Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies

Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci

Abstract—
Vulnerable dependencies are a known problem in today's free open-source software ecosystems because FOSS libraries are highly interconnected, and developers do not always update their dependencies. Our paper proposes Vuln4Real, the methodology for counting actually vulnerable dependencies, that addresses the over-inflation problem of academic and industrial approaches for reporting vulnerable dependencies in FOSS software, and therefore, caters to the needs of industrial practice for correct allocation of development and audit resources. To understand the industrial impact of a more precise methodology, we considered the 500 most popular FOSS Java libraries used by SAP in its own software. Our analysis included 25767 distinct library instances in Maven. We found that the proposed methodology has visible impacts on both ecosystem view and the individual library developer view of the situation of software dependencies: Vuln4Real significantly reduces the number of false alerts for deployed code (dependencies wrongly flagged as vulnerable), provides meaningful insights on the exposure to third-parties (and hence vulnerabilities) of a library, and automatically predicts when dependency maintenance starts lagging, so it may not receive updates for arising issues.

Index Terms—Vulnerable Dependency; Free Open Source Software; Mining Software Repositories

1 Introduction

The inclusion of free open-source software (FOSS) 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 FOSS [1]. For example, SAP is an active user of and contributor to FOSS[1]. Modern dependency management tools (such as *Maven*, *Ivy*, and *Gradle* for Java, or *npm, pip* for other languages) automates part of the process of managing such libraries, so the developers could focus on the interaction with the libraries they directly invoke (usually called 'direct dependencies') and treat the rest of the codebase as a black-box.

The price to pay is that the *opportunity* of using mature, high-quality FOSS components conflicts with the *need* of maintaining a secure software supply chain, and therefore, effective vulnerability analysis and management for one's software dependencies. This problem is worsened as dependency analysis methodologies are based on assumptions which are suitable for a research analysis but are not valid in an industrial context. They may not distinguish dependency scopes (e.g. [2]) which may lead to reporting vulnerabilities that are not exploitable in the field or consider only direct dependencies (e.g. [3]) although security issues may be introduced transitively [4]. Dependency analysis methodologies also miss several important factors. For example, some dependencies are maintained and released together (they may belong to the same project), and therefore, should be treated as a single unit, when constructing dependency trees and reporting results of a dependency study. Another example is the presence of dependencies whose development had been suspended for an unspecified time. Such a dependency may turn to be harmful to a dependent project in case of a vulnerability discovery as there might be no new release that fixes the issue[2].

Hence, the current approaches may present a distorted view of the situation with vulnerable dependencies:

1) *Inflation of unexploitable vulnerabilities* - a non-negligible number of development-only dependencies could not be possibly exploited;
2) *Underestimation of transitive vulnerabilities* - transitive dependencies may as well introduce vulnerabilities;
3) *Imprecise vulnerability mapping* - manual or name-based mapping is error-prone, and therefore, not reliable;
4) *Misrepresentation as somebody's else problem* - separately considered dependencies that belong to same projects reduce the visibility of the nodes that can be directly changed from an analysed library;
5) *Misreporting that nobody is in charge* - the mitigation strategy should consider the fact that maintenance of a library has significantly delayed.

In this paper we build on our case study [5] as follows

- We provide richer details on the Vuln4Real methodology for reliable measurement of vulnerable dependencies in free open-source software, by transforming observations into actionable steps that could be applied for analysis of dependencies of various dependency

- *I.Pashchenko (corresponding author) is with University of Trento, Italy.*
- *H.Plate, S.E.Ponta, and A.Sabetta are with SAP Security Research, France.*
- *F.Massacci is with University of Trento, Italy and Vrije Universiteit Amsterdam, The Netherlands.*

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

[2] For example, there is no fixed version available for the halted library org.springframework:spring-dao with CVE-2014-1904. Although the latest version of the Spring framework does not depend on the spring-dao library, the dead library is present in Maven Central and 43 other libraries still use it (as reported by mvnrepository.com).

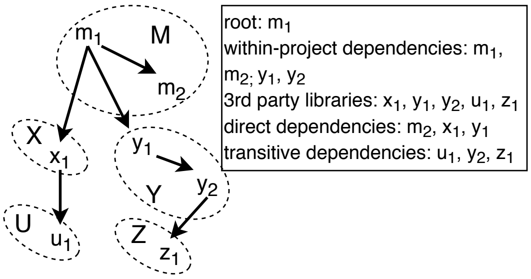

M
m_1
m_2
X
x_1
y_1
Y
y_2
U
u_1
Z
z_1
root: m_1
within-project dependencies: m_1, m_2; y_1, y_2
3rd party libraries: x_1, y_1, y_2, u_1, z_1
direct dependencies: m_2, x_1, y_1
transitive dependencies: u_1, y_2, z_1

Figure 1. Dependency tree

ecosystems and also discussing how Vuln4Real can be adapted from Java to Javascript and Python;
- we provide two different perspectives to analyze such data (the traditional) *ecosystem view* of research papers, and a *developer view* on the impact that it might have on the analysis of an individual, average library;
- we support the analysis with a tool[3] a to perform large-scale studies of (Maven-based) FOSS libraries and to determine whether any of their dependencies are affected by known vulnerabilities;
- we validate the approach by running an empirical study of 500 Java Maven-based FOSS libraries (corresponding to 25767 versions) that are most frequently used in SAP software.

We found that Vuln4Real changes the hopelessly insecure ecosystem perception, by showing that the developers of the analysed libraries can potentially fix 80% of vulnerable dependencies by updating the direct dependencies of their projects, in contrast to the state-of-the-art approaches where it seems that by manipulating direct dependencies, developers can fix only 37% of vulnerable dependencies[4] The proposed methodology also identifies and removes alerts for 27% of vulnerable direct and 21% of vulnerable transitive dependencies that could not be exploited.

These results also transfer from the ecosystem view to the developer view. This transfer is not obvious and has a major impact on industrial applicability. A 'significant' finding when analyzing an ensemble of 25K instances might not transfer to the rank-and-file developer who manages on average a dozen direct dependencies for his application.

Our simulation of the state of the art and Vuln4Real reports of the dependency analysis methodologies to understand what the individual developer would see shows that an individual developer would receive 27% less false alerts (i.e. save 4 bogus vulnerability alerts out of 9-11 that might be presented to him). Also, Vuln4Real helps planning the mitigation activities by showing which safe versions of the affected dependencies the developer can adopt directly and for which vulnerable libraries more complex mitigations should be considered.

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[5] and Apache Maven[6]:

- 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*[7] 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*[8] 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 *within-project dependencies*, while library instances maintained within other projects are *third-party dependencies*.
- A *deployed* dependency is delivered with the application or system that uses it, while a *development-only* dependency is only used 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.
- A library instance is *outdated* if there exists a more recent instance of this library at the time of analysis. A *lagging behind* library is such that the next estimated release time has been passed by far based on the interval of past releases (see Step 4 of Section 6).

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 *within-project 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.

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

- Although, from the perspective of the build system, *within-project dependency* m_2 is just a direct dependency, in practice, it is a piece of vulnerable code shipped as part of project M. Hence, the vulnerability should be fixed as part of the project development, i.e., by directly changing its source code.

[3] Similarly to releasing Eclipse Steady (https://eclipse.github.io/steady/) after publishing [6], we are planning to undergo the SAP procedure for publishing the tool behind Vuln4Real as FOSS. A reader interested in accessing the tool may contact the corresponding author.

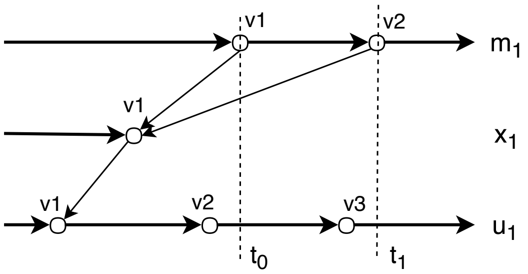
[4] A user of a library can fix all vulnerabilities by accessing and modifying the code base of dependencies, but developers tend to avoid it [7]. Equally, 'simply updating' ain't so simple [8].

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

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

[7] 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.

[8] Although dependency relations mathematically represent a graph (one dependency may have several dependent library instances), we use the term *dependency tree* to be consistent with an industrial usage: after the resolution step, dependencies of a library instance are typically presented in a form of a tree.



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. x_1 might be only slowed down development (e.g. moving from a monthly to a quarterly schedule) but from the perspective of m_1 this might be a significant risk.

Figure 2. A Dependency Lagging Behind

- Developers of M can variate the version of y_2 by selecting a suitable y_1: if a fixed version of y_1 is released, they should update project M to use it.
- Usage of dependency z_1 cannot be controlled without 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 the dependency z_1 up-to-date.

Even if library dependencies are not affected by known vulnerabilities, presence of dependencies lagging behind may lead to costly mitigations in future: if a security vulnerability is discovered in a library that is no longer actively developed, there may be no version of this library that fixes the vulnerability[9]. Hence, being a dependency, this library will introduce the vulnerability to all its dependents.

Additionally, a lagging 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 lagging dependency x_1, which, in turn, uses an "alive" dependency u_1. Although both versions $v1$ and $v2$ of library m_1 use the latest available version of direct dependency x_1, outdated transitive dependency u_1 would be also present.

3 Motivating example

Figure 3a shows the dependency migration analysis [2] of the `xalan:xalan` library. The number of appearances of each library version in the dependency trees of the analyzed libraries is reported on the ordinates for each year.

In the span of 12 years different versions of `xalan:xalan` appear in 197 dependency trees of the analyzed libraries in our dataset (See further Section 7). The versions of `xalan:xalan` prior to 2.7.2 are affected by CVE-2014-0107. The red dashed line shows the variation of the number of analyzed libraries that depend on a vulnerable version of `xalan:xalan` in time, while the green solid line represents the variation for the analyzed libraries that adopted the safe version 2.7.2.

