation could be the fact that the MvnRepository ranking includes dependencies with the scope *test* (in total 8) and Kotlin, Scala, and Closure dependencies (in total 13), which we excluded.

To study how often the sample set is used within open-source projects, we extracted the number of usages for all 723 artifacts (GAVs) – all observed versions of the 20 most-used dependencies – as reported by MvnRepository [32], resulting in the log-normal distribution in Figure 1. The figure shows that the dependencies in the sample set are also regularly used in open-source projects.

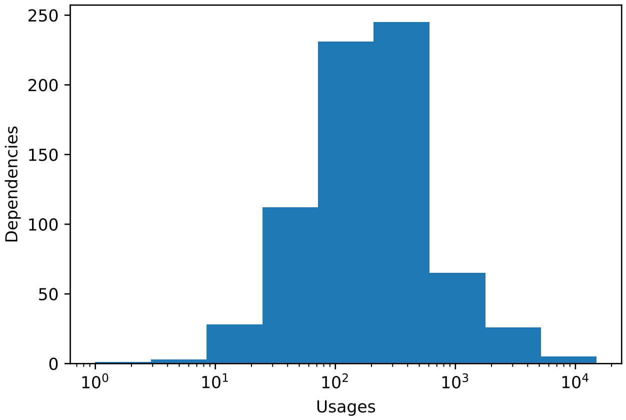

Figure 1: The #usages of the sample 723 distinct artifacts (GAVs) as reported by mvnrepository.com.

4.2.3 Vulnerable Dependency Identification

As described in Section 3, all current approaches for matching vulnerabilities and OSS are prone to false-positives and false-negatives. Moreover, the databases are not complete w.r.t. the reported vulnerabilities and the set of CPEs, resulting in false-negatives.

To reduce the likelihood of false-positives and false-negatives, we used three different vulnerability scanners for identifying vulnerable dependencies: the open-source scanner Eclipse Steady and OWASP, and the commercial scanner C3. The scanners apply different matching strategies: Eclipse Steady applies a code-based matching, whereas OWASP uses name-based matching. For C3 there is no public information available describing the underlying approach. By choosing these scanners our results are based on three different vulnerability databases: Eclipse Steady uses its open-source database [10], [33], OWASP uses the NVD [19], and C3 uses a commercial database. Thereby, we aim to improve the validity of our results and *Achilles* by balancing out the shortcomings of one particular database.

Finally, we classified the scanners' reports in semi-manual reviews (cf. Section 5.2) into true- and false-positives. In total, we classified 723 distinct artifacts (GAVs) – considering all versions of the selected GAs – and 2,505 reports.

4.2.4 Identification of Modifications on Maven Central

To assess the prevalence of the identified modifications outside *SAP*, we check how often they occur on Maven Central. To do so, we first checked for the vulnerabilities identified in RQ2 what class-files (contained in the dependency's JAR) are vulnerable by investigating commits fixing those vulnerabilities. Second, we computed how often the identified vulnerable classes occur in modified form on Maven Central w.r.t. the identified modification types 1–4.

To check if the bytecode of a vulnerable class matches the found classes on Maven Central, we compared their bytecode using the tool SootDiff [34]. SootDiff's comparison was specifically designed to be resistant to changes induced by various compilation schemes, and thus allows us to check for bytecode equivalence even if a modification has been applied to one of the classes.

5 Use of OSS at *SAP*

In this section, we address the research questions RQ1, RQ2, and RQ3 by investigating the practices regarding the use of OSS in an industrial context. To this end, we conducted an empirical study on 7,024 Java projects developed at *SAP*, one of the world's largest software development companies.

5.1 RQ1: What are the practices in using OSS at *SAP*?

To answer RQ1, we first computed the average **number of dependencies per project**. Therefore, we calculated the number of distinct dependencies' GAVs (uniq. identifier: vendor, component, version), calculate the arithmetic mean, and standard deviation [35]. We found that, on average, a project includes 94.78 direct and transitive dependencies with distinct GAVs, with a standard deviation of 124.61. The high standard deviation shows that the size of a BoM heavily varies among projects.

Counting the distinct GAVs seems simple, however, it may suffer from several issues described by Pashchenko et al. [14]. For instance, if a developer declares a dependency on `spring-context`, its transitive dependencies with the same groupId are counted as separate dependencies. This overweights dependencies that are released as multiple JARs, e.g., frameworks like Spring or Struts. To overcome this issue, we count the number of distinct groupIds per project as proposed by Pashchenko [14]. This resulted in 36.34 ($sd = 35.84$) direct and transitive dependencies per project, with a median of 25. The number of dependencies highly varies per project, ranging from 0 to 228 dependencies. Unless stated otherwise, we use this grouping for the remainder of the paper.

Second, we computed the **ratio of direct to transitive dependencies**, shown in Figure 2. The figure shows that only 21% are direct dependencies, whereas 79% are transitive.

More than 50% of the studied projects incorporate at least the framework Spring or Struts. Although we only count distinct GAs (uniq. identifier: vendor, component), these frameworks include a high number of transitive dependencies with different `groupIds`. For instance, the framework `spring-boot-starter:2.17` introduces 17 dependencies with various groupIds, e.g., `javax.*`, `ch.qos.logback`, `org.apache.*`. The high number of transitive dependencies is caused by the fact that each dependency has dependencies on its own, and so on. Since all *but* the direct dependencies of the project itself are considered transitive by Maven, their amount is high in the studied projects.

