UNIVERSITÀ DEGLI STUDI DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

Decision Support of Security Assessment of Software Vulnerabilities in Industrial Practice

Ivan Pashchenko

Advisor

Prof. Fabio Massacci

Università degli Studi di Trento

External reviewers

Dr. Achim D. Brucker

Prof. Paolo Tonella

September 2019

Acknowledgements

On the path towards this thesis, I met many great people who guided, supported, and taught me. I am very grateful to all of you and I believe that without you this thesis and my PhD would not be possible. Thank you all for being with me.

My special words are dedicated to my advisor, whom I value, proud of, and always respectively refer to as Professor Massacci (University of Trento, Italy). I want to thank you a lot for being exacting enough and honest with me. Under your supervision I have grown both as a researcher and as a person.

The applicable industrial part of this work would not be possible without the Vulas team of the SAP Security Research (SAP Labs France): Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. While working with you I always felt myself being on a team. Thank you for teaching me the practical thinking and for providing me with the unique opportunity to impact on the workflow of hundreds of SAP employees.

I want to thank Prof. Bruno Crispo (University of Trento, Italy), Dr Achim D. Brucker (University of Exeter, United Kingdom), and Prof. Paolo Tonella (Università della Svizzera Italiana, Switzerland) for being the PhD defense committee members. This dissertation greatly benefited from your valuable comments.

I want to thank PhD students of DISI, and especially my former colleagues Dr Stanislav Dashevskyi and Dr Katsiarina Labunets. You supported me since the very first steps of my way towards PhD. Thank you for your patience and your priceless advice for both research and social sides of being a PhD student.

I want to thank all my friends who timely distracted me from working. The activities you involved me in have always recharged and inspired me for the new steps of my research.

My special gratitude goes to my parents. Although you were distant, I always felt your presence and your support. And I especially want to underline dearest and most valuable person in my life, Ismagilova Zilia, who went with me through all the happy and challenging moments of the path towards this thesis.

Abstract

Software vulnerabilities are a well-known problem in current software projects. The situation becomes even more complicated, due to the ever-increasing complexity of the interconnections between both commercial and free open-source software (FOSS) projects. In this dissertation, we are aiming to facilitate the security assessment process in an industrial context.

We start from the level of the own code of an individual software project, for which we propose a differential benchmarking approach for automatic assessment of static analysis security testing tools. We have demonstrated this approach, using 70 revisions of four major versions of Apache Tomcat with 62 distinct vulnerability fixes as a ground-truth set to test 7 tools.

Since modern software projects often import functionality via software dependencies, that can also introduce vulnerabilities into the dependent project, we propose a methodology for counting actually vulnerable dependencies. We have evaluated the methodology on the set of 200 most used industry-relevant FOSS libraries, that resulted in 10905 distinct library instances when considering all the library versions.

Finally, we have investigated the situation on the level of the FOSS ecosystem. Here we have studied decision-making strategies of developers for selecting and updating dependencies, as well as the influence of security concerns on the developers' decisions from quantitative and qualitative perspectives. For the qualitative study we have run 15 semi-structured interviews with software developers from 15 companies located in 7 countries.

Keywords

Software Vulnerabilities, Benchmarking, Dependency Management, FOSS Ecosystem, Empirical Study

Contents

List of Tables

List of Figures

Chapter 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 [81]. By incorporating community-developed functionalities, software vendors reduce dramatically development and maintenance costs and can concentrate their investments on differentiating capabilities.

Unfortunately, the software industry is facing the difficult challenge of reconciling the cost saving *opportunity* represented by the availability of mature, high-quality FOSS components [65, 75, 81], with the *need* of maintaining a secure software supply chain and an effective vulnerability management process. Security flaws expose systems to attacks that may have a significant cost for the affected companies and, more generally, for our society (around 4 million dollars on average per incident, up $\simeq$ 30% from 2013, according to a report published in 2016 [83]). An analysis of more than 1000 applications performed by BlackDuck showed that 96% of the commercial solutions considered depended on FOSS components, and more than 60% of them included versions that were publicly known to be vulnerable and for which a more recent, non-vulnerable version existed [81].

As information security is a complete process, in this dissertation we consider various levels of granularity of interdependence of software projects. First, we look at a software project as a separately shipped unit, i.e., not connected with other projects. Then we make a step aside and consider relations of a software project with other FOSS projects. More specifically, we consider the bugs and security vulnerabilities that may be introduced into a software project while importing already implemented functionality distributed via separately packaged software libraries. Finally, we refer to a general interdependence between FOSS software projects and consider them as an interconnected ecosystem. In the next section we specify the problems addressed within this dissertation to facilitate the security assessment process of software projects at each level of granularity.

1.1 The Problem

To automate vulnerability identification *in the own code of software projects*, development teams may use static analysis security testing (SAST) tools. However, such tools are known to demonstrate different efficiency depending on the usage scenario [6], and therefore, have to be evaluated on their ability to identify vulnerabilities specific to a particular software project. This leads us to the following problem:

Problem 1: Static analysis security testing tools may be evaluated using synthetic micro benchmarks and benchmarks based on real-world software. However, the existing SAST tool benchmarks have the limitations, like the lack of vulnerability realism or uncertain ground truth. Hence, the state-of-the-art benchmarking approaches do not provide *the way to automatically select the best static analysis tool, that caters to industrial needs.*

Nowadays software projects often import some functionality from other libraries by including those libraries into their projects as software dependencies [81]. Since security vulnerabilities may be also introduced by the third-party components of a software project [36, 49], we identify the second research problem as follows:

Problem 2: Past dependency studies did not consider several key distinctions between (i) development-only and deployed dependencies (the former are not part of the final, running software system), (ii) outdated and "dead" libraries (the transitive dependencies of the latter would never be updated), (iii) dependencies under and out of direct control of the developer (as the latter require complicated and costly mitigation strategies), (iv) information obtained from natural-language advisories (e.g., Common Vulnerabilities and Exposures[1]) and information extracted from code analysis (the former might contain over-approximation errors). All these aspects expose automatic dependency analysis to misleading over-inflated results, that reduce the value of the analysis reports. Hence, there exists a need of *reporting only the dependencies affected by security vulnerabilities, relevant to the analysed project.*

Thanks to the modern development strategies and tools that facilitate dependency management, current software projects are highly interconnected. I.e., they do not evolve stand-alone, but rather consider the requirements of their users (e.g., dependent projects) and the development way of their dependencies. Hence, we may talk about both commer-

[1] `https://cve.mitre.org/`

cial and FOSS projects as *an ecosystem*, which we have been also interested to study from both quantitative and qualitative points of view. More specifically, we have identified the following problems of the FOSS ecosystem:

Problem 3 (quantitative point of view): Both commercial and FOSS projects may introduce technical debt by adopting software dependencies whose improper maintenance may result in serious security incidents, like the *Equifax* data breach, where over 100'000 credit card records were leaked due to the vulnerability introduced into the project by an outdated software dependency[2]. From this perspective, the FOSS ecosystem acts similar to financial markets, where widespread and high values of leverages led to a financial crisis. However, it is not known *if current financial models can be applied to the FOSS ecosystem to predict the risk of technical bankruptcy or to calculate the thresholds for software developers to work on new functionality for their libraries, to reduce their technical debt, or to quit the maintenance.*

Problem 4 (qualitative point of view): Current empirical studies provide very little insights on the developers' perception of FOSS dependencies: they either report some developers' feedback as a support to the proposed metrics [19,49] or focus the data collection within one software development company [9,80]. Hence, they do not provide any evidence on *how developers currently address the issues introduced by software dependencies.*

1.2 Contributions

Considering the problems listed above, this dissertation has the following contributions:

A differential approach for automatic comparison of static analysis security testing tools on the basis of historical vulnerability fixes in real-world software projects. To test our approach, we used 7 state of the art SAST tools against 70 revisions of four major versions of Apache Tomcat spanning 62 distinct Common Vulnerabilities and Exposures (CVE) fixes and vulnerable files totalling over 100K lines of code as the source of ground truth vulnerabilities.

A methodology for extracting and counting vulnerable dependencies in software projects. The methodology takes into account the industrial usage of third-party components, and therefore, caters to the needs of actual software developers. To understand the industrial impact of a more precise methodology, we considered the 200 most popular FOSS Java libraries used by SAP in its own software. Our analysis included

[2]`https://blogs.apache.org/foundation/entry/media-alert-the-apache-software`

10905 distinct GAVs (group, artifact, version) in Maven when considering all the library versions.

A quantitative study of the levels of technical debt and leverage in the FOSS ecosystem. The analysis presented in the study suggests the possibility of modelling and predicting the technical bankruptcy risks in the FOSS ecosystem. For the study we have used the set of 10905 library versions of the most popular Java libraries from the set of libraries that we considered for the validation of the methodology for counting actually vulnerable dependencies.

A qualitative study of the developers' perception of software dependencies. We have run 15 semi-structured interviews with software developers coming from 15 companies located in 7 different countries to understand whether our qualitative findings correspond to the day-by-day management of dependencies in industrial context.

Part of the work was performed in collaboration with an industrial partner - SAP. However, this thesis represents the research carried out by the author and does not necessarily represent the official position or research interests of SAP.

1.3 Thesis Structure

This thesis is structured as follows:

Chapter 2 provides an overview on the state-of-the-art approaches for benchmarking static analysis security testing tools, counting vulnerable dependencies, and modelling the FOSS ecosystem. Additionally, the chapter presents the studies that report developers' information needs. This chapter is partially published as the related works, background, and motivation sections of the research papers.

Chapter 3 presents the Delta-Bench, a differential benchmark for comparing static analysis security testing tools (Fortify SCA, Coverity, SonarQube, etc.), using historical fixes in real-world software as a ground-truth vulnerability source. This chapter was partially published in the following papers:

[77] Ivan Pashchenko, Stanislav Dashevskyi, and Fabio Massacci. "Delta-bench: differential benchmark for static analysis security testing tools." *In Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 163-168, 2017.

[76] Ivan Pashchenko. "FOSS version differentiation as a benchmark for static analysis

security testing tools." *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 1056-1058. ACM, 2017.

Chapter 4 describes 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. The chapter is partially published in:

[78] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. "Vulnerable open source dependencies: counting those that matter." *In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, p. 42. ACM, 2018.

The full version of Chapter 4 was submitted to:

Fabio Massacci, Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta. "Vuln4Real: A Methodology For Counting Actually Vulnerable Dependencies." *Submitted to ACM Transaction on Software Engineering.*

Chapter 5 presents the levels of technical debt and leverage in the FOSS ecosystem and provides a preliminary evidence of a possibility of applying financial models to the ecosystem of FOSS projects to estimate the risks of technical bankruptcy (the situation, when developers of FOSS projects abandon maintenance of their projects, since they do not see any added value), and therefore, avoid (or at least minimize) negative consequences.

Fabio Massacci and Ivan Pashchenko. "Technical Debt and the Risk of Leverage in the Open Source Software Ecosystem." *Working paper.* To be submitted to journal.

Chapter 6 reports the results of the empirical study done with the aim to validate the actual usefulness of the proposed approaches and underlined problems in the previous chapters of this thesis. The study is based on the semi-structured interviews with professional software developers on their perception of software dependencies. The chapter is planned to be submitted to:

Ivan Pashchenko, Ly Vu Duc, and Fabio Massacci. "FOSS Dependencies: A Qualitative Study on Developers' Perseption." *Working paper.* To be submitted to journal.