Figure 3b shows the dependency migration plot after considering the five issues of the current state-of-the-art dependency analysis approaches (See Section 1). By removing development-only versions, and eliminating the cases where `xalan:xalan` itself was part of the analyzed project, we observe a reduction of the number of (falsely-reported) usages of the vulnerable versions (the peak on Figure 3a).

The presence of lagging dependencies has a major impact on a library maintenance strategy. Indeed, in Figure 3b, the only three libraries that depend on the vulnerable version of `xalan:xalan` even after more than two years since the release of the safe version, depend on a vulnerable version of `xalan:xalan` via direct lagging dependencies. In this case, a different mitigation strategy might be needed: (i) contribute to the lagging library, i.e., to develop its new release; or (ii) fork the lagging library and continue its maintenance as part of the dependent library.

4 Related Works

Table 1 presents the existing approaches for analyzing software dependencies.

Accounting for Deployment

Kula et al. [2] studied whether developers update dependencies of their projects. They 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, the reported quotes of software developers reveal that the paper actually included development-only dependencies in its study:

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

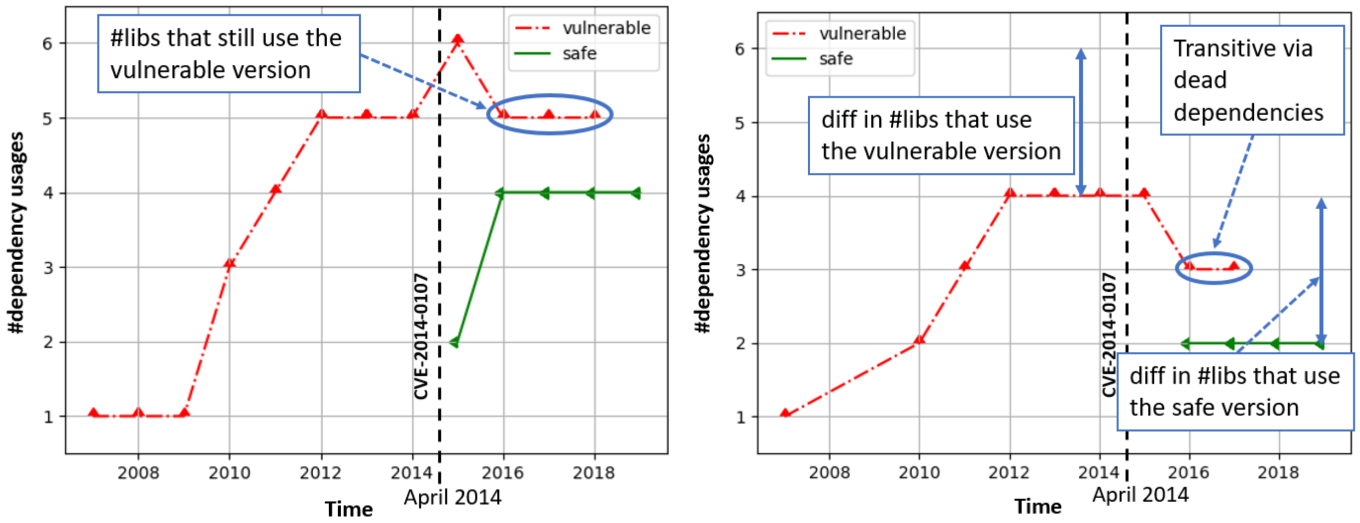
As a result vulnerable dependency count presented in [2] may be over-inflated (see Figure 3).

Several other works [3], [9], [10], [13] do not mention explicitly that they consider only deployed dependencies. Hence, their results and conclusions may be affected by low-priority non-exploitable vulnerabilities in development-only dependencies of the analysed projects.

Accounting for Transitivities

Transitive dependencies are known to be the source of vulnerabilities in software projects. For example, the first large scale study of JavaScript open source projects done by Lauinger et al. [4] underlines the finding that transitive dependencies of a project are more likely to be vulnerable, since developers (i) may not be aware of their existence and (ii) they have less control on them.

[9] There may be cases when a certain library does not receive new commits for a long time, but its developers still quickly react on arising issues. For example, although there were no releases of the Apache commons-collection library for 7 years, its developers quickly provided a fix for a vulnerability discovered in 2015 and released it within a new version. Alternatively, another organization may decide to fork an abandoned library and fix the arising security issues, as, for example, Apache Software Foundation did for the beanshel:bsh library. However, such outcomes are not guaranteed, since library developers may decide to move on and no other organization may want to support it (e.g., Apache moved from Axis to Axis2 project, but, according to mvnrepository.com, 176 libraries still depend on the vulnerable axis:axis library).



(a) Threat inflation based on state-of-the-art methods

(b) Reality

State of the art methodologies for counting the usage of vulnerable libraries over-inflate the actual risks as they count vulnerabilities that are by construction not-exploitable being part of the development and test libraries. After the release of a safe version, three libraries did not adopt the safe version of the analyzed dependency for the simple reason that maintenance and development of those libraries had halted.

Figure 3. Threat inflation vs reality for Apache's Xalan vulnerable libraries in Maven

Table 1
Distinctions considered in the related works

		Considered aspects					Consequences				
Sample language	Rel work	Only deployed	Includes transitive	Vuln mapping	Dep groups	Dead deps	Over-inflation	FN alerts	Unreliable mapping	Misleading dep picture	No is-dead analysis
JS	[9]		✓	Name-based			✗		✗	✗	✗
	[10]		✓	Name-based			✗		✗	✗	✗
	[4]	✓	✓	Manual						✗	✗
	[11]	✓						✗	✗	✗	✗
JS, Ruby, Rust	[12]		✓(no resolution)				✗		✗	✗	✗
Java	[2]			Manual			✗	✗	✗	✗	✗
	[3]			Name-based + manual			✗	✗	✗	✗	✗
	Ours	✓	✓	Code-based	✓	✓					

Several recent studies [2], [3], [11], [12] do not consider transitive dependencies. A plausible reason is that the analysis of transitive dependencies is technically complex because it requires one to follow the dependency tree construction and the resolution procedures of a specific dependency management system.

For example, Wittern et al. [11] in their study of the npm ecosystem did not follow the dependency tree construction algorithm and only considered the (direct) dependencies specified in the *package.json* files. Similarly, both Kula et al. [2] and Cox et al. [3] extracted dependencies from project configuration *pom.xml* files in their studies of the Maven ecosystem. Hence, the studies reported results only for direct dependencies and did not apply the resolution procedure of the analysed dependency management systems.

Kikas et al. [12] in their study of the dependency network structure and evolution of the JavaScript, Ruby, and Rust ecosystems considered both direct and transitive dependencies. However, the authors used dependency versions as they were specified in the project configuration files (i.e., they did not resolve versions for transitive dependencies), since the implementation of the resolution procedure of the dependency management systems required too many resources for their study.

Table 2
Approaches for Identification of Vulnerable Dependencies

Name	Approach	Advantages	Disadvantages
[14] [13]	name-based matching	High performance	Prone to FP and FN
[15]	semantic-web name matching	High performance	Prone to FP and FN (5% more than OWASP Dependency Check)
[6]	Patch-base matching	High precision	Manual effort required to create Vuln DB

Vulnerability Matching Approaches

Table 2 presents the most popular approaches to identify whether a certain library is affected by a vulnerability.

The main source of vulnerabilities in software components is the National Vulnerability Database (NVD[10]) that uses the Common Platform Enumeration (CPE) standard for enumerating the affected components. The NVD represents the most complete, public source of vulnerabilities[11] albeit

[10] https://nvd.nist.gov/

[11] Other sources of vulnerabilities are software-specific advisories and bug tracking systems which are used to report and solve security issues. Some of them might be product or vendor-specific, e.g. MSFA for Mozilla's Firefox browser.

it does not cover all FOSS projects with the same accuracy. Moreover, CPE names used to denote the affected software, use a different granularity and convention than software package repository coordinates.

False negatives easily result from the fact that the NVD is not complete and whenever the assigned CPEs are not listing all required software (e.g., in some cases the NVD assigns vulnerabilities to products rather than the responsible libraries). For example, 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, [16], [17] for the study of browser vulnerabilities and the large presence of false positives).

OWASP Dependency Check[12] is a tool that 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 an approach has high performance, it fully relies on the information present in the NVD, and therefore, may be exposed to both false positive and false negative issues.

Cadariu et al. [13] 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 acknowledged that the matching mechanism based on comparing library names with CPEs yields many false positives. Moreover, at the time of publication of [13] VAS was capable only to provide information regarding direct dependencies, while vulnerabilities may be also introduced via transitive dependencies [10].

Alqahtani et al. [15] used a semantic-web approach for mapping CVE descriptions from the 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 [15] may provide an 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. [18] and Ponta et al. [6], who propose a precise approach to use the patch-based mapping of vulnerabilities onto the affected components (see Section 6).

Accounting for within-project Dependencies

To the best of our knowledge, none of the existing dependency studies considers the fact that certain software libraries belong to the same project.

Although the concept seems intuitively simple, failure to distinguish within-project and third-party dependencies may incorrectly present as an insecure ecosystem with several vulnerable dependencies (a "dependency hell" [19]) what in reality is just a project that has broken its components into several libraries. 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.