Third, we investigated how many dependencies would be deployed with a release. Therefore, we computed the ratio of scopes declared by the developers (cf. Figure 2). The results show that most dependencies 58.1% have the default scope *compile*, and thus are available during compilation and runtime. 4.9% have the scope *runtime*, and thus are available on the runtime classpath. 20.2% have the scope *provided/system*, and thus are available at runtime only. 16.8% have the scope *test*, and thus are development-only dependencies, not deployed in production. The high number of dependencies with the scopes *runtime* and *provided/system* shows that 25.1% of the dependencies are not shipped with the application but are pre-installed or provided by the system on which the application is executed, which may differ from the build system. However, vulnerability scanners are executed on the build system, and thus are unable to identify vulnerabilities in dependencies that are provided later – during runtime.

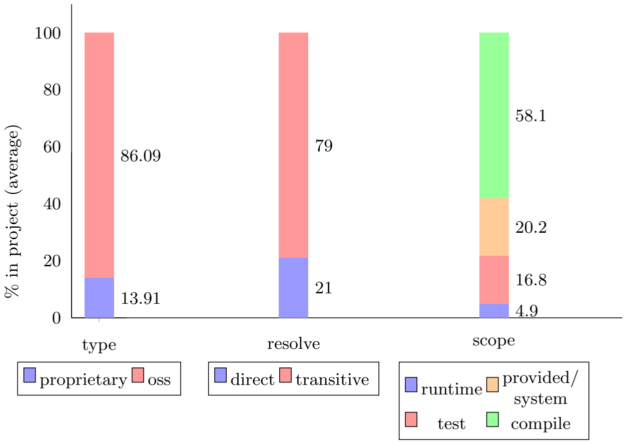

Figure 2: BoM metrics for dependencies with distinct GA

Fourth, to check to **what extent OSS** is used in an industrial context, we computed the ratio of open-source to proprietary dependencies. To distinguish between open-source and proprietary dependencies, we classified each dependency that is hosted on Maven Central as open-source and any other as proprietary. Figure 2 shows the results. As other repositories also exist, e.g., Sonatype, JCenter, or Redhat JBoss, the computed ratio is only a lower bound.

On average, 86.09% of the dependencies are open-source, whereas only 13.91% are proprietary. Moreover, the use of open-source dependencies is ubiquitous as 95.43% of the projects include at least one OSS library.

> On average, an industrial Java project includes 36 direct and transitive dependencies, 86% of them being open-source. The majority of dependencies (79%) are transitive and (58.1%) are release dependencies.

5.2 RQ2: What vulnerabilities affect the most-used dependencies?

We investigated how many vulnerabilities affect the 20 most-used dependencies that we selected as described at Section 4.2.2. As vulnerabilities affect specific version ranges, we considered all versions of the dependencies in our data set. In total, our data set contains 723 different artifacts (GAVs) for the 20 most-used dependencies (GA).

To identify known-vulnerable dependencies, we used the vulnerability scanners: Eclipse Steady, OWASP, and C3. Since these scanners rely on different vulnerability databases we aim to improve the validity of the results, as described in Section 4.2.3. As input for those scanners, we created for each GAV a separate Maven project with a direct dependency on that GAV, including all optional dependencies.

Table 2 shows the number of findings separated per tool per GA and the number of distinct reported vulnerabilities. The column *Dependency* shows the GAs that were declared as a direct dependency in the created Maven projects. The column *Reported vulnerable Dependency* shows the (direct or transitive) dependency that was reported as being vulnerable. The column *#Dist. Findings* contains the number of found vulnerabilities. The brackets contain the number of distinct artifacts (GAVs) which were reported. Highlighted cells indicate cases in which the scanners reported the direct dependency as vulnerable.

Table 2 includes both true- and false-positives, which we later classified in semi-manual reviews. Moreover, Eclipse Steady and C3 also reported vulnerabilities that were not officially reported to the NVD, and thus do not have a *common vulnerabilities and exposures* (CVE) number, e.g., bugs or vulnerabilities disclosed in bug trackers. Though we considered these vulnerabilities in Table 2, we ignored them for *Achilles*, as different scanners may name the same bug differently or may not consider them as vulnerabilities, thereby hindering a fair comparison. In total, the scanners reported 249 distinct CVEs for 527 of the 723 different artifacts (GAV). Since a single CVE usually affects multiple versions, e.g., CVE-2016-3720 affects `jackson-dataformat-xml` version 2.0.0 to 2.7.4, the scanners generated 2,505 distinct findings (GAV, vulnerability).

The table shows that scanners reported the most vulnerabilities for transitive dependencies. Note that the vulnerability scanners did not report anything for `commons-codec`, `commons-io`, `commons-logging`, `commons-lang3`, `gson`, `httpcore`, `jackson-annotations`, `jackson-core`, and `slf4j-api`, which are therefore omitted.

> Steady, OWASP, and C3 generated 2,505 findings for 723 different artifacts and 249 CVEs. The majority of vulnerabilities affect transitive dependencies.

To investigate the ratio of true-positive and false-positive reported vulnerabilities, we semi-manually classified the 2,505 reports, using the following procedure:

1. We checked if the NVD [19] or Eclipse Steady's database contains a reference to a commit, issue, or pull request fixing the reported vulnerability. We successfully found, in total, 96 source-code commits and identified 254 vulnerable classes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3101739, IEEE Transactions on Software Engineering

Table 2: Sample of TP & FP Vulnerabilities reported by Eclipse Steady, OWASP Dependency-Check, C3

Dependency (#GAVs*)	Reported vulnerable Dependency (#GAVs*)	#Dist. Findings			
		#Dist. Vuln.	Steady	OWASP	C3
guava (35)	guava (25)	1	12	25	25
httpclient (22)	httpclient (18)	7	4	9	40
jackson-databind (54)	groovy (1)	2	4	0	0
	jackson-databind (53)	16	227	203	218
spring-aop (61)	spring-core (52)	25	17	374	29
spring-beans (61)	groovy-all (8)	2	30	18	24
	spring-core (52)	25	23	29	375
spring-context (60)	bsh (2)	1	60	60	55
	hibernate.validator (2)	1	0	4	0
	groovy-all (10)	2	45	28	34
	hibernate-validator (5)	2	7	8	7
	jruby (4)	6	1	18	2
	jsoup (1)	1	0	2	0
	spring-core (51)	25	28	368	26
	spring-expression (46)	2	92	0	0
spring-core (62)	commons-collections (1)	4	5	10	8
	spring-core (53)	24	20	390	31
spring-expression (57)	spring-core (48)	25	4	29	11
	spring-expression (10)	2	20	0	0
spring-web (46)	axis (1)	3	0	9	0
	axis-saaj (1)	3	0	9	0
	commons-fileupload (3)	6	50	49	9
	commons-httpclient (1)	1	3	0 0	
	groovy-all (8)	2	18	12	17
	guava (1)	1	9	9	9
	httpasyncclient (1)	1	0	1	0
	httpclient (7)	5	26	5	24
	jackson-databind (28)	13	396	134	134
	jackson-dataformat-xml (20)	5	23	90	14
	jetty-http (21)	9	116	192	74
	jetty-security (15)	1	33	0	0
	jetty-server (20)	6	148	0	109
	jetty-servlet (19)	1	38	0	0
	jetty-util (20)	4	75	0	68
	netty-all (9)	2	10	15	4
	okhttp (2)	1	3	3	3
	org.apache.axis (1)	1	3	0	0
	protobuf-java (1)	1	0	31	31
	spring-core (43)	25	22	305	20
	spring-expression (37)	2	74	0	0
	spring-oxm (14)	4	24	0	8
	spring-web (44)	12	42	0	105
	taglibs (1)	1	0	1	1
	tomcat-embed-core (7)	8	30	15	32
	undertow-core (4)	6	12	0	44
spring-webmvc (45)	bcprov-jdk14 (2)	14	180	481	37
	bcprov-jdk15on (1)	3	16	24	16
	castor (1)	1	0	5	0
	commons-beanutils (3)	2	88	45	43
	commons-collections (4)	4	18	46	21
	commons-compress (1)	1	33	0	33
	commons-fileupload (1)	5	5	0	0
	dom4j (1)	1	1	0	0
	groovy-all (8)	2	17	12	15
	guava (1)	1	24	25	25
	itextpdf (1)	2	2	0	3
	jackson-databind (28)	13	407	139	139
	jackson-dataformat-xml (19)	5	21	85	13
	jasperreports (8)	6	0	68	0
	lucene-queryparser (1)	1	29	0	0
	ognl (1)	1	43	44	44
	poi (8)	7	35	107	72
	poi-ooxml (3)	24	24	0	24
	spring-core (38)	23	12	20	251
	spring-expression (33)	2	66	0	0
	spring-oxm (13)	5	12	0	19
	spring-tx (1)	1	1	0	1
	spring-web (40)	13	75	0	100
	spring-webmvc (38)	8	28	28	99
validation-api (5)	bsh (1)	1	4	0	0

* GAV - a unique identifier in Maven for an artifact at a specific version
in orange - direct dependency

2. If we found a commit, we checked if the reported JAR file contains the vulnerable bytecode.

2.1. To do so, we first determined the vulnerable method(s), class(es), static initializer(s), changed by the commit using Eclipse Steady and its database [11]. Thereby, we used the bytecode of the vulnerable artifacts that Steady identified as vulnerable for further comparison.

2.2. Second, we compared the bytecode of the vulnerable classes, methods, or static initializer with the bytecode contained in the reported JAR using SootDiff [34].

2.3. If the bytecode of at least one class, method, or static initializer matched the vulnerable code, we classified the finding as true-positive. In these steps, we classified 428 reports as true-positive, 792 as false-positive, and left 1285 for further investigation.

3. If we did not find a commit or SootDiff's comparison failed, we searched the NVD and Eclipse Steady's database for links to issue boards or bug trackers.

4. If we found a link to an issue or bug tracker, we checked whether the description states the vulnerable artifacts and versions.

4.1. If a description existed and matched the reported artifact, we classified the finding as true-positive.

4.2. If a description existed but did not match the reported artifact or the NVD entry referred to a different artifact, we classified the finding as false-positive. In total, we classified 306 reports as true-positive, 821 as false-positive, and 158 as ambiguous.

5. If we could not find a link or the description was ambiguous, we checked the set of CPEs in the NVD [19].

5.1. If the CPE exactly matched the reported artifact, we classified the finding as true-positive.

5.2. If the CPE did not match, we classified the finding as false-positive. Finally, we classified 15 reports as true-positive, 141 as false-positive, and 2 as ambiguous.

We could not classify two findings using the steps above: one finding referenced CVE-2013-5855 and one referenced CVE-2014-7810. For these findings, we looked into the two reported JARs manually and found that they only contain the API (abstract classes, and interfaces) of the vulnerable artifacts but no implementation. Thus, we classified them as false-positive. In total, we classified 859 (34%) as true-positive, and 1,646 as false-positive of the 2,505 reports.

> We classified 859 of the 2,505 findings in reviews as true-positive, and the rest as false-positive.

5.3 RQ3: How do developers include OSS?

In our study, we observed *four types* of modification that may affect OSS dependencies.

Unmodified: Most commonly, we observed that developers include an OSS directly from Maven Central using its plain GAV.

Type 1 (patched): We noticed that developers include OSS with a slightly modified GAV, e.g., `com.google:guava:-23.0_fix3`. We investigated that these type 1 dependencies occur if developers or distributors fork the source-code of an OSS and modify or patch it slightly, and indicate these changes by appending a suffix string like `fix` to the version. The JAR file does not contain the original bytecode but the (modified) re-compiled source-code, changing the classes' digests and timestamps [9], [11].

In our study, we observed the modifications re-bundling, metadata-removal, and re-packaging along with so-called Uber-JARs – sometimes also called fat-JARs. Uber-JARs merge multiple OSS artifacts into a single JAR to ease, for instance, deployment or distribution. In the following, we further elaborate on Uber-JARs and how they relate to the observed modifications re-bundling, metadata-removal, and re-packaging.

Type 2 (Uber-JAR): We found projects that include a few dependencies with GAVs that do not indicate which OSS the JAR file contains, e.g., `com.my:servicebundle:1.0`.

Such dependencies re-bundle multiple OSS (and transitive dependencies) into a single JAR file, so-called *Uber-JAR*. Examples are, for instance, the `jar-with-dependencies` files, which can be commonly found on Maven Central and can be generated with Maven, e.g., the assembly or the shade plugin. Note that in contrast to type 1, these plugins preserve the original bytecode, digest, and timestamp.

For *Uber-JAR*s, we found two further sub-types.

Type 3 (bare Uber-JAR): In rare cases, multiple OSS dependencies are merged into a *Uber-JAR*, but the `pom.xml` files, the folders `META-INF`, and file timestamps are removed. Since Maven's shade and assembly plugin preserve the `pom.xml` by default, this case is supposedly relevant for legacy Uber-JARs built before the advent of these plugins, e.g., with Ant.

Type 4 (re-packaged Uber-JAR): Similar to type 2, but the *Uber-JAR* contains re-packaged classes, i.e., classes with a string prepended to the original class name. Here, the classes' bytecode, digest, and timestamp are changed. Such re-packaging can be configured with the Maven shade plugin and is usually used to avoid name clashes.

> We identified four types of modification: patched, Uber-JAR, bare Uber-JARs, and re-packaged Uber-JARs.

6 Prevalence & Impact of modified OSS

In this section, we address the research questions RQ4 and RQ5. Therefore, we investigate the prevalence of the identified modifications on Maven Central and their impact on the performance of vulnerability scanners.

6.1 RQ4: How prominent are the modifications outside *SAP*?

To check that the observed modifications are not specific to our data set, which is based on Java projects developed at *SAP*, we computed their prevalence on Maven Central. As a basis for detecting modifications, we used the vulnerable open-source dependencies that we identified in RQ2. In particular, we investigated how often the vulnerable code of the found vulnerable dependencies is subject to modification types 1–4. To do so, we first identified which classes are changed in the 96 commits fixing the 249 CVEs that we identified in RQ2 (Section 5.2). Table 3 shows the prevalence of type 1-4 modifications for the 254 vulnerable classes, which we identified based on the 96 commits, from 38 open-source dependencies hosted on Maven Central.

Table 3: Prevalence of Modifications on Maven Central

	Type 1	Type 2	Type 3	Type 4
# classes subject to	143	222	222	17
# affected GAV*	5,919	36,609	24,500	168
# affected GA†	360	6,728	3,882	89

* GAV - distinct versions; † GA - distinct vendor, artifact

Type 1 (patched) Patching and forking an OSS usually only changes a subset of the classes in a JAR. However, it requires the re-compilation of all classes. To measure how prevalent type 1 modifications are, we, thus, checked how often the vulnerable classes (a subset of the classes in a JAR) have been re-compiled on Maven Central. Note that we fail to identify an artifact as a form of modification type 1, if during patching or forking the source-code of a class has been modified such that its compiled bytecode differs so much from the original, vulnerable bytecode that SootDiff's comparison fails. Thus, our results are only a lower bound.

Since re-compilation may change a class' bytecode and SHA1 but not the fully-qualified name (FQN), we used the following approach. We checked how many classes on Maven have identical FQN as the 254 vulnerable classes and equivalent bytecode, according to SootDiff [34], but a different SHA1. We found 50,702 artifacts on Maven Central that contain a class with at least one FQN. Further, we found for 143 (56%) of the 254 classes re-compiled versions on Maven Central that have a different SHA1 but equivalent bytecode. In total, we found such duplicates in 5,919 (11.6%) artifacts. These artifacts spread across 360 distinct GAs.

Type 2 (Uber-JAR) Uber-JARs do not change the bytecode nor the metadata, e.g., timestamp, SHA1, but only merge the files contained in multiple JARs into a single JAR. Consequently, to check the prevalence of *Uber-JARs* we computed how frequently the vulnerable classes (same FQN and identical SHA1) are copied into artifacts with distinct GAs. We found that 36,609 artifacts on Maven Central contained copies of the 254 vulnerable classes. We identified re-bundling for 222 out of the 254 classes (87%) across 6,723 artifacts with different GA. Further, we found that commonly classes are re-bundled in two or three artifacts with different GAs (with quartiles Q1: 2, Q2: 2 Q3: 3), thus re-bundling often occurs within the same groupId and artifactId. However, we also found at max that the class `org.bouncycastle.math.ec.custom.sec.SecP256R1Curve` was re-bundled in 27 artifacts with distinct GAs.

Type 3 (bare Uber-JAR) Further analysis showed that 3,882 (57%) out of those 6,723 artifacts do not contain a `pom.xml` in the `META-INF` folder, and thus are *bare*.

Type 4 (re-packaged Uber-JAR) To avoid name-clashes between classes, Uber-JARs may also re-package classes by prepending a string to the original FQN before merging them. Since the FQNs are embedded within a class' bytecode, the bytecode itself and the SHA1 also change.

To cope with changed bytecode, we checked how often classes with an identical filename and nearly equal bytecode, in terms of local-sensitive hash distances [36], are contained in JARs with different GAs. We found that 16,665 artifacts contain a class with the same filename as a vulnerable class but a different FQN, for 174 of the 254 classes. To check if the bytecode is similar to one of a vulnerable class, we compared the bytecode using SootDiff and computed local-sensitive hashes (TLSH) [36]. If the TLSH distance was lower than 20, we considered it as re-packaging. In total, we found re-packaging for 17 classes in 89 distinct GAs.

> Re-bundling, re-packaging, and re-compilation are common in industrial and open-source projects. We found that more than 87% of the checked classes are re-bundled, and more than 56% are re-compiled on Maven Central. Thus, vulnerabilities reported for one OSS actually affect many other projects as well.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3101739, IEEE Transactions on Software Engineering

6.2 RQ5: What is the impact of the modifications on vulnerability scanners?

To assess the impact of the identified modifications on the performance of vulnerability scanners, we evaluate the performance of the open-source vulnerability scanners OWASP and Eclipse Steady, of GitHub Security Alert, and the commercial scanners C1, C2, and C3[1] w.r.t. these modifications.

As test cases, we selected from the 20 most-used open-source dependencies the seven that were themselves vulnerable, and selected their most recent vulnerabilities (cf. Table 2). Table 4 shows the test cases that we used as input. Column *Reported By* shows which scanners reported them and column *TP* shows whether they are true- or false-positives. As an example, all vulnerabilities for `spring-core` are false-positives. Since we created the tests based on the results of OWASP, Eclipse Steady, and C3, two of the scanners reported the dependency as vulnerable but the manual inspection showed that the vulnerabilities affect `spring-web`.

Table 4: Test Cases: Artifact, Vulnerability, Classification

Artifact	Vulnerability	TP	Reported By
httpclient 4.1.3	CVE-2012-6153	yes	Steady, C3
	CVE-2014-3577	yes	Steady, OWASP, C3
	CVE-2015-5262	yes	Steady, OWASP, C3
jackson-databind 2.9.7	CVE-2018-19362	yes	Steady, OWASP, C3
	CVE-2018-19361	yes	Steady, OWASP, C3
	CVE-2018-19360	yes	Steady, OWASP, C3
spring-webmvc 5.0.0.RELEASE	CVE-2018-1271	yes	Steady, OWASP, C3
spring-core 5.0.5.RELEASE	CVE-2018-1258	no	OWASP, C3
	CVE-2018-11039	no	OWASP
	CVE-2018-1257	no	OWASP
	CVE-2018-11040	no	OWASP
spring-expression 5.0.4.RELEASE	CVE-2018-1270	yes	Steady, OWASP
	CVE-2018-1275	yes	Steady, OWASP
spring-web 5.0.5.RELEASE	CVE-2018-15756	yes	Steady, OWASP, C3
	CVE-2018-11039	yes	Steady, OWASP, C3
guava 23.0	CVE-2018-10237	yes	Steady, OWASP, C3

We evaluated the vulnerability scanners' performance for the four modification types that we discovered in our study (cf. Section 5.3). For all types, we used the same test cases (cf. Table 4) but applied different modifications using *Achilles*.

Unmodified All dependencies keep their original GAV, their metadata is preserved, they are kept as separate JAR files, and the bytecode is not modified, providing an anchor for comparing the modifications' impact.

Type 1 (patched) All dependencies get a slightly modified GAV (appending the string `fix` or `patch`), the metadata is preserved, they are kept as separate JAR files, but the classes are re-compiled.

Type 2 (Uber-JAR) All dependencies are merged into a single Uber-JAR with a random GAV, the metadata of the original artifacts is preserved, and the original bytecode and the timestamps of the original files are untouched.

Type 3 (bare Uber-JAR) All dependencies are merged into a single Uber-JAR with a random GAV, all metadata is removed (manifest files, `pom.xml`), the original classes are kept, but the timestamps of the files are updated.

Type 4 (re-packaged Uber-JAR) All dependencies are merged into a single Uber-JAR with a random GAV, the metadata is kept, and the original classes are re-packaged changing the bytecode and the class-files timestamps.

For each type, we generated the modified JAR file(s) and a Maven project (`pom.xml`) declaring the respective GAV(s) as dependencies. Then we executed the vulnerability scanners on each project and computed their precision, recall, and F1-score. Table 5 shows the results. The table shows that the scanners heavily differ even for unmodified JARs. The commercial scanner C2 does not find any vulnerabilities in types 1–4, and thus, seems to be unable to deal with modifications at all. GitHub Security Alerts does not find any vulnerabilities in types 2–4, and detects the same vulnerabilities for unmodified and type 1 JARs. C3 performs similar to GitHub Security Alerts but with higher precision and recall. Based on Table 5, Security Alerts, C2, and C3 seem to rely heavily on metadata to detect vulnerable artifacts, as they do not detect vulnerabilities in types 2–4.

The results further show that the commercial scanner C1 fails to detect vulnerabilities in type 4. OWASP and C1 are the only scanners reporting vulnerabilities in type 4. Eclipse Steady performs best for unmodified JARs. The table shows that Eclipse Steady performs better for type 2 and 3 than for type 1. Thus, re-compilation and patching decrease the performance more than lost or modified metadata.

Note that *Achilles* is based on the results obtained from OWASP, Eclipse Steady, and C3. The fact that Eclipse Steady achieves perfect precision and recall for unmodified JARs means that all findings that we use in the case study were marked as true-positives in the manual review, and there were no false-negatives w.r.t. the findings of OWASP, and C3. However, the results show that the performance of all scanners heavily decreases with the modifications.

All scanners struggle to identify vulnerable dependencies if the JAR files are modified (type 1-4).

7 Discussion

In the following, we summarize and discuss the observations that we made in our study. To do so, we link them to previous studies, where applicable.

Vulnerable OSS The results for RQ1 and RQ2 emphasize that the use of OSS is established practice, even in industrial applications. This aligns with previous studies [1], [8], [10], [14], [17], which studied the use of vulnerable OSS in industrial and open-source applications.

Remarkably, the amount of vulnerable dependencies, which we identified, is lower than the amount presented in the studies [1], [8] who state that each application contains 22.5 different vulnerabilities on average and 81.5% of the applications use outdated dependencies.

Scalability The results for RQ1 show that the number of dependencies heavily differs per project, ranging from 0 to 228. Additionally, RQ1 shows that vulnerability scanners must not only check dependencies with the scope *compile* but also must check all release (transitive) dependencies as they constitute a relevant share. Previous studies [1], [8], [14], which point out the importance of transitive dependencies, validate this observation.

Detection Performance The classification in RQ2 shows that vulnerability scanners tend to produce a large number of false-positives, as only 859 from 2,505 findings are true-positives. While in an early development phase, updating

Table 5: Vulnerability Scanners' metrics for Type 1–4; * used in the construction of the test cases; **bold** the highest score

	Unmodified			Type 1 (patched)			Type 2 (Uber-JAR)			Type 3 (bare Uber-JAR)			Type 4 (re-pack. Uber-JAR)		
	precision	recall	F1	precision	recall	F1	precision	recall	F1	precision	recall	F1	precision	recall	F1
OWASP*	0.34	0.92	0.50	0.35	**0.92**	0.51	0.17	0.17	0.17		–		0.17	0.17	0.17
Eclipse Steady*	**1.00**	**1.00**	**1.00**	0.38	0.75	0.50	**0.41**	0.75	**0.53**	**0.41**	**0.75**	**0.53**		–	
Security Alerts	0.60	0.50	0.55	0.60	0.50	0.55		–			–			–	
C1	0.64	0.58	0.61	0.32	0.58	0.41	0.34	**0.96**	0.51		–		**0.78**	**0.58**	**0.67**
C2	0.75	0.75	0.75		–			–			–			–	
C3*	0.71	0.83	0.77	**0.71**	0.83	**0.77**		–			–			–	

a dependency is relatively unproblematic, whereas updates during the release or the operational lifetime impact the schedule, may cause downtimes, or introduce unexpected defects [4], [11]. Thus, the number of false-positive must be kept low. Similarly, RQ5 shows that scanners highly divert in performance even for unmodified OSS.

Modified JAR Files A major finding in RQ3 and RQ4 is the fact that industrial applications, as well as open-source projects, include modified OSS. Since the modifications repackaging, re-bundling, re-compilation, Uber-JARs, modify a dependency's metadata, e.g., GAV, digest, as well as the code, they pose a major challenge for vulnerability scanners.

RQ5 shows that current scanners struggle to identify vulnerabilities in modified OSS. Only the scanners C2 and Eclipse Steady provide reasonable results in the presence of re-compilation and lost metadata. Since RQ3 and RQ4 show that all modifications (type 1–4) occur at *SAP* and Maven Central, future development must address the related challenges. The case study shows that all scanners need to improve further w.r.t. their performance, especially concerning the modification types 3 and 4. Remarkably, our case study shows that – even the vulnerability scanners used in the construction of the test cases – fail to deal with *Modified JAR Files*.

8 Achilles - Test Suite

As, to the best of our knowledge, no test suite exists to replicate challenges for vulnerability scanners, we developed *Achilles*. *Achilles* allows replicating the challenges *Detection Performance* and *Modified JAR Files* for evaluating the performance of vulnerability scanners. Therefore, *Achilles* provides the options to include (vulnerable) OSS dependencies, remove vulnerable classes, and apply the modifications 1–4. As an input *Achilles* uses a set of GAVs and a ground truth, stating the vulnerabilities. With Achilles, we provide 2,505 test cases and ground truth, derived from our study.

Since benchmarks play a strategic role in computer science research and development by providing a ground truth for evaluating algorithms and tools, we constructed *Achilles* based on the following criteria, introduced by the widespread DaCapo benchmark [28].

Diverse real-world applications: The test cases should not consist of artificially created programs. Instead, the benchmark should contain dependencies and vulnerabilities collected from real-world projects to provide a compelling focus for evaluating real-world usage.

Detecting vulnerable OSS (precision and recall): The test cases and ground truth should enable measuring if a vulnerability scanner successfully detects included dependencies with published vulnerabilities (recall) and to what extent a scanner raises false warnings (precision).

Automation and ease of use: The test cases should be in a format consumable by vulnerability scanners and enable the measurement of their accuracy.

In the following, we explain how *Achilles* implements these criteria, its organization, and its use.

8.1 Diverse real-world application

To closely resemble real-world applications, we directly select the test cases from our study. We thus choose as test data the 2,505 findings from *RQ2* for the 723 artifacts and 249 distinct vulnerabilities, which we (semi-)automatically classified into true- and false-positives (cf. Section 5.2). Further, test cases can be easily added as JSON files.

As we identified that the same open-source dependencies are also used in the open-source community, the test cases not only replicate the settings at *SAP* but also in open-source projects.

To account for the identified modifications, *Achilles* allows to apply these modifications to the test data before serving them as input to a vulnerability scanner.

8.2 Detecting vulnerable OSS

For evaluating a vulnerability scanner's precision and recall, each test case is specified as a human-readable JSON file, stating the published vulnerability, the (affected) GAV, a short description, and the ground truth. Moreover, each test case also contains a timestamp specifying when the ground truth was updated. Listing 1 shows exemplary a test case for the vulnerability *CVE-2016-3720*.

```
"cve": "CVE-2016-3720", "timestamp": "2018-11-30",
"gav": {"version": "2.4.3", ,"artifactId":
     "jackson-dataformat-xml", "groupId":
     "com.fasterxml.jackson.dataformat"},
"vulnerable": true, "comment" :"XML Injection",
     "details": [{"contained": true, "affectedFile":
     "com/fasterxml/jackson/dataformat/xml/XmlFactory.class",
"fqn": "com.fasterxml.jackson.dataformat.xml.XmlFactory"}]
```

Listing 1 Test (jackson-dataformat-xml, CVE-2016-3720)

Optionally, a test case also contains the FQN of the vulnerable methods, classes, the commit, as well as SootDiff's result if the artifact contains the vulnerable bytecode – we could identify 96 source-code commits and included the SootDiff's results (cf. Section 5.2).

8.3 Automation and ease of use

Achilles provides a graphical user interface to compose a consumable Maven project (`pom.xml`), the ground truth, and for applying the modifications.

For applying *Achilles* to evaluate the performance of a vulnerability scanner the process is as follows. First, users select the test cases (vulnerabilities and GAVs) that they want to use. Next, users choose if the JAR files should be modified. Building on top of the four types of modifications identified in *RQ3*, users can choose the following options, shown in Figure 3 as a feature diagram:

GAV To evaluate to what extent vulnerability scanners are resilient to simple changes in the GAV (*type 1*) or to a novel GAV due to re-bundling (*type 1–4*): the original GAV can be kept, can be modified by appending a version suffix (e.g., the string `-fix-01`) or replaced by a random GAV.

Metadata To evaluate to what extent name-based scanners are resilient to modified metadata (*type 3*): the metadata, in particular the `pom.xml` file in the `META-INF` folder, can be removed or maintained.

JAR To encompass *Uber-JAR* (*type 2–4*): all artifacts can be kept as separate JAR files, or bundled into a single *Uber-JAR*. Optionally, the original timestamps of the files in the JAR can be kept, when re-bundling or re-compiling them.

Classes To encompass re-compilation (*type 1, 4*): the original class-files can be copied, the source-code can be re-compiled using the original FQN, or the classes can be re-package by prepending the string `com.repackage` to each FQN.

Vulnerable Code Optionally, *Achilles* can remove the vulnerable classes from a JAR file. This can be used to evaluate if a vulnerability scanner correctly checks whether the vulnerable code is contained in a given JAR.

By default *Achilles* applies the settings highlighted in Figure 3, producing unmodified JAR files.

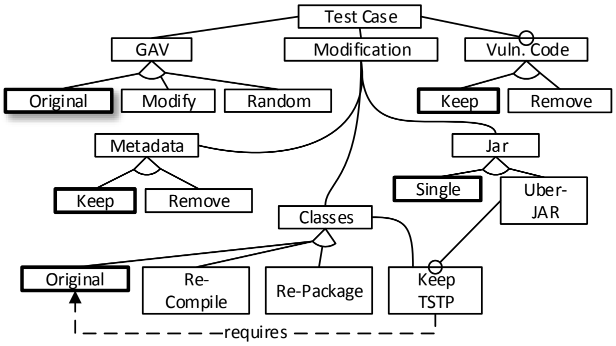

Figure 3: *Achilles* Feature Diagram - Test Case Generation; highlighted are the default settings (unmodified JAR)

Based on the chosen configuration, *Achilles* creates a Maven project consisting of a `pom.xml` with the (modified) JAR files as dependencies. The generated project can be used directly as an input for a vulnerability scanner. To measure a scanner's precision and recall, its findings are compared with the ground truth.

8.4 Organization and Distribution

To allow the community to extend *Achilles* and provide further test cases, we distribute *Achilles* publicly on GitHub.[2]

9 Threats to Validity

9.1 Use of OSS at *SAP*

Since we conducted our study on projects developed at *SAP*, the applicability to industrial Java projects, in general, is limited as the impact of development practices, tools, and guidelines must be evaluated. The studied projects already apply Eclipse Steady, which may bias our results, as the development teams may update dependencies regularly. Nevertheless, *SAP* is one of the world's leading software development companies, with a diverse product portfolio and our study aligns with previous open-source and industrial case studies [1], [8], [10], [12], [14], [17]. Additionally, the 20 most-used dependencies are also popular within the open-source community (cf. Section 4.2), and the modifications also occur on Maven, indicating that our results are also applicable to other – particularly to open-source – projects.

Our decision to check vulnerabilities for the 20 most-used OSS and the choice of the scanners influences the number of findings, false-positive, and false-negative. Since we semi-manually classified the 2,505 findings to achieve soundness, we had to restrict ourselves. To limit the likelihood of false-negative, we used the scanners Steady, OWASP, and C3. They apply different matchings: name-based vs. code-based; and rely on different databases: Steady's Database [37], NVD [19], and a commercial database. All scanners and databases are continuously updated and cannot guarantee the absence of false-negative or false-positives. Thus, the absolute number of false-positives may differ with the chosen scanners. Nevertheless, our results regarding the use of vulnerable OSS and the ratio of false-positives align – or are even less – than reported by previous studies [1], [8], [10], [14].

9.2 Prevalence & Impact of modified OSS

The prevalence of the modifications is based on the vulnerable classes that we identified in *RQ2*. This selection may be inaccurate because we do not know how often these vulnerable classes are re-compiled or re-bundled compared to other classes, e.g., if the found classes are more often or rarely re-bundled. Hence, our results may under- or over-approximate but still show that these modifications are popular on Maven Central and must be addressed by vulnerability scanners.

A potential pitfall is the fact that the evaluated vulnerability scanners Steady, OWASP, and C3 were also used in the creation of *Achilles*. Although we semi-manually classified the 2,505 findings to ensure soundness, we cannot ensure completeness, as false-negatives (findings missed by all scanners) may occur.

A bias towards Eclipse Steady may only occur in the case of unmodified JAR, as the tests used to create the benchmark were confirmed as correct during the semi-manual review. For modified OSS, instead, the generator creates new dependencies with new metadata, e.g., digest, GAV. As the modified dependencies are unknown to all scanners, they allow fair comparison.

The relevance of modifications is independent of any particular scanner, as even the scanners used in the construction fail – to different degrees – to deal with modified JARs. Especially, the good performance of C1 for type 4 shows that the study does not favor the scanners that have been used in *Achilles*' creation. Foremost, the results show that the performance of all scanners heavily decreases with modification, and must be addressed in the future.

10 Conclusion

The paper presents a case study investigating the use of (modified) vulnerable OSS and their impact on vulnerability scanners. We conducted a case study on 7,023 Java projects developed at *SAP* regarding the use of OSS. We found that the majority of dependencies (86%) are OSS, and that most dependencies are included transitively (79%). The (in)security of OSS is thus a major issue for Java projects. Further, our study shows that modified OSS, compromising re-compiled, re-packaged, or re-bundled classes from other open-source projects, commonly occur on Maven Central.

We found that such modifications heavily decrease the precision and recall of vulnerability scanners by checking the performance of the open-source scanners Dependency-Check and Eclipse Steady, GitHub Security Alerts as well as three commercial tools. The results show that all vulnerability scanners struggle to cope with modified OSS, and that further research and development are needed.

To facilitate comparisons w.r.t. the identified modifications, we present *Achilles* – a novel test suite for evaluating the performance of vulnerability scanners. *Achilles* is comprised of 2,505 test cases, directly derived from our case study. *Achilles* is the first test suite designed to foster research and development by enabling evaluation and comparison of vulnerability scanners. *Achilles* is publicly available, evolving, and open for feedback and contributions.

References

[1] M. Pittenger, "The State of Open Source Security in Commercial Applications," Black Duck Software, Tech. Rep., 2016.

[2] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," in *Proceedings of the 12th International Conference on Top Productivity through Software Reuse*, ser. ICSR'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 207–222.

[3] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, ser. ICSM'12. USA: IEEE Computer Society, 2012, p. 483–492.

[4] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, oct 2015.

[5] Forbes, "Equifax," feb 2017. [Online]. Available: https://www.forbes.com/sites/thomasbrewster/2017/09/14/equifax-hack-the-result-of-patched-vulnerability/

[6] B. Krebs. (2018) Equifax Breach. [Online]. Available: https://krebsonsecurity.com/tag/equifax-breach/

[7] NVD. (2017, feb) Cve-2017-5638. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-5638

[8] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, feb 2018.

[9] 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.

[10] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. IEEE Press, 2019, p. 383–387.

[11] 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 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 449–460.

[12] 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.

[13] SourceClear. (2020) Evaluation framework for dependency analysis. [Online]. Available: https://github.com/srcclr/efda

[14] 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*, ser. ESEM '18. New York, NY, USA: ACM, 2018.

[15] Sonatype. (2020, feb) Central download statistics for OSS projects. [Online]. Available: https://blog.sonatype.com/2010/12/now-available-central-download-statistics-for-oss-projects/

[16] BlackDuck, feb 2020. [Online]. Available: https://www.blackducksoftware.com/technology/vulnerability-reporting

[17] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," Constrast Security, Tech. Rep., 2014.

[18] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "DéjàVu: a map of code duplicates on GitHub," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, oct 2017.

[19] NIST. (2020) Nvd. [Online]. Available: https://nvd.nist.gov/

[20] V. H. Nguyen and F. Massacci, "The (un)reliability of NVD vulnerable versions data: An empirical experiment on Google Chrome vulnerabilities," in *ASIA CCS 2013 - Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. New York, New York, USA: ACM Press, 2013, pp. 493–498.

[21] Pivotal Software. (2020, Dec) CVE-2018-1271. [Online]. Available: https://pivotal.io/security/cve-2018-1271

[22] OffSec Services, "Exploit database," feb 2020. [Online]. Available: https://www.exploit-db.com/

[23] GitHub, "Security alerts," feb 2020. [Online]. Available: https://help.github.com/articles/about-security-alerts-for-vulnerable-dependencies/

[24] G. Nilson, K. Wills, J. Stuckman, and J. Purtilo, "Bugbox: A vulnerability corpus for PHP web applications," in *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*. Washington, D.C.: USENIX, 2013.

[25] NIST. (2018) Samate - software assurance metrics and tool evaluation. [Online]. Available: https://samate.nist.gov/

[26] B. Livshits. (2012) Securibench. [Online]. Available: https://suif.stanford.edu/~livshits/securibench/

[27] NIST. (2017, oct) Juliet test suite. [Online]. Available: https://samate.nist.gov/SARD/testsuite.php

[28] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," *SIGPLAN Not.*, vol. 41, no. 10, pp. 169–190, Oct. 2006.

[29] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," in *2010 Asia Pacific Software Engineering Conference*. IEEE, nov 2010, pp. 336–345.

[30] J. Dietrich, H. Schole, L. Sui, and E. Tempero, "XCorpus - An executable corpus of Java programs," *Journal of Object Technology*, vol. 16, pp. 1–24, 2017.

[31] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, apr 2009.

[32] MvnRepository. (2020, feb) 100 popular projects. [Online]. Available: https://mvnrepository.com/popular

[33] SAP. (2020, feb) Vulnerability Assessment Knowledge Base. https://github.com/SAP/project-kb.

[34] A. Dann, B. Hermann, and E. 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*, ser. SOAP 2019. New York, NY, USA: ACM, 2019, pp. 14–19.

[35] H. Sajnani, V. Saini, J. Ossher, and C. V. Lopes, "Is Popularity a Measure of Quality? An Analysis of Maven Components," in *2014 IEEE International Conference on Software Maintenance and Evolution*, sep 2014, pp. 231–240.

[36] J. Oliver, C. Cheng, and Y. Chen, "TLSH – A Locality Sensitive Hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, nov 2013, pp. 7–13.

[37] Eclipse. (2020) Steady. [Online]. Available: https://projects.eclipse.org/projects/technology.steady