Finally, Chapter 7 reflects on the main contributions of this work, and provides discussion on the future work.

Chapter 2

State of the Art

In this chapter we provide an overview of how state-of-the-art studies approach the problems addressed in this dissertation. In section 2.1 we describe the existing ways of benchmarking SAST tools. Then in section 2.2 we provide the overview of the approaches for counting vulnerable dependencies. Finally, in sections 2.3 and 2.4 we present the current studies of the FOSS ecosystem from the quantitative and qualitative perspectives.

2.1 Own Code: Existing SAST tool benchmarks

The current approach for comparing performance of static analysis security testing tools (SAST) is to run a tool on a code fragment with a known bug or security vulnerability and evaluate the tool on its ability to identify the presence of the error, i.e., true positive (TP) / false negative (FN) evaluation. Similarly, the static analyser can be run on a fixed code fragment to evaluate its ability to avoid generation of alerts on a safe fragment of code, i.e., false positive (FP) / true negative (TN) evaluation.

Both TP/FN and FP/TN evaluations are quite straightforward in case there exists a set of safe (or fixed) and vulnerable code fragments, where the exact lines of code introducing the vulnerability are identified: a tool should produce some alerts pointing to the vulnerable lines of code and do not produce any alerts for the safe code fragment.

Unfortunately, creation of a database for evaluation of SAST tools often requires significant manual efforts (i.e., it took over 8 years and 5 large-scale public events for security researchers and software developers to create Software Assurance Reference Dataset[1]). On the other hand, various SAST tools may have different outputs [1]: one tool may report actual lines of code, while another tool may indicate the presence of a problematic data

[1]`https://samate.nist.gov/SARD/`

Table 2.1: Comparison of features of the existing benchmarks
"✓" means, that a benchmark provides a certain feature, while "na" shows that a certain feature is not applicable for it. We use a specific scale to distinguish TP/FN and FP/TN component size features: "+" means, that the size of an average component is smaller than 100 lines of code, "++" stands for average component sizes in a range from 100 to 1000 lines of code (possibly spread through several files), and "+++" shows that test case may include large files containing more than 1000 lines of code.

	[73]	[28]	[100]	[55]	[47]	[1]	[41]	[94]	[6]	[50]	[27]	[79]	Ours
Automatic testcase generation							✓				✓	✓	✓
Signature/ pattern independence	na	na	na	na	na	na		na	na	na			✓
Evaluation methodology							✓	✓	✓	✓			✓
Automatic TP/FN classification	✓		✓		✓	✓	✓	✓	✓				✓
Automatic FP/TN classification	✓		na		na	na	na	na	✓		na		✓
TP/FN component size	++	+++	+	+++	+	+	+	+	++	+++	++	++	+++
FP/TN component size	++	+++	No TN	+++	No TN	No TN	No TN	No TN	++	+++	na	++	+++

flow, and therefore, point to the lines of code where it starts (source) or finishes (sink). Hence, automatic comparison of tool alerts and identification of whether they are related to the actually vulnerable lines of code are not trivial.

To summarize, a thorough SAST tool evaluation requires the following:

- a large and reliable set of both buggy/vulnerable and fixed/safe code fragments;
- a benchmarking approach to distinguish the performance of the evaluated SAST tools.

Table 2.1 presents the comparison of features of the existing benchmarks. We do not find any benchmark, that works without a signature or a pattern of a vulnerability. Hence, the existing benchmarks are quite specific and require manual work to be extended for other vulnerability types. Moreover, we observe, that only few existing benchmarks support automatic testcase generation, which implies significant manual efforts in case one decides to extend a benchmark or adapt it for a specific use case. Majority of existing benchmarks (8 out of 12) consists of separated testcases and do not provide any methodology for evaluating SAST tools. Although 7 out of 12 benchmarks support automatic TP/TN classification of findings, only 2 of them provide a way to automatically perform FP/TN classification. The last feature is very important for the industrial usage of SAST tools, since they are well known to generate false positive alerts [86].

2.1.1 Collections of synthetic test cases

A large part of existing benchmarks for software analysis tools is based on "synthetic" software. For instance, Wilander and Kamkar [100] prepared a micro benchmark consisting from just one C file, containing 23 insecure and 21 safe API calls. Kratkiewicz [47] presented a synthetic set of 291 files, each representing a different buffer overflow vulnerability case in C programs. Shiraishi et al. [94] proposed a set of 1276 test suites, which cover 51 defect types in C programs.

In 2008 NIST initiated the Software Assurance Metrics And Tool Evaluation (SAMATE) project dedicated to improving software assurance by developing methods for enabling software analysis tool evaluation, measuring the effectiveness of tools and techniques, and identifying gaps in the existing tools and methods. The first large-scale public event named Static Analysis Tool Exposition (SATE), aiming to accumulate test data, was also conducted in 2008. From 2008 to 2016 there were five different stages of the event [10, 70–73]. One of the main outcomes of this project is the creation of the Software Assurance Reference Dataset (SARD), which contains more than 180k test cases from 3 Stand-alone suites, 38 test suites, and 18 applications for SAST tool comparison.

Another initiative for creating a large-scale collection of test suites for SAST tool benchmarking was done by NSA Center for Assurance Software (CAS). This resulted in the Juliet [1] test suite, which contains more than 86k buildable test cases, each containing only one malicious flaw.

While such artificial test suits allow us to clearly distinguish the vulnerable code and increase the maintainability of a benchmark, such test cases do not represent the complexity of real-world software, which may significantly change the results of the evaluation.