Maintenance of Software Libraries

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 [20]. However, this becomes impossible, if a FOSS library becomes inactive [2]:

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

Zerouali et al. [21] proposed a framework to measure the technical lag (i.e., the time and/or number of versions between the last released library version and the actually used dependency) in open source repositories. In their later study, Zerouali et al. [22] claimed that the technical lag of third-party components might lead to the presence of vulnerabilities in Docker images. However, the proposed way to calculate the semantic technical lag is might be misleading as it by far overestimates the actual semantic difference between library versions[13]. Therefore, claims that an increasing technical lag leads to more vulnerabilities needs to be reviewed with a correct definition of semantic technical lag. An interesting issue that the paper [22] does not touch is automatically establishing when a particular project is lagging behind too far to warrant an action from the developers to decrease the risk of being vulnerable as captured in Figure 2.

The recent work by Coelho et al. [23] presented a machine learning based approach that uses standard metrics (number of commits, pull requests, contributors, etc.) extracted from Github to classify, whether maintenance of a particular project becomes dead. However, such features are particular to Github and may not be available for the libraries stored in other places.

Other academic approaches rely on the time of the latest commit in a certain software project. For example, Khondhu et al. [24] in their study of SourceForge projects define a project to become *dormant* if the latest commit occurred more than one year ago. The same time threshold is used by Mens et al. [25], Izquierdo et al. [26], and Coelho et al. [27]. However, the one-year threshold used by the above-mentioned studies is arbitrary, since various software projects have different development strategies, and therefore, different intervals between commits and releases. Hence, the time threshold to count project as lagging behind should vary depending on a project development strategy. In this respect, we rely on an individual project history to predict whether its maintenance is likely to have lagged or even halted.

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

[13] The authors count the total number of minor versions and patches without resetting them after each major version. For instance, a library updating from 2.0.0 to 3.0.0 is just an update of 1 major version from the perspective of a developer. If there were intermediate versions 2.1.0 and 2.2.0, the proposed measure would instead give a distance of one major version and two minor versions and all possible patches in between.

5 Research Questions and Validation of the Methodology

A dependency measurement methodology may impact the analysis of vulnerable dependencies for a sample of selected libraries in several different ways. To understand such impact, we structure our analysis according to the principles of Empirical Based Software Engineering (EBSE) [28, Chapter 8]: we propose a number of research questions and related hypotheses, which will be then tested and analyzed.

RQ1: Does Vuln4Real significantly reduce the number of false alerts for deployed code (dependencies wrongly flagged as vulnerable)?

> ***Hypothesis 1 (H1).*** Considering only deployed dependencies reduces the number of security alerts in deployed dependencies.

RQ2: Does Vuln4Real provide insights on the exposure to third-parties functionalities (and hence vulnerabilities) of a library?

> ***Hypothesis 2 (H2a).*** Grouping software dependencies by projects reveals a significant number of dependencies whose versions could be directly controlled by the developers of the libraries under analysis.
>
> ***Hypothesis 2 (H2b).*** Grouping software dependencies by projects reveals a lower level of exposure to third parties vulnerabilities.

RQ3: Can Vuln4Real predict when a library's development is lagging behind?

> ***Hypothesis 3 (H3).*** The history of library releases can be used for predicting if a library maintenance lags.

RQ4: Can the number of dependencies be used as a predictor for a number of vulnerabilities in a library?

> ***Hypothesis 4 (H4).*** The number of dependencies can be used to predict the number of vulnerabilities in a library.

Before going to the actual evaluation we observe that for each research question there are *two* distinct and equally important viewpoints:

1) The *ecosystem view* assesses the hypotheses by considering the ensemble of libraries as a whole. It is interesting from research and systemic perspective (i.e. the perspective of a CTO using our 500 libraries). If our hypotheses were true we would expect variations in the population parameters such as mean and median or, at least, in the tails of the distribution.
2) The *developer view* simulates what would change for the average developer who is using the methodology and whether she would see visible changes in the daily activity for the average number of dependency alerts. It is important from an industrial perspective.

Given a large number of libraries (more than 25K GAVs) a measurable change in the overall distribution might still be into an invisible change for the individual developer working with a dozen of libraries.

To understand the difference, suppose we find that the average number of dependencies falsely reported as vulnerable is bounded by a 95% confidence interval at $[0.01, 0.02]$. When multiplied by a large number, this is going to be a very large effect. Our hypothesis of a significant effect is definitely satisfied. Yet, to the individual developer, it might remain immaterial as she cannot 'experience' it: a concrete dependency is always a unit and cannot be 0.2 of the unit. For the dozen of dependencies of her library she might not see any visible change as, at worst, she will have 0.24 false reports i.e., mostly none. If such falsely reported vulnerabilities are clustered at the tail among infrequently unused libraries she might never face the difference. As we shall see later this is not the case for us (Table8. The difference is materially visible.

Hence, it is important to check whether a methodology has an impact on *both* the ecosystem view and the individual developer's view. From the perspective of EBSE this means that we would have to check our hypothesis twice: one for each viewpoint.

To answer the research questions for the ecosystem view we have collected both direct and transitive dependencies of the library instances. First, we treated them according to a SoA approach affected by all the dependency presentation issues (See Section 1), which corresponds to approaches presented in, for example, [2] or [3]. Then we applied Vuln4Real and compared the number of vulnerable/ non-vulnerable dependencies calculated according to both approaches.

To identify a typical industrial library, we have extracted the number of direct dependencies for each SAP software project in the proprietary repository. We assume that the number of direct FOSS 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 to 11.

Then we have artificially constructed dependency trees for 100 software projects according to Algorithm 1. This approach could be adapted to any dependency methodology by replacing Vuln4Real with one's own.

6 Vuln4Real Methodology

Table 3 overviews the Vuln4Real methodology for counting vulnerable dependencies.

Step 1: Extraction of a dependency tree for a library

The extraction of a dependency tree for a library includes two steps:

- full dependency tree construction that contains all the dependencies as they are specified in the configuration files of the dependency tree nodes;
- resolution of conflicts between dependency versions when the full dependency tree contains several different instances of the same library.

In many cases a dependency management system provides the functionality to extract the dependency tree for a specific library instance and to resolve the conflicts. For example, to have a dependency tree of a Maven based library instance, one may execute the *dependency:tree* goal of the Apache Maven Dependency Plug-in[14] and the *dependency:resolve* goal to have the version conflicts resolved. The

[14] https://maven.apache.org/plugins/maven-dependency-plugin/index.html

Algorithm 1: Evaluation of a Developer's View

```
input : Sample of analysed libraries $AnalysedLibs$, sets of
        deployed dependencies, grouped dependencies, and
        lagging dependencies
output: Impact on the dependency analysis of a 'sample'
        library
$Vuln\_Paths \leftarrow \emptyset$ // Output according to the
    standard approach
$Vuln\_Paths\_filtered \leftarrow \emptyset$ // Output according to
    Vuln4Real
$Lagging\_deps \leftarrow \emptyset$ ;
$i = 0$ ;
while $i < 100$ do
    // Random selection of 12 libraries
    $l = 0$;
    $Libs \leftarrow \emptyset$ ;
    while $l < 12$ do
        $lib \leftarrow \{Random(lib) | lib \in AnalysedLibs\}$
            // random selection of a library
        $lib\_version \leftarrow \{Random(version) | version \in lib\}$
            // random selection of a library
            version
        $Libs \leftarrow Libs \cup lib\_version$ ;
        $l = l + 1$;
    end
    // Calculation of the results according to
        the standard approach
    $Vuln\_Paths \leftarrow VulnPaths(Libs)$ ;
    // Calculation of the results according to
        the proposed methodology
    $Vuln\_Paths\_filtered \leftarrow DeployedOnly(Vuln\_Paths)$
        // Leave only deployed deps
    $Vuln\_Paths\_filtered \leftarrow Group(Vuln\_Paths\_filtered)$
        // Group coupled deps
    $Lagging\_deps \leftarrow Lagging(Vuln\_Paths\_filtered)$
        // Get lagging deps
    $i = i + 1$ ;
end
```

JavaScript packet manager *npm* provides the *npm ls* command to display the dependency tree of a specified package[15]. Alternatively, there exists the *dependency-tree* plug-in[16] that also handles version resolution conflicts. Although the Python package manager *pip* does not provide a default functionality to display the dependency trees, tools like *pipdeptree*[17] or *pipenv*[18] support this. Those tools do not provide the functionality for resolving version conflicts, however the current resolution procedure is simple - *pip* performs the breadth-first traversal of the dependency tree and picks the first instance of a library it encounters[19].

Step 2: Identification of development-only dependencies

We identify development-only dependencies as follows:

- we rely on the dependency management system (or project configuration files) to provide us with additional information about the dependency type[20];
- we use this information to classify dependencies in the dependency tree.

For example, in Maven we extract the dependency *scope*: the dependencies with scope *test* are used only for development purposes. In npm development-only dependencies are collected within the *devDependencies* section of the configuration file, while in pip such dependencies are specified as *extraRequirements*.

Step 3: Identification of within-project dependencies

To identify dependencies that are maintained and released simultaneously, we perform the following procedure:

- we refer to the development practices adopted by the developers within the corresponding dependency management systems;
- we use these practices to identify a project that includes the analysed dependency and other within-project libraries of this project.

Maven libraries are grouped into multi-module projects where each module is released as a separate artifact. According to the Maven naming conventions[21], within-project dependencies of a multi-module project have the same *groupId*. Hence, within-project dependencies can be easily identified by comparing their *groupId*s. JavaScript and Python developers may follow the monorepo development strategy, when several software libraries are stored in the same repository[22]. Such library groups do not share a common identifier, however, they still can be distinguished by analysing monorepos separately. Although in these cases the step of identification of within-project dependencies would require additional efforts, it allows library developers to receive the meaningful (and correct) presentation of the dependency analysis results.

Step 4: Identification of lagging dependencies

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 a 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 [33]:

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

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

where Release time$_i$ is the time interval 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

[15] https://docs.npmjs.com/cli/ls.html
[16] https://www.npmjs.com/package/dependency-tree
[17] https://pypi.org/project/pipdeptree/
[18] https://pypi.org/project/pipenv/
[19] https://github.com/pypa/pip/issues/988
[20] Such information is always available, albeit in possibly different formats.
[21] https://maven.apache.org/guides/mini/guide-naming-conventions.html
[22] The monorepo development strategy is widely adopted by large software development companies, such as Google [30], Facebook [31], and Microsoft [32]

Table 3
Vuln4Real methodology overview

Step 1: Extraction of a dependency tree for a library	
INPUT	Source code of an analysed library
OUTPUT	Resolved dependency tree for an analysed library
PROCEDURE	Identify dependencies of an analysed library and represent them in a form of a dependency tree: • Employ the mechanism of a dependency management system to construct dependency tree of a library • Apply the dependency management system resolution procedure to resolve version conflicts • Extract the resolved dependency tree
Step 2: Identification of development-only dependencies	
INPUT	Resolved dependency tree for an analysed library
OUTPUT	The set of development-only dependencies
PROCEDURE	Identify dependencies that are used only during development of the library, i.e., are not shipped with this library: • Extract dependency scopes • Mark dependencies in scopes that are not shipped with the analysed library as test. For example, in Maven dependencies with scope *test* are not shipped with the library, npm has a set of *devDependencies* that are used only for development purposes, and in pip such dependencies are specified as *extra* requirements.
Step 3: Identification of within-project dependencies	
INPUT	Resolved dependency tree for an analysed library
OUTPUT	The set of groups of within-project dependencies
PROCEDURE	Identify within-project dependencies: • Identify dependencies that are maintained and released simultaneously. In Maven the libraries that have a common *groupid* are parts of a single multi-module project, while in npm and pip dependencies are joined into monorepos.
Step 4: Identification of lagging dependencies	
INPUT	Resolved dependency tree for an analysed library
OUTPUT	The set of lagging dependencies
PROCEDURE	Identify dependencies of the analysed library that are no longer maintained: • Refer to the dependency repository to extract the release times for all dependency instances • Use release times to estimate the expected time of the next release • In case the time of observation does not exceed the estimated time, count such dependency as alive, otherwise count it as lagging
Step 5: Identification of dependencies with known vulnerabilities	
INPUT	Resolved dependency tree for an analysed library
OUTPUT	The set of dependencies with known vulnerabilities
PROCEDURE	Employ code-base matching procedure to check whether a dependency is affected by a known security vulnerability: • If a patch to fix the vulnerability in the dependency exists use the code-base approach by Plate et al. [18] and [29] to compare the nodes of the dependency tree of the version of interest to check whether one of them is affected • If no such fix exists report the vulnerability according to database approach (i.e., listed version in the vulnerability dataset).
Step 6: Path extraction	
INPUT	The dependency tree of an analysed library, the sets of development-only dependencies, groups of within-project dependencies, lagging dependencies, and dependencies with known vulnerabilities
OUTPUT	Dependency analysis report
PROCEDURE	We use the following algorithm to construct paths from vulnerable nodes to the analysed libraries: • Remove development-only dependencies (Step 2) and their subtrees from the dependency tree • Use the output from Step 5 to identify nodes affected by known vulnerabilities • For each node in the dependency tree from Step 1, extract the shortest path between the vulnerable dependency and the analysed library • Substitute a group of consecutive within-project dependencies in the path with the closest to the vulnerable node dependency from the group. • Use the output of Step 4 to identify lagging dependencies.

time intervals decreases[23]. 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. Considering, the error threshold, we count the *Estimated next release date* to be within the following interval for a library to be *lagging*

$$\text{Now} > \quad \text{Last release} + 2 \cdot \text{Expected Release Interval}$$

The proposed model based on release dates is conservative since it provides a 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 lagging when it is still under development. However, such finding would mean that a library does not receive a fix for a long time, during which a zero-day vulnerability remains exploitable. Hence, even in case of “false positives”, our model provides valuable information for developers.

[23] The observation of released dates for the analyzed libraries suggests 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.

Step 5: Identification of dependencies with known vulnerabilities

To avoid the false positive and false negative inflation introduced by name-based vulnerability matching ([13], [14], [15]), we leverage on precise code-based approaches to vulnerability detection such as Ponta et al. [6] and Dashevskyi et al. [29]. Starting from known vulnerabilities disclosed in the NVD, advisories, bug tracking systems, etc., we manually identified and analyzed the commit fixing the vulnerability. This activity results in a list of code changes. All software constructs (e.g., constructors, methods) included in such a 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 within-project libraries of the project and all its dependencies is collected by performing a code-level matching of the vulnerable fragment following the approach of [6]. Whenever the vulnerable code fragment is contained within a dependency, the corresponding vulnerability is automatically reported for our analysis. If a fixed version is not yet available, we have to resort to the traditional approach of dataset-based identification, i.e. the versions mentioned as vulnerable in a dataset (e.g., the NVD) are marked as vulnerable. This case is rare as the process of responsible disclosure [34] becomes increasingly adopted.

Step 6: Path construction

We use the resulting dependency tree and the outputs of the steps 2-5 of the proposed methodology to identify whether the dependencies belong to one of the following groups:

- development-only dependencies;
- within-project dependencies;
- lagging dependencies;
- dependencies with known vulnerabilities.

Vulnerable dependencies represent the most valuable assets, 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 within-project dependencies within one path and substitute them in the path with the library instance, closest to the vulnerable dependency.

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).

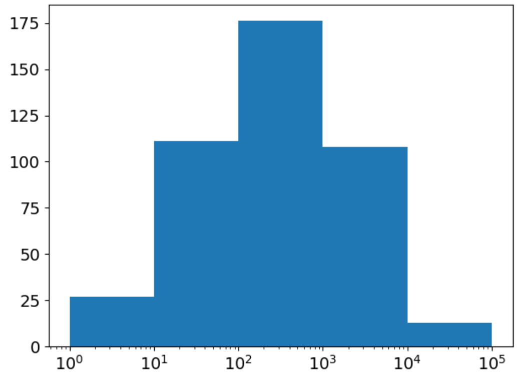
7 Data Collection

Considering the popularity and industrial relevance of Java[24], in the following we demonstrate the proposed methodology on Java projects.

Table 4
Descriptive statistics of the library sample

We considered the 500 most popular FOSS Java libraries used by SAP in its own software, which resulted in 25767 distinct GAVs when considering all library versions.

	μ	σ	min	max	Q25%	Q50%	Q75%
#GAVs	53.8	105.0	1.0	846.0	11.3	26.0	67.0
#deps	14.5	12.9	1.0	128.0	5.0	12.0	19.0
#direct	4.2	4.8	0.0	39.0	1.0	2.0	6.0
#trans	10.3	10.2	0.0	103.0	2.0	7.0	17.0
#vuln deps	1.3	1.9	0.0	24.0	0.0	1.0	2.0
#direct	0.4	0.9	0.0	10.0	0.0	0.0	0.0
#trans	0.9	1.5	0.0	15.0	0.0	0.0	1.0
rel intervals (days)	43.6	106.2	0.0	3204	1.8	14.2	43.0



To understand how the selected sample is also used by the broader community, this graph reports the #usages as reported in mvnrepository.com, it shows a log-normal distribution (X-axis has logarithmic scale).

Figure 4. Distribution of library usages in the sample within Maven ecosystem

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 and Gradle (which is gaining popularity)[25], however Maven still has the largest share of users[26]. Hence, we use it to demonstrate the proposed mitigations for each problem described in Section 3.

In Maven the name of a component is standardized[27] 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)

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.

7.1 Library selection - incorrect way

Initially, we followed the approach of [35] and selected the number of usages of a library instance as a proxy for its popularity. By usage we understood the number of direct

[24] 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.

[25] https://gradle.org/

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

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

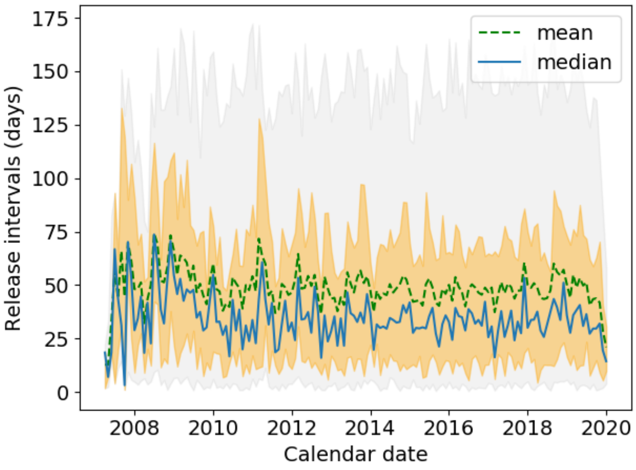


Figure 5 shows the per-month distribution of times till the next release for the cleaned release time dataset. The lines represent median and mean time of the next release, the orange area shows the Q25% – Q75% release interval range, and the background grey area shows the 95% confidence interval.

Figure 5. Release intervals of libraries in our sample

dependent library instances of a library instance of interest[28]. We should note that the number of library instance usages does not add up to the total number of usages of its library, i.e. $\sum_v Usage(GA_v) \neq Usage(GA)$.

When we extracted the list of top 100 most used libraries, the resulting 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 within-project libraries in different projects: if a project has 100 within-project 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 library instances in Maven Central), so we had to find another way to construct the list of libraries for our study.

7.2 Library selection - the way we followed

To identify the relevant 'main libraries', one needs an outside anchor. Indeed, just using the number of usage of within Maven itself appeared to be severely biased as it makes service libraries to be disproportionately selected. However, this does not correspond to the popularity of the software in the world (which makes the study interesting).

To ensure industrial relevance of our study, we selected the top 500 FOSS 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 and org.apache.httpcomponents:httpclient, and correspond to 25767 library instances when considering all versions (see Table 4 for descriptive statistics of the selected sample). We have also extracted the number of usages of the libraries in our sample as reported in mvnrepository.com. Figure 4 shows that the libraries in our sample are popular[29] within FOSS projects: 121 libraries are used by more than 1000 other libraries (and even more library versions), while median library has 291 dependents.

For the collected sample of libraries and their dependencies, which resulted in 906 distinct libraries, we have collected 54475 release intervals.

To validate the model for identification of dead dependencies, we have extracted release intervals for all distinct library instances and dependencies in our sample. As developers of several libraries support several versions of the same library simultaneously (e.g., developers of org.springframework.boot:spring-boot-starter-web supported 1.5.x, 2.0.x, 2.1.x, and 2.2.x versions in parallel), to remove the possible bias introduced by simultaneous releases of different versions, we have considered only the releases of the library with increasing version labels[30]. For example, if a library has the following versions order sorted according to their release dates: 2.1.1, 1.5.19, 2.1.2, 1.5.20 - we do not include releases 1.5.19 and 1.5.20 in our analysis.

We noticed that the dataset of release intervals has outliers: mean release interval is 43.6 days, while the longest release interval is 3204 days. To decrease the influence of the outliers, for our further analysis we have considered only the libraries that have release intervals not exceeding 365 days (are within 95% confidence interval of the original release interval dataset). This resulted in 35256 release intervals of 632 libraries. Figure 5 and Table 4 describe release intervals of libraries in our sample.

To automate our dependency study we implemented a tool that:

- 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 library instance.
- uses the code-based approach of [6] to annotate dependency trees with the vulnerability data at our disposal. In particular, when a vulnerable library instance is found among the dependencies of an analyzed root library, 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.

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

[29] We have manually checked 27 libraries that have less than 10 usages and we found that they might be industry-specific libraries: e.g., they may have groupIds like com.sap.*. However, their share in the selected library sample is quite small (≈ 5%).

[30] If library version labels do not allow us to determine increasing order of library versions (e.g., non-numeric labels are used to mark library releases), we assumed that the library developers supported only one version of the library at a time.

8 Evaluation: Ecosystem View

RQ1: Does Vuln4Real reduce the number of false alerts in the deployed code (dependencies wrongly flagged as vulnerable)?

Vulnerabilities in development-only dependencies cannot be exploited once the library is deployed because the vulnerable code is simply not there. The library might still be vulnerable for other - so far unknown - reasons, but from the perspective of any developer this would be a false alert.

Figure 6a visually compares the per library instance distributions of the total number of dependencies and the number of deployed[31] dependencies ($p\text{-}value \approx 0$, Wilcoxon test), while Figure 6b shows the difference between the per library instance distributions of all and deployed dependencies affected by known security vulnerabilities ($p\text{-}value \approx 0$, Wilcoxon test).

These observations confirm H1 and allow us to conclude that the proposed methodology reduces the number of low-priority security alerts.

Discussion: We observe that development-only vulnerable dependencies are widely used within the analysed libraries; for some library instances there are only development-only dependencies. Hence, following the SoA approach, developers would have to face a big number of alerts for dependencies not included into the deployed version of their libraries (for some library instances their amount exceeds the alerts in deployed dependencies by up to 3 times). Their analysis may require a significant amount of expensive developers' time and, consequently, decrease the value and trust in the dependency analysis findings. Instead, Vuln4Real allows developers to receive trustworthy dependency analysis reports.

RQ2: Does Vuln4Real provide insights on the level of exposure to third-parties vulnerabilities of a given library?

To make an application safe, its developers need to be sure that they address all the vulnerable dependencies. SoA approach suggests that direct dependencies of a software project are within the full control of its developers. However, such an approach misses the fact that within-project dependencies should also be considered to correctly report the number of controlled dependencies in the analysed projects.

The dependency grouping procedure shortens the dependency paths (by grouping dependencies belonging to same projects), so some direct vulnerable dependencies appear to be within-project dependencies of the root libraries, while the versions of some vulnerable transitive dependencies appear to be in direct control from the root libraries.

Vulnerable within-project dependencies. Since the analysed library instances may as well be parts of multi-module projects while reporting the results it is also important to correctly distinguish between the "true" number of third-party and within-project dependencies of the analysed libraries as the latter should be fixed by the developers of those libraries by directly changing their code.

Figure 7a shows that within-project dependencies of root libraries are often present in dependencies despite the popularity of the analysed libraries. Several library instances only have within-project dependencies. Figure 7b shows the percentage of within-project dependencies of the analysed libraries affected by known security vulnerabilities. These vulnerabilities affect components of the analysed libraries, and therefore, represent the parts of the multimodule project the analysed library is included in.

Vulnerable direct dependencies. Figure 8a shows the difference in the number of direct dependencies per library before and after grouping libraries by software projects. We observe that Vuln4Real allows us to reveal up to 60 additional dependencies, whose versions could be directly controlled by developers of the analysed libraries. Figure 8b shows the difference in the number of direct and the number of revealed vulnerable dependencies which versions could be directly updated by the developers of the analysed libraries. We observe, that the proposed methodology suggests that developers of the analysed libraries could have directly adopted fixed versions of up to 10 dependencies affected by known vulnerabilities. The negative values on both Figures correspond to the within-project dependencies of the analysed libraries.

Grouping dependencies by their projects allows Vuln4Real to reveal (i) vulnerable dependencies whose code should be directly changed by the developers of the analysed libraries as these dependencies are parts of their project (within-project dependencies), (ii) vulnerable dependencies, safe versions of which could be directly adopted by the developers of the analysed projects. Hence, H2a is confirmed.

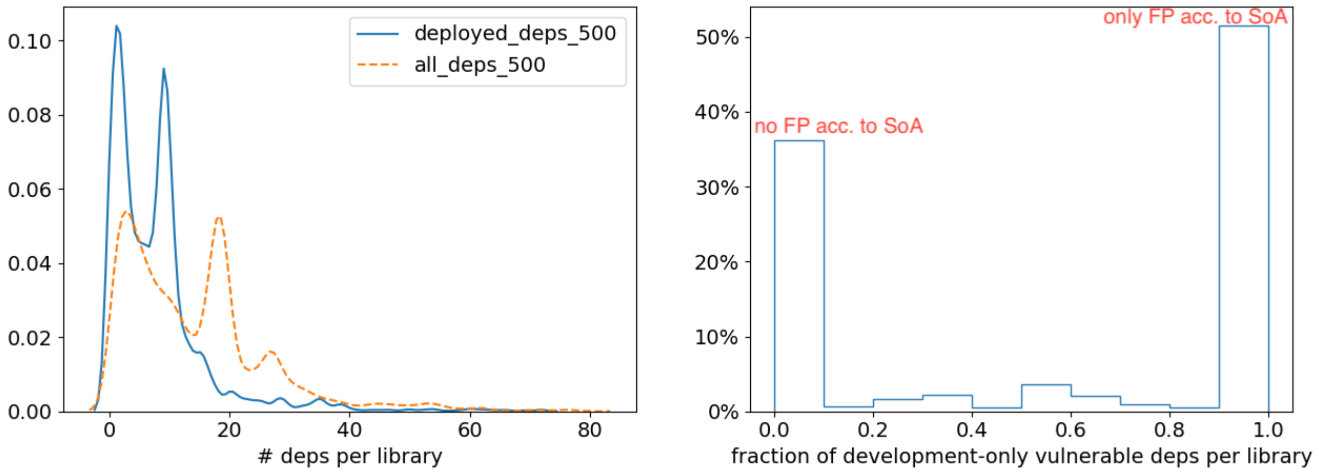
Discussion: We observe that many direct vulnerable dependencies were presented as transitive for the developers of the analysed libraries. This may influence developers to select a wrong mitigation strategy, i.e., to wait for the dependencies to adopt the fixed versions of vulnerable dependencies, instead of fixing them directly from the analysed projects. Hence, the SoA way of presenting the vulnerable dependencies that can be fixed by updating direct dependencies of the analysed libraries hides several dependencies, safe versions of which could be directly adopted from the root libraries (i.e., false negatives from the perspective of a software developer). For several libraries, the amount of such FN alerts is equal to the number of TPs. Hence, Vuln4Real allows us to reveal direct vulnerable dependencies, which were falsely hidden by the SoA approach.

Figure 9 allows us to visually compare the amount of dependencies reported by SoA and Vuln4Real. Each category of dependencies is presented as a rectangular area, where the center has the mean numbers of vulnerable and not vulnerable dependencies as coordinates and the height and width are the respective 95% confidence intervals.

Results presented according to the SoA approach suggest that there are more transitive dependencies and they introduce more vulnerabilities ($Trans_{SoA}(\mu_{vuln}) = 0.51$), rather than direct dependencies ($Direct_{SoA}(\mu_{vuln}) = 0.37$) per library instance (p-value $\ll 0.05$, Wilcoxon test).

In contrast, our methodology dramatically changes this picture. Filtering out deployed dependencies decreased

[31] To identify deployed dependencies in Maven we have excluded dependencies in *test* and *provided* scopes, since dependencies in both scopes do not appear as transitive dependencies in the dependency trees of the dependent libraries.

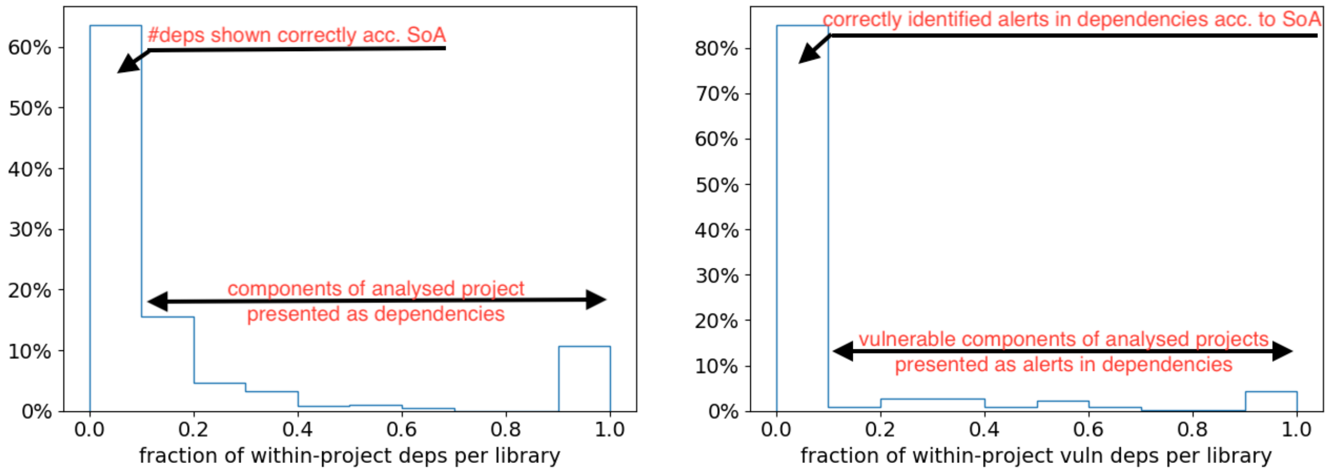


(a) #deployed vs #all dependencies

(b) Percentage of vulnerabilities reported only by development-only dependencies

To plot figures 6a and 6b, we have removed 84 library instances of 8 libraries that have more than 80 dependencies (outliers). The figures show a visible difference between the distributions when all dependencies are considered and development-only dependencies are removed. The vulnerabilities affecting latter ones have low priorities for software developers of the dependent libraries.

Figure 6. RQ1: per library instance comparison of all, deployed, and development-only dependencies.



(a) Percentage of root library components

(b) Percentage of vulnerable components of root libraries

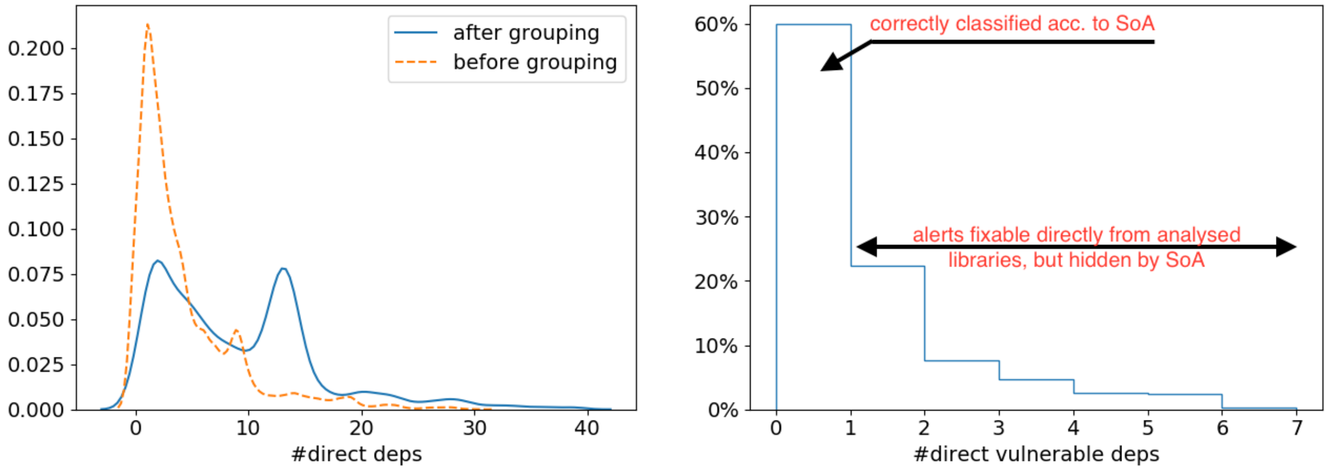
Figure 7. RQ2: Distribution of components of root libraries presented as its dependencies (within-project dependencies)

the mean number of non vulnerable dependencies from $All_{SoA}(\mu_{\neg vuln}) = 13.18$ to $All_{deployed}(\mu_{\neg vuln} = 9.34$ and the mean number of vulnerable dependencies withing the analysed library sample from $All_{SoA}(\mu_{vuln}) = 1.30$ to $All_{deployed}(\mu_{vuln}) = 0.94$. Furthermore, after the grouping procedure, the mean number of both vulnerable ($Direct_{within-project\&3rdPty}(\mu_{vuln} = 0.65)$ and non vulnerable ($Direct_{within-project\&3rdPty}(\mu_{\neg vuln} = 6.01)$ direct dependencies has become bigger than the mean number of vulnerable ($Transitive_{within-project\&3rdPty}(\mu_{vuln} = 0.10)$ and non vulnerable ($Transitive_{within-project\&3rdPty}(\mu_{\neg vuln} = 1.58)$ transitive dependencies.

While SoA presents that there are more transitive dependencies and they are the main source of vulnerabilities, Vuln4Real shows that most vulnerabilities affect within-project and 3rdPty dependencies, safe versions of which could be directly adopted from the root libraries (H2b is confirmed).

Discussion: The prevalence of transitive dependencies and security issues affected them might create a feeling of absence of responsibility for fixing security issues in software dependencies. I.e., if a problem is too large it cannot fit into the next release interval of a library sprint. Hence, it might be postponed until the moment when the issues make development impractical and library dies. In this respect, showing that most of the issues could (and should) be fixed from the root libraries might be an important starting point for developers to consider vulnerabilities in their depen-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3025443, IEEE
The final version of Transactions on Software Engineering http://dx.doi.org/10.1109/TSE.2020.3025443



(a) #direct dependencies per library

(b) Difference in #vuln direct dependencies per library

Figure 8. RQ2: differences of the number of direct dependencies per library before and after grouping

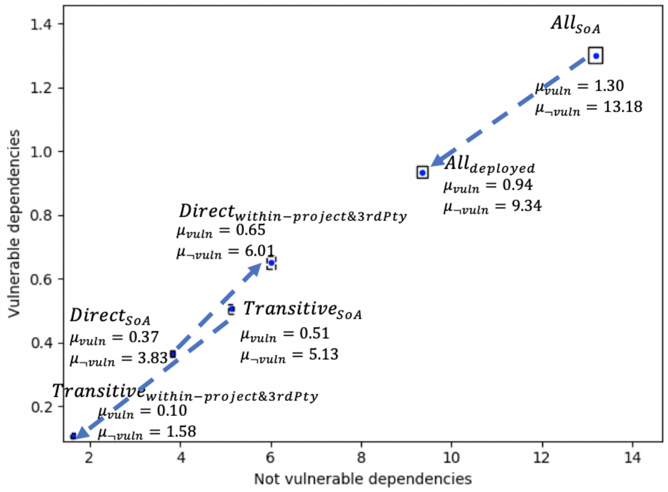


Figure 9. Number of direct and transitive dependencies calculated according to SoA and Vuln4Real

dencies, so the task of adopting safe versions of vulnerable dependencies becomes feasible and developers start fixing dependency issues regularly.

RQ3: Can Vuln4Real predict when a library lags behind?

To understand which features have the biggest impact on the release time of a library, we introduce two regression models (Table 5):

- *Int* assumes, as Vuln4Real, that the release time of a library can be essentially captured by observing the past release intervals,
- *All* combines the *Int* model with the intuition that the increasing complexity of library dependencies might lead to a longer release interval.

To check the effectiveness of the models and compare their performance with the Vuln4Real model based on Simple Exponential Smoothing, we have run them against the release dates dataset. We have considered decision threshold to be twice the predicted release time as specified in Section 6.

Table 5
Regressions

(a) Int: new release $\sim$ release interval$_{t-1}$ + release interval$_{t-2}$ + release interval$_{t-3}$ ($R^2 = 0.345$, p-values $\ll 0.05$ for all explanatory variables)

	estimate	std. error
intercept	8.0272	0.307
$interval_{t-1}$	0.3264	0.007
$interval_{t-2}$	0.2071	0.007
$interval_{t-3}$	0.1776	0.007

(b) All: new release $\sim$ release interval$_{t-1}$ + release interval$_{t-2}$ + release interval$_{t-3}$ + within-project + direct + trans + within-project vuln + direct vuln + trans vuln ($R^2 = 0.354$, p-values $\ll 0.05$ for all explanatory variables)

	estimate	std. error
intercept	11.4463	0.468
$interval_{t-1}$	0.3097	0.007
$interval_{t-2}$	0.1963	0.007
$interval_{t-3}$	0.1655	0.007
#within-project deps	0.0960	0.185
#direct deps	-0.5681	0.052
#transitive deps	0.4198	0.057
#within-project vuln deps	-0.2606	0.754
#direct vuln deps	2.7882	0.251
#trans vuln deps	-2.4354	0.387

Table 6
Comparison of models for prediction of whether a library is lagging

	Vuln4Real	Int	All
next releases within 2pred	78%	86%	70%
next releases later 2pred	22%	14%	30%

The coefficients of *Int* model shows that more recent release intervals have higher impact in comparison to older release intervals. This corresponds to our intuition and the observations presented in [5].

Table 6 compares the performance of Vuln4Real and the regression models. The simple exponential decay of Vuln4Real correctly predicted the next release date for 56% of release intervals within the lengthened period. Additionally, 22% of library instances had next releases earlier than the expected release date. Hence, the Vuln4Real model has correctly classified libraries as *alive* in 78% of cases.

The proposed model based on a library release history

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3025443, IEEE Transactions on Software Engineering
The final version of record is available at http://dx.doi.org/10.1109/TSE.2020.3025443

could be used for screening test to check that the developed of a library is slower than expected and therefore could lag behind (H3 is partly confirmed).

Discussion: The *All* model coefficients generally support the intuition behind the model: the increasing number of within-project and transitive dependencies corresponds to increasing release intervals, while the number of vulnerabilities in transitive dependencies indicates its decrease. However, the positive coefficients of within-project and direct vulnerable dependencies seem counter-intuitive. This might be caused by the necessity of developers of an analysed library to assess the vulnerability whether it affects their project, check the suggested fix on breaking changes, and thoroughly test the library.

Although, the regression models have a slightly higher correct prediction to test whether a library is *alive*, they are harder to apply in practice as they require a very large sample of release intervals to fit them. In contrast, the proposed model has a sufficiently high prediction rate and requires only a handful of data points for prediction (and no regression on 25K GAVs).

RQ4: Can the number of dependencies be used as a predictor for a number of vulnerabilities in a library?

Although several studies name transitive dependencies as one of the main vulnerability sources [4], [10], Vuln4Real changes the distributions of vulnerabilities between direct and transitive dependencies. Hence, we are interested in studying the influence of software dependencies on the number of vulnerabilities in the analysed libraries.

To do this, we count the number of vulnerabilities in an analysed library instance V to be a function of its *own code*, *within-project dependencies*, *direct*, and *transitive dependencies*:

$$V \sim \text{own code} + \text{within-project dep} + \text{direct dep} + \text{trans dep} \tag{1}$$

Then we compute the linear model for (1) and estimate coefficients for each of the supposed 'predictors'. Table 7 presents the estimated coefficients and their descriptive statistics when we have considered the number of dependencies to be the values of the independent variables for the linear model. The SoA approach does not distinguish within-project dependencies, hence, we have used the *root* of the dependency tree (the analysed library instance) to represent the *own code* in (1). The p-value $\ll 0.05$ for all the predictors, hence they all have a statistically significant influence on the dependent variable (number of vulnerabilities). The model for estimating the number of vulnerabilities according to the SoA approach has $R^2_{SoA} = 0.603$ and stochastically distributed residual errors, hence, is appropriate for the description of the situation with dependencies.

We use the results returned according to the Vuln4Real methodology to model the number of vulnerabilities in direct and transitive dependencies. We used both root and the number of within-project dependencies of the analysed library to represent the *own code*. The root estimates for both models (direct and transitive vulnerabilities) have p-value > 0.05 and therefore are not significant. Other predictors are significant (p-values $\ll 0.05$). The models have $R^2_{\text{\# vulns direct}} = 0.523$ and $R^2_{\text{\# vulns trans}} = 0.623$, residual errors are stochastically distributed. Therefore, the number of dependencies can be used for predicting both the number of direct and transitive dependencies.

The regression analysis suggests that **the number of dependencies have a significant impact on the number of vulnerabilities in a software library (H4 is confirmed)**.

Discussion: The analysis of the coefficients of the model for predicting the total number of vulnerabilities suggests that both the number of direct and transitive dependencies lead to an increase in the number of vulnerable dependencies in this library. This observation corresponds to a general intuition: the more dependencies there is in a library, the higher chance that one of them is affected by a vulnerability.

The regression for the number of vulnerable direct dependencies suggests that the number of third party direct dependencies (3rdPty) have a positive influence, while the increase in within-project dependencies reduces the number of direct vulnerable dependencies. This finding suggests direct control to be the best assurance for quality criteria.

The coefficients of regression for the number of vulnerable transitive dependencies are positive. It is intuitive that the increasing number of transitive dependencies might lead to an increase in the number of vulnerabilities affecting them. The positive influence of within-project and 3rdPty direct dependencies might also happen because their addition might increase the number of transitive dependencies in a library, which indirectly increases the chance of having a transitive dependency affected by a security vulnerability.

9 Evaluation: Developer View

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 FOSS) is a continuing effort that spans the development and the operational phases of a product lifetime.

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.

It is therefore important to analyze whether the global analysis also has an impact as perceivable by the individual developer. Table 8 shows the effect of Vuln4Real on the dependency analysis results for a typical industrial library.

9.1 RQ1: Deployed dependencies

We observe from Table 8 that the 95% Confidence Interval of the reported number of vulnerable dependencies is significantly different between Vuln4Real and the state of the art due to filtering out (falsely reported) findings of development-only dependencies with known vulnerabilities. This phenomenon is extremely visible already from the lack of problems on the eight dependencies potentially lagging behind on average (seven at the very least). They become non-problematic in our analysis. They might have

Table 7
The influence of software dependencies on the number of vulnerabilities in the analysed libraries

The table shows the estimates for regression models of the total number of vulnerable dependencies ($R^2 = 0.603$, the p-value for all explanatory variables $\ll 0.05$), the number of vulnerable direct dependencies according to Vuln4Real ($R^2 = 0.523$, the p-value for all explanatory variables $\ll 0.05$), and the number of vulnerable transitive dependencies ($R^2 = 0.623$, the p-value for all explanatory variables $\ll 0.05$)

	#vulns SoA		#vulns direct		#vulns trans	
	estimate	std error	estimate	std error	estimate	std error
root	-0.4113	0.012	Not Applicable			
direct deps	0.1612	0.002				
transitive deps	0.1007	0.001				
within-project deps	Not Applicable		-0.0400	0.005	0.0063	0.003
3rdPty direct deps			0.1101	0.001	0.0012	0.001
transitive			-0.0021	0.001	0.0949	0.001

Table 8
Impact of the Vuln4Real methodology on the view of a single developer

The SoA and Vuln4Real columns show how many known vulnerable (or not vulnerable) dependencies would appear in the corresponding vulnerability report with a 95% Confidence Interval. The $\Delta\mu$ column show the means. From the total of known vulnerabilities, the individual developer will see four false alerts disappearing from one's to-check list.

Issues	SoA	Ours	
	CI	CI	$\Delta\mu$
Not Known vulnerable	[88, 103]	[89, 104]	+1
in lagging deps	–	[7, 9]	+8
Known Vulnerable			
total	[9, 11]	[6, 7]	-4
in your code	[0.3, 0.6]	[0.3, 0.6]	0
in your within-project deps	–	[0, 1]	+1
in your direct deps	[3, 4]	[4, 5]	+1
in your transitive deps	[6, 8]	[1, 2]	-6
in lagging deps	–	[0.1, 0.5]	+0.3

other problems, but surely not the misleading ones that some library used for their development was vulnerable.

The same situation is present on the dependencies affected by known vulnerabilities. Four of them on average are not affected at all (see the -4 in total row), and at the very least a developer will have two less false alerts (the difference between the lower bound of CI of the SoA and the upper bound of our approach. Similarly for other cases. Vuln4Real introduces no difference to the dependency analysis result from the perspective of an individual developer only for one's own individual library.

Hence, H1 is confirmed also from the view of an individual developer.

Discussion: 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 FOSS dependencies, these mitigations may consist of dependency updates, or in adhoc 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 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.

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.

The distinction between deployed and development-only components allows quick and reliable pre-filtering of not exploitable vulnerable dependencies, as they are not part of the deployed product. Any metrics reporting the "danger" of using FOSS libraries that do not discriminate between the deployed and development-only dependencies would lead to a wrong allocation of costly development and audit resources.

9.2 RQ2: Grouping dependencies by software projects

The simulation shows that according to the SoA approach the developer of an average software library would be notified that the majority of vulnerable dependencies are coming from transitive dependencies (7 out of 10). With a 95% confidence interval our methodology changes this view: only two vulnerabilities are introduced by transitive dependencies, while five are coming from direct dependencies. Moreover, one out of the seven vulnerable direct dependencies is the within-project dependency of the library. **The visible difference in the number of direct and transitive dependencies allows us to conclude that H2a is confirmed from the perspective of the individual developer**.

Discussion: 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 project dependencies. Failing to group dependency nodes that are updated together (e.g., belong to the same FOSS project), 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).

9.3 RQ3: Dead dependencies

For a 'sample' library, Vuln4Real reports presence of eight dead dependencies with the 30% – 50% chance that one of these dead dependencies to be affected by a security vulnerability. **This observation only partly confirms H2b** as we don't have a difference of a whole dependency among the confidence intervals.

Discussion: 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 FOSS 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.

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

10 Threats to Validity

Threats to *construct validity* concern the appropriateness of inferences made based on observations or measurements.

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 attributes, such as *Repository*, *ProjectID*, and others (which might be specific to certain language ecosystems).

The proposed conservative model for identification of whether a certain dependency has become dead may introduce some misclassifications. The model depends on the α parameter that defines the rate of decrease of the influence of more recent releases on the estimation of the next release date. Following our observations [5], the last three releases have the biggest impact on the next release date. The analysis of the reliability of the proposed model suggests that the model has correctly predicted that the library was 'alive' in 78% of cases. We believe this result to be sufficient to be used for preliminary evaluation of the status of a dependency, given the lightweight nature of the model. E.g., it only needs to have access to release history of an analysed library, in contrast with more sophisticated models based on, for example, linear regression that requires a sufficiently large dataset of release dates for training.

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

The selection of FOSS libraries is based on the number of usages from within SAP. Such a selection criterion may yield a sample, not representative of what libraries are most relevant for other industrial companies or FOSS developers. To check the popularity of the studied libraries within the FOSS community, we obtained the information about library usages from MVNRepository and the number of FOSS contributors that claimed to use the selected libraries from BlackDuck Openhub[32]. The results obtained from both sources suggested us that selected libraries are popular within the FOSS developers. Since SAP is a large multinational software development 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 FOSS projects developed in Java. The coverage is closer to 100% when considering the FOSS 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 a very comfortable way to handle dependency management and is wildly used within both FOSS 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. Any tooling for these languages would likely require fine-tuning on a case basis.

Vuln4Real targets on dependencies of software libraries. Since the configuration of a dependency tree might differ depending on the dependencies that appear on the higher levels of a dependency tree (e.g., direct dependencies), resolving the dependent libraries impacted by a vulnerable library may be not trivial. Moreover, developers of a library are mostly interested in its own dependencies and typically do not alert their users. Hence, we keep the analysis of dependent libraries out of scope for Vuln4Real. To find whether there is an impact of a vulnerable dependency on a dependent library, one can start the analysis by applying our methodology to the library of interest.

Threats to *reliability* concern the reliability of the tools and methods we have used in our paper.

Vuln4Real methodology is evaluated through a self-developed tool. Although tool development could have potentially introduced bugs that might affect the results presented in the paper, we tried our best to reduce the probability of bugs. The code of the tool was developed by one researcher and then carefully reviewed by other researchers. The results corresponding to the SoA approach, being compared with another dependency analysis tool internally used within SAP, did not reveal errors. Hence, we believe that the evaluation results presented in the paper are minimally affected by implementation bugs.

Table 9 shows the potential impact of the threats to validity discussed above on each step of Vuln4Real.

[32] https://www.openhub.net/

Table 9
Possible errors at each step of the Vuln4Real methodology

#	name of step	FP	FN	Reason
1	Extraction of a dependency tree			We employ actual mechanisms of a dependency management system to extract dependencies and resolve version conflicts.
2	Identification of development-only dependencies		✗	FN may happen if some of the dependencies are specified as excluded, and therefore, not shipped with the dependent library instance.
3	Identification of within-project dependencies	✗	✗	In case of Maven both FP and FN are possible if a project does not follow Maven name conventions.
4	Identification of dead dependencies	✗		A library may be falsely classified as dead due to an unusually long release time interval of a new version.
5	Identification of dependencies with known vulnerabilities		✗	Due to the limitations of the code-centric vulnerability mapping approach [6], not all vulnerable libraries could be identified (e.g., vulnerabilities whose fixes do not involve code changes or vulnerabilities that are due to the deserialization of untrusted data).
6	Path extraction			This step implies only postprocessing of the results, and therefore, is not affected by any errors, besides the implementation mistakes (that we tried our best to reduce).

11 Conclusions

In this paper, we have proposed the Vuln4Real methodology for a reliable measurement of vulnerable dependencies in FOSS libraries. In particular, the proposed methodology extends the state-of-the-art approaches to analysing software dependencies by applying several steps, such as (i) filtering development-only dependencies, (ii) grouping dependencies on their belonging to software projects, and (iii) determining whether a certain dependency is dead.

To demonstrate Vuln4Real, we selected the 500 most used FOSS Maven-based libraries from within SAP. To perform the analysis we have built a tool that leverages the functionality of Apache Maven to extract the library dependencies and applies the Vuln4Real postprocessing steps.

The results of our study demonstrate that the proposed methodology has visible impacts on both ecosystem and individual library developer views of the situation regarding software dependencies: Vuln4Real significantly reduces the number of false alerts for deployed code (dependencies wrongly flagged as vulnerable), provides meaningful insights on the exposure to third-parties (and hence vulnerabilities) of a library, and automatically predicts when dependency maintenance starts lagging, so it may not receive updates for arising issues.

An interesting direction for future research is to understand how important is the list of vulnerabilities. Obviously, any improvement in the precision of the list of vulnerabilities will give better results for *some* libraries. However, at the level of the ecosystem, when more than 20K GAV are analyzed his might only be visible in 1/2 percentage points. The same consideration might apply to the Developer's view. We also plan to complement this work with a qualitative study on the reasons why developers do not update dependencies with an investigation of developers' behaviour concerning security-related updates.

Acknowledgments

We would like to thank A.Brucker and P.Tonella for their insightful comments on early drafts of this work. This work was partly funded by the European Union under the H2020 Programme under grant n. 830929 (CyberSec4Europe).

References

[1] M. Pittenger, "Open source security analysis: The state of open source security in commercial applications," Black Duck Software, Tech. Rep., 2016.

[2] 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. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5

[3] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Proc. of ICSE'15*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 109–118. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819027

[4] 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.

[5] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proc. of ESEM'18*, 2018.

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

[7] I. Pashchenko, D. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proc. of CCS'20*. ACM, 2020.

[8] J. Huang, N. Borges, S. Bugiel, and M. Backes, "Up-to-crash: Evaluating third-party library updatability on android," in *Proc. of EuroS&P'19*. IEEE, 2019, pp. 15–30.

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

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

[11] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proc. of MSR'16*. IEEE, 2016, pp. 351–361.

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

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

[14] S. J. Long, "Owasp dependency check," 2015.

[15] 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.*, vol. 121, pp. 153–175, 2016.

[16] 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*. ACM, 2013, pp. 493–498.

[17] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Emp. Soft. Eng. Journ.*, vol. 21, no. 6, pp. 2268–2297, 2016.

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

[19] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *LJ*, vol. 2014, no. 239, p. 2, 2014.

[20] D. J. Reifer, V. R. Basili, B. W. Boehm, and B. Clark, "Eight lessons learned during cots-based systems maintenance," *IEEE Softw. Journ.*, vol. 20, no. 5, pp. 94–96, 2003.

[21] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, and G. Robles, "A formal framework for measuring technical lag in component repositories—and its application to npm," *J. Softw.-Evol. Proc.*, vol. 31, no. 8, p. e2157, 2019.

[22] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the impact of outdated and vulnerable javascript packages in docker images," in *Proc. of SANER'19*. IEEE, 2019, pp. 619–623.

[23] J. Coelho, M. T. Valente, L. L. Silva, and E. Shihab, "Identifying unmaintained projects in github," in *Proc. of ESEM'18*. ACM, 2018, p. 15.

[24] J. Khondhu, A. Capiluppi, and K.-J. Stol, "Is it all lost? a study of inactive open source projects," in *In Proc. of IFIP OSS'13*. Springer, 2013, pp. 61–79.

[25] T. Mens, M. Goeminne, U. Raja, and A. Serebrenik, "Survivability of software projects in gnome–a replication study," *Proc. of SATToSE'14*, p. 79, 2014.

[26] J. L. C. Izquierdo, V. Cosentino, and J. Cabot, "An empirical study on the maturity of the eclipse modeling ecosystem," in *Proc. of MODELS'17*. IEEE, 2017, pp. 292–302.

[27] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proc. of FSE'17*. ACM, 2017, pp. 186–196.

[28] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 201–228.

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

[30] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Commun. ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[31] D. Goode. (2014) Scaling mercurial at facebook. [Online]. Available: https://code.fb.com/core-data/scaling-mercurial-at-facebook/

[32] R. B. Harrys. (2017) The largest git repo on the planet. [Online]. Available: https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/

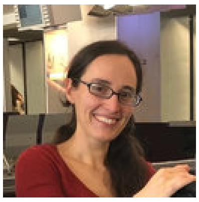
[33] R. G. Brown, *Statistical forecasting for inventory control*. McGraw/Hill, 1959.

[34] H. Cavusoglu, H. Cavusoglu, and S. Raghunathan, "Emerging issues in responsible vulnerability disclosure." in *Proc. of WEIS'05*, 2005.

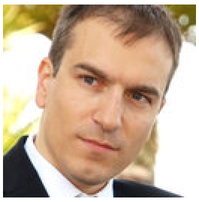
[35] H. Sajnani, V. Saini, J. Ossher, and C. V. Lopes, "Is popularity a measure of quality? an analysis of maven components," in *Proc. of ICSME'14*. IEEE, 2014, pp. 231–240.

Ivan Pashchenko is a Postdoctoral Researcher at the University of Trento. He received his Ph.D. in *Information and Communication Technology* from the University of Trento in 2019. His research interests include open-source software security, software verification, and machine learning for security. Contact him at *ivan.pashchenko@unitn.it*.

Henrik Plate is a senior researcher at SAP Security Research. He received his MSc in *Computer Science and Business Administration* in 1999 from the University of Mannheim. His current research focusses on the security of software supply chains, esp. the use of open source components with known vulnerabilities and supply chain attacks. Contact him at *henrik.plate@sap.com*.



Serena Elisa Ponta is a senior researcher at SAP Security Research. She received her Ph.D in *Mathematical Engineering and Simulation* in 2011 from the University of Genova and previously her MSc in *Computer Engineering* in 2007 from the same university. Her current research focuses on security of open-source libraries. Serena's prior research topics include security policies, configuration validation, and security validation for business processes. Contact her at *serena.ponta@sap.com*.



Antonino Sabetta is a senior researcher at SAP Security Research. He received his PhD in Computer Science and Automation Engineering from the University of Rome Tor Vergata, Italy in 2007. Antonino's most recent research interest is in the security of open-source software, especially using machine learning to analyze source code. Contact him at *antonino.sabetta@sap.com*.

Fabio Massacci is a full professor at the University of Trento. He has a Ph.D. in Computing from the University of Rome La Sapienza in 1998. He has been in Cambridge (UK), Toulouse (FR) and Siena (IT). Since 2001 he is in Trento. He has published more than 250 articles on security and his current research interest is in empirical methods for security. He participates to the FIRST SIG on CVSS and was the European Coordinator of the multi-disciplinary research project SECONOMICS on socio-economic aspects of security. Contact him at *fabio.massacci@unitn.it*.