Empirical Software Engineering (2020) 25:3175–3215
https://doi.org/10.1007/s10664-020-09830-x

Detection, assessment and mitigation of vulnerabilities in open source dependencies

Serena Elisa Ponta[1] · **Henrik Plate**[1] · **Antonino Sabetta**[1]

Published online: 30 June 2020

Abstract

Open source software (OSS) libraries are widely used in the industry to speed up the development of software products. However, these libraries are subject to an ever-increasing number of vulnerabilities that are publicly disclosed. It is thus crucial for application developers to detect dependencies on vulnerable libraries in a timely manner, to precisely assess their impact, and to mitigate any potential risk. This paper presents a novel method to detect, assess and mitigate OSS vulnerabilities. Differently from state-of-the-art approaches that depend on metadata to identify vulnerable OSS dependencies, our solution is code-centric, and combines static and dynamic analyses to determine the reachability of the vulnerable portion of libraries, in the context of a given application. Our approach also supports developers in choosing among the existing non-vulnerable library versions, with the goal to determine and minimize incompatibilities. Eclipse Steady, the open source implementation of our *code-centric* and *usage-based* approach is the tool recommended to scan Java software products at SAP; it has been successfully used to perform more than one million scans of about 1500 applications. In this paper we report on the lessons learned when maturing the tool from a research prototype to an industrial-grade solution. To evaluate Eclipse Steady, we conducted an empirical study to compare its detection capabilities with those of OWASP Dependency Check (OWASP DC), scanning 300 large enterprise applications under development with a total of 78165 dependencies. Reviewing a sample of the findings *reported only by one of the two tools* revealed that all Steady findings are true positives, while 88.8% of the findings of OWASP DC for vulnerabilities covered by our code-centric approach are false positives. For vulnerabilities not caused by code but due, e.g., to erroneous configuration, 63.3% of OWASP DC findings are true positives.

Communicated by: David Lo and Foutse Khomh

This article belongs to the Topical Collection: *Software Maintenance and Evolution (ICSME)*

✉ Serena Elisa Ponta
serena.ponta@sap.com

Henrik Plate
henrik.plate@sap.com

Antonino Sabetta
antonino.sabetta@sap.com

[1] SAP Security Research, Mougins, France

Keywords Open source software · Publicly known vulnerabilities · Code-centric vulnerability analysis · Combination of static and dynamic analysis · Usage-based update support

1 Introduction

The use of OSS libraries as well as the number of available libraries are ever-increasing: Synopsys Black Duck (2019) reports that over 96% of the applications they analyzed include OSS libraries, whose code often weighs more than 50% of the average code-base; Snyk (2019) reports a growth of 102% in the number Java OSS libraries available in Maven Central and of 40% for Python libraries in the Python package index (Pypi).

Though the use of OSS libraries speeds up development, it comes at a cost, as vulnerabilities discovered in OSS libraries may affect the dependent applications. While *using OSS components with known vulnerabilities* is included in the *OWASP Top 10 Application Security Risks* since 2013 (OWASP Foundation 2013, 2017), the problem is still far from being solved. OSS vulnerabilities keep on hitting the headlines of mainstream media: from the Heartbleed[1] and Shellshock[2] bugs in 2014 to the (in)famous *Equifax* incident[3] in 2017, which was caused by a missed update of a widely used OSS component, and which compromised the personal data of over 140 million U.S. citizens. Moreover, Snyk (2019) reports that the number of vulnerabilities disclosed for OSS libraries keeps on increasing (by 43% in 2017 and 33% in 2018). It thus comes at no surprise that, as reported by Snyk (2017), OSS vulnerabilities were the root cause of the majority of the data breaches that happened in 2016.

Establishing effective OSS vulnerability management practices, supported by adequate tools, is broadly understood as a priority in the software industry, and tools helping to *detect* known vulnerable libraries are available nowadays, either as OSS, e.g., OWASP Dependency Check[4] (OWASP DC) and Retire.js,[5] or as commercial products, e.g., WhiteSource[6] and Synopsys Black Duck.[7]

These tools differ in terms of detection capabilities, but (to the best of our knowledge) the approaches they use rely on the assumption that the metadata associated to OSS libraries (e.g., name, version), and to vulnerability descriptions (e.g., technical details, list of affected components) are always *available* and *accurate*. Unfortunately, these metadata, which are used to map each library onto a list of known vulnerabilities that affect it, are often incomplete, inconsistent, or missing altogether (Nguyen and Massacci 2013; Plate et al. 2015). Therefore, the tools that rely on them may fail to detect vulnerabilities (false negatives), or they may report as vulnerable artifacts that do not contain the code that is the actual cause of the vulnerability (false positives). To overcome the before-mentioned issues, the providers of commercial tools and services usually maintain proprietary vulnerability databases that map vulnerabilities onto the affected artifacts; however, the creation and maintenance such

[1] http://heartbleed.com/
[2] https://www.minttm.com/takeover-shellshocker-net
[3] https://investor.equifax.com/news-and-events/news/2017/09-15-2017-224018832
[4] https://www.owasp.org/index.php/OWASP_Dependency_Check
[5] https://retirejs.github.io/retire.js/
[6] https://www.whitesourcesoftware.com/oss_security_vulnerabilities/
[7] https://www.blackducksoftware.com/

mappings requires substantial human effort that is multiplied across vendors and whose results are error-prone.

Furthermore, merely detecting the inclusion of vulnerable libraries does not cater for the needs of the entire software development life cycle. In the early phases of development, updating a library to a more recent release is relatively unproblematic, because the necessary adaptations in the application code can be performed as part of the normal development activities. On the other hand, as soon as a project gets closer to the date of release to customers, and during the entire operational lifetime, all updates need to be carefully pondered, because they can impact the release schedule, require additional effort, cause system downtime, or introduce new defects.

To evaluate precisely the need and the urgency of a library update, it is necessary to answer the key question: "is the vulnerability *exploitable*, given the particular way the library is used within the application?". Answering this question is extremely difficult: vulnerabilities are typically described in advisories that consist of short, high-level, textual descriptions in natural language, whereas a reliable assessment of the exploitability and the potential impact of a vulnerability demands much lower-level, detailed, technical information.

Our first steps in tackling this problem were documented in Plate et al. (2015), where we introduced a method to analyze the code changes introduced by security fixes and to assess the impact of the vulnerability for a given application using *dynamic analysis*.

Our follow-up paper (Ponta et al. 2018) builds on Plate et al. (2015), proposing a more comprehensive *code-centric* and *usage-based* approach to *detect*, *analyze*, and *mitigate* OSS vulnerabilities. Its central claim is that code-centric and usage-based approaches outperform approaches relying on meta-data, namely regarding the accuracy of vulnerability detection, the possibility to perform application-specific impact assessments with help of reachability analyses, and mitigation support on the basis of metrics considering application-specific library use.

More precisely, in Ponta et al. (2018):

- We generalize the vulnerability detection approach of Plate et al. (2015) by considering fixes independently of the vulnerable libraries;
- We use static analysis to determine whether vulnerable code is reachable and through which call paths;
- We introduce a novel *combination* of static and dynamic analysis to overcome their mutual limitations;
- We define metrics that support the choice of alternative library versions that are not vulnerable, highlighting which options are more likely to minimize the update effort as well as the risk of incompatibility and regressions.

In this paper, compared to our previous two works:

- We provide an updated presentation of our approach, including an extended description of the method we use to establish whether an OSS library contains vulnerable code.
- We propose five criteria to automatically establish whether a given OSS library contains the vulnerable or the fixed code with respect to a given vulnerability.
- We present the results of an empirical study we conducted to compare the detection capabilities of our approach with those of OWASP DC, which is a well-established and mature flagship project of the OWASP Foundation.
- We revised the overall presentation of the existing material and we updated the section on related work to account for the most recent research results.

– We reflected the move of the open source implementation to the Eclipse Foundation under the name Eclipse Steady (incubator project).

Our approach is implemented as a tool known as Vulas internally to SAP, that is available as open source under the name Eclipse Steady. It is adopted at SAP as the recommended solution to scan Java software, and has been successfully used to scan about 1500 applications under development (as of May 2019). Vulnerable code was found reachable for 213 of them, and we found that in 11.7% of the cases this was only determined by the combination of static and dynamic analysis that is unique to our approach. We report on our experience and on the lessons we learned when maturing Steady from a research prototype to an industrial-grade solution that has been used to perform over one million scans since December 2016.

To evaluate our approach, we performed a comparative empirical study by running both Steady and OWASP DC on 300 large enterprise projects under active development, either products or internal tools, and which have an average of 260 dependencies (including test dependencies). The study targets vulnerability detection, as OWASP DC does not offer assessment and mitigation capabilities. As our approach is code-centric, it covers the detection of vulnerabilities that are fixed by changing the source code. For those we obtained over 2000 findings reported by both tools, about 1800 identified only by Steady, and over 17000 only by OWASP DC. To explain this difference, we manually reviewed a sample of the findings reported only by one of the tools. The review revealed that all the Steady findings are true positives because the vulnerable fragment of code corresponding to a vulnerability was found in a dependency. On the other hand, 88.8% of the findings of OWASP DC were false positives, caused by the inherent imprecision of the mechanism that the tool uses to match libraries with vulnerabilities. This result shows the advantages of our code-based approach. Eclipse Steady can also use manually-provided annotations to specific library versions, based on which it can detect vulnerabilities that are fixed without changing the source code (e.g., erroneous default configuration). For those we obtained 843 findings reported by both tools, 27 only by Steady (all true positives), and 1728 only by OWASP DC. A sample review of the latter showed that 63.3% are true positives. Most of them are not found by Steady as they affect libraries re-bundling the affected one. Though Steady did not report any false positive, the presence of false negatives shows the limitations of relying on manual annotations.

The remainder of the paper is organized as follows: Section 2 describes the technical approach, Section 3 defines the update metrics, Section 4 illustrates our approach in practice, and Section 5 discusses the comparative empirical study. In Section 6, we report on our experience, lessons learned and the challenges we identified. Section 7 discusses related literature and Section 8 concludes the paper.

2 Technical Description of the Approach

In this section we illustrate our code-based approach. This material is partly based on Plate et al. (2015) where we first introduced the idea of shifting the problem of establishing whether an application incorporates OSS components that have *exploitable* vulnerabilities, to the problem of assessing whether the vulnerable code of those components is *reachable*.

The code-centric and usage-based approach presented in the following Sections overcomes weaknesses of approaches and tools based on meta-data. Phenomenons like library

re-bundling[8] or the fact that single open source projects commonly distribute multiple, fine-grained libraries containing a subset of the original code base[9] represent significant challenges for approaches and tools based on meta-data. Their vulnerability detection suffers from false positives and false negatives, and the maintenance of corresponding vulnerability databases is costly and error-prone. Besides precise vulnerability detection (cf. Section 2.2), the code-centric approach also supports functionalities out of reach for approaches based on meta-data, especially the analysis whether vulnerable code can be executed in the context of a given application (cf. Sections 2.3 to 2.5), which is needed in order to prioritize findings, as well as update metrics considering the actual library use with the goal to reduce regressions (cf. Section 3).

Compared to previous works, Sections 2.1, 2.2 and 2.3 *generalize* (Plate et al. 2015), and Sections 2.2, 2.4, 2.5 *extend* it with unique novel contributions. In particular Sections 2.4, 2.5 are the basis of the update metrics presented in Section 3.

2.1 Representing vulnerabilities at the code level

Our approach is based on the idea that a vulnerability can be characterized, thus detected and analyzed, by the set of program constructs (such as methods), that were modified, added, or deleted to fix that vulnerability (Plate et al. 2015).

Definition 1 A *program construct* (or simply *construct*) is a structural element of the source code characterized by a construct *type* (e.g., package, class, constructor, method), a *language* (e.g., Java, Python), and a *unique identifier* (e.g., the fully-qualified name).

Example 1 The fully-qualified name of method baz(int) in class Bar and package foo is foo.Bar.baz(int). The type of this construct is method.

It is important to remark that the term *type* as used in this definition denotes the different kinds of syntactic constructs that form the structure of a program (as illustrated in the examples above); the same term is used with a different meaning in the domain of programming languages. The two meanings should not be confused.

Changes to program constructs are performed through *commits* in a source code repository; therefore, the set of changes that fix a vulnerability can be obtained from the analysis of the corresponding *fix commit*. Note that in cases where a commit includes not only a vulnerability fix but also unrelated changes, a post-processing of the construct changes is required.

Definition 2 We define a *construct change* as the tuple

$$(c, t, \mathtt{AST}_f^{(c)}, \mathtt{AST}_v^{(c)})$$

where c is a construct, t is a change operation (i.e., addition, deletion or modification) on the construct c, and $\mathtt{AST}_v^{(c)}$, $\mathtt{AST}_f^{(c)}$ are, respectively, the abstract syntax trees of the vulnerable and of the fixed c.

[8]The inclusion of code taken from other libraries with the intention to create self-contained library artifacts or self-contained applications.

[9]The open source project Apache POI, for instance, offers support for different Microsoft office formats, e.g., Word documents or PowerPoint spreadsheets. For each supported format, the project distributes separate library artifacts that can be used independently by developers.

Notice that for deleted (added) constructs only $\texttt{AST}_v^{(c)}$ ($\texttt{AST}_f^{(c)}$) exists.

In practice, the typical source of source code changes (from which we extract construct changes) are *commits* coming from code versioning systems: to a *commit* (that modifies source code) corresponds a set of *construct changes*.

These *fix commits* represent the main input to our approach, and its implementation, Eclipse Steady, requires the maintenance of a knowledge-base with triples each comprising a *vulnerability identifier*, a *URL* of the versioning control system of the vulnerable open source project and the identifiers of fix commits (cf. Sections 5 and 6.2.3). Steady automatically processes those fix commits in order to determine all changed constructs and the $\texttt{AST}_f^{(c)}$ and $\texttt{AST}_v^{(c)}$, so that this information is available when analyzing concrete applications and the libraries they depend upon (directly or transitively). Differently from proprietary vulnerability databases, mentioned in Section 1, our knowledge-base is open source. Furthermore, identifying (typically few) fix commits is less expensive and error-prone than identifying all the library versions affected by a given open source vulnerability, especially because of a popular technique called *re-bundling*, where code from one open source library is copied into other libraries distributed with different identifiers. The identification and enumeration of affected library versions with help of, e.g., Common Platform Enumeration[10] identifiers (CPE) or Maven coordinates,[11] is what we consider metadata, and is used by many state-of-the-art approaches.

When a fix is implemented over multiple commits, we rely on commit timestamps to compute the set of construct changes by comparing the source code before the first and after the last commit. If the vulnerability fix includes changes in a nested construct (e.g., a method of a class), two distinct entries are included in the set of construct changes, one for the outer construct (the class), one for the nested construct (the method).

While, ideally, fix commits should be systematically indicated by the developers of open source libraries (e.g., as part of security advisories), they are not always disclosed explicitly. Some OSS projects (e.g., Apache Tomcat) provide such information via security advisories; others reference issue tracking systems, which, in turn, describe the vulnerability being solved; some other OSS projects do not explicitly refer to vulnerabilities being fixed. Thus reconciling the information based on the textual description and code changes still requires manual effort (see Section 6.2.3). A broader discussion of the data integration problem can be found in Plate et al. (2015).

Differently from Plate et al. (2015), we provide a definition of construct change and consider the ASTs of the modified program constructs. This is used in Section 2.2 to establish whether a given library artifact includes the changes introduced by the fix.

2.2 Vulnerability detection

In this section we introduce the principles of our approach to detect the presence of vulnerabilities, based on the concepts introduced in the previous subsection. A concrete example of how this approach is applied in practice is illustrated in Section 4.1-(1).

Figure 1 shows how a vulnerability j is associated to an application a. C_j is the set of the constructs obtained by analyzing the fix commits of j, as described above. The set S_i contains all the constructs of the OSS library i used by the application a, whereas S_a is the set of all constructs of the application itself.

[10] https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe/

[11] https://maven.apache.org/pom.html

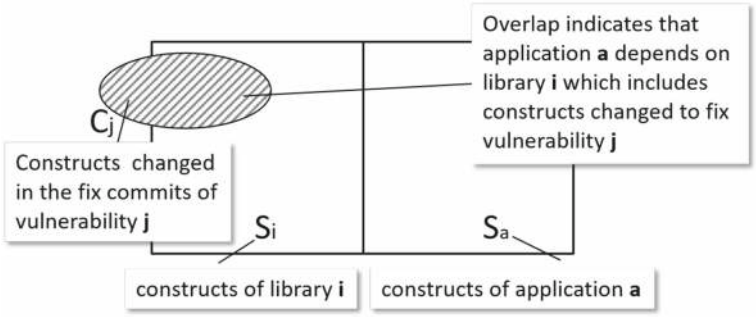

Fig. 1 Vulnerability detection

Definition 3 An application depends on a vulnerable library i, affected by vulnerability j, if that library includes code vulnerable to j (referred to as *vulnerable constructs*, i.e., constructs that have been changed in the fix commits of j), thus, if

$$(C_j \cap S_i) \neq \emptyset \wedge \forall c \in (C_j \cap S_i), \mathtt{AST}^{(c)} = \mathtt{AST}_v^{(c)} \tag{1}$$

According to Definitions 1 and 2, the list of construct changes contains constructs whose types are not limited to functions and methods but also include the outer constructs, e.g., the class. As a result, even if a vulnerability is fixed by adding new methods to an existing class, the intersection $C_j \cap S_i$ is not empty, as it would contain a construct corresponding to the class.

Condition (1) in Definition 3 requires the abstract syntax tree of each construct of library i that was changed in the fix commits to be strictly equal to the vulnerable abstract syntax tree. If such condition holds, it is then straightforward to conclude that the library is vulnerable to j. However, such a strict condition can hardly cope with real world scenarios. In fact, the vulnerable and fixed abstract syntax trees of the construct change are representations of the code at a single point in time, i.e., before and after the patch is applied, whereas it is often the case that several versions of a library (e.g., Spring Framework v4.x, v5.x) are maintained in parallel and thus several, possibly different, corrections may be applied for the different versions. If the same construct is changed in different ways in different versions, a given library version would only contain one of such vulnerable or fixed abstract syntax trees, thereby violating condition (1) of Definition 3.[12] Moreover the code evolves over time and thus the code where the correction has to be applied may have undergone numerous refactorings during the history of the library. Similarly, the fixed code may be modified and refactored while moving forward. It is thus clear that in the case of library versions released long before or after the application of the patch, it is unlikely to have an exact match on the abstract syntax trees.

To cope with the above challenges, we relaxed condition (1) of Definition 3 as follows to only require an exact match for a single construct as long as the remaining ones do not raise any inconsistencies. Such inconsistencies occur if a single library version contains

[12]Note that, for the ease of readability, the formal notation used in this paper covers the case of a patch in a single branch, however the existing implementation supports corrections applied in different repository branches.

both fixed constructs with $\texttt{AST}_f^{(c)}$ and vulnerable ones with $\texttt{AST}_v^{(c)}$, and have to manually resolved.

$$\exists\, c_a \in (C_j \cap S_i) \mid \texttt{AST}^{(c_a)} = \texttt{AST}_v^{(c_a)} \wedge \nexists\, c_b \in (C_j \cap S_i) \mid \texttt{AST}^{(c_b)} = \texttt{AST}_f^{(c_b)} \quad (2)$$

Similarly, a library version S_i includes the patch fixing vulnerability j if

$$\exists\, c_a \in (C_j \cap S_i) \mid \texttt{AST}^{(c_a)} = \texttt{AST}_f^{(c_a)} \wedge \nexists\, c_b \in (C_j \cap S_i) \mid \texttt{AST}^{(c_b)} = \texttt{AST}_v^{(c_b)} \quad (3)$$

Whenever condition (2) ((3), resp.) holds, we conclude that a library version i is vulnerable (fixed, resp.) to j according to criterion *AST equality*.[13]

For a given version of a library, it may happen that no equality is found while comparing abstract syntax trees. In this case, to draw a conclusion as to whether that library version contains the vulnerable or the fix version of the code, other versions of the same library can be considered. The evaluation based on multiple library versions uses both the result of the comparison of the AST of each construct changed and the order of the library versions.

The comparison of the AST of each construct changed (c) is used to determine a *distance* between $\texttt{AST}^{(c)}$ and $\texttt{AST}_v^{(c)}$ ($\texttt{AST}_f^{(c)}$). To obtain the distance between abstract syntax trees we use tree differencing algorithms (Falleri et al. 2014; Fluri et al. 2007). In particular we use (Fluri et al. 2007) that, given two abstract syntax trees, provides an edit script to transform the former in the latter. The number of edit operations in the edit script is used as distance in our approach.

To order library versions we rely on semantic versioning, following the usual schema MAJOR.MINOR.PATCH, extended with a fourth segment (BUILD). This extension was made to cover the versioning schema of libraries like Apache Struts that use four segments (MAJOR.MINOR.PATCH.BUILD).

Given a library name (e.g., the Maven group and artifact identifier), we order library versions by constructing a set of *release tree*s, where a release tree is defined as follows.

Definition 4 A *release tree* is a binary tree such that

- the root node always contains the first release of a given MAJOR.MINOR, i.e., X.Y.0.0;
- the left child contains the next patch release, i.e., given a node $\texttt{X.Y.Z}_1\texttt{.0}$ the left child connected through the edge p is $\texttt{X.Y, Z}_2\texttt{.0}$ where $\texttt{Z}_2 > \texttt{Z}_1$ (denoted as $\texttt{X.Y.Z}_1\texttt{.0} \prec_p \texttt{X.Y, Z}_2\texttt{.0}$);
- the right child contains the next build release, i.e., given a node $\texttt{X.Y.Z.W}_1$ the right child connected through the edge b is $\texttt{X.Y.Z.W}_2$ where $\texttt{W}_2 > \texttt{W}_1$, (denoted as $\texttt{X.Y.Z.W}_1 \prec_b \texttt{X.Y.Z.W}_2$).

Figure 2 shows an example of a generic release tree. Note that, when omitted, the BUILD segment of the version corresponds to 0.

In the following we present the additional criteria based on which we determine whether a vulnerability should be reported for a certain library version based on abstract syntax tree comparisons and release trees. Differently from criterion *AST equality* that may be applied to a single library version i, these additional criteria use the knowledge of several library versions. In fact, they require that both a release tree and the results of the abstract syntax tree comparisons be available. These criteria are only used when conditions (2) and (3) do not hold and they are applied in order of precedence.

[13] Note that in the existing implementation this criterion only works for constructs of type method, constructor or function as the AST is not available for classes and packages.

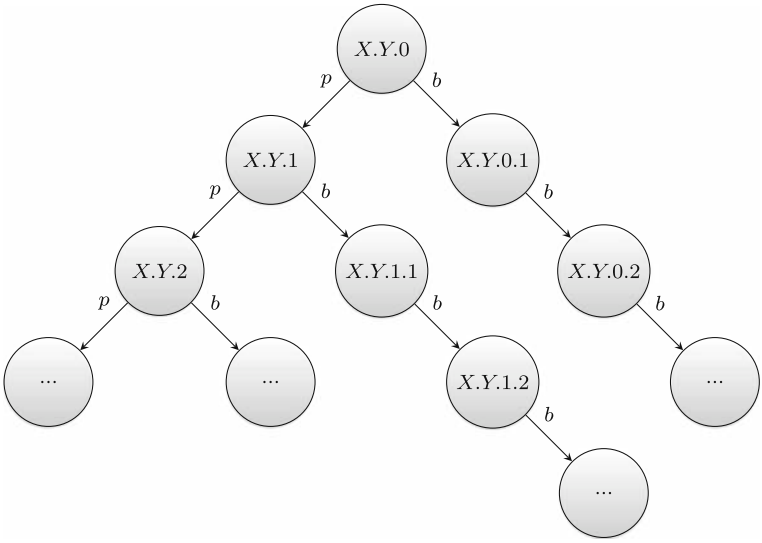

Fig. 2 Release tree

Intersection The *intersection* criterion considers library versions that are adjacent in the release tree (i.e., directly connected) and checks whether the abstract syntax tree comparisons show that the ancestor library version is *more similar* to the vulnerable code whereas the successor is *more similar* to the fixed code. If this is the case it concludes that the ancestor contains the vulnerable code whereas the successor contains the correction.

Definition 5 Given libraries S_1, S_2 in a release tree such that

$$S_1 \prec_p S_2 \wedge \nexists\, S_3 \mid S_1 \prec_p S_3 \wedge S_3 \prec_p S_2$$

or

$$S_1 \prec_b S_2 \wedge \nexists\, S_3 \mid S_1 \prec_b S_3 \wedge S_3 \prec_b S_2$$

(i.e., S_2 is a direct child of S_1 either through the patch or build branch), then S_1 is said to be vulnerable and S_2 is said to be fixed with respect to vulnerability j according to criterion *intersection* if there exists a construct whose `AST` in S_1 is more similar to the vulnerable `AST` whereas the `AST` for the same construct in S_2 is more similar to the fixed one, and no other construct exists for which the opposite holds true. Let $\texttt{diff}(\texttt{AST}_{S_1}{}^{(c)}, \texttt{AST}_{S_2}{}^{(c)})$ be the number of changes required to transform $\texttt{AST}_{S_1}{}^{(c)}$ into $\texttt{AST}_{S_2}{}^{(c)}$ (i.e., the abstract syntax tree edit distance), then the *intersection* criterion establishes that S_1 contains the vulnerable code and S_2 contains the corrected code if

$$\begin{aligned}\exists\, c \in (C_j \cap S_1 \cap S_2) \mid \texttt{diff}(\texttt{AST}_{S_1}{}^{(c)}, \texttt{AST}_v^{(c)}) < \texttt{diff}(\texttt{AST}_{S_1}{}^{(c)}, \texttt{AST}_f^{(c)}) \\ \wedge\, \texttt{diff}(\texttt{AST}_{S_2}{}^{(c)}, \texttt{AST}_v^{(c)}) > \texttt{diff}(\texttt{AST}_{S_1}{}^{(c)}, \texttt{AST}_f^{(c)}),\end{aligned} \tag{4}$$

and

$$\begin{aligned}\nexists\, c_1 \in (C_j \cap S_1 \cap S_2) \mid \texttt{diff}(\texttt{AST}_{S_1}{}^{(c_1)}, \texttt{AST}_v^{(c_1)}) > \texttt{diff}(\texttt{AST}_{S_1}{}^{(c_1)}, \texttt{AST}_f^{(c_1)}) \\ \wedge\, \texttt{diff}(\texttt{AST}_{S_2}{}^{(c_1)}, \texttt{AST}_v^{(c_1)}) < \texttt{diff}(\texttt{AST}_{S_1}{}^{(c_1)}, \texttt{AST}_f^{(c_1)}).\end{aligned} \tag{5}$$

Note that these conditions hold also when either $\texttt{diff}(\texttt{AST}_{S_1}{}^{(c)}, \texttt{AST}_v^{(c)}) = 0$ or $\texttt{diff}(\texttt{AST}_{S_2}{}^{(c)}, \texttt{AST}_f^{(c)}) = 0$, i.e., if S_1 is vulnerable by criterion *AST equality* or S_2 is fixed by criterion *AST equality*.

Example 2 Consider the release tree of Fig. 2 and let

- $S_1 = X.Y.1$
- $S_2 = X.Y.2$
- $\texttt{diff}(\texttt{AST}_{S_1}{}^{(c)}, \texttt{AST}_v^{(c)}) = 1$
- $\texttt{diff}(\texttt{AST}_{S_1}{}^{(c)}, \texttt{AST}_f^{(c)}) = 3$
- $\texttt{diff}(\texttt{AST}_{S_2}{}^{(c)}, \texttt{AST}_v^{(c)}) = 4$
- $\texttt{diff}(\texttt{AST}_{S_2}{}^{(c)}, \texttt{AST}_f^{(c)}) = 2$

In Fig. 3, we connect the distances of S_1 and S_2 to the vulnerable AST and the ones to the fixed AST. The former can be distinguished by the use of a bold line. As it becomes clear from Fig. 3, the distances above represent an intersection as the distance is closer to the vulnerable AST for S_1 (i.e., the tree edit distance is smaller) whereas it is closer to the fixed one for S_2. As a result, according to criterion *intersection* we conclude that S_1 is vulnerable and S_2 is fixed.

Major release The underlying intuition of the *major release* criterion is that once the correction of a security defect is included in a library version, all the versions that follow also include the correction. Thus, the release tree for a given MAJOR.MINOR is used to establish the ordering of library versions, and to determine those that are descendants of versions that are known to be fixed by criteria *AST equality* or *intersection*.

Definition 6 Given the libraries S_1 and S_2 such that S_1 is fixed according to criteria *AST equality* or *intersection*, then S_2 is said to be fixed according to criterion *major release* if

- $S_1 \prec_p S_2$, or
- $S_1 \prec_b S_2$

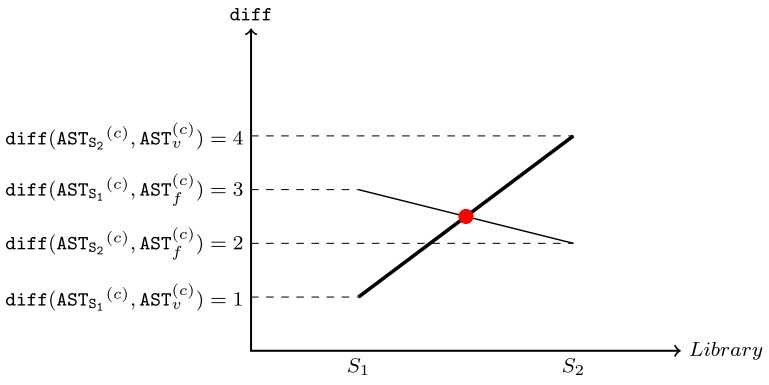

Fig. 3 Example of intersection

Example 3 Consider the example of Fig. 4 where nodes marked with double circles are fixed to vulnerability j by criteria *AST equality* or *intersection*. Then according to criterion *major release* we can conclude that

- library versions $X.Y.3$ and $X.Y.4$ are fixed as $X.Y.2 \prec_p X.Y.3$ and $X.Y.2 \prec_p X.Y.4$, i.e., they follow the fixed one $X.Y.2$;
- library versions $X.Y.0.2$ and $X.Y.2.1$ are fixed as $X.Y.0.1 \prec_b X.Y.0.2$ and $X.Y.2 \prec_b X.Y.2.1$, i.e., they follow the fixed ones $X.Y.0.1$ and $X.Y.2$, resp.

Minor release Similarly to criterion *major release*, the idea of the *minor release* criterion is that library versions preceding one containing the vulnerable code also contain the vulnerable code. Again, the release tree is used to establish the ordering of library versions. The library versions for which the criteria *AST equality* or *intersection* concluded that they contain the vulnerable code are used as starting point.

Definition 7 Given the library versions S_1 and S_2 such that S_1 is vulnerable according to criteria *AST equality* or *intersection*, S_2 is said to be vulnerable according to criterion *minor release* if

- $S_2 \prec_p S_1$, or
- $S_2 \prec_b S_1$

Example 4 Consider the example of Fig. 5 where dashed nodes are vulnerable to vulnerability j by criteria *AST equality* or *intersection*. Then according to criterion *minor release* we can conclude that

- library versions $X.Y.1$ and $X.Y.0$ are vulnerable as $X.Y.1 \prec_p X.Y.2$ and $X.Y.0 \prec_p X.Y.2$, i.e., they are ancestors of the vulnerable one $X.Y.2$;
- library versions $X.Y.0.1$ and $X.Y.0$ are vulnerable as $X.Y.0.1 \prec_b X.Y.0.2$ and $X.Y.0 \prec_b X.Y.0.2$, i.e., they are ancestors of the vulnerable one $X.Y.0.2$.

Note that library version $X.Y.0$ is vulnerable according to both conditions.

Greater release Criterion *greater release* is meant to cope with the case of library versions released long after the vulnerability was found and henceforth fixed. The underlying idea

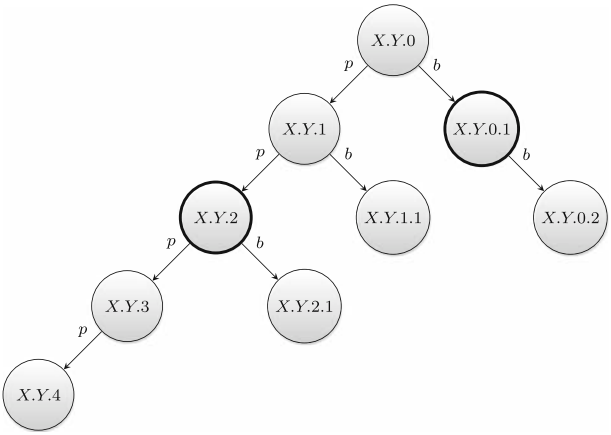

Fig. 4 Example of major release criterion

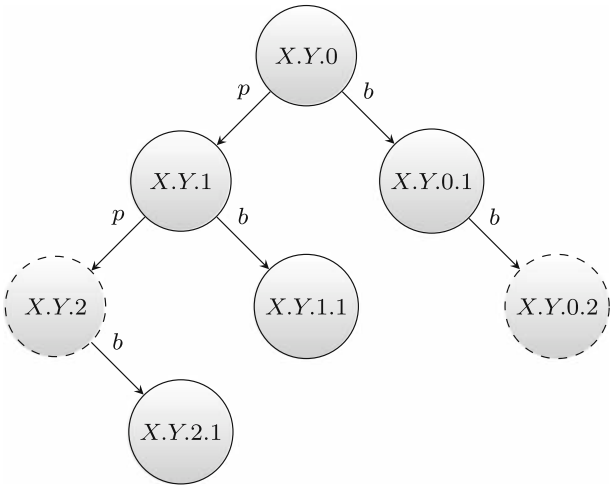

Fig. 5 Example of minor release criterion

is to check whether the entire release tree was created temporally after all the versions that were found fixed according to criteria *AST equality* or *intersection*.

Example 5 Consider a library having two release trees: the one of Fig. 4 and a tree with root node $A.B.0$. All versions belonging to the release tree $A.B$ are fixed according to criterion *greater release* if the release date of $A.B.0$ is greater than the one of $X.Y.2$ and $X.Y.0.1$ which are the versions that were found fixed according to criteria *AST equality* and *intersection*.

Manual inspection is still required whenever no automated conclusion can be taken, e.g., when

$$\exists\, c_1, c_2 \in (C_j \cap S_i) \mid \mathtt{AST}^{(c_1)} = \mathtt{AST}_v^{(c_1)} \ \wedge\ \mathtt{AST}^{(c_2)} = \mathtt{AST}_f^{(c_2)}.$$

Differently from Plate et al. (2015), we define the set of constructs C_j as independent of any library i, which takes into consideration that single constructs, e.g., classes or packages, are copied to (included in) libraries other than the original ones produced by the respective open source project. A vulnerability in a library, be it one produced in the context of the original open source project or one produced by other developers who just copied (included) the code of other libraries, is then detected through the intersection of its constructs with C_j. This approach has several advantages: First, it makes it explicit that the vulnerable constructs responsible for a vulnerability j can be contained in any library artifact i, hence, the approach is robust against the prominent practice of re-bundling the code of OSS libraries. Second, it is sufficient that a library version includes a subset of the vulnerable constructs for the vulnerability to be detected. Last, it improves the accuracy compared to approaches based on metadata, which typically flag entire open source projects as affected, even if projects release functionalities as part of different libraries. Apache POI,[14] for instance, while developed in a single source code repository, is released as a set of distinct, independent libraries. Because Plate et al. (2015) focuses on *newly-disclosed* vulnerabilities, it

[14] https://poi.apache.org/

assumes that, *at the time of disclosure*, every library that includes constructs changed in the fix commit must be vulnerable. While this assumption is valid at that moment in time, it is not valid for old vulnerabilities, which require that one establishes whether a given library contains the fixed code. We achieve this by comparing the AST of constructs in use with those of the *affected* and *fixed* versions and providing a set of criteria to automatically conclude.

2.3 Reachability of vulnerable code: Dynamic assessment

After having determined that an application depends on a library version that *includes* vulnerable constructs, it is important to establish whether these constructs are *reachable*. In this paper, we use the term *reachable* to denote both the case where dynamic analysis shows that a construct is *actually executed* and the case where static analysis shows *potential* execution paths. The underlying idea is that if an application executes (or may execute) vulnerable constructs, there exists a significant risk that the vulnerability can be exploited. The dynamic assessment described here is borrowed from our previous work (Plate et al. 2015).

In the following we explain how we use dynamic reachability analysis in our approach; in Section 4.1-(2-3) we illustrate its application in practice.

Figure 6 illustrates the use of dynamic analysis to assess whether the vulnerable constructs are reachable by observing actual executions. T_{ai} represents the set of constructs, either part of application a or its bundled library i, that were executed at least once during some execution of the application. To increase readability, the figure only visualizes one such library, while the actual analysis always considers all libraries that are directly or transitively used by the application. The intersection $C_j \cap T_{ai}$ comprises all those constructs that are both changed (added, deleted or modified) in the fix commits of j and executed in the context of application a because of its (direct or transitive) use of library i.

The collection of actual executions of constructs can be done at different times: during unit tests, integration tests, and even during live system operation. In our implementation for Java, for instance, the collection is accomplished with a Java instrumentation agent. If enabled using a command line argument of the Java Virtual Machine (JVM), the agent adds suitable Java statements to each method when the corresponding class is loaded for the first time. This happens regardless of whether a class belongs to a directly or indirectly used library, since they are all included in the JVM's classpath. Moreover, this implementation does not require specific test cases, but relies on existing (unit, integration, or manual) tests. Therefore, the effectiveness of dynamic analysis in discovering the execution of vulnerable code significantly depends on the coverage achieved by such tests.

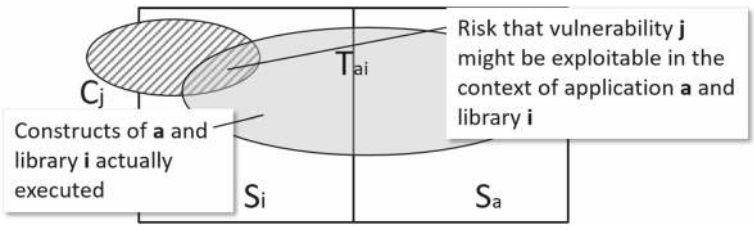

Fig. 6 Dynamic vulnerability analysis

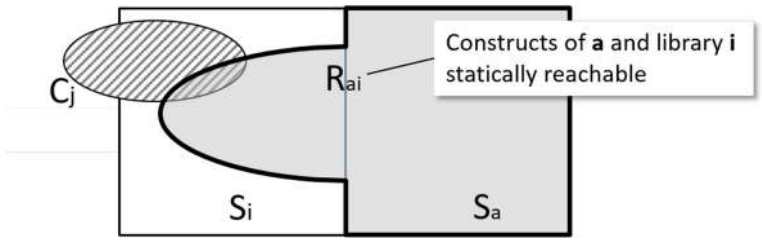

Fig. 7 Static vulnerability analysis

2.4 Reachability of vulnerable code: Static assessment

In addition to the analysis of *actual* executions (dynamic analysis), our approach uses static analysis to determine whether the vulnerable constructs are *potentially* executable.

An example of how this approach is applied in practice is presented in Section 4.1-(4).

Our method uses static analysis in two different ways.

First, we use it to *complement* the results of the dynamic analysis, by identifying the library constructs reachable from the application.

Second, (Section 2.5) we *combine* the two techniques by using the results of the dynamic analysis as input for the static analysis, thereby overcoming limitations of both techniques: static analyzers are known to struggle with dynamic code (such as, in Java, code loaded through reflection (Landman et al. 2017)); on the other hand, dynamic (test-based) methods suffer from inadequate coverage of the possible execution paths.

Figure 7 illustrates how we use static analysis to complement the results of dynamic analysis. R_{ai} represents the set of all constructs, either part of application a or a bundled library i, that are found reachable starting from the application a and thus can be potentially executed. Again, for the sake of readability, the figure only visualizes one such library, while the actual analysis always considers all libraries that are directly or transitively used by the application. Static analysis is performed by using a static analyzer, e.g., the T.J. Watson Libraries for Analysis[15] or the Soot framework,[16] to compute a graph of all constructs of all libraries reachable from the application constructs. The implementation of the approach, Steady, only requires the distributed Java archives (JARs) of each library the application depends on (but not their source code). However, as those archives are also needed in other contexts, e.g., to compile or test the application source code, their presence does not represent an additional requirement. The intersection $C_j \cap R_{ai}$ comprises all constructs that are both changed in the fix commit of j and are part of the call graph, thus, can be potentially executed.

2.5 Combination of dynamic and static assessment

This subsection presents our method to combine static and dynamic reachability analysis. The application in practice is illustrated in Section 4.1-(5).

As shown in Fig. 8, in the combined method we use the set of constructs actually executed, T_{ai}, as starting point for the static analysis. The result is the set $R_{T_{ai}}$ of constructs reachable starting from the ones executed during the dynamic analysis. The intersection

[15] http://wala.sourceforge.net/wiki/index.php/Main_Page

[16] http://sable.github.io/soot/

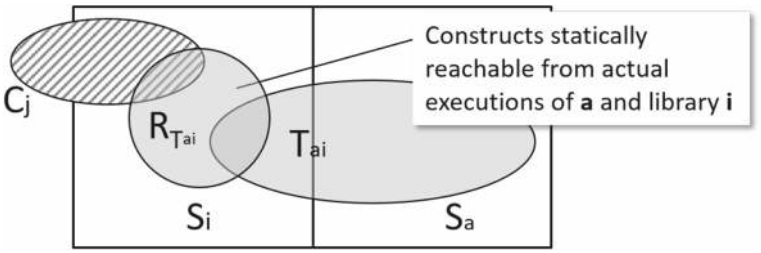

Fig. 8 Combination of static and dynamic vulnerability analysis

$C_j \cap R_{T_{ai}}$ comprises all constructs that are both changed in the fix commit of j and can be potentially executed. Note that library i, as in Figs. 6 and 7, can be directly or transitively used by the application.

We explain the benefits of the combinations of the two techniques through the example in Fig. 9. In the following, we denote a library bundled within a software program with the term *dependency*.

Example 6 Let S_a be a Java application having two direct dependencies S_1 and S_f where S_1 has a direct dependency S_2 that in turn has a direct dependency S_3 (thereby S_2 and S_3 are transitive dependencies for the application S_a). S_1 is a library offering a set of functionalities to be used by the application (e.g., Apache Commons FileUpload[17]). Moreover the construct γ of S_1 calls the construct δ of S_2 dynamically, e.g, by using Java reflection, which means the construct to be called is not known at compile time. S_f is what we call a "framework" providing a skeleton whose functionalities are meant to call the application defining the specific operations (e.g., Apache Struts,[18] Spring Framework[19]). The key difference is the so-called *inversion of control* as frameworks call the application instead of the other way round.

With the vulnerability detection step of Section 2.2, our approach determines that S_a includes vulnerable constructs for vulnerabilities j_1 and j_2 via the dependencies S_f and S_3, respectively. Note that even if S_3 only contains two out of the three constructs of C_{j_2}, our approach is still able to detect the vulnerability.

We start the vulnerability analysis by running the static analysis of Section 2.4 that looks for all constructs potentially reachable from the constructs of S_a. The result is the set R_{a1} including all constructs of S_a and all constructs of S_1 reachable from S_a. As expected, S_f is not reachable in this case as frameworks are not called by the application. Moreover, it is well known that static analysis cannot always identify dynamic calls like those performed using Java reflection. As the call from γ to δ uses Java reflection, in this example only S_1 is statically reachable from the application. As shown in Fig. 9 R_{a1} does not intersect with any of the vulnerable constructs.

The dynamic analysis of Fig. 6 produces the set T_a (omitted from the figure) of constructs that are actually executed. Though no intersection with the vulnerable constructs is found, the dynamic analysis increases the set of reachable constructs ($R_{a1} \cup T_a$ in Fig. 9). In particular, it complements static analysis revealing paths that static analysis missed. First,

[17] https://commons.apache.org/proper/commons-fileupload/
[18] https://struts.apache.org/
[19] https://projects.spring.io/spring-framework/

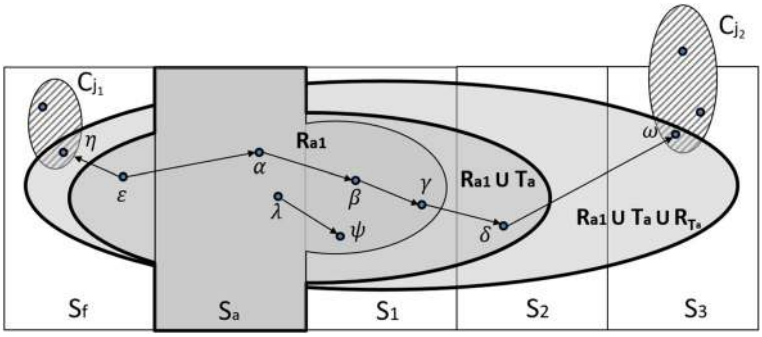

Fig. 9 Vulnerability analysis example

it contains construct ϵ of framework S_f that calls construct α of the application. Second, it follows the dynamic call from γ to δ.

Combining static and dynamic analysis, as shown in Fig. 8, we can use static analysis with the constructs in T_a as starting point. The result is the set R_{T_a} (omitted in the figure) of all constructs that can be potentially executed starting from those actually executed T_a.

After running all the analyses, we obtain the overall set $R_{a1} \cup T_a \cup R_{T_a}$ (shown with solid fill in Fig. 9) of all constructs found reachable by at least one technique. Its intersection with C_{j1} and C_{j2} reveals that both vulnerabilities j_1 and j_2 are reachable, since one vulnerable construct for each of them is found in the intersection and is thus reachable ($\eta \in C_{j_1}$ is reachable from ϵ and $\omega \in C_{j_2}$ is reachable from δ).

2.6 Applicability of the approach to different programming languages

This section discusses the possibility to apply the above-presented approach to different programming languages and ecosystems, considering their respective particularities.

A prerequisite for both vulnerability detection and reachability analysis is the unique identification of constructs, both during the analysis of source code (altered by fix commits) as well as during the analysis of distributed packages (downloaded during the build process of downstream dependents).

The generic concept of unique construct identifiers, introduced with Definition 1, has been implemented for the Java and Python programming languages. In both cases, constructs are identified using their fully-qualified name.

In Java, the fully-qualified name of, e.g., method baz(int) in class Bar and package foo is foo.Bar.baz(int). The Java compilation process does not remove or alter such identifiers (apart from constructors of non-static inner classes), thus, they can be easily built from source code and Java archives (JARs) containing Java bytecode.

In Python, the fully-qualified name of, e.g., function loads(s,**kwargs) in module json and package flask is flask.json.loads(s,**kwargs). Python package formats, e.g., egg[20] or wheel,[21] contain the Python source code as-is, thus, the same parser can be used to identify Python constructs in source code and in distributed packages.

[20] https://setuptools.readthedocs.io/en/latest/formats.html
[21] https://www.python.org/dev/peps/pep-0427/

The implementation choice of using fully-qualified names supports other programming languages as long as they have a comparable naming scheme allowing construct identification in source and distributed code. Moreover, its development community must follow consistent naming conventions (that is, construct names can be assumed to be globally unique). However, those properties are not satisfied for certain languages and cases.

Specifically, we acknowledge that fully-qualified names can hardly be used for programming languages compiled into machine code, e.g., C and C++. For such languages, it is very hard to recognize and compare constructs found in source code with those in the distributed binary packages. But even in case of interpreted languages, the code of distributed packages can deviate significantly from its corresponding source code. JavaScript libraries running in browsers, for instance, are commonly minimized and obfuscated in order to reduce their download size, which makes it hard to relate constructs identified in source code to their counterparts in the distributed libraries. Such transformations are less common for JavaScript running on servers, e.g., in the Node.js runtime.

Where fully-qualified names cannot be used, one has to identify constructs using other information, e.g., characteristics of method or function bodies. This, however, comes with its own challenges and represents a separate body of research.

As described in Section 2.2, vulnerable detection is possible as soon as constructs can be uniquely identified and compared, and has been successfully implemented for Java and Python.

The subsequent reachability analyses, dynamic as described in Section 2.3 and static as described in Section 2.4, depend on other language characteristics, and have only been implemented for Java. However, before mentioning potential problems in other languages, one has to note that the absence of one or the other analysis for a given programming language does not render an implementation useless. The precision of code-based vulnerability detection is valuable by itself, even when reachability analyses cannot be performed.

Static program analysis commonly struggles with dynamic languages such as Python and JavaScript. Characteristics like the absence of static type information or dynamic code execution through statements like `eval` make it hard to build accurate call graphs. Depending on the chosen analysis approach, the approximated call graphs can be either too small or too big, resulting in false negatives and false positives respectively.

Dynamic program analysis is considered straightforward in cases where tracing tools already exist. In other cases, especially for interpreted languages, the source code of downloaded open source packages can be altered prior or during test execution, e.g., using techniques like aspect-oriented programming.

3 Vulnerability Mitigation

The analysis presented in Sections 2.3 to 2.5 provides in-depth information about which parts of the code of the application and of all its dependencies (both direct and transitive) are executed (or could be executed). In the following, we show how we leverage this information to support application developers in mitigating vulnerable dependencies. As long as non-vulnerable library versions are available, updating to one of those is the preferred solution to fix vulnerable application dependencies. And since the approach described in Section 2.1 depends on the presence of at least one fix commit, such a non-vulnerable library version becomes available when the respective open source project releases a version including this commit.

In practice, such updates are straightforward for direct dependencies. Typically, in case dependencies are specified in a declarative manner, e.g., using Maven pom.xml files, the application developer just needs to modify the version of the respective declaration. In case of transitive dependencies, there exist two possibilities: Ideally, it is possible to update the direct dependency, which is responsible for pulling in the transitive one, to a newer version such that a non-vulnerable version is pulled. If that is not possible or wanted, e.g., no such version of the direct dependency exists or the update has other side effects, the developer can transform the transitive dependency into a direct one by adding a corresponding declaration, thereby pointing to a non-vulnerable version. Typically, dependency managers prioritize versions of dependencies that are closer to the root of the dependency tree, hence, the vulnerable version of the transitive dependency is ignored. Even though the introduction of a new direct dependency conflicts with the original idea of transparent dependency management, it is an effective means to fix vulnerable transitive dependencies until all open source projects of the respective dependency branch have updated their declarations.

In the following, our focus lies on the update of the vulnerable library, no matter whether it is a direct or transitive dependency, and how the update is achieved in practice. In this context, it is well known that developers are reluctant to update dependencies because of the risk of breaking changes, the difficulties in understanding the implications of changes, and the overall migration effort (Kula et al. 2018; Mostafa et al. 2017). Such risk and effort depend on the usage the application makes of the library, and on the amount of changes between the library version currently in use and the respective non-vulnerable version. As a result of the analysis described in Sections 2.3 to 2.5, the reachable share of each library is known. Whether a construct with a given identifier is also available in other versions of a library can be determined, for instance, by comparing compiled code with tools such as Dependency Finder.[22] Among all the reachable constructs, of particular importance in the scope of mitigation are those which are called directly from the application, as they provide a measure of how many times the application developer explicitly used the library. We define a *touch point* as a pair of constructs (c_1, c_2) such that $c_1 \in S_a$ is an application construct, $c_2 \in S_i$ is a library construct, and there exists a call from c_1 to c_2. We define *callee* the library construct called directly from the application, i.e., c_2. In the example of Fig. 9 there are two touch points: (α, β) and (λ, ψ), with β and ψ being the callees.

Given a library version in use S_i and its candidate replacement S_j, we define the following update metrics.

Callee Stability (*CS*) Intuitively, this metric represents the share of touch-point callees in S_i that also exists in a given non-vulnerable library version S_j. Let $c_k^{(S_i)}$ with $k = 1, \ldots, n$ be the callees of S_i, and $c_k^{(S_j)} = 1$ if $c_k^{(S_i)} \in S_j$, 0 otherwise. Then the callee stability is the number of callees of S_i that exist in S_j over the number of callees of S_i:

$$CS = \sum_{k=1}^{n} c_k^{(S_j)} / |\{c_1^{(S_i)}, \ldots, c_n^{(S_i)}\}| = \sum_{k=1}^{n} c_k^{(S_j)} / n$$

If S_j contains all the callees of S_i, then the callee stability is 1, to indicate that the constructs of S_i called by the application exist also in library version S_j. In case S_j does not contain all the callees of S_i, then the callee stability is smaller than 1 and reaches 0 when none of the callees of S_i is present in S_j.

[22] https://github.com/jeantessier/dependency-finder

Development Effort (*DE*) Intuitively, this metric represents the share of touch-point callers in S_a that has to be changed due to non-existing callees in S_j. Let $a_k^{(S_i)}$ with $k = 1, \ldots, n$ be the calls from the application to the callees of S_i, and $a_k^{(S_j)} = 1$ if $a_k^{(S_i)} \notin S_j$, 0 otherwise. The development effort for updating from library version S_i in use by the application to library version S_j is defined as the number of application calls that require a modification due to callees of S_i that do not exist in S_j.

$$DE = \sum_{k=1}^{n} a_k^{(S_j)}$$

Compared to the callee stability, the development effort takes into account the fact that each callee can be called multiple times within an application.

In Fig. 9, for instance, each callee is called only once by α and λ respectively. However, assuming that β is called by two application constructs in addition to α, and that it is not contained in the new library version S_j, $CS = 1/2$ whereas the $DE = 3$. This reflects the fact that multiple calls need to be modified as a result of a change in a single callee. The notion of development effort we use does not take into account the complexity of each modification; rather, it focuses on the number of modifications required by the application, considering that each modification comes at the cost not only of updating the code (which could be automated to some extent) but also of testing it.

Reachable Body Stability (*RBS*) The reachable body stability is calculated in the same way as the callee stability, but instead of callees, it considers the reachable share of a library version, i.e., the set of all library constructs reachable according to static analysis, dynamic analysis, or both. Given the total number of constructs of S_i reachable from the application, it measures the ratio of those that are contained as-is, i.e., with identical identifier and byte code, also in S_j. By quantifying the share of modified reachable constructs, this metric provides the likelihood that the behavior of the library changes from S_i to S_j. In case all reachable constructs of S_i exist in S_j, then $RBS = 1$ and thus there is a higher likelihood that the library change does not break the application.

Overall body stability (*OBS*) The overall body stability is calculated similarly to *RBS* but now considers all the constructs of S_i. This metric provides the same indication as the one above but, by considering the entire library rather than only its reachable share, it is independent of the application-specific usage.

The above metrics support the application developer in estimating the effort and risks of updating a library. When several non-vulnerable library versions exist that are newer than the one in use, they are all candidate replacements. By quantifying the changes to be performed on the application and the changes that the library underwent, our update metrics allow the developer to take an informed decision.

Note that the callee stability and development effort metrics only apply for dependencies that are called directly from the application, whereas the reachable and overall body stability also apply for transitive dependencies and frameworks.

4 Illustrative Example

In this section we illustrate how our approach works in a typical scan, applying it to a SAP-internal web application that we adapted, for illustrative purposes, to include vulnerable OSS. The application allows users to upload files (such as documents or compressed

Fig. 10 Vulnerable constructs for CVE-2017-5638 affecting a struts2-core artifact. Static analysis starting from traced methods provides further evidence this CVE is relevant for the application at hand

archives) through an HTML form, inspects the file content and displays a summary to the user. It is built using Maven,[23] and it depends on popular open source libraries from the Apache Software Foundation, such as Struts 2.3.24 (released on 3 May 2015), Commons FileUpload 1.3.1 (6 February 2014), POI 3.14 (6 March 2016) and HttpClient 4.5.2 (21 February 2016). Overall, the application has 12 direct and 15 transitive compile-time dependencies.

The analysis is performed using Steady, the open source implementation of the approach described previously. This implementation is released under the Apache license version 2. Sections 2.2, 2.4, and 2.5 are implemented as *goals* of a Maven plugin; the collection of traces during the dynamic analysis (Section 2.3) is performed by instrumenting all classes of both the application and all its dependencies as described in Plate et al. (2015). This happens either at runtime, when classes are loaded, or by modifying the byte code of the application before deploying it in a runtime environment such as Apache Tomcat. The knowledge-base of vulnerabilities used by Steady is described in Ponta et al. (2019) and available at https://github.com/SAP/vulnerability-assessment-kb.

4.1 Detection and analysis

To illustrate the benefits of our approach, we go through the analysis steps and highlight selected findings. To demonstrate the added value of static analysis compared to Plate et al. (2015), we perform it after dynamic analysis. However, our implementation also supports changing their order (or executing only a subset of the steps).

(1) Vulnerability detection The first step is to create a *bill of materials* (BOM), consisting of the constructs of the application and *all its dependencies*, as explained in Section 2.2.

Vulnerabilities in a library artifact are detected by intersecting the set of constructs found in (the BOM of) that artifact with the vulnerable constructs of all the vulnerabilities in our knowledge-base. As an example, the bottom part of Fig. 10 shows a table listing the vulnerable constructs for CVE-2017-5638 (columns *Type* and *Qualified Construct Name*) together with the respective change operation (column *Change*), as well as the information that those constructs are actually present in the Java archive corresponding to struts-core:2.3.24 (column *Contained*). The same figure also illustrates the application of reachability analysis, which is explained later in this section.

[23] https://maven.apache.org/

Fig. 11 Criterion AST Equality establishes that struts-core:2.3.24 contains code vulnerable to CVE-2017-5638

Additionally, Fig. 11 shows that the criterion *AST equality* was applied to establish that such Java archive contains vulnerable constructs for which condition (2) holds. In more detail, the AST of three out of the four constructs modified in the fix commit is equal to the vulnerable one (column *Distance to V*) whereas there is a distance of five to the fixed one (column *Distance to F*); the AST of the fourth construct is not equal to the vulnerable nor the fixed one but is more similar to the former.

The vulnerability detection step reveals that our sample application includes vulnerable code related to 25 different vulnerabilities, affecting nine different compile-time dependencies: seven are *direct*, while the remaining two (ognl:3.0.6 and xwork-core:2.3.24,[24] pulled in through struts2-core:2.3.24) are transitive.

(2) Dynamic assessment (Unit tests) This step and the next use the method explained in Section 2.3. The execution of unit tests reveals that vulnerable constructs related to three vulnerabilities are executed, e.g., the method URIBuilder.normalizePath(String),[25] which is part of httpclient:4.5.2 and subject to vulnerability HTTPCLIENT-1803.[26] Another example is shown in Fig. 12: method SharedStringsTable.readFrom(InputStream), which is part of poi-ooxml:3.14 and subject to CVE-2017-5644, is invoked in the context of unit test openSpreadsheetTest. The fact that reflection is heavily used inside the poi-ooxml method XSSFFactory.createDocumentPart(Class,Class[],Object[]) (as visible from a sequence of four invocations of newInstance, see figure) makes it difficult for static analysis to determine the reachability of the vulnerable method.

(3) Dynamic assessment (Integration tests) The execution of integration tests is done using an instrumented version of the application deployed in a runtime container. They

[24] Maven dependencies are denoted using their artifact identifier followed by, where necessary, a colon and their version. Group identifiers are omitted for brevity.

[25] Where possible, Java package and class names are omitted for brevity.

[26] https://issues.apache.org/jira/browse/HTTPCLIENT-1803

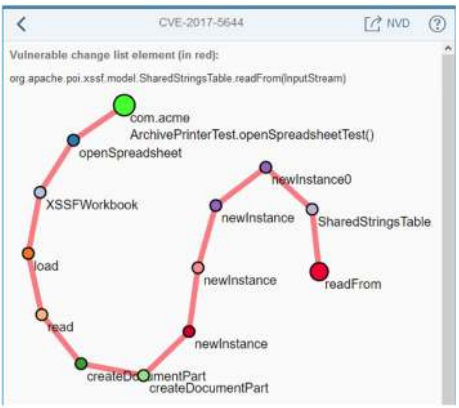

Fig. 12 Unit tests reveal the execution of vulnerable construct SharedStringsTable.readFrom (InputStream) in poi-ooxml

reveal the execution of vulnerable code related to eight additional vulnerabilities, all affecting struts2-core, or its dependencies ognl and xwork-core. As an example, the last line of the table in Fig. 10 shows a vulnerable construct of CVE-2017-5638, FileUploadInterceptor.intercept(ActionInvocation), whose actual execution is traced (column *Traced*) at the reported time. This method is included in struts2-core 2.3.24 which is part of the Struts2 framework and exemplifies the inversion of control (IoC) happening when frameworks invoke application code.

(4) Static assessment Using the method introduced in Section 2.4, the static reachability analysis starting from application constructs reveals that the constructor MultipartStream(InputStream,byte[],int,[4]ProgressNotifier), part of commons-fileupload:1.3.1 and subject to CVE-2016-3092, is reachable from the application. Dynamic analysis was not able to trace its execution due to the limited test coverage.

On the other hand, static analysis starting from the application constructs falls short in the presence of IoC. As application methods are called *by the framework*, there is no path on the call graph starting from application and reaching framework constructs that are involved in the IoC mechanism.

(5) Combination of static and dynamic assessment This step combines the two complementary approaches to reachability analysis, as explained in Section 2.5. The static analysis starting from constructs traced with dynamic analysis provides additional evidence regarding the relevance of CVE-2017-5638 (the vulnerability that was exploited in the Equifax breach). In addition to the execution of method FileUploadInterceptor.intercept(ActionInvocation) during step 3, the combination of static and dynamic analysis reveals that method MultiPartRequestWrapper.buildErrorMessage(Throwable, Object[]), included in struts2-core 2.3.24, is reachable with two calls from the traced method Dispatcher.wrapRequest(HttpServletRequest), as shown in Fig. 13. Its reachability is indicated with the red paw icons in the table containing the construct changes for CVE-2017-5638 (cf. the two right-most columns of the table in Fig. 10).

Fig. 13 Combined analysis reveals the reachability of MultiPartRequestWrapper.buildErrorMessage(...) in struts2-core

4.2 Mitigation

During the execution of dynamic analysis (steps 2 and 3) and static analysis (steps 4 and 5), touch points and reachable constructs are collected. They are the basis for the computation of the update metrics for the application at hand.

Figure 14 shows that for httpclient:4.5.2, one of the direct dependencies of the application, there are nine touch points between the application and the library. The application method ArchivePrinter.httpRequest2(String), for instance, calls the constructor HttpGet(String) (cf. first table in the figure). This invocation was observed during dynamic analysis, and was also found by static analysis (cf. rightmost columns in the first table in the figure). The second table of the figure shows the number of constructs of httpclient:4.5.2 by type. For example, of the 608 constructors (CONS), 199 were found reachable by static analysis, and 117 were actually executed during tests.

The table at the bottom of Fig. 14 shows the update metrics that can guide the developer in the selection of a non-vulnerable replacement for httpclient:4.5.2. Each table row corresponds to a release of Apache HttpClient that is not subject to any vulnerability known to our knowledge-base, hence, the developer is advised to choose among the three versions:

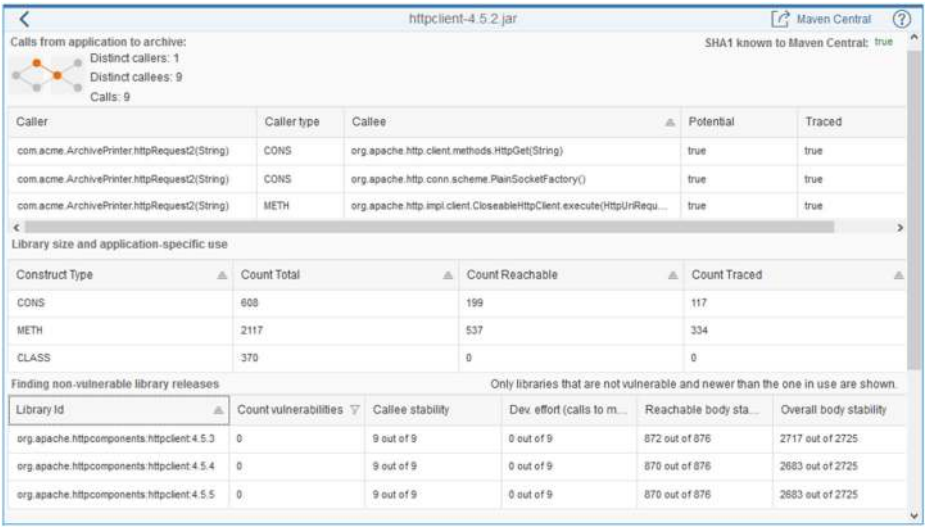

Fig. 14 Touch points, reachable constructs and update metrics for httpclient:4.5.2

Library Id	Coun...	Reachable body stability	Overall body stability
org.apache.struts:struts2-core:2.3.34	0	885 out of 887	3095 out of 3101
org.apache.struts:struts2-core:2.5.14	0	862 out of 887	2782 out of 3101
org.apache.struts:struts2-core:2.5.14.1	0	862 out of 887	2782 out of 3101
org.apache.struts:struts2-core:2.5.16	0	862 out of 887	2781 out of 3101

Fig. 15 Update metrics for struts2-core:2.3.24

4.5.3, 4.5.4 and 4.5.5. The callees of all touch points exist in all of those versions, hence, the update to any of those would not result in signature incompatibilities (cf. columns 3 and 4). The *RBS* metric indicates that 872 out of 876 reachable constructs of type method and constructor are also present in release 4.5.3 (870 out of 876 in 4.5.4 and 4.5.5). The *OBS* metric is also relatively high for all three non-vulnerable releases, thus, the developer would likely choose httpclient:4.5.5 in order to update the vulnerable library.

While the update decision is relatively straightforward for httpclient, it is more difficult for struts2-core, since there are non-vulnerable replacements from both the 2.3 and the 2.5 branch (cf. Fig. 15). Here, the *RBS* and *OBS* metrics indicate a more significant change of constructs between the current version struts-core:2.3.24 and the latest version of the 2.5 branch (*RBS*=862/887 and *OBS*=2781/3101) than between the current version and the latest version of the 2.3 branch (*RBS*=885/887 and *OBS*=3095/3101). Hence, the developer may be more inclined to stick to the 2.3 branch, thus updating to struts-core:2.3.34 rather than to struts2-core:2.5.16.

5 Empirical Evaluation

5.1 Goals of the Study

This section presents a study of the effectiveness of our approach when used to detect vulnerabilities in open source dependencies of real-world enterprise applications.

An open source implementation of the approach presented in this paper for the analysis of Java and Python applications is available on GitHub at https://github.com/eclipse/steady. The tool was first developed at SAP and has been released as open source in 2018. At the time of writing, the tool became an incubator project of the Eclipse Foundation under the name Eclipse Steady.

This study compares the results produced by Steady with those of OWASP DC in term of findings, i.e., vulnerabilities reported for a dependency of a project (or project module in case of multi-module projects). We selected OWASP DC, because it is an established open source tool for the detection of OSS vulnerabilities in Java projects, developed since 2012 and a mature flagship project of the OWASP foundation. Since OWASP DC does not offer neither reachability analysis nor mitigation capabilities, the comparison is limited to the *detection* of vulnerable dependencies.

More precisely, the goals of the study are the following:

– To determine the differences in the findings reported by the two tools;

– To evaluate the coverage of Steady's vulnerability database;
– To identify strengths and weaknesses of the two approaches.

5.2 Methodology

Selection of projects Eclipse Steady, in its current architecture, is continuously operated at SAP since December 2016. Depending on the tool configuration, the location of an application's versioning control system (VCS) is collected and persisted. In order to run the study, we selected all applications whose GitHub repository URL is available, and obtained a set of 444 large enterprise projects, either products or internal tools, under active development.

The selected projects are regularly scanned with Steady as part of SAP's secure development life cycle, whose goal is to ensure security and quality. As a result, OSS vulnerabilities are regularly mitigated by updating the vulnerable dependency and/or by introducing a security control in the library or in the application. Since our study aims at comparing the findings of Steady and OWASP DC, we targeted the project repositories at a point in time where the Steady findings were not yet addressed.

In most cases, the project descriptor (pom.xml file) is located in the root folder of the source code tree; however, the folder structure can vary from one project to another, and in some cases the descriptor is located in some sub-folder.

For practical reasons, we further restricted the analysis to Maven projects whose project descriptor (pom.xml file) is located in the root folder, thereby reducing the number of projects under analysis to 300.

Coping with the other cases (i.e., automatically locating the descriptor when it is in an arbitrary sub-directory) is not straightforward and would would have complicated our experimental setup considerably. Based on our experience, we have no reasons to believe that the projects we excluded would be easier (or harder) to analyze with Steady and OWASP DC; nonetheless, we cannot exclude that this selection criterion could have an impact on the generalizability of our findings.

Vulnerability data Steady and OWASP DC use vulnerability data coming from different sources; also, the nature of contents of the vulnerability knowledge-bases used by the two tools differs significantly.

OWASP DC uses the data feeds published regularly on the NVD website.[27] These feeds expose the content of each CVE record in JSON format. A record typically contains a textual description of the vulnerability, a set of references to other Web resources related to the vulnerability (e.g., vendor advisories, blog posts, GitHub issues, and the like), a severity assessment expressed using the Common Vulnerability Scoring System[28] (CVSS), and a set of Common Platform Enumeration[29] identifiers (CPE) that indicate which products are affected by the vulnerability at hand. Before each run, OWASP DC downloads these feeds (or updates them, if the last download is recent enough).

Steady on the other hand uses its own vulnerability knowledge-base (whose content is also released under the same license as the tool, Apache v.2.0) that we actively maintain since 2014. Essentially, the Steady knowledge-base contains, for each vulnerability, the

[27] https://nvd.nist.gov/vuln/data-feeds
[28] https://nvd.nist.gov/vuln-metrics/cvss
[29] https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe/

1	CVE-2017-4971	https://github.com/spring-projects/spring-webflow	57f2ccb66946943fbf3b3f2165eac1c8eb6b1523
2	CVE-2018-1000134	https://github.com/pingidentity/ldapsdk	8471904a02438c03965d21367890276bc25fa5a6
3	CVE-2016-8749	https://github.com/apache/camel	57d01e2fc8923263df896e9810329ee5b7f9b69
4	CVE-2017-1000393	https://github.com/jenkinsci/jenkins	d7ea3f40efedd50541a57b943d5f7bbed046d091
5	CVE-2018-8034	https://github.com/apache/tomcat	2835bb4e030c1c741ed0847bb3b9c3822e4fbc8a
6	CVE-2016-8747	https://github.com/apache/tomcat85	9601a937ff3b7ef5d04b2a0e950d0e44e1bb4cbd
7	CVE-2009-3555	https://github.com/apache/tomcat55	359c7ee17f5759cc99988e1cc9e971fe4a6ffad5
8	CVE-2017-12612	https://github.com/apache/spark	772a9b969aa179150aa216e9efd950e512e9d0b4
9	CVE-2013-4322	https://github.com/apache/tomcat	d6a9898125f34e593de426e8c7dabb0f224fc00f
10	CVE-2014-0119	https://github.com/apache/tomcat	50311bed8d87e452ff0e69838ba312c4fe899b2d

Fig. 16 Excerpt from the Steady knowledge-base

identifiers (i.e., repository URL and commit hash) of the commits that fix it, as shown in the excerpt in Fig. 16. The knowledge-base is described in further detail in Ponta et al. (2019), and is available for download at https://github.com/SAP/vulnerability-assessment-kb.

For this study, we used a more recent (thus larger) snapshot, than the one described in Ponta et al. (2019), covering 813 vulnerabilities (May 2019).

Retrieval of project data and setup of automated scans Using a set of Python scripts we retrieve the projects and we prepare them for the analysis. A first script is used to clone all repositories. A second script modifies the pom.xml files by removing the name tag (if present) to ensure that both Steady and OWASP DC report the findings referring to the projects by their artifactId.[30] A third python script generates (in the root folder of every project) two bash files containing the commands that are necessary to run the vulnerability detection with either tool. Steady was run using its Maven plugin, which offers the goal vulas:app[31] to retrieve all information about the application and its dependencies, and the goal vulas:report to generate a report containing the detected vulnerable dependencies (i.e, vulnerabilities whose code is included in application dependencies). The commands for Steady are as follows:

```
mvn -fn com.sap.research.security.vulas:plugin-maven:3.0.16:app
mvn -fn com.sap.research.security.vulas:plugin-maven:3.0.16:
report
```

The -fn flag forces Maven to build each project module, even in case a build failure occurs for one of them. Further information on how to run Steady can be found at https://eclipse.github.io/steady/. Note that the vulas:report goal of Steady must be run separately, once the vulas:app goal has been executed for all the modules in the case of multi-module projects, in order to aggregate the results for the entire project.

OWASP DC also offers a Maven plugin and the analysis was run using a goal (aggregate) that runs the analysis and generates the report. A dedicated configuration was used to include findings for dependencies declared in scope TEST as follows

[30] This is necessary because OWASP DC would use the project *name* to identify the project, unless that tag is missing. Steady uses the artifactId by default.

[31] The old name of Vulas is still visible in the goal names and in the source code of Eclipse Steady

```
mvn -fn -Dformat=ALL
    -DassemblyAnalyzerEnabled=false
    -DskipProvidedScope=false
    -DskipRuntimeScope=false
    -DskipSystemScope=false
    -DskipTestScope=false
org.owasp:dependency-check-maven:5.0.0-M2:aggregate
```

Note that both tools produce a single report for multi-module projects.

Collection and post-processing of findings Both Steady and OWASP DC produce reports in several formats. For gathering the results we used the JSON format for both tools. We collected the reports and processed them with help of a Jupyter notebook[32] that parses the reports, retrieves all the findings and gathers them in a data structure containing the following information:

- the name of the GitHub repository hosting the project under analysis;
- the artifactId of the affected project or module in the case of multi-module projects;
- the library dependency for which the finding was reported;
- the vulnerability identifier;
- the information whether Steady reported the finding;
- the information whether OWASP DC reported the finding;
- the CPE identifiers that OWASP DC used to associate the vulnerability to the dependency;
- the source for the vulnerabilities reported by OWASP DC (either the National Vulnerability Database (NVD) or RETIRE.JS were found in our study),
- the information whether Steady threw an error during the analysis;
- the information whether OWASP DC threw an error during the analysis.

As the study targets Java, we then filtered out the OWASP DC findings for source RETIRE.JS that provides JavaScript vulnerabilities, and obtained a set of 27769 findings.

Finally, we enriched the dataset with the following additional information:

- whether the vulnerability is available in the NVD, and
- whether the vulnerability has vulnerable constructs (rather than only involving configuration changes).

5.3 Analysis of findings

In this section we compare the findings reported by Steady and OWASP DC according to eight different categories.

Note that three of those categories relate to so-called *vulnerabilities without vulnerable constructs*, which are vulnerabilities whose fix does not involve any code change, but, for instance, changes of default configuration. Though the approach presented in this paper is based on the presence of vulnerable code, the existing implementation, which is used in a productive setting to scan commercial applications, has been extended to also cover such kinds of vulnerabilities. This is done by a manual effort to identify affected artifacts, e.g.,

[32] https://jupyter.org/

Table 1 Findings of Steady (S) and OWASP DC (O) by category

Category	Findings	Distinct vulnerabilities	Distinct vulnerabilities known to NVD
SO_C	2099	92	92
SO_{NC}	843	10	10
S_C	1829	148	128
S_{NC}	27	10	9
O_C	17062 (2909)	187	187
O_{NC}	1728 (237)	27	27
O_{NOKB}	4124 (633)	146	146
U_C	57	7	7

The letters of the category names indicate which tool reported the findings (S indicates *Steady only*, O indicates *OWASP DC only*, SO indicate *both tools*). The subscripts C, NC indicate that the findings refer to vulnerabilities with or without construct changes respectively. The subscript $NOKB$ is applied to the findings that refer to a vulnerability that is not present in the Steady knowledge-base (of course, these only apply to O findings). The category U_C contains the findings of Steady for which the tool could not determine automatically whether the constructs identified contain or not the correction to the vulnerability at hand. The figures reported in parentheses are the number of findings affecting project modules that Steady failed to analyze (e.g., because of dependency resolution issues)

using its Maven coordinates, thus, it is subject to false negatives in case an affected dependency is missed. Also note that the current implementation also considers code changes not involving methods or constructors (e.g., changes in a static block) as vulnerabilities without vulnerable constructs.

Accordingly, by using the Jupyter notebook described above, we cluster the findings in the following eight categories:

- S_C: Findings reported *only by Steady* for vulnerabilities *with* vulnerable constructs;
- S_{NC}: Findings reported *only by Steady* for vulnerabilities *without* vulnerable constructs;
- O_C: Findings reported *only by OWASP DC* for vulnerabilities *with* vulnerable constructs;
- O_{NC}: Findings reported *only by OWASP DC* for vulnerabilities *without* vulnerable constructs;
- O_{NOKB}: Findings reported *only by OWASP DC* for vulnerabilities *not present in the Steady knowledge-base*;
- SO_C: Findings reported *both by Steady and OWASP DC* for vulnerabilities *with* vulnerable constructs;
- SO_{NC} :Findings reported *both by Steady and OWASP DC* for vulnerabilities *without* vulnerable constructs;
- U_C: Findings reported *only by Steady* for which it is *unclear* whether the vulnerable or the fixed version of the construct is included.

Table 1 shows the analysis results by category:[33]

[33]Note that the findings also include vulnerabilities for dependencies in scope TEST, which are less critical (as they are not present at runtime) but still relevant. In our study, the vulnerable dependencies in scope TEST represent 4, 82% of the total.

Table 2 Manual review of sample findings for categories S_C, O_C, S_{NC}, and O_{NC}

Category	Proportion of TP
S_C	30/30 (0.943 ± 0.0568)
O_C	3/30 (0.145 ± 0.111)
S_{NC}	27/27 (1.0)
O_{NC}	19/30 (0.618 ± 0.163)

The table reports the proportion of true positives and confidence intervals (Wilson score, $\alpha = 0.05$). The class S_{NC} only has 27 elements, we reviewed them all

Categories SO_C and SO_{NC} represent cases where the tools agree and, as expected, those findings arise for vulnerabilities available in the NVD. In particular, 92 CVE identifiers for vulnerabilities with vulnerable constructs yield 2099 findings, and 10 CVE identifiers for vulnerabilities without vulnerable constructs yield 843 findings.

Categories S_C, S_{NC}, O_C, and O_{NC} represent cases where the tools disagree. It is important to notice that Steady requires the project dependencies to resolve successfully (by running, e.g., mvn dependency:resolve) in order to run the analysis, whereas OWASP DC seems to work even if some dependencies cannot be resolved (we were not able to establish whether such cases have an impact on the quality of the findings reported). As a result, categories O_C, O_{NC}, and O_{NOKB} also contain findings for projects (project modules) for which the dependency resolution failed and thus Steady was not executed. The number of findings reported for projects (i.e., "modules" in the Maven terminology) that could not be analyzed by Steady (because of dependency resolution issues) is shown in brackets for each category where it applies in Table 1. In order to establish whether the findings of categories S_C, S_{NC}, O_C, and O_{NC} are true or false positives, we manually reviewed 30 findings for each of them. The manual reviews were performed for findings of projects (project modules) for which both tools successfully ran. Also note that categories S_C and O_C are the most relevant for the evaluation of our code-based detection approach, whereas the comparison for categories S_{NC} and O_{NC} mostly evaluate the quality of our manual efforts in covering such cases.

Table 2 summarizes the results of the manual evaluation.

Category S_C (Vulnerabilities with construct changes, found only by Steady) Manual review showed that all the findings of Steady are true positives. It also revealed that 2 out of the 30 findings are also found by OWASP DC, however they show up in this category (and O_C resp.) rather than SO_C because Steady uses an identifier different from the CVE. This usually happens in case the vulnerability is added to Steady's knowledge-base at a time the CVE does not exist yet. The remaining 28 findings are false negatives for OWASP DC. We then investigated the possible reason for OWASP DC to miss those findings and, to the best of our knowledge,

- 3 are caused by vulnerabilities affecting library artifacts bundled in a Java Web application archive (WAR) dependency of the application (inside the WAR folder WEB-INF/lib);
- 6 are caused by vulnerabilities not available in the NVD;[34]

[34] Although it did not occur in our sample, this category would also contain findings for CVEs already published in the NVD but whose metadata are not available yet.

- 4 originate from vulnerabilities whose affected products listed by the NVD seem to considerably differ from the library metadata;
- for 1 finding it remains unknown why OWASP DC didn't report it;
- for 14 cases it is not clear why OWASP DC didn't even list the vulnerable dependency among the complete list of application dependencies.

As an example, OWASP DC fails at reporting CVEs for the Bouncy Castle[35] library whose artifact often uses the abbreviation bcprov whereas the vulnerabilities are assigned to CPEs using a variety of naming (e.g. cpe:2.3:a:bouncycastle:[4]bouncy_castle_crypto_package, cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle-java-cryptography-api) or are assigned to the overarching archive as in the case of CVE-2016-6644 that is assigned to cpe:2.3:o:google:android.

Category O_C (Vulnerabilities with construct changes, found only by OWASP DC**)** The manual review showed that 3 out of 30 findings are true positives as OWASP DC correctly identifies that vulnerable libraries where re-bundled in another one. Steady was not able to report those findings as the signature of the vulnerable construct was altered while being re-bundled (i.e., a new Java package name-space was created by concatenating the one of the outer and the one of the re-bundled library). The remaining 27 findings are false positives as the vulnerability does not affect the application dependency but another Java archive built from the same library project. This commonly happens when open source projects release a set of Java archives (with disjoint constructs), and when applications can depend on a subset of those, according to the required functionalities. As an example, according to the NVD description and to the advisory page https://pivotal.io/security/cve-2018-11040, CVE-2018-11040 applies to the part of the Spring Framework responsible for the handling of requests via JSONP; however OWASP DC reports it for dependency spring-expression-4.3.16.RELEASE.jar, which does not contain any related functionality or code.

Category S_{NC} (Vulnerabilities without construct changes, found only by Steady**)** Manual review showed that all 27 findings of Steady are true positives. To the best of our knowledge, OWASP DC did not detect them because

- 17 findings are for vulnerabilities affecting library artifacts bundled in a WAR dependency;
- 3 findings are for vulnerabilities that are not available in the NVD;
- 4 findings most likely are for dependencies whose metadata are too far from the affected products listed by the NVD;
- 3 findings are reported for dependencies that are not listed by OWASP DC (considering all application dependencies).

Category O_{NC} (Vulnerabilities without construct changes, found only by OWASP DC**)** The manual review showed that 19 findings are true positives whereas 11 are false positives. Among the false positives, 3 are for dependencies whose version is not listed as fixed by the NVD (though the library maintainers confirmed it is), and 8 findings are reported for an archive which is not affected (whereas another archive published starting from the same library project is). Most of the true positives (15) are not found by Steady as

[35] https://www.bouncycastle.org/

they are reported for libraries that re-bundle the affected artifact while modifying the construct signatures; the remaining 4 are due to some missing affected library versions in our knowledge-base.

To summarize, we can conclude that none of the tools includes all findings of the other. Steady didn't report any false positives but was subject to few false negatives. OWASP DC complements our approach in the sense that it is able to identify vulnerable libraries whose signature was altered while being re-bundled; however such findings needs to be manually identified while reviewing a non-negligible share of false positives. In particular, for categories S_C and O_C, which are those suitable to compare the effectiveness of our code-based approach, only a small fragment of OWASP DC findings are true positives.

Category O_{NOKB} Category O_{NOKB} in Table 1 contains findings of OWASP DC for vulnerabilities that are not known to Steady. The 4124 findings originate from 146 distinct vulnerabilities available in the NVD. We manually reviewed those vulnerabilities and concluded that:

- 80 are out of scope as they affect products that are not open source or do not contain any Java code;
- 12 affect the non-Java part of library artifacts that also include Java code; and
- 54 relate to vulnerabilities in Java libraries that are not covered by the Steady knowledge-base.

Note, however, that the 54 missing vulnerabilities also include, e.g., disputed CVEs for which the library developers question the presence of a vulnerability and, hence, did not create a fix. The 12 vulnerabilities whose vulnerable code is not in Java could only be covered by the current implementation of Steady as vulnerabilities without vulnerable constructs, but they could be covered by our code-centric approach once implemented for the respective languages.

It is interesting to observe that, despite the number of vulnerabilities (813) in the Steady knowledge-base being orders of magnitude smaller than the number of vulnerabilities considered by OWASP DC (the entire NVD, with 123031 vulnerabilities), when running our experiments only 66 distinct vulnerabilities could not be detected because they were missing from the Steady knowledge-base. This observation confirms that the criteria we use to maintain the vulnerability knowledge-base are correct (that is, we are focusing on the vulnerabilities that matter for enterprise products that incorporate OSS components). After running the experiments we incorporated many of those missing CVEs in the Steady knowledge-base (we left out CVEs that are disputed or that affect OSS software implemented in programming languages other than those currently supported by Steady, i.e., Java and Python).

Category U_C This category contains findings of Steady where the tool was able to detect the presence of constructs modified to fix a vulnerability but was not able to conclude whether it is contained in its vulnerable or fixed form (i.e., whether it contains the correction to the vulnerability). These are cases where the criteria of Section 2.2 were not able to automatically draw a conclusion and thus a manual review is needed.

We consider the number of findings in category U_C to be low, the reason being that the criteria presented in Section 2.2 were automatically evaluated for the majority of libraries and vulnerabilities. Table 3 shows the corresponding data collected when running Steady

Table 3 Evaluation of vulnerability detection criteria

Criteria	Conclusions	Verified OK	Verified Err
AST Equality	82490	9252	312
Intersection	4915	1226	88
Major release	6932	915	23
Minor release	4679	646	162
Greater release	18382	2233	5

for more than two years: The column *Conclusions* contains the number of cases for which the criteria were able to establish whether a given library version contained the vulnerable or the fixed code for a given vulnerability. Criterion *AST equality* was the one concluding in the majority of cases (82490), thereby establishing whether a vulnerability affects a given library version.

Columns *Verified OK* and *Verified Err* show in how many cases our manual reviews confirmed the results of the automated assessment (*Verified OK*) and in how many cases a wrong assessment had to be overwritten (*Verified Err*). Considering the conclusions manually reviewed, across all detection criteria, we observe that they concluded correctly in 97.5% of the cases.

However, when compared to the other criteria, the *minor release* criterion yields a relatively high-number of wrong assessments, roughly 20%, even though the criterion only considers versions having the same `MAJOR.MINOR`. All of those correspond to false positives, because a library version is wrongly assessed as being vulnerable. This observation relates to the question with which version a given vulnerability has been introduced, which is out of scope of the proposed approach. Nguyen et al. (2016) investigate this problem, and question the common approach of vulnerability databases such as the NVD to highlight *all* previous versions as being vulnerable, even those having smaller `MAJOR` or `MINOR` versions. In the context of using Steady in a productive setting, considering that the *minor release* criterion is responsible for the smallest number of automated assessments (4679 compared to a total of 117,398), and that the alternative would be to not have any assessments at all for those 4679 cases, it was decided to accept its false positive assessments.

6 Lessons Learned

6.1 From research prototype to industrial-grade solution

As introduced in the previous section, the approach presented in this paper was first implemented as a SAP-internal tool known as Steady. Initially, a research prototype was used to run pilots with a small number of development units, to clarify the needs of real-life development projects and to evaluate the viability of our approach.

From the feedback gathered from the users of the prototype, we could make two clear observations:

Precision Developers feared that using *yet another tool* (general static code analyzers as well as dynamic security testing tools were widely available to scan the code of open source

dependencies used in SAP's products) would mean being flooded with more findings referring to *potential* issues, many of which irrelevant (not exploitable) in practice. Especially the more security-aware among developers were reluctant to use metadata-based tools, because of their excessive rate of false positives and false negatives. As a matter of fact, frequent false positives can challenge the adoption of a tool as much as false negatives do. This observation confirmed the importance of our decision to strive for a *reliable, precise* method to detect the *actual presence* of vulnerable *code* in a given library artifact.

Unobtrusiveness and automation A tool whose adoption requires changes in the development practices is extremely hard to promote. We made the choice to integrate with the de-facto standard build processes and tools, making Steady a pluggable element of the automated build tool-chain that could be enabled with minimal configuration effort. Automating vulnerability scans has the additional benefit that issues are detected in a more timely manner, allowing better prioritization and effort allocation to identify and apply cost-effective fixes. Of course, the integration into commit-triggered build pipelines also comes with requirements regarding the tool's availability and performance: For instance, regarding the entire build infrastructure, developers expect a service availability of 99.95%.

After the prototyping phase, as the demand for Steady started to increase, the tool underwent a major re-implementation in order to make it more flexible and scalable, by adopting a micro-services architecture and deploying it onto SAP's internal cloud infrastructure. With the growth of the user base, we could observe that the tool is used differently depending on the phase of the development life-cycle. To deal also with legacy applications, built without modern tools such as Maven[36] or Gradle,[37] a dedicated command-line version of Steady is often used.

Quick vs. Deep scans Projects in the earliest phases of development, and particularly those that have not yet been released to customers, are mainly concerned with identifying as early as possible the dependency on a vulnerable library. The large majority of the projects that adopted Steady are scanned routinely, as part of an automated build pipeline, in which vulnerability detection (based purely on the analysis of the constructs of the application and its libraries as presented in Section 2.2) is performed at each commit; these projects perform a deeper analysis during nightly (or weekly) jobs. The performance offered by the vulnerability detection implemented in Steady is adequate for frequent scans, since its average execution time is about 76 seconds (static reachability analysis can take hours to complete, depending on the size of the application).

We observed that in this phase, the resistance to updates (especially, to minor releases) is quite low, and developers tend to update their open source dependencies in a short timeframe.

Deeper analysis of the vulnerability becomes more and more critical the closer the project gets to the date of release to customers, and stays so throughout the operational lifetime of the application because new vulnerabilities impacting the application could be discovered at any point in time after the release. After the release date, Steady is used mostly in manual scans, as a *program comprehension* tool, to achieve a deeper understanding of whether and how vulnerable code is reachable in a target application, what its concrete impact is, and what remediation options are available. We observed, for instance, that users run several iterations of dynamic and static scans (using the collected execution traces as entry

[36] https://maven.apache.org/
[37] https://gradle.org/

points for the call graph construction), whereby they observe and step-wise increase the test coverage throughout those iterations. In fact, the value of this combination of two reachability analysis techniques becomes evident when considering all applications scanned with our approach: vulnerable constructs are reachable, statically or dynamically, in 213 out of 1556 applications. In particular, we observed 960 pairs of applications and vulnerabilities whose constructs were reachable. In 113 cases, the reachability could only be determined through the combination of techniques, which represents a 11.7% increase of evidence that vulnerable code is potentially executable.

At the time of writing, Steady is the recommended tool at SAP to scan Java projects and it is used in over 1500 distinct development projects. The number of dependencies of the applications scanned (as of May 2019) is plotted in the distribution graph of Fig. 17. It becomes visible, for instance, that only very few of those 1500 projects have more than 1000 dependencies.

Despite its success, several challenges, pertaining to either organizational or technical aspects, remain to be addressed. In the remainder of this section we briefly discuss the most important ones.

6.2 Challenges

6.2.1 Developer opt-in vs. Central scans

To foster the uptake by developers, one has to minimize the required changes of development artifacts and processes. Plug-ins for common build tools support this goal, and the provision of templates for continuous integration pipelines goes as far as enabling in-depth Steady scans by means of boolean flags. Still, the tool adoption ultimately depends on the developer's initiative, even at that degree of integration and automation. Awareness and training campaigns, and other organizational measures are one means to this end. However, they require significant resources in large, international and heterogeneous development organizations. Future work aims at overcoming such issues by running fully-automated scans at a few central elements of an organization's development infrastructure, e.g., its source code or artifact repositories.

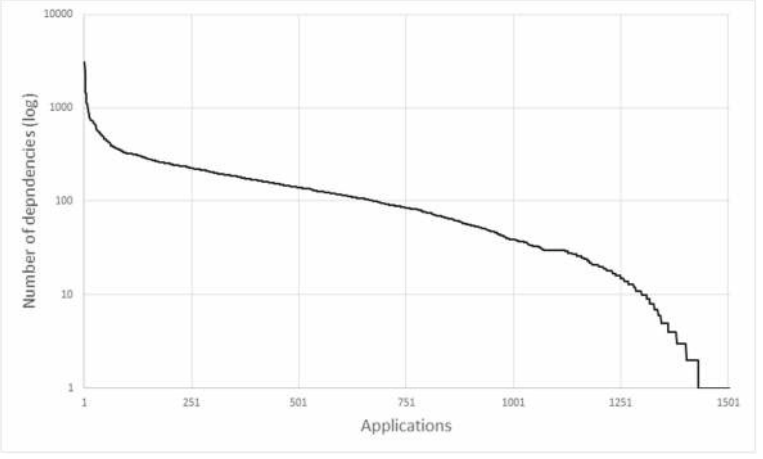

Fig. 17 Number of dependencies in the applications analyzed (May 2019)

6.2.2 Decision-support vs. Decision-making

Steady is essentially a *fact-finding* tool, whose goal is to provide comprehensive evidence that vulnerable code is *included* and *reachable* in a given application. Clearly, it cannot prove (decide) whether vulnerable code can or cannot be *exploited*. However, we occasionally observe this expectation, particularly when library updates are difficult and expensive. Again, security training and awareness campaigns are essential to ensure that developers internalize that they are responsible for drawing the final conclusion in regards to the exploitability of each vulnerability, especially before deciding to stick to a vulnerable version of an open source dependency. Also, while such conclusions and the decision to upgrade (or not) a library have to be documented, we decided to keep this functionality out of the scope of Steady and to stick to its fact-finding mission.

6.2.3 Vulnerability knowledge-base

The creation and maintenance of a comprehensive vulnerability database is key for our approach. However, the required vulnerability information is scattered across many different sources, e.g., public vulnerability databases, issue trackers and security advisories of individual projects, or source code repositories. Moreover, as discussed in Plate et al. (2015) and Alqahtani et al. (2016), different sources like the National Vulnerability Database and code repositories are difficult to integrate using the existing data. Based on a study that we performed on the NVD, we estimate our knowledge-base (Ponta et al. 2019) to cover about 90% of all vulnerabilities affecting Java open source projects and that are part of the NVD, which is consistent with the outcome of the review of all CVEs originating the findings of category O_{NOKB} in Section 5.

At the time of writing, we are working with other members of the SAP Security Research organization to experiment with machine-learning methods to automate the identification of fix commits. Such automated support has the potential to ease significantly the maintenance of a rich knowledge-base linking vulnerabilities to the corresponding source-code fixes. Our early attempts to tackle this problem are reported in Sabetta and Bezzi (2018), where source code changes are represented using a *bag-of-words* model; more recently we explored the use of AST embeddings and transfer learning in an approach called COMMIT2VEC (Cabrera Lozoya et al. 2019), but this very promising investigation is still in progress.

Complementary the introduction of automation in the curation of accurate vulnerability data, a coordinated approach to vulnerability disclosure and patch release across the open source community, through the governance exercised by established institutions (such as the Apache Software Foundation, the Eclipse Foundation, and the like) would be extremely beneficial to the maintenance of a comprehensive, high-quality, open vulnerability knowledge-base.

6.2.4 Shallow vs. Deep updates

The metrics of Section 3 support the update to non-vulnerable library versions. For transitive dependencies, however, developers are required to interfere with the transparency and automation of the dependency resolution mechanism. In fact, one would need to add the updated non-vulnerable version of the library as a direct dependency, thereby taking the risk of future incompatibilities. Therefore, the mitigation strategy could be optimized to recommend the update of the dependency that is "nearest" to the application (shallowest in the

dependency tree). As an example, to mitigate the vulnerable library version S_3 of Fig. 9, a newer version of S_1 can be recommended if it pulls in a fixed version of S_3.

6.2.5 Problematic types of vulnerabilities

By identifying a vulnerable dependency through the presence of vulnerable code, our approach is robust against false positives and false negatives, as typical for solutions based on the mapping of library and vulnerability metadata. As a drawback, the fraction of vulnerabilities whose fix does not involve any code change, e.g., those that are fixed by modifying a default configuration, cannot be covered by our code-centric approach described in Section 2.1. Nevertheless, Steady is able to cover such cases, which are relatively rare compared to code-related vulnerabilities, by flagging entire libraries as affected (cf. categories O_{NC} and S_{NC} in Section 5).

By assessing the exploitability of a vulnerability in terms of potential or actual code execution, our approach provides evidence about the application-specific usage of vulnerable code. However, this approach does not work with vulnerabilities that are due to the de-serialization of untrusted data, where the mere presence of so-called *de-serialization gadgets* in the application *classpath* can cause the application to be vulnerable (regardless of whether they can be reached during normal program execution). Attackers exploit the behavior of the Java serialization mechanism, which creates objects (through de-serialization) as long as the definition of the respective class is known, regardless of whether the application actively uses it or not.

7 Related work

7.1 Vulnerability detection

There exist several free (e.g., OWASP DC) and commercial tools (e.g., WhiteSource,[38] snyk,[39] Black Duck,[40] SourceClear[41]) for detecting vulnerabilities in OSS components. In Plate et al. (2015) we compared the vulnerability detection capabilities of our approach with state-of-the-art tools at the example of an application we developed to include vulnerable libraries. In this work we evaluate our approach against OWASP DC by analyzing 300 enterprise applications under development and observe that while all Steady findings are true positives, OWASP DC has a high-rate of false positives. To the best of our knowledge, the combination of static and dynamic techniques and the metric-based support to mitigation offered by our approach are unique.

OWASP DC is used in Cadariu et al. (2015) to create a vulnerability alert service and to perform an empirical investigation about the usage of vulnerable components in proprietary software. The results showed that 54 out of 75 of the projects analyzed have at least one vulnerable library. However the results had to be manually reviewed, as the matching of vulnerabilities to libraries showed low precision. Alqahtani et al. proposed an ontology-based approach to establish a link between vulnerability databases and software repositories (Alqahtani et al. 2016). The mapping resulting from their approach yields a

[38] https://www.whitesourcesoftware.com/oss_security_vulnerabilities/
[39] https://snyk.io/
[40] https://www.blackducksoftware.com/technology/vulnerability-reporting
[41] https://www.sourceclear.com/

precision that is 5% lower than OWASP DC. All these approaches and tools differ from ours in that they focus on vulnerability detection based on metadata, and do not provide application-specific reachability assessment nor mitigation proposals. Dashevskyi et al. propose a screening test based on slicing to identify which older versions of FOSS components are likely to be affected by newly disclosed vulnerabilities (Dashevskyi et al. 2018). Though this approach is able to quickly establish whether a given library version in a versioning control system (VCS) is affected, it relies on the availability of the complete history of the library (which may not be available in cases where, e.g., a library is migrated to a different VCS), and it does not link the VCS revisions to the library versions available in artifact repositories such as Maven Central, which are ultimately used to retrieve application dependencies. On the contrary our approach aims at establishing whether the actual application dependencies (the packaged artifacts) include vulnerable code.

7.2 Reachability of vulnerable code

In our previous work (Plate et al. 2015), we used reachability analysis (and in particular, dynamic analysis) to establish whether vulnerable code is actually executed and thus to make a first step beyond the mere detection of vulnerabilities. In this work, we extend (Plate et al. 2015) by including static reachability analysis and presenting a novel combination of static and dynamic reachability analysis. To the best of our knowledge, none of the existing works and tools combine static and dynamic reachability analysis, or provide mitigation proposals according to the application-specific use of library code. Zapata et al. (2018) performed a manual inspection of 60 npm projects to establish whether the vulnerable constructs of three vulnerabilities affecting the projects' dependencies were reachable. They found that up to 73.3% of the projects depending on vulnerable versions were actually safe. This work confirms that the analysis at level of libraries is indeed an overestimation and underlines the importance of a code-centric approach like the one we propose in this work.

7.3 Library update

The empirical study conducted by Kula et al. on library migrations of 4600 GitHub projects showed that 81.5% of them do not update their direct library dependencies, not even when they are affected by publicly known vulnerabilities (Kula et al. 2018). In particular, that study highlights the lack of awareness about security vulnerabilities. Considering 147 Apache software projects, Bavota G et al. (2015) studied the evolution of dependencies and found that applications tend to update their dependencies to newer releases containing substantial changes. Tools based on reward or incentives to trigger the update of outdated dependencies exist (e.g., the David project,[42] Greenkeeper[43]), however as shown in Mirhosseini and Parnin (2017), project developers are mostly concerned about breaking changes and mechanisms are needed to provide–next to transparency–a motivation for the update and confidence measures to estimate the risk of performing the update. By automatically detecting vulnerabilities, providing evidence about the reachability of the vulnerable code, and supporting mitigation via update metrics, our work addresses the need of motivating updates and estimating effort and risk.

[42] https://david-dm.org/
[43] https://greenkeeper.io/

Raemaekers et al. (2012) propose four metrics to measure the stability of libraries through time. In particular they consider the removal of units (constructs in our context), the amount of change in existing constructs, the ratio of change in new and old constructs, and the percentage of new constructs. Similar to our work, the metrics are meant to be representative for the amount of work required to update a certain library and thus they also consider usages of library methods in other projects. However the main focus of Raemaekers et al. (2012) is the *library*, and the metrics are used to measure its stability over time given a set of projects. Though some of their metrics *ingredients* can also be considered in our work, the metrics we propose are about the application-specific library usage. Moreover, our metrics benefit of our in-depth analysis of the application (e.g., some usages of the libraries that can only be observed with a combination of static and dynamic reachability analysis).

Raemaekers et al. studied breaking changes in Java library releases over seven years and showed that they occur with the same frequency in major and minor releases (Raemaekers et al. 2014). This shows that the rules of *semantic versioning*, according to which breaking changes are only allowed in major releases, are not always followed in practice, at least not in the Java/Maven ecosystem. They also showed that the top 3 most frequent breaking changes involve a deletion of methods, classes, fields, respectively. This result reinforces our belief that our update metrics based on measuring the removal of program constructs provides a critical information. However, regarding compliance with semantic versioning in other ecosystems than Java/Maven, Decan and Mens showed that it generally increases during an observation period of five years, but also depends on ecosystem-specific factors such as notations or maturity (Decan and Mens 2019).

Mileva et al. (2009) studied the usage of different library versions and provided a tool to suggest which one to use based on the choice of the majority of similar users. Our work differs from theirs, in that we provide quantitative measures to support the user in selecting a non-vulnerable library.

7.4 Library migration

Existing works on library migration (Hora and Valente 2015; Dagenais and Robillard 2009; Nguyen et al. 2010) are complementary to our approach in that they support developers in evolving their code to adapt to new libraries or library versions. Hora and Valente (2015) propose a tool that keeps track of API popularity and migration of major frameworks/libraries, amounting to 650 GitHub projects resulting in 320000 APIs at the time of publication. Dagenais and Robillard (2009) describe tools able to recommend replacements for framework methods accessed by a client program and deleted over time. Nguyen et al. (2010) present a tool able to recommend complex adaptations learned from already migrated clients or the library itself.

8 Conclusion

The unique contributions of this paper are *(i)* the detection of vulnerabilities based on abstract syntax tree comparisons, *(ii)* the use of static reachability analysis and its combination with dynamic reachability analysis to support the application-specific assessment and mitigation of open source vulnerabilities, and *(iii)* the comparative empirical study to evaluate the effectiveness of our approach to vulnerability detection. This approach further advances the code-centric detection and dynamic reachability analysis of vulnerable dependencies we originally proposed in Plate et al. (2015).

The accuracy and application-specific nature of our method improves over state-of-the-art approaches, which commonly depend on metadata. Steady, the implementation of our approach for Java, was chosen by SAP among several candidates as the recommended OSS vulnerability scanner. Since December 2016, it has been used for over one million scans of about 1500 applications, which demonstrates the viability and scalability of the approach.

The variety of programming languages used in today's software systems pushes us to extend Steady to support languages other than Java. However, as demonstrated by our comparative evaluation of Section 5, fully-qualified names can be inadequate to uniquely identify the relevant program constructs in certain languages, so we are considering the use of information extracted from the construct bodies.

Finally, the problem of systematically linking open source vulnerability information to the corresponding source code changes (the fix) remains open. Maintaining a comprehensive knowledge-base of rich, detailed vulnerability data is critical to all vulnerability management approaches and requires considerable effort. While this effort could be substantially reduced creating specialized tools, we strongly believe that the maintenance of this knowledge-base should become an industry-wide, coordinated effort, whose outcome would benefit the whole software industry.

References

Alqahtani SS, Eghan EE, Rilling J (2016) Tracing known security vulnerabilities in software repositories–a semantic web enabled modeling approach. Sci Comput Program 121:153–175

Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2015) How the Apache community upgrades dependencies: An evolutionary study. Empirical Soft Eng 20(5):1275–1317. https://doi.org/10.1007/s10664-014-9325-9

Cabrera Lozoya R, Bzumann A, Sabetta A, Bezzi M (2019) Commit2vec: Learning distributed representations of code changes. arXiv:1911.07605

Cadariu M, Bouwers E, Visser J, van Deursen A (2015) Tracking known security vulnerabilities in proprietary software systems. In: 2015 IEEE 22nd international conference on software analysis, evolution and reengineering (SANER), IEEE, pp 516–519

Dagenais B, Robillard MP (2009) SemDiff: Analysis and recommendation support for API evolution. In: Proceedings of the 31st international conference on software engineering. IEEE Computer Society, Washington, pp 599–602, https://doi.org/10.1109/ICSE.2009.5070565. ICSE '09

Dashevskyi S, Brucker AD, Massacci F (2018) A screening test for disclosed vulnerabilities in foss components. IEEE Trans Softw Eng 2018:1–1. https://doi.org/10.1109/TSE.2018.2816033

Decan A, Mens T (2019) What do package dependencies tell us about semantic versioning? IEEE Transactions on Software Engineering

Falleri J, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and accurate source code differencing. In: ACM/IEEE international conference on automated software engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, pp 313–324. https://doi.org/10.1145/2642937.2642982

Fluri B, Wuersch M, PInzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans Softw Eng 33(11):725–743. https://doi.org/10.1109/TSE.2007.70731

Hora A, Valente MT (2015) Apiwave: Keeping track of API popularity and migration. In: 2015 IEEE international conference on software maintenance and evolution (ICSME), pp 321–323. https://doi.org/10.1109/ICSM.2015.7332478

Kula RG, Germán DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? - an empirical study on the impact of security advisories on library migration. Empirical Software Engineering 23(1):384–417. https://doi.org/10.1007/s10664-017-9521-5

Landman D, Serebrenik A, Vinju JJ (2017) Challenges for static analysis of Java reflection: literature review and empirical study. In: Uchitel S, Orso A, Robillard MP (eds) Proceedings of the 39th international conference on software engineering, ICSE 2017. IEEE / ACM, Buenos Aires, pp 507–518. https://doi.org/10.1109/ICSE.2017.53

Mileva YM, Dallmeier V, Burger M, Zeller A (2009) Mining trends of library usage. In: Proceedings of the joint international and annual ERCIM workshops on principles of software evolution (IWPSE) and software evolution (Evol) Workshops. ACM, New York, pp 57–62, https://doi.org/10.1145/1595808.1595821. IWPSE-Evol '09

Mirhosseini S, Parnin C (2017) Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: Proceedings of the 32Nd IEEE/ACM international conference on automated software engineering. IEEE Press, Piscataway, pp 84–94. ASE. http://dl.acm.org/citation.cfm?id=3155562.3155577

Mostafa S, Rodriguez R, Wang X (2017) Experience paper: A study on behavioral backward incompatibilities of Java software libraries. In: Bultan T, Sen K (eds) Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis. ACM, Santa Barbara, pp 215–225. https://doi.org/10.1145/3092703.3092721

Nguyen HA, Nguyen TT, Wilson G Jr, Nguyen AT, Kim M, Nguyen TN (2010) A graph-based approach to API usage adaptation. In: Proceedings of the ACM international conference on object oriented programming systems languages and applications. ACM, New York, pp 302–321, https://doi.org/10.1145/1869459.1869486. OOPSLA '10

Nguyen VH, Massacci F (2013) The (un)reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities. In: Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security. ACM, New York, pp 493–498, https://doi.org/2484313.2484377. ASIA CCS '13

Nguyen VH, Dashevskyi S, Massacci F (2016) An automatic method for assessing the versions affected by a vulnerability. Empirical Software Engineering 21(6):2268–2297. https://doi.org/10.1007/s10664-015-9408-2

OWASP Foundation (2013) OWASP Top 10 - 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10

OWASP Foundation (2017) OWASP Top 10 - 2017. https://www.owasp.org/index.php/Top_10_2017-Top_10

Plate H, Ponta SE, Sabetta A (2015) Impact assessment for vulnerabilities in open-source software libraries. In: 2015 IEEE international conference on software maintenance and evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015, pp 411–420. https://doi.org/10.1109/ICSM.2015.7332492

Ponta SE, Plate H, Sabetta A (2018) 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. IEEE Computer Society, Madrid, pp 449–460

Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C (2019) A manually-curated dataset of fixes to vulnerabilities of open-source software. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), pp 383–387. https://doi.org/10.1109/MSR.2019.00064

Raemaekers S, van Deursen A, Visser J (2012) Measuring software library stability through historical version analysis. In: 2012 28th IEEE international conference on software maintenance (ICSM), pp 378–387. https://doi.org/10.1109/ICSM.2012.6405296

Raemaekers S, van Deursen A, Visser J (2014) Semantic versioning versus breaking changes: A study of the Maven repository. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp 215–224. https://doi.org/10.1109/SCAM.2014.30

Sabetta A, Bezzi M (2018) A practical approach to the automatic classification of security-relevant commits. In: 34th IEEE international conference on software maintenance and evolution (ICSME)

Snyk (2017) Which of the owasp top 10 caused the world's biggest data breaches? https://snyk.io/blog/owasp-top-10-breaches/, accessed: 2018-03-30

Snyk (2019) The state of open source security report. https://res.cloudinary.com/snyk/image/upload/v1551172581/The-State-Of-Open-Source-Security-Report-2019-Snyk.pdf, accessed: 2019-05-27

Synopsys Black Duck (2019) 2019 open source security and risk analysis. Tech. rep., Black Duck Software

Zapata RE, Kula RG, Chinthanet B, Ishio T, Matsumoto K, Ihara A (2018) Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, pp 559–563. https://doi.org/10.1109/ICSME.2018.00067