2.1.2 Databases of real-world bugs

Extraction of real-world bugs requires development expertise for locating and fixing them, and therefore, the very first benchmarks were presented by large software companies. For example, Hutchins et al. [38] presented the benchmark suite developed by Siemens. It consists of 7 small programs written in C. However, the bugs in the benchmark were introduced manually by the authors. In contrast, Emanuelsson and Nilsson [28] used the real historical security vulnerabilities in the Erickson software to benchmark static analysis tools. However, they did not make the benchmark available.

The software-artifact infrastructure repository (SIR) [26] is one of the early attempts to provide an open database of realistic bugs. It consists of 81 test cases, written in Java, C, C++, and C#. However, most of the defects in SIR are hand-seeded or obtained from mutation, and therefore, SIR may not be fully considered as a database of real bugs. The iBugs [22] and Defects4J [42] are the databases of real bugs for Java programs. iBugs contains 223 bugs with an exposing test case, while Defects4J provides 357 real bugs for 5 large real-world programs. While these databases allow benchmarking of software analysis tools on the real bugs, they contain a limited number of bug types. In particular, they do not focus on security, and therefore, cannot be used for assessing SAST tools.

An academic initiative to develop 8 Java web applications, that intentionally contain security vulnerabilities resulted in Stanford SecuriBench [55] benchmark, that covers SQL-injection, XSS, Path traversal, and HTTP splitting vulnerability types. Kupsch and Miller [50] performed manual analysis of the large FOSS project Condor, and identified 15 vulnerabilities. Then they used these vulnerabilities to evaluate two commercial static analysis tools. Reis and Abreu [88] mined 248 FOSS projects stored in Github to build a database of vulnerability. However, creation and maintenance of such collections of test cases is expensive, since it requires significant manual effort of software developers and security researchers. Moreover, such test suite often provides only a set of code fragments to be used for SAST analysis and do not facilitate tool evaluation, hence separate approaches are required to use the proposed benchmarks (i.e., [99] for the Juliet test suite).

2.1.3 Existing benchmarking approaches

There exist several studies that survey various features of SAST tools. For example, Li and Cui [54] provided a technical description of seven open source tools. The authors shared their experience with three of them in terms of false positive and false negative rates, after applying these tools against their own test code. Emanuelsson and Nilsson [28] described four commercial tools (CodePro, Coverity, FlexeLint, and Klockwork), providing case studies on their evaluation at the Ericsson company. Shiraishi et al. [94] performed an evaluation of 3 SAST tools, using generally used metrics detection rate and false positive rate, as well as productivity and cost efficiency of used tools. Oyetoyan et al. [74] performed evaluation of 5 open source tools (FindBugs, FindSecBugs, SonarQube, JLint, and Lapse+) and a "mainstream commercial tool", using SARD as a set of testcases to run the SAST tools. The authors used the following metrics to report the tool performance: #True Positives (TP), #False Positives (FP), #Discriminations, #Incidental flaws, Precision, Recall, and Discrimination rate.

Johns and Jodeit [41] introduced a common methodology for systematic evaluation of SAST tools using a benchmark composed of small programs that contain artificially injected vulnerabilities. The methodology references such characteristics as quality of the analysis, precision and scalability of the tools, set of known vulnerability types, and usability of the tools.

Antunes and Vieira [6] proposed an approach for assessing vulnerability testing tools by their performance on web services. They constructed VDBenchWS-pd and PTBenchWS-ud benchmarks and evaluated 8 static analysis and penetration testing tools.

Such comparisons provide valuable insights for researchers and security practitioners. Unfortunately, they cannot be easily generalized to a large population of SAST tools and source code bases, hence, it may be difficult to use them for tool selection in practice.

2.1.4 Automatic generation of benchmarks

A possible way to adapt real-world software for benchmarking purposes is to modify the original source code of an application to increase the potential coverage of SAST tools. Examples are mutation [40] and metamorphic [15] testing. Although such techniques may expand the applicability of static analysis tools to real-world software, they do not help automatic warning classification and do not solve a problem on how to compare outputs of different SAST tools [1].

Authors [79] propose EvilCoder, an approach for introducing software vulnerabilities by detecting and removing security mechanisms in real programs.

The solution proposed by Dolan-Gavitt et al. [27] (LAVA) suggests an artificial injection of vulnerabilities into the source code of real applications. Although this technique allows benchmarks to be created automatically, it does not allow false positive evaluation of SAST tools [27, §VIII]. The current implementation of LAVA is limited only to one vulnerability type (buffer overflow), and some vulnerability types cannot be injected using LAVA approach (e.g., logic errors, crypto flaws, and side-channel vulnerabilities).

2.2 Project Level: Counting Vulnerable Dependencies

Table 2.2 compares the existing approaches for analyzing software dependencies according to the considered aspects.

2.2.1 Accounting for Deployment

Kula et al. [49] 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

[2]As we reuse the vulnerability matching procedure introduced by Ponta et al. [84].

Table 2.2: Aspects considered in the related works

Rel work	Only deployed	Includes transitive	Vuln mapping	Dep groups	Dead deps
[101]		✓	Name-based		
[36]		✓	Name-based		
[51]	✓	✓	Manual		
[19]			Name-based + manual		
[102]	✓				
[49]			Manual		
[45]		✓(no resolution)			
Ours	✓	✓	Code-based[2]	✓	✓

owners to be unaware of vulnerable dependencies in their projects. Although the authors provide a thorough insight into developers' motivation, the reported developers' quotes 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 [49] may have been over-inflated (see the motivating example of Figures 4.3 and 4.4).

Several other works [14, 19, 36, 101] 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.

2.2.2 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. [51] underlines the finding that transitive dependencies of a project are more likely to be vulnerable, since developers (i) may not be aware about their existence and (ii) they have less control on them.

Since analysis of transitive dependencies requires one to follow the dependency tree construction and resolution procedures of a dependency management system, the analysis

Table 2.3: Approaches for Identification of Vulnerable Dependencies

Name	Approach	Advantages	Disadvantages
[57]	name-based matching	High performance	Prone to FP and FN
[14]			
[4]	semantic-web name matching	High performance	Prone to FP and FN (5% more than OWASP Dependency Check)
[84]	Patch-base matching	High precision	Manual effort required to create vulnerability database

may be both implementation-wise complicated and computation intense. This might be a reason for several recent studies [19, 45, 49, 102] to not consider transitive dependencies.

For example, Wittern et al. [102] in their study of the npm ecosystem did not follow the dependency tree construction algorithm and instead only considered the (direct) dependencies specified in the *package.json* files. Similarly, both Kula et al. [49] and Cox et al. [19] extracted dependencies from project configuration *pom.xml* files in their studies of the Maven ecosystem. This means that the studies reported results only for direct dependencies and did not apply resolution procedure of the analysed dependency managers.

Kikas et al. [45] 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 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.

2.2.3 Vulnerability Matching Approaches

Table 2.3 presents the approaches that are currently used to identify whether a certain library is affected by a security vulnerability.

The main source of vulnerabilities in software components is the National Vulnerability Database (NVD[3]) that uses the Common Platform Enumeration (CPE) standard for enumerating the affected components. The NVD represents the most complete, public source of vulnerabilities[4] albeit it does not cover all OSS projects with the same accuracy.

[3]`https://nvd.nist.gov/`

[4]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.

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 softwares (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, [68, 69] for the study of browser vulnerabilities and the large presence of false positives).

OWASP Dependency Check[5] 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 approach has high performance, it fully relies on the information present in the NVD.

Cadariu et al. [14] enhanced the OWASP Dependency Check tool to create a Vulnerability Alert Service (VAS) to provide the information about vulnerable dependencies used by clients of the Software Improvement Group (SIG). However, the authors discovered that the matching mechanism based on comparing library names with CPEs yields many false positives. Moreover, at the time of publication of [14] VAS was capable only to provide information regarding direct dependencies, while vulnerabilities may be also introduced via transitive dependencies [36].

Alqahtani et al. [4] used a semantic-web approach for mapping CVE descriptions from NVD database to the corresponding Maven 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 [4] may inaccurately estimate the number of vulnerable dependencies in the FOSS projects being affected by both FP and FN.

We rely on the works from Plate et al. [82] and Ponta et al. [84], who propose a precise approach to use the patch-based mapping of vulnerabilities onto the affected components (see Section 4.4).

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

2.2.4 Accounting for Own Dependencies

To the best of our knowledge, current dependency studies do not consider the fact that certain software libraries belong to the same project that uses them.

Although the concept seems intuitively simple, failure to distinguish own and third-party dependencies may incorrectly present as an insecure ecosystem with several vulnerable dependencies (a "dependency hell" [64]) 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 (only by changing versions of direct dependencies).

2.2.5 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 [87]. However, this becomes impossible, if an OSS library becomes dead [49]:

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

Automatic detection of the point in time, where a particular project becomes dead may be tricky. The recent work by Coelho et al. [18] 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. [44] 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. [63], Izquierdo et al. [39], and Coelho et al. [17]. However, the one-year threshold used by the above mentioned studies is arbitrary. Moreover, various software projects have different development strategies, and therefore, should have different intervals between commits and releases. Hence, the time threshold to count project as dead should vary depending on a project development strategy.

2.3 FOSS Ecosystem: Quantitative studies

Several technical studies [19,36,49,51,78] showed that FOSS dependencies, although being widely used by both commercial and FOSS projects, are not often maintained properly: a large share of projects (up to 81%) have outdated dependencies. Several of them (69%) are not aware that some of those dependencies introduce serious bugs and security vulnerabilities [49]. As Allman [3] drew parallels between technical and monetary debts, one may relate dependencies in FOSS to the well-studied financial leverage instruments whose excessive use might cause a financial crisis. However, we do not find a study that would try to model the situation in a FOSS ecosystem in this perspective.

Manikas and Hansen [60] presented a systematic literature review of 90 papers on the studies regarding software ecosystems. Although the number of software ecosystem research papers is increasing, the majority of studies are report papers. Hence, the authors reported the lack of analytic studies of software ecosystems. This statement is supported by another extensive literature review of 213 papers on software ecosystems [58]. Similar results are found by Manikas [59] in a more recent literature review of 56 empirical studies spanning over 55 software ecosystems: there exists a lack of deeper investigation of technical and collaborative aspects.

Boucharas et al. [12] proposed a standards-setting approach to software product and software supply network modelling. Although this allows developers to anticipate upcoming changes in the software ecosystems, the approach aims at development within one company, and therefore, does not suit the purpose of modelling FOSS infrastructure.

Bonaccorsi and Rossi [11] proposed a simple model that helps software developers to decide on whether to include FOSS components into their projects based on the value of the component. However, the proposed model does not consider the actual value of FOSS libraries, but rather count the value of receiving additional support from the developers of an FOSS community.

2.4 FOSS Ecosystem: Qualitative Studies

Several studies describe information needs of software developers. Such research studies aimed at finding the needs of industrial practitioners, and are, therefore, very important for academic researchers. I.e., they allow them to identify practical problems and focus research activities on solving them.

One of the first studies in this area was done by Sillito and Murphy [95]. The authors reported the results of their field study on the information, that developers need to change source code of their project. Ko and DeLine [46] used a speak aloud protocol. I.e., the authors collected information from developers' communications, while they were working on real tasks. In this way the study found 21 information needs of software developers. However, both studies focused on developers working on the code of their own projects and did not include the concept of a software project in an ecosystem of other projects.

Phillips et al. [80] performed an interview study at a large software development company to identify information needs that support integration decision-making in parallel development. The authors report awareness of software developers about software dependencies, and their impact on the software integration process. However, the study included only 7 developers from one company and was not focused on software dependencies, and therefore, provides only initial incites on the topic of developers perception of software dependencies.

Begel et al. [9] reported results of a survey done with Microsoft engineers on inter-team coordination. Similarly, Haenni et al. [33] made a survey of software developers in the OSS ecosystem to identify their information needs with respect to their upstream and downstream projects. The authors underlined the finding, that due to a lack of specialized tools, developers use non-specific ways to find such information. Although the studies report the need of software developers to have specialized tools that provide them with the information regarding inter-project collaboration, they do not provide any insights on the kind of information needed by developers in terms of fixing bugs and software vulnerabilities introduced by software dependencies.

Cox et al. [19] conducted 5 interviews with software developers to validate the proposed metric of dependency freshness. Although the study provides some insights on the developers' opinions regarding software updates, the number of interviews is too small to be generalized for all software developers.

Kula et al. [49] is the only study we are aware of, that reports the developer opinions regarding security vulnerabilities in the dependencies of their projects. However, the authors did not do an empirical study, but rather reported several interesting citations from the developer feedback on the reports regarding the existence of vulnerable dependencies in their projects.

Chapter 3

Delta-Bench: Differential Benchmark for Static Analysis Security Testing Tools

In this chapter we aim to address the limitations of the existing SAST tool benchmarks: lack of vulnerability realism, uncertain ground truth, and large amount of alerts not related to analyzed vulnerability. For this purpose we propose Delta-Bench – a novel approach for the automatic construction of benchmarks for SAST tools based on differencing vulnerable and fixed versions in Free and Open Source (FOSS) repositories. To test our approach, we used 7 state of the art SAST tools against 70 revisions of four major versions of Apache Tomcat spanning 62 distinct Common Vulnerabilities and Exposures (CVE) fixes and vulnerable files totalling over 100K lines of code as the source of ground truth vulnerabilities. Delta-Bench allows SAST tools to be automatically evaluated on the real-world historical vulnerabilities using only the alerts that a tool produced for the analyzed vulnerability.

3.1 Introduction

Designing a benchmark with real-world software is a challenging task [1]. Therefore, existing approaches either insert bugs artificially [21, 27], or use historical bugs from software repositories [42]. Artificial bug injection is often difficult to verify (see [21, p.2]), whilst historical vulnerabilities may represent only a subset of the ground truth.

Purely synthetic benchmarks [41] eliminate the above problems by isolating vulnerabilities into atomic tests that represent small applications, so that each of them contains only the code relevant to a vulnerability to be tested or a deliberately inserted FP, and some other closely related code which may be required for the vulnerable code to compile.

Still, for practical purposes one would like to know how a tool scales when moving from synthetic to real-world software. The biggest problem of using real-world software for benchmarking is that the code usually contains several "issues" simultaneously. Hence, the tool may produce many alarms not related to the vulnerability type for which we would like to use the software as a benchmark (*Background Noise*). Some of those alerts may be wrong but others may be "true" for other issues (see the discussion on the Juliet test suite [1, p.2]).

Developers perceive these large amounts of alerts to be a "pain in the neck" that goes along with the practical usage of static analysis security testing (SAST) tools [16]. Therefore, we aim to devise a methodology for benchmarking SAST tools that would combine both benefits of synthetic benchmarks and real-world software and answer the following research questions (RQ):

- RQ1: Can we "isolate" SAST tool alerts relevant to the ground truth vulnerability of a particular testcase?
- RQ2: How do the SAST tool alerts generated due to code issues not relevant for the ground truth vulnerability of a particular testcase affect the results of a benchmark?

3.2 Benchmark construction

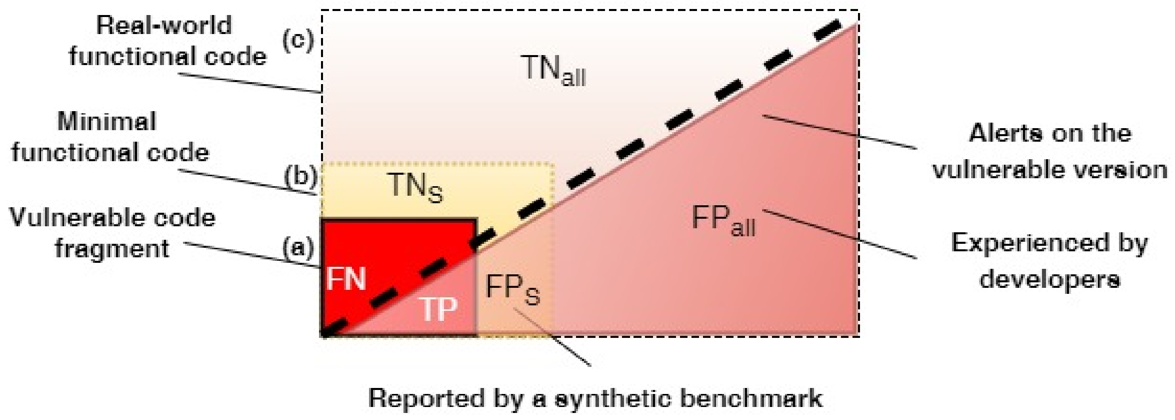
Similarly to Livshits and Lam [56], and Delaitre et al. [25], we intend to use large FOSS projects – these projects are well documented, their source code is publicly available, and their software repositories contain many historical vulnerabilities. Therefore, they can be used for identifying the ground truth – the expected correct output of a tool.

In a software repository we typically have available:

- C_{fixed} – the source code of a revision of a software project that was created to fix a security vulnerability.
- C_{vuln} – the source code of the last vulnerable revision that precedes the C_{fixed}.

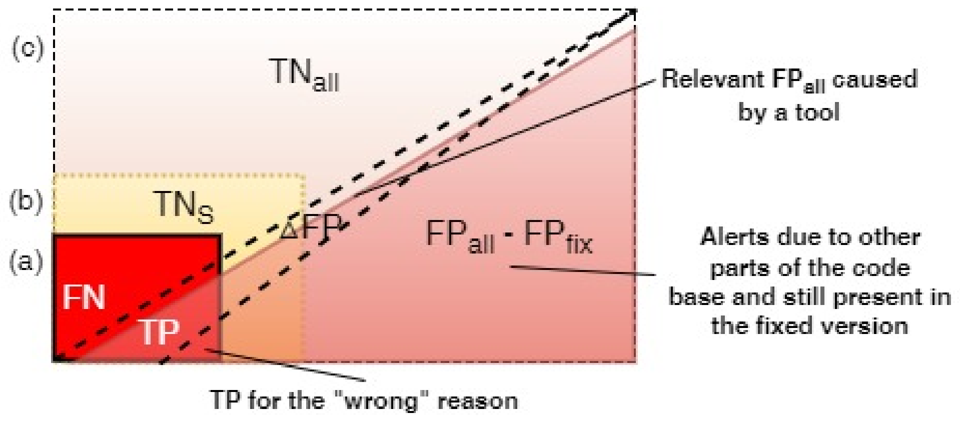
Figure 3.1 demonstrates a typical situation regarding the alerts of a SAST tool when running it on C_{vuln}. Ideally, the tool output for C_{vuln} should contain only alerts related to the vulnerability (TP area of the (a) square in Figure 3.1). A tool may not identify all the code related to the vulnerability in C_{vuln} (FN area in the (a) square in Figure 3.1).

SAST tools tend to generate many false alerts [86], so the tool output may contain


Real-world functional code
(c)
TNall
Minimal functional code
(b)
TNS
Vulnerable code fragment
(a)
FN
TP
FPS
FPall
Alerts on the vulnerable version
Experienced by developers
Reported by a synthetic benchmark

The tool output after analyzing a vulnerable version would likely contain many alerts not related to the analyzed vulnerability

Figure 3.1: Directly running the tool on the vulnerable version


(c)
TNall
Relevant FPall caused by a tool
(b)
TNS
ΔFP
FPall - FPfix
(a)
FN
TP
Alerts due to other parts of the code base and still present in the fixed version
TP for the "wrong" reason

The alerts not related to the analyzed vulnerability should be also present in the tool output on a fixed version. Hence, the alert subtraction may significantly decrease the amount of irrelevant alerts.

Figure 3.2: Considering different alerts on vulnerable and fixed versions

false positive alerts (FP_S) related to the vulnerable set of files in the squared area (b)[1]. In case of a specific synthetic benchmark there would be no other alerts, since a typical synthetic test case is focused and compact (i.e., contains only one vulnerability).

Unfortunately, real-world software projects usually contain many “issues” distributed between all the project files. Hence, a SAST tool would generate many more alerts (the FP_{all} area in the (c) square). The false alerts FP_{all} may be distracting, and therefore, unwanted by developers [16].

Some of the alerts may correspond to other flaws present in a project but unrelated to the vulnerable code fragment of the benchmark test. As these alerts are likely to be

[1]We consider such alerts as false positives, since we concentrate only on one vulnerability at a time.

present in the tool outputs for both C_{vuln} and C_{fixed}, we call them *Background Noise*. The successful fix of a vulnerability implies that the source code does not contain the vulnerable code anymore. Hence, the tool output on C_{fixed} should not contain alerts related to the vulnerability and observed in C_{vuln}. We can then subtract common alerts and evaluate the "actual" tool performance considering only the alerts relevant to the analyzed vulnerability (i.e., *Signal*). Figure 3.2 shows the alert distribution in the vulnerable code base after eliminating the alerts common for C_{vuln} and C_{fixed}.

The above intuition corresponds to our proposed process to generate a benchmark:

1. Determine the ground truth:
    - Identify a project that provides sufficient information about security fixes, so they can be identified in the source code (e.g., Common Vulnerabilities and Exposures (CVE) entries in the Git logs).
    - For each fixed vulnerability, identify a pair $\langle C_{vuln}, C_{fixed} \rangle$ – this information can be obtained either from the repository commit logs, vulnerability databases, or security notes.
    - Extract the source code constructs (files, and lines of code) modified during a fix (thus, likely vulnerable): we use the *diff* tool of a version control system.
2. Separate the *Background Noise* from the alerts related to the specific vulnerability – *Signal*:
    - Run a tool on a vulnerable version of the software C_{vuln}, and on the fixed version C_{fixed} (the fix must be the only difference between the two versions);
    - The *Signal* are the alerts differing between the tool outputs on C_{vuln} and C_{fixed}. Metrics (TP, FP, etc.) are only calculated on the alerts related to *Signal*. As *Background Noise* we consider the lines of code from the same files that were reported for both C_{vuln} and C_{fixed}.

The next step in the process is to assess the tool alerts and classify them as true or false positives.

Unfortunately, various tools may return different code lines for the same issue [1]. Moreover, a security fix may not touch the exact vulnerable line, but may modify a line that is relevant to the vulnerable one and is located "closely" to it (i.e., within the same method). An insertion of a sanitization mechanism for the user input may be an example of such a fix. Hence, a direct comparison of the lines reported by a tool with the code lines changed during a security fix would be misleading.

In this chapter we use files as a first approximation for alert classification: a TP is a file that has been changed during the security fix and for which there exists an alert pointing to that file. We extended our approach to work with methods, and plan to extend it to

Algorithm 1: Differential tool assessment

```
input  : A vulnerable revision $C_{vuln}$ and a fixed revision $C_{fixed}$
output: Differential assessment of tool alerts on file-level
// identification of the ground truth
1  $GTF \leftarrow \{file|file \in diff(C_{fixed}, C_{vuln})\}$ // $diff(C_1, C_2)$ is a diff tool of a version
      control system
2  $BackgroundNoise \leftarrow \emptyset$ ;
   // $Alerts(C)$ represents a tool output after running on $C$ and returns a set of
      $< file, line >$.
3  for each $< file, lile > \in Alerts(C_{fixed})$ do
       // $Adjust(file, line, C_1, C_2)$ converts positions of lines in $C_1$ into relative
          positions in $C_2$
4      $line^* \leftarrow Adjust(file, line, C_{fixed}, C_{vuln})$;
5      if $< file, line^* > \in Alerts(C_{vuln})$ then
6          $BackgroundNoise \leftarrow BackgroundNoise \cup$
                              $\{< file, line^* >\}$;
7      end
8  end
9  $Signal \leftarrow Alerts(C_{vuln}) \setminus BackgroundNoise$;
   // identification of a set of correct alerts
10 $TP_\Delta \leftarrow \emptyset$;
11 for each $< file, line > \in Signal$ do
12     if $file \in GTF$ then
13         $TP_\Delta \leftarrow TP_\Delta \cup \{file\}$;
14     end
15 end
   // classification of all the remaining alerts
16 $FN_\Delta \leftarrow GTF \setminus TP_\Delta$;
17 $FP_\Delta \leftarrow Alerts(C_{vuln}) \setminus GTF$;
18 $TN_\Delta \leftarrow file(C_{vuln}) \setminus (GTF \cup TP_\Delta)$;
```

Table 3.1: Software projects used for evaluation in this paper
The table shows the characteristics of an average vulnerable version (C_{vuln}) extracted from both Scanstud and Apache Tomcat: the total number of files in the revision, the number of vulnerable files, and the Prevalence rate (the ratio of vulnerable files in the revision). For Scanstud each vulnerable revision consists of one vulnerable file, while for Apache Tomcat an average revision may contain more than 1600 files with only 2-3 actually vulnerable files.

	#Files	#Vuln files	Prevalence Rate
	μ ($\pm$ σ)	μ ($\pm$ σ)	μ ($\pm$ σ)
Scanstud	1 (± 0)	1.0 (± 0.0)	1.000 (± 0.000)
Tomcat	1626 (± 318)	2.5 (± 3.3)	0.001 (± 0.002)

hunks (sequences of lines common to two files, interspersed with groups of differing lines[2]) and program slices.

Algorithm 1 shows how to filter *Background Noise* and classify the “clean” tool alerts. Since a line of code in a vulnerable revision may have a different position in a fixed revision, we have to convert the positions of the identified lines obtained after running a tool on the fixed version, in order to make the set of code lines comparable (we used *lhdiff* [7] for this purpose).

3.3 Data selection for evaluation

First, we select an appropriate synthetic benchmark, as well as a real-world software project to compare their discriminative power, i.e., the difference in results when comparing the performance of various tools. To have a fair comparison, both the synthetic benchmark and the project should be written in the same programming language (we selected Java, which is the most popular programming language since 2004[3]).

We used Scanstud by Johns and Jodeit [41] as a synthetic benchmark, since it contains a large number of tests and provides both “vulnerable” and “fixed” versions of each test.

We used Apache Tomcat as a real-world application, since it is mainly written in Java, and contains a large number of historical vulnerabilities that can be easily identified in its source code repository. This project has more than 800 thousands of lines of code, more than 15 thousands of commits and 30 unique contributors.

[2] `https://www.gnu.org/software/diffutils/manual/html_node/Hunks.html`

[3] 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.

To demonstrate our approach we identified 38 vulnerable-fixed file pairs from Scanstud. From Apache Tomcat we extracted 70 revisions with 62 distinct CVEs, which contain 178 vulnerable files out of the total amount of 113842 files. There are some common CVEs for different versions of the project. A revision was selected if it was possible (i) to precisely identify that the particular CVE was fixed, and (ii) to successfully build the project version. Table 3.1 shows the averages and standard deviations of total number of files, number of vulnerable files, and the prevalence rate in one experimental unit extracted from both Apache Tomcat and Scanstud, and Table 3.3 lists the vulnerability types present in both code bases.

To select SAST tools for benchmarking we considered the lists created by OWASP[4] and SAMATE[5]. Out of these lists we selected the tools that (1) support Java, (2) are specifically created for finding security vulnerabilities, and (3) can be easily automated. From the commercial tools, we could obtain an academic license for Fortify SCA (Checkmarx asked for several thousands euros a year). Table 3.2 contains the list of the selected SAST tools. All the tools were used in their default configuration. Due to the licensing issues, we obfuscate the real names of the tools while presenting their results.

We could not use all the tools from Table 3.2 for evaluation. One tool generated many issues both on Scanstud and Tomcat, but there were no security issues among them. FindBugs identified 21 out of 38 issues on Scanstud, but was not able to spot any vulnerabilities in Tomcat. This might happen because Tomcat contains many different vulnerability types, not all supported by FindBugs. However, the most likely reason is that Apache Tomcat developers actually used FindBugs (and also Coverity)[6], hence they may have already fixed the reported issues before committing code to the repository. We assume that this also caused the absence of alerts from three other tools on Tomcat. Moreover, two of them are unable to identify the vulnerability types present in Scanstud.

Instead, Tool A and Tool B use smart algorithms of data and control flow analysis and are constantly updated, and therefore, they identified some vulnerabilities in both Scanstud and Apache Tomcat. