Vulnerable Open Source Dependencies: Counting Those That Matter

Ivan Pashchenko
University of Trento, IT
ivan.pashchenko@unitn.it

Henrik Plate
SAP Security Research, FR
henrik.plate@sap.com

Serena Elisa Ponta
SAP Security Research, FR
serena.ponta@sap.com

Antonino Sabetta
SAP Security Research, FR
antonino.sabetta@sap.com

Fabio Massacci
University of Trento, IT
fabio.massacci@unitn.it

ABSTRACT

Background: Vulnerable dependencies are a known problem in today's open-source software ecosystems because OSS libraries are highly interconnected and developers do not always update their dependencies.

Aim: Our paper addresses the over-inflation problem of academic and industrial approaches for reporting vulnerable dependencies in OSS software, and therefore, caters to the needs of industrial practice for correct allocation of development and audit resources.

Method: Careful analysis of deployed dependencies, aggregation of dependencies by their projects, and distinction of halted dependencies allow us to obtain a counting method that avoids over-inflation. To understand the industrial impact of a more precise approach, we considered the 200 most popular OSS Java libraries used by SAP in its own software. Our analysis included 10905 distinct GAVs (group, artifact, version) in Maven when considering all the library versions.

Results: We found that about 20% of the dependencies affected by a known vulnerability are not deployed, and therefore, they do not represent a danger to the analyzed library because they cannot be exploited in practice. Developers of the analyzed libraries are able to fix (and actually responsible for) 82% of the deployed vulnerable dependencies. The vast majority (81%) of vulnerable dependencies may be fixed by simply updating to a new version, while 1% of the vulnerable dependencies in our sample are halted, and therefore, potentially require a costly mitigation strategy.

Conclusions: Our case study shows that the correct counting allows software development companies to receive actionable information about their library dependencies, and therefore, correctly allocate costly development and audit resources, which is spent inefficiently in case of distorted measurements.

ESEM '18, October 11–12, 2018, Oulu, Finland
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5823-1/18/10…$15.00
https://doi.org/10.1145/3239235.3268920

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software libraries and repositories**; **Open source model**;

KEYWORDS

Vulnerable Dependency, Open-Source Software, Mining Software Repositories

ACM Reference Format:
Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable Open Source Dependencies: Counting Those That Matter. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239235.3268920

1 INTRODUCTION

The inclusion of free open-source software (OSS) components in commercial products is a consolidated practice in the software industry: as much as 80% of the code of the average commercial product comes from OSS [13]. SAP is an active user of and contributor to OSS[1]. In this paper we report our hands-on experience on the industry relevant measurement of vulnerable dependencies in OSS.

Current dependency analysis methodologies are based on assumptions that are not valid in an industrial context. They may not distinguish dependency scopes [8] (which may lead to reporting non-exploitable vulnerabilities), or consider only direct dependencies [4] (although security issues may be introduced transitively [9]). On the other hand, dependency analysis methodologies miss several important factors at all. For example, we could not find studies that distinguish dependencies, whose development had been suspended for unspecified time, although they may still introduce bugs and security vulnerabilities transitively. Additionally, current dependency management practices do not consider the fact that some dependencies are maintained and released simultaneously, and therefore, should be treated as a singular unit, while constructing dependency trees and reporting results of a dependency study.

These issues lead to an inefficient allocation of costly development and audit resources due to the distorted measurements of vulnerable dependencies.

Hence, we make the following contributions:

[1] https://archive.sap.com/documents/docs/DOC-29056

- A precise approach, that caters to the needs of industrial practice, for reliable measurement of vulnerable dependencies in open-source software;
- A tool to perform large-scale studies of (Maven-based) OSS libraries and to determine whether any of their dependencies are affected by known vulnerabilities
- An empirical study of 10905 library instances of the 200 Java Maven-based open-source libraries that are most frequently used in SAP software.

We found that as many as 20% of the dependencies affected by a known vulnerability are *not deployed*, and therefore, do not introduce vulnerabilities in the dependent library instances. Also, we found that the developers of the analyzed libraries could directly fix 82% (45% more comparing to a traditional approach) of their vulnerable dependencies. Our study indicates that, under a conservative model to characterize halted dependencies, 14% of the total number of dependencies are halted, and therefore, do not receive updates (including security fixes). Such dependencies should be used with caution, since mitigations of their vulnerabilities are costly.

2 TERMINOLOGY

In this paper we rely on the terminology established among practitioners and used in well-known dependency management tools such as Apache Ivy[2] and Apache Maven[3]:

- A *library* is a separately distributed software component, which typically consists of a logically grouped set of classes (objects) or methods (functions). To avoid any ambiguity, we refer to a specific version of a library as a *library instance*.
- A *dependency*[4] is a library instance, some functionality of which is used by another library instance (the *dependent* library instance).
- A dependency is *direct* if it is *directly* invoked from the dependent library instance.
- A *dependency tree*[5] is a representation of a software library instance and its dependencies where each node is a library instance and edges connect dependent library instances to their direct dependencies.
- A *transitive dependency* is connected to the root library instance of a dependency tree through a path with more than one edge.
- A *project* is a set of libraries developed and/or maintained together by a group of developers. Dependencies belonging to the same project of the dependent library instance are *own dependencies*, while library instances maintained by other projects are *third-party dependencies*.
- A *deployed* dependency is actually delivered with the application or system that uses it, while a *non-deployed* dependency is only needed at the time of development (e.g., for testing) but is not a part of the artifact that is eventually released and operated in a production environment.

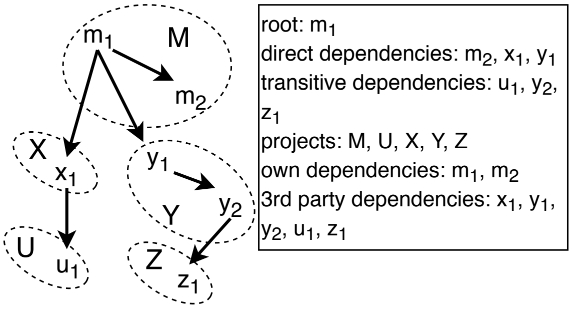

m1
M
m2
X
x1
y1
Y
y2
U
u1
Z
z1
root: m1
direct dependencies: m2, x1, y1
transitive dependencies: u1, y2, z1
projects: M, U, X, Y, Z
own dependencies: m1, m2
3rd party dependencies: x1, y1, y2, u1, z1

Figure 1: Dependency tree

- A library instance is *outdated* if there exists a more recent instance of this library at the time of analysis. A *halted* library is such that the next estimated release time has been passed by far based on the interval of past releases (see §3 and §4.2).

To illustrate how this terminology is used in practice, we refer to Figure 1, which depicts the dependency tree for a library instance m_1. The library instance under analysis m_1 is the root, m_2, x_1, and y_1 are direct dependencies, while u_1, y_2, and z_1 are transitive dependencies. Library instances m_1, m_2 and y_1, y_2 are *own dependencies* of projects M and Y respectively, while library instances x_1, y_1, y_2, u_1, and z_1 are *third-party* dependencies of project M.

3 PROBLEM STATEMENT

The construction of the complete bill of materials (BoM) of a project is a necessary preliminary step to determining which dependencies of a project are vulnerable and assessing the risk they represent and the effort needed to mitigate it.

Several approaches exist for analyzing software dependencies (Table 4). However, they do not provide a reliable measurement of the situation with software dependencies, because they do not consider several key aspects:

- a non-negligible number of dependencies that appear in the BoM could not be possibly exploited because they are only used at development time (e.g., for testing purposes) and are not delivered with the actual software system in operation;
- libraries from the same project should be treated differently than third-party libraries (the former should be maintained by the same team, which should fix them rather than wait for another project team to release a new non-vulnerable version);
- the mitigation strategy that should be used to deal with each vulnerable library depends on the above two and on the fact that the library might not be maintained any longer (*halted*).

RQ1: How many actually vulnerable dependencies does a library have?

A dependency tree for a library may include dependencies that are used only for testing or development purposes and are not deployed in the released version. Since they are not shipped with the product, they cannot possibly be exploited. Hence, allocating resources to fix or mitigate these vulnerabilities is pointless. This is well-known to software developers [8]:

> "…In this case, it's a test dependency, so the vulnerability doesn't really apply …"

[2] http://ant.apache.org/ivy/history/latest-milestone/ivyfile/dependency.html

[3] https://maven.apache.org/pom.html#Dependencies

[4] For the sake of consistency with the terminology used in Maven, we use the term 'dependency' to denote a node (not an edge) of a dependency tree.

[5] We use the term *dependency tree* rather than *dependency graph* to be consistent with Maven, where the resolved graph of dependencies never contains cycles and each dependency appears once.

> "…It's only a test scoped dependency which means that it's not a transitive dependency for users of XXX so there is no harm done …"

Several recent works [3, 4, 8] do not mention explicitly that they consider only deployed dependencies (we discuss this further in Section 8). Indeed, the very quotes above in [8] show that the paper actually included such dependencies in its study. As a result vulnerable dependency count may become severely over-inflated.

RQ2: Who is responsible for vulnerable dependencies?

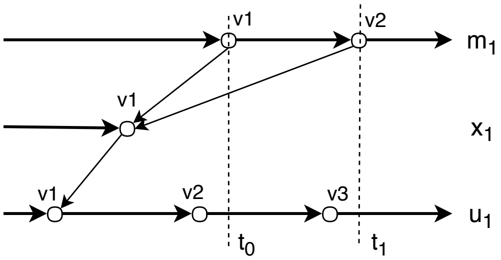
A key question for the user of a vulnerable library is to attribute responsibility for fixing it (or avoiding projects with bad security discipline altogether). Developers of a software project are responsible for own code of their project and its direct dependencies (i.e., to keep them up-to-date). Although the concept seems intuitively simple, the following issues may occur:

- *Own vs third-party dependencies*: Failure to distinguish them may incorrectly present as an insecure ecosystem with several vulnerable dependencies (a "dependency hell" [10]) what in reality is just a project that has broken its components into several libraries and did not fix its own vulnerable code.
- *Direct vs transitive dependencies*: A dependency tree may include several library instances that belong to the same project. Such dependencies should not be considered separately, since an update of one of those dependencies would automatically bring the new versions of all other dependencies from the same project. Hence, some transitive dependencies may actually be controlled directly from the project under analysis.

To illustrate these issues, we refer to the example of a dependency tree shown in Figure 1. Both library instances m_1 and m_2 belong to the same project M. They are maintained and released simultaneously by the same team: if developers wanted to fix a bug in m_2, then they include the fix within the new release of the project M and, at the same time, should update the versions of all their own libraries of project M (m_1 and m_2). If they don't do so this might be a sign of a poor management within the project.

Suppose now that m_2, y_2, and z_1 are affected by known security vulnerabilities.

- Library instance m_2 is an *own* dependency of m_1 because m_1 and m_2 belong to the same project M, and therefore, the source code of m_2 is under the control of developers of that project. Hence, the vulnerability should be fixed as part of the development of the project itself, i.e., by directly changing its source code. While from the perspective of the build system, m_2 is just a dependency, in practice it is a piece of vulnerable code developers are shipping as part of their project.
- Dependency y_2 does not appear within configuration files of project M, but it comes together with y_1 (since both y_1 and y_2 belong to project Y), which, in turn, is a direct dependency of project M. Hence, developers of M can control the version of y_2 by selecting a suitable y_1: if a newer version of y_1 is released, they should update project M to use it.
- Dependency z_1 appears within the dependencies of project M, since it is introduced transitively through project Y (via y_2). Usage of dependency z_1 cannot be controlled without changing M and transforming the (transitive) dependency z_1 into a direct dependency of the project. Since this would break the "black-box" dependency management principle, such a solution is not likely to be adopted. As a matter of fact, it is a responsibility of the developers of project Y to keep the version of z_1 up-to-date.



Library m_1 has a halted dependency x_1. In case a vulnerability is discovered in x_1 or its dependency u_1, there would be no version of x_1 that fixes such a vulnerability or adopts a fixed version of u_1.

Figure 2: Halted dependency

Proper distinction of these cases is very important for selection of an appropriate mitigation strategy and correct allocation of development resources for fixing security issues introduced by vulnerable dependencies.

RQ3: How many direct dependencies can be actually fixed?

If an outdated direct dependency is affected by a known vulnerability, the simplest solution to mitigate this vulnerability is to update the dependent library to use the fixed version of the dependency [16]. However, this becomes impossible, if an OSS library becomes inactive [8]:

> "…our project has been inactive and production has been halted for indefinite time"

If a security vulnerability is discovered in a no longer actively developed library, there would be no version of this library that fixes the vulnerability. Hence, being a dependency, this library will introduce the vulnerability to all its dependents. Additionally, a halted dependency may transitively introduce outdated dependencies and expose the root library instance to bugs and security vulnerabilities (Figure 2): the root library instance m_1 depends on the last version of halted dependency x_1, which, in turn, uses an "alive" dependency u_1. Although both versions $v1$ and $v2$ of library m_1 technically use the latest available version of direct dependency x_1, there would always present outdated transitive dependency u_1. Hence, the halted dependencies should be considered separately.

Clearly, the presence of halted dependencies has a major impact on a company maintenance strategy. Indeed, any user of library x_1 would not obtain any benefit from switching to its latest version. The vulnerable version of u_1 would always be present. A different mitigation strategy might be needed: (i) contribute to the halted library, i.e., to develop its new release; or (ii) fork the halted library and continue its maintenance as part of the dependent library.

4 COUNTING DEPENDENCIES IN MAVEN

Considering the popularity and industrial relevance of Java[6], in the following we demonstrate our approach on Java projects.

Over the past decade, Apache Maven established itself as a standard solution in the Java ecosystem for dependency management and other tasks related to build processes. Other solutions exist, such as Apache Ivy [7] and Gradle (which is gaining popularity)[8], however Maven[9] still has the largest share of users[10]. Hence, we use it to demonstrate the proposed mitigations for each problem described in Section 3, although the concepts presented below can be easily extended to other dependency management systems.

In Maven the name of a component is standardized[11] and represented as *groupId*:*artifactId*:*version*. Hence:

- a "project" may be referenced as Maven *groupId*
- a "library" corresponds to *groupId*:*artifactId* (GA)
- a "library instance" corresponds to the name of Maven component *groupId*:*artifactId*:*version* (GAV)

4.1 Dependency resolution

For each of the library instances in our sample, we use Maven to determine the complete set of dependencies. Before doing so, Maven requires that the Project Object Model (POM) files be installed in the local repository. Once a POM is installed locally, we use the standard Maven goals[12] *dependency:tree* and *dependency:resolve* to construct the dependency tree of each POM and to resolve conflicts and duplications.

4.2 Post-processing of the results

The next step of our data collection process is a post-processing of the data obtained after the dependency resolution step to address the problems discussed in Section 3.

Filter non-deployed dependencies. To control whether a dependency is deployed with an artifact, Maven provides a possibility for a software developer to specify the *scope*. The dependencies with scopes *provided* and *test* are used only for development purposes and are not shipped with a released artifact, hence, we do not consider them for the further analysis as non-deployed dependencies.

Dependency grouping. The Maven dependency resolution process starts from the *POMs* under analysis as the major source of the necessary information to build the dependency trees. However, at the final step of our analysis, the vulnerable dependency represents the most valuable asset. Hence, we perform the final aggregation of the results in the opposite direction, i.e., considering the paths from vulnerable dependencies to libraries under analysis:

- we group all GAVs with the same groupId within one path and substitute them in the path with the GAV, closest to the vulnerable GAV

Consider the example of a dependency tree from Figure 1: let dependencies x_1 and z_1 be affected by known security vulnerabilities. Initially there are two paths from vulnerable dependencies to the analyzed root library: (x_1, m_1) and (z_1, y_2, y_1, m_1). In the second path library instances y_1 and y_2 belong to the same project Y, hence, they are grouped. So, the analysis results in two vulnerable paths: (x_1, m_1) and (z_1, y_2, m_1).

Identification of halted dependencies. Public software package repositories keep all published library instances, since there is a possibility to break a build of a library in case a library dependency is removed [7]. So, even if a certain library is no longer maintained, it is still available for download from a software package repository.

At the same time, when selecting a mitigation strategy, software developers need to know that a fixed version of a vulnerable dependency is going to appear (otherwise, a costly mitigation strategy is required). Some projects publish information about their decision to stop maintenance of certain libraries. Monitoring these sources of information requires tracking notifications for every dependency separately; some projects do not publish such data frequently, or stop publishing it at all. At this time, there is no systematic and scalable approach to determine if an OSS component has reached the end of its lifetime.

We propose to consider the amount of time library developers require to release a new version for determining whether a library development becomes halted.

Some libraries may have varying time intervals between releases due to different release strategies adopted within development teams, as well as the maturity of a certain library: at earlier stages of development it needs to have more updates than an established library with a long development history. An example of a mature library is the Apache commons-logging package. Released on 2007-11-26 version 1.1.1 was the latest available version for more than 5 years till the release of version 1.1.2 on 2013-03-16.

Since the time difference between recent releases should have bigger impact on the *Last release interval* comparing to the time difference between older releases, the typical statistical model that describes such a process is a simple Exponential Smoothing model [2]:

$$\text{Last release interval} = \quad \alpha \sum_{i=0}^{n} \left\{ (1-\alpha)^i * \text{Release time}_{n-i} \right\}$$

$$\text{Expected release date} = \quad \text{Last release} + \text{Last release interval}$$

where Release time$_i$ is the time needed to release the i-th version of a library, $0 < \alpha < 1$ is the smoothing parameter that shows how fast the influence of previous time intervals decreases[13]. We estimate the *Expected release date* for a library by adding the *Last release interval* to the release date of the latest available version of the library. Then we determine the status of the library as follows:

- *next release date* < *TIME*: the library is *halted*
- *next release date* ≥ *TIME*: the library is *alive*

TIME represents the date, for which the library status is calculated. In our study, for each analyzed library instance we will identify its release date and use it to calculate whether any of the

[6] Java is estimated to be the most popular programming language since 2004, according to the two indexes used by IEEE Spectrum (http://spectrum.ieee.org/) to assess popularity of a programming language: (i) Tiobe index (http://www.tiobe.com/tiobe-index/), which combines data about search queries from 25 most popular websites of Alexa; and (ii) PYPL index (http://pypl.github.io/PYPL.html), which uses Google search queries.

[7] http://ant.apache.org/ivy/

[8] https://gradle.org/

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

[10] https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/

[11] https://maven.apache.org/guides/mini/guide-naming-conventions.html

[12] In Maven terminology, *goal* can be thought of as a synonym of a *command.*

[13] Our hands-on experience (which is also supported by the observation of released dates for the analyzed libraries) suggest, that the last three releases have the major impact on the *Expected release date* of a library, and therefore, in this paper we count $\alpha = 0.6$. For libraries with less than 3 releases, we take the *Last release interval* equal 3 months.

dependencies were halted. To know the current status, *TIME* should be equal to the current date.

The proposed model based on release dates is conservative, since it provides the lower bound for the estimation of the *Expected release date* for a library. Hence, it is more likely to be affected by false positives, i.e., to classify a library as halted, when it is still under development. However, such finding would mean, that a library does not receive a fix for a long period of time, which increases chances of a zero day vulnerability to be exploited. Hence, even in case of "false positives", our model provides a valuable information for a software developer.

To examine the reliability of the proposed model, we randomly selected 100 distinct library instances identified to be halted. Then we manually looked for any available information of whether a new version of a halted library is planned to be released. For this purpose, for every halted library we checked (when possible) (i) their software repositories, (ii) release pages, or (iii) other available resources returned by Google searches. The manual analysis did not reveal any libraries falsely reported to be halted.

4.3 Identification of vulnerabilities

Once all the dependencies of each subject project are determined, we lookup any known vulnerabilities associated with them. Most approaches for the identification of vulnerable dependencies use the NVD and try to map CPE names to the language-specific naming, e.g., Maven coordinates. This is, for example, the case for OWASP Dependency Check[14]. Such approaches suffer from both false positives and false negatives. In particular, many false positives come out of the fact that CPE names are more coarse grained than Maven coordinates: a vulnerability only affecting the poi-ooxml artifact within the Apache Poi project, would be assigned to the entire project in the NVD, thereby resulting in false positives whenever an application only uses 'Poi' artifacts other than poi-ooxml. This might be further exacerbated since the NVD might use an over-approximation rule 'X and all previous versions' for marking vulnerable versions (See, for example, [11, 12] for the study of browser vulnerabilities and the large presence of false positives).

To improve our precision we leverage on code based approaches to vulnerability detection such as Ponta et al. [15] and Dashevskyi et al. [5]. Starting from known vulnerabilities from the NVD, advisories, bug tracking systems, etc., the commit fixing the vulnerability is identified manually and analyzed resulting in a list of code changes. All software constructs (e.g., constructors, methods) included in such list are the so-called *vulnerable code*. The creation of such knowledge is a one-time effort for each vulnerability. Then, for every analyzed project, the list of all own libraries of the project and all its dependencies is collected by performing a code-level matching of the vulnerable fragment following the approach of [15]. Whenever the vulnerable code fragment is contained within a dependency, the corresponding vulnerability is automatically reported for our analysis.

4.4 Dependency resolution tool

To automate our dependency study we implemented a tool that:

Table 1: Descriptive statistics of the library sample

We considered the 200 most popular OSS Java libraries used by SAP in its own software, which resulted in 10905 distinct GAVs when considering all library versions.

	mu	median	$sigma$	min	max	Q1	Q3
#GAVs	54.52	35.0	49.24	1.0	248	15.0	87.0
#dependencies	11.89	3.0	18.54	0.0	131	0.0	16.0
#direct deps	4.26	2.0	6.80	0.0	51	0.0	6.0
#transitive deps	7.63	1.0	13.56	0.0	92	0.0	11.0
#usages	55.96	5.0	508.41	1.0	29 472	1.0	23.0

- wraps *dependency:tree* and *dependency:resolve* Maven commands, which helps us get a more manageable (and a machine-readable) representation of the results of the resolution mechanism. This allows us to construct the resolved dependency tree for each analyzed library instance.
- uses the code-based approach of [15] to annotate dependency trees with the vulnerability data at our disposal. In particular, when a vulnerable library instance is found among the dependencies of one of the analyzed root libraries, our tool produces in the output (i) the identifier of the vulnerability, (ii) the library instance importing it, and (iii) the complete dependency path leading from the root library to the vulnerable dependency.
- applies path simplifications and produces the results in the form of a human-readable report.

5 DATA COLLECTION

Processing of a full Maven Central repository with almost 2,7 million GAVs would be impractical and especially would include artifacts of no relevance in industrial practice. Hence, for this paper we take a sample from Maven Central, as explained below.

Library selection - incorrect way. Initially, we followed the approach of [17] and selected the number of usages of a library instance as a proxy for its popularity. By usage we understood the number of direct dependent library instances of a library instance of interest[15].

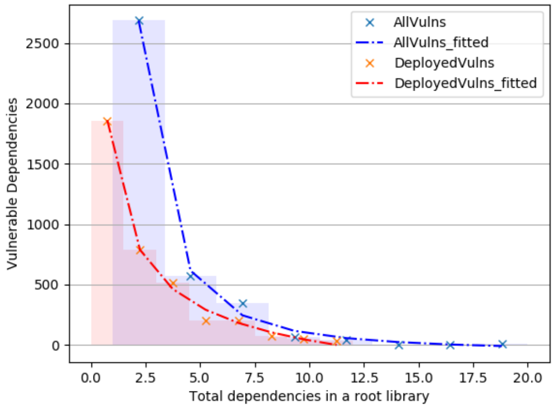
However, when we extracted the list of top 100 most used libraries, the resulted list had an unbalanced usage distribution: scala and spring-framework projects were over-represented, while some well-known projects, like Apache Tomcat, were not present in the list. A possible reason may be in the large difference in numbers of own libraries in different projects: if a project has 100 own libraries and they directly depend on a certain library instance, then this library instance would be "used" 100 times, while in reality there is only one usage.

This approach may have potentially allowed us to receive a "good" list of libraries, if as a proxy for popularity we used the number of dependent projects. However, such information is not easily available (to obtain it, we would have to build dependency trees for all GAVs in Maven Central), so we had to find another way to construct the list of libraries for our study.

Library selection - the way we followed. To ensure industrial relevance of our study, we selected the top 200 OSS libraries used by a set of more than 500 Java projects developed at SAP; these include actual SAP products and software developed by the company for internal use. Those libraries comprise, for instance, org.slf4j:slf4j-api

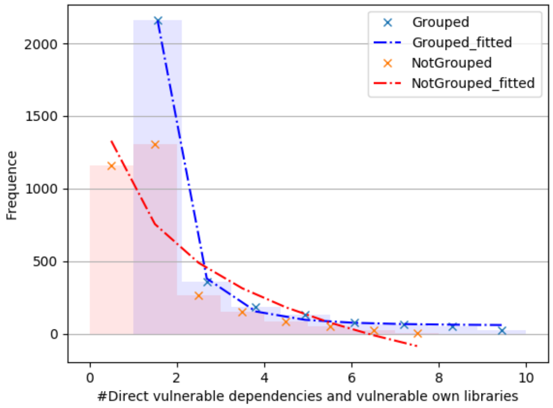
[14] https://www.owasp.org/index.php/OWASP_Dependency_Check

[15] We used the data from MVNrepository (https://mvnrepository.com/).



Significant amount of vulnerabilities are coming from non-deployed dependencies. However, these vulnerabilities are not exploitable, and therefore, do not introduce any danger to the analyzed library instances.

Figure 3: All vs Non-deployed vulnerable dependencies



The grouping step allows us to report many vulnerable cases that software developers of the analyzed libraries are actually in direct control of, since those vulnerabilities are present in either direct dependencies or own libraries of the analyzed projects.

Figure 4: The cases when developers of analyzed libraries are actually responsible for fixing vulnerable dependencies

and org.apache.httpcomponents:httpclient, and correspond to 10905 library instances when considering all versions (see Table 1 for descriptive statistics of the selected sample).

6 RESULTS AND DISCUSSION

In this Section we answer the research questions and present how each step of our approach influences the results of a dependency study for the complete sample of selected libraries. Then we show the impact of the proposed approach on the results of a dependency analysis for an average industrial software library. Below we present the results from the perspective of developers of the analyzed libraries.

RQ1: How many actually vulnerable dependencies does a library have? To answer RQ1 we collected both direct and transitive dependencies without applying other simplification steps. The left part of Table 2 shows the total amount of both vulnerable and safe dependencies in our sample. We found that non-deployed dependencies represent 45% of direct and 34% of transitive dependencies, wherein 22% of transitive and 20% of direct vulnerable dependencies are non-deployed.

Figure 3 shows the distributions of the total number of vulnerable dependencies and the number of vulnerable dependencies of an analyzed library that are actually deployed. We observe, that those distributions are different: the number of actually deployed dependencies is significantly smaller than the total number of dependencies in a library (p-value=$1.467 * 10^{-190}$, Wilcoxon test). In some cases only 22% of vulnerable dependencies are released with the library, while the majority of the analyzed libraries in our sample have up to 12 vulnerable dependencies, three of which are non-deployed (25%).

We observe that non-deployed dependencies are a significant share of vulnerable libraries: **every fifth dependency affected by a known vulnerability is non-deployed, and does not bring any danger to the analyzed library**. Hence, they should be excluded (or separately marked) from the results of a dependency study.

RQ2: Who is responsible for vulnerable dependencies? To make an application safe, its developers need to be sure that they address all the vulnerable dependencies. Direct dependencies and own libraries of a software project are within the full control of its developers but fixing vulnerabilities in transitive dependencies may require opening the "black-box" approach for dependency management, and may be significantly more expensive. To answer the RQ2 we need to determine the following:

- the difference between the "true" number of own libraries and direct dependencies in the dependency trees of analyzed libraries;
- the actual number of vulnerable dependencies that developers of analyzed libraries are responsible for fixing.

To do this, firstly, we filtered out non-deployed dependencies. Then we identified the own dependencies and compared the number of direct dependencies before and after the grouping procedure.

The effect of the grouping procedure on the aggregated numbers of dependencies for the studied library sample is shown in the right part of Table 2. We found that paths of 148 (out of 10905) analyzed library instances include own dependencies of the analyzed projects. However, those own dependencies are not affected by known vulnerabilities. This most probably reflects the quality of the selected OSS library sample: the OSS libraries used by SAP belong to well-organized software projects. However, these own dependencies may still introduce some noise while creating a bill of materials to plan allocation of development resources.

Besides own dependencies, software developers are also responsible for fixing their direct dependencies (See section 3 for detailed discussion). After the grouping step we observe the surprising increase of the number of direct dependencies by as much as 87%. This most likely happens because the dependency grouping procedure shortens the dependency paths (by grouping dependencies belonging to same projects), and therefore, it reduces the appalling feeling of an unmanageable 'dependency hell'.

Figure 4 shows the difference between the distributions of the number of vulnerable dependencies that developers of analyzed

Table 2: The effects of considering only deployed dependencies (RQ1) and grouping dependencies by software projects (RQ2)

The left part of the table shows the number of vulnerabilities within all and deployed dependencies. Non-deployed dependencies represent significant part within both vulnerable (45% of direct and 34% of transitive) and non-vulnerable (20% direct and 22% transitive) dependencies. The right part of the table shows the effect of the grouping step, which allowed us to reveal that developers of analyzed libraries could directly fix 82% of deployed vulnerable dependencies (direct vulnerable dependencies). Moreover, dependency trees of 148 analyzed library instances included own dependencies. Although they are not affected from known vulnerabilities, they may still introduce some noise, while planning allocation of development resources.

	Not vuln		Vuln	
	Direct dep.	Trans. dep.	Direct dep.	Trans. dep.
Deployed	22 464	42 519	3 078	5 282
All dep.	42 590	65 753	3 868	6 788

	Not vuln		Vuln		Not vuln		Vuln	
	Direct dep.	Trans. dep.	Direct dep.	Trans. dep.	Own lib.	3rdParty dep.	Own lib.	3rdParty dep.
Grouped	40 865	22 478	6 879	1 481	148	63 343	0	8 360
Not grouped	22 464	42 519	3 078	5 282	NA		NA	

libraries are in direct control of (own libraries and direct dependencies) before and after application of the grouping step (the difference is statistically significant, p-value=$1.648 * 10^{-285}$). The dependency grouping allows us to reveal up to eight additional vulnerable dependencies under the direct control of the developers of an analyzed library instance.

We observe that developers of the analyzed libraries *could* fix (either by directly correcting a bug in the library instances belonging to their project or by updating direct dependencies to newer versions) the major part of vulnerable dependencies, because these are under their responsibility. **Without the dependency grouping, it may seem that developers of the analyzed libraries have direct control of only 37% of the vulnerable dependencies, while in reality they are responsible for fixing 82% of the deployed vulnerable dependencies**.

RQ3: How many direct dependencies can be actually fixed? To answer RQ3 we considered only deployed dependencies, grouped according to the software projects they belong to. We found that 13% of the overall number of direct dependencies and 16% of transitive dependencies are halted. Some of them (69 direct and 5 transitive out of 9047 halted dependencies) are affected by known security vulnerabilities. Although this number is not big, each case of a halted dependency is very important. Such dependencies do not have a fixed version, and therefore, a costly mitigation is needed to fix such vulnerabilities.

Additionally, within the sample of 10905 analyzed libraries, we found five library instances that have transitive vulnerable dependencies via a halted direct dependency. All these dependencies are outdated and there exist safe versions of them. However, these safe versions would not be adopted by halted libraries, and therefore, developers of analyzed libraries have to apply a non-trivial mitigation strategy: to artificially convert those dependencies into direct dependencies of their libraries.

The proposed approach allowed us to identify that **14% of the dependencies in our sample are halted, while 1% of them are affected by known security vulnerabilities**. Moreover, **direct halted dependencies also transitively introduced 565 dependencies, 7 of which are vulnerable**. All these vulnerabilities require specific costly mitigation strategies.

Effect of the proposed approach on an individual software library. To identify a typical industrial software library, we have extracted the number of direct dependencies for each SAP software library in the proprietary Nexus repository. We assume that the number of direct OSS dependencies in a typical industrial library is equal to the mean number of direct dependencies that SAP projects have, which we found to be equal 12.

Then we have artificially constructed dependency trees for 100 software projects:

1. From the overall sample of analyzed libraries we randomly select 12 libraries
2. For each selected library in the step 1, we randomly pick its version
3. We calculate the difference between the results received according to the "standard" dependency study approach (used, for example, in [8]) and the proposed approach
4. We repeat steps 1–3 100 times to receive the data for the specified number of simulated projects

Figure 5 shows the effect of the proposed methodology for a typical industrial library. We observe that the number of deployed vulnerable dependencies is always lower, than the total number of vulnerable dependencies (Figure 5a). At the same time we see that the proposed methodology allows us to distinguish additional dependencies that developers of the simulated industrial libraries are responsible for (own and direct dependencies). Additionally, we found that an average library in our simulation have a 9,5 % chance to have a vulnerable halted dependency ($\sigma = 0.252$) with a maximum number of 2 vulnerable halted dependencies.

Hence, we can conclude that **the proposed approach has a positive impact on the correct resolution of dependency analysis results of a single industrial library**.

7 IMPLICATIONS ON INDUSTRIAL PRACTICE

In an industrial setting, the practical negative impact of using an *inadequate* measurement method can be substantial. Ensuring a healthy supply chain of third-party dependencies (of which the large majority is OSS) is a continuing effort that spans the development and the operational phases of a product lifetime.

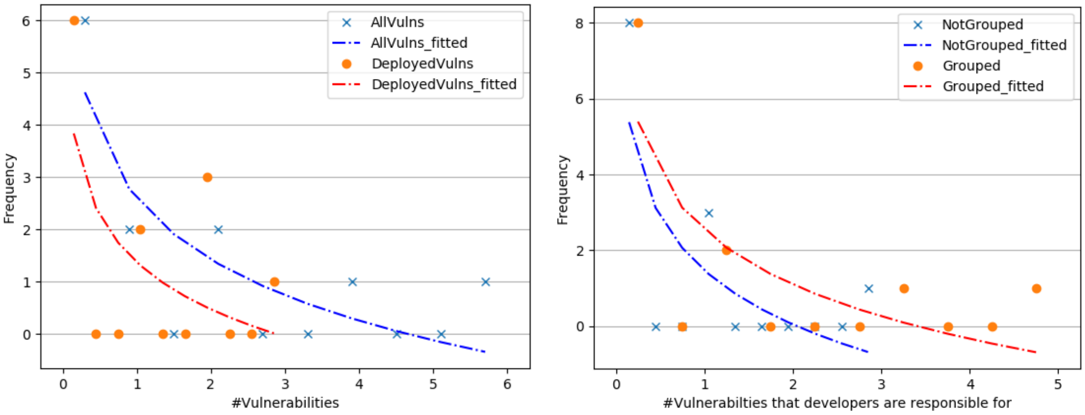
As part of SAP's secure development life-cycle, all development projects go through several validation steps and each single finding has to be audited, assessed, and mitigated. After the product is released to customers, and for its entire operational lifetime, its own security and the security of its third-party dependencies are continuously monitored. When a vulnerability is detected in one of the dependencies, timely mitigations need to be developed and deployed to all affected systems. In the case of OSS dependencies, these mitigations may consist of dependency updates, or in ad-hoc fixes in the product that relies on the affected library or in the dependency itself (through a company-internal fork that can be temporary or persistent). When the product portfolio of a company

Table 3: The effect of halted dependencies (RQ3)

14 % of dependencies of the analyzed library instances are halted, while 1 % of them are affected by known vulnerabilities. Moreover, the right part of the table shows, that direct halted dependencies introduced 7 vulnerable dependencies transitively.

	Not vuln		Vuln	
	Direct	Transitive	Direct	Transitive
Halted	5 369	3 678	69	5
All dep.	40 865	22 478	6 879	1 481

	Not vuln			Vuln		
	Halted	Outdated	Up-to-date	Halted	Outdated	Up-to-date
Transitive via halted	187	378	0	0	7	0
All dep.	9 047	54 836	1 731	74	8 286	0



(a) RQ1 for a typical industrial library **(b) RQ2 for a typical industrial library**

The proposed methodology has a positive impact on the dependency analysis results of a typical industrial library, since it allows us to distinguish deployed vulnerable dependencies from all vulnerable dependencies for a typical industrial library (Figure 5a), as well as, to report the actual number of dependencies in direct control of developers of industrial libraries (Figure 5b).

Figure 5: Effect of the proposed methodology for a typical industrial library

includes thousands of products, whose support period can extend to decades, wrong assessments lead to inadequate risk management and inefficient allocation of resources, which ultimately translate to increased chances of security incidents and financial loss.

The distinction between deployed and non-deployed components allows quick and reliable pre-filtering of not exploitable vulnerable dependencies, since they are not part of the deployed product. From our analysis of a sample of over 550 OSS libraries used by SAP projects, as many as 20% of all the dependencies are *non-deployed*. Any metrics reporting the “danger” of using OSS libraries that do not discriminate between those two classes would lead to a wrong allocation of costly development and audit resources.

The granularity at which dependencies are analyzed and the reliability with which vulnerabilities affecting them are detected are essential to obtaining a meaningful view of the (security) health of the dependencies of a project. Approaches that use imprecise vulnerability detection methods and that ignore the interdependencies among the individual nodes of the dependency tree yield a distorted view, which requires tedious, manual reviews to be correctly interpreted and that cause precious resources to be wasted. Failing to group dependency nodes that belong to the same group (e.g., to the same OSS project), and that are updated together, makes the update of certain libraries appear more problematic than it is. The vulnerability may affect a node that is deep in the dependency tree, while the node that the application developer would need to update might be much shallower (e.g., it could even be a direct dependency). More in general, imprecise approaches to vulnerability management undermine the trust of developers on automated analysis because the dependencies identified as problematic do not correspond to those that must be actually acted upon to address the reported issues. As a consequence, despite the promises of automation, considerable additional human effort and expert judgment is required to determine the appropriate mitigation strategy.

Finally, determining precisely whether a dependency could be upgraded to a non-vulnerable version or not (because such a version does not exist, and perhaps will never exist, if the dependency is no longer maintained) is the key to choosing the correct mitigation strategy. Addressing vulnerabilities in OSS components that are alive, but for which a fixed release does not exist *yet*, requires to act fast, so that an emergency solution can be rolled-out as fast as possible to all customers. Being temporary and urgent, such mitigation might not be optimal. An upgrade to a non-vulnerable version of the dependency will eventually be done. Conversely, if a vulnerability affects a dependency that is no longer maintained, fixing the code of the dependency would effectively mean creating a company-internal fork, whose long-term support could require substantial additional investments and maintenance effort.

8 RELATED WORK

8.1 Dependency Studies

Williams and Dabirsiaghi [18] report that 26% of open-source libraries downloaded by organizations from Maven Central to have known vulnerabilities and average software projects to contain at least one vulnerable dependency. The authors refer to a lack of meaningful controls of the components used in the proprietary software projects as a possible reason for such a high number of usage of vulnerable dependencies.

Hejderup [6] studied the npm registry of JavaScript modules and found that one-third of all modules use vulnerable dependencies. Besides the lack of awareness of developers, the study suggests context usage of a module and breaking changes to be the possible reasons for not fixing vulnerable dependencies. However, the authors did not distinguish deployed and non-deployed dependencies, hence the results may be reported for low-priority libraries.

The first large scale study of JavaScript open source projects was done by Lauinger et al. [9]. The authors underline the finding that transitive dependencies of a project are more likely to be vulnerable, since developers (i) may not be aware about their existence and (ii) they have less control on them. However, this relation between direct and transitive dependencies seems to be specific for the JavaScript environment, since it allows different versions of the same dependency to be included several times. Moreover, the authors say that main sources of transitive dependencies in the web sites are advertisement, tracking or social widget code, security side of which is not very well maintained. The other dependency management system may not have such problems by design. For example, Maven allows a project to use only one version of a dependency, while open-source Java projects typically do not include advertisement or social widget contents.

The authors investigate the relation between outdated dependencies and dependencies with known vulnerabilities. However, Cox et al. [4] extract dependencies from project *pom.xml* files, which means that the study report results only for direct dependencies and do not apply Maven version resolution procedure. Although the latter does not have a high impact while working with direct dependencies, it may introduce errors when transitive dependencies are involved. Moreover, the study might include low-priority findings, since it does not explicitly mention that non-deployed dependencies were filtered out. We propose to use Maven resolution procedure and consider results for both direct and transitive dependencies. Cox et al. [4] rely on name-based matching of CVEs onto library dependencies, which may have a high number of false positives [3]. Instead, we propose a precise matching approach, which relies on code-level matching.

Kula et al. [8] report 81,5% of the studied projects to have outdated dependencies, and 69% of the project owners to be unaware of vulnerable dependencies in their projects. Although the authors provide a thorough insight into developers' motivation of keeping dependencies outdated, the study uses manual analysis to map security advisories onto affected project versions, and therefore, cannot be easily applied to a large number of software projects (also, the study provides insights for only nine versions of three libraries). For the study of dependencies with known vulnerabilities Kula et al. [8] used security advisories for just five CVEs of two types - Denial of Service and "man in the middle". Hence, the results of the study might not cover important aspects of the problem of outdated dependencies. Also, as the reported developer comments reveal, the study did not filter out non-deployed dependencies and did not consider grouping dependencies by projects.

Table 4: Aspects considered in the related works

Related work	RQ1: only deployed?	RQ2: grouped?	RQ3: halted deps	Vuln. matching
Williams and Dabirsiaghi [18]	No	No	No	Name-based
Hejderup [6]	No	NA	No	Name-based
Lauinger et al. [9]	Yes	NA	No	Manual
Cox et al. [4]	No	No	No	Name-based+ manual
Kula et al. [8]	No	No	No	Manual
OWASP Dep Check	NA	NA	NA	Name-based
Cadariu et al. [3]	No	No	No	Name-based
Alqahtani et al. [1]	No	No	No	Semantic-web
Ponta et al. [15]	NA	NA	NA	Code-based

8.2 Identification of Vulnerable Dependencies

OWASP Dependency Check[16] is a tool, which provides the functionality to automatically extract a list of project dependencies and check if this list contains any libraries with known security vulnerabilities. The tool allows automatic matching of a library with an associated CVE by comparing the name of a library with a CPE version indicated in the description of a vulnerability (CVE) in NVD. Although such approach has high performance, it fully relies on the information present in the NVD.

Cadariu et al. [3] enhanced the OWASP Dependency Check tool to create a Vulnerability Alert Service (VAS) to provide the information about vulnerable dependencies used by clients of the Software Improvement Group (SIG). However, the authors discovered that the matching mechanism based on comparing library names with CPEs yields many false positives. Moreover, at the time of publication of [3] VAS was capable only to provide information regarding direct dependencies, while vulnerabilities may be also introduced via transitive dependencies [6].

Alqahtani et al. [1] used a semantic-web approach for mapping CVE descriptions from NVD database to the corresponding Maven library identifiers. However, the precision of the approach is 5% lower when compared to OWASP Dependency Check (and consequently to VAS). Hence, the results reported in [1] may provide inaccurate estimation of the number of vulnerable dependencies in the open-source projects being affected by both FP and FN.

We rely on the works from Plate et al. [14] and Ponta et al. [15], who propose a precise approach to use the patch-based mapping of vulnerabilities onto the affected components (see Section 4.3).

9 THREATS TO VALIDITY

Threats to *internal validity* concern the external factors not considered in our study:

The selection of OSS libraries is based on the number of usages from within SAP. Such selection criterion may yield a sample not representative of what libraries are most relevant for other industrial companies or OSS developers. To check the popularity of the studied libraries within the OSS community, we obtained the information about library usages from MVNRepository and the number of OSS contributors that claimed to use the selected libraries from BlackDuck Openhub[17]. The results obtained from both sources suggested us that selected libraries are popular within the OSS developers. Since SAP is a large multinational software development

[16] https://www.owasp.org/index.php/OWASP_Dependency_Check
[17] https://www.openhub.net/

company with a significant number of Java projects, we believe that the threat of industrial non-representativeness is minimal.

The vulnerability database used for our case study may not cover all known vulnerabilities. To minimize this threat SAP conducted an internal study of the vulnerability dataset, which concluded that it covers 90% of all NVD vulnerabilities reported for OSS projects developed in Java. The coverage is closer to 100% when considering the OSS projects most relevant for SAP. Hence, we believe that this threat has minimal influence on the results of our analysis.

Threats to *external validity* concern the generalization of results of a case study:

Currently we considered only Maven based projects. We used Maven, because it provides very comfortable way to handle dependency management and is wildly used within both OSS and commercial projects. Clearly, dependency analysis can be enlarged to other build automation systems, like Ant or Gradle. Although our tool depends significantly on Maven, the approach that we present in this paper is language independent and it only relies on the availability of a dependency management mechanism, such as those provided for Java (Maven, Gradle), Javascript (npm), Python (pip), PHP (pear), and so forth.

We use Maven groupIds as an approximation for a project. This may potentially lead to an incorrect grouping of libraries because some projects may use the same cross-project groupIds, or conversely, different groupIds to identify their components. The former threat has a minimal impact, since the Maven naming convention of assigning different group identifiers to distinct projects is quite well established. We observed the latter case for test or example libraries, e.g., org.apache.activemq has a subgroup org.apache.activemq.tooling. We considered two groupIds as equal if one of the two includes the other groupId (as in the activemq example). The projects that cannot be distinguished only by groupId could be distinguished using additional atributes, such as *Repository*, *ProjectID*, and others (which might be specific to certain language ecosystems).

10 CONCLUSIONS AND FUTURE WORK

In this paper we have proposed an approach for a reliable measurement of vulnerable dependencies in OSS libraries. To demonstrate our approach, we selected 200 most used OSS Maven based libraries from within SAP. However, the underlined concepts apply to any dependency management system.

To perform the analysis we have built a tool that leverages the functionality of Apache Maven to extract the library dependencies and applies code-level matching approach to identify the known vulnerabilities affecting them. We have also performed several post-processing steps, such as (i) filtering non-deployed dependencies, (ii) grouping dependencies on their belonging to software projects, and (iii) determining whether a certain dependency is halted.

The results of our study demonstrate that all the suggested post-processing steps have a positive impact:

- every fifth dependency affected by a known vulnerability is non-deployed, hence our approach allows reported results of a dependency study to be free from a significant number of vulnerable dependencies that do not introduce any harm to the analyzed libraries;
- the grouping step of the proposed approach allows us to reveal 82% (45% more comparing to a regular approach) of vulnerable dependencies the developers of the analyzed libraries are responsible for fixing;
- the results of the dependency study suggest that 14% of the total number of dependencies are halted, and therefore, do not receive updates (including security fixes). Such dependencies should be used with caution, since mitigations of their bugs and bugs of their dependencies are costly;
- the library simulation shows that the proposed approach has a positive impact on the correct resolution of dependency analysis results of a single industrial library.

As future directions of our research we take the following steps:

- to investigate the situation outside of the Maven ecosystem, for example targeting *npm* or *pip*.
- to extend this study to analyze all the existing libraries in Maven Central;
- to identify a precise model for automatic identification of whether a certain library is halted;
- to complement the existing studies on the reasons why developers do not update dependencies with an investigation of developers' behavior with regard to security-related updates.

REFERENCES

[1] S. S. Alqahtani, E. E. Eghan, and J. Rilling. Tracing known security vulnerabilities in software repositories–a semantic web enabled modeling approach. *Sci. Comp. Program.*, 121:153–175, 2016.

[2] R. G. Brown. *Statistical forecasting for inventory control.* McGraw/Hill, 1959.

[3] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *Proc. of SANER'15*, pages 516–519. IEEE, 2015.

[4] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser. Measuring dependency freshness in software systems. In *Proc. of ICSE'15*, ICSE '15, pages 109–118, Piscataway, NJ, USA, 2015. IEEE Press.

[5] S. Dashevskyi, A. D. Brucker, and F. Massacci. A screening test for disclosed vulnerabilities in foss components. *TSE*, 2018.

[6] J. Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? 2015.

[7] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. Structure and evolution of package dependency networks. In *Proc. of MSR'17*, pages 102–112. IEEE, 2017.

[8] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. Do developers update their library dependencies? *Emp. Soft. Eng. Journ.*, May 2017.

[9] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *Proc. of NDSS'17*, 2017.

[10] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *LJ*, 2014(239):2, 2014.

[11] V. H. Nguyen, S. Dashevskyi, and F. Massacci. An automatic method for assessing the versions affected by a vulnerability. *Emp. Soft. Eng. Journ.*, 21(6):2268–2297, 2016.

[12] V. H. Nguyen and F. Massacci. The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities. In *Proc. of ASIACCS'13*, pages 493–498. ACM, 2013.

[13] M. Pittenger. Open source security analysis: The state of open source security in commercial applications. Technical report, Black Duck Software, 2016.

[14] H. Plate, S. E. Ponta, and A. Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *Proc. of ICSME'15*, pages 411–420. IEEE, 2015.

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

[16] D. J. Reifer, V. R. Basili, B. W. Boehm, and B. Clark. Eight lessons learned during cots-based systems maintenance. *IEEE Softw. Journ.*, 20(5):94–96, 2003.

[17] H. Sajnani, V. Saini, J. Ossher, and C. V. Lopes. Is popularity a measure of quality? an analysis of maven components. In *Proc. of ICSME'14*, pages 231–240. IEEE, 2014.

[18] J. Williams and A. Dabirsiaghi. The unfortunate reality of insecure libraries. *Asp. Sec.*, pages 1–26, 2012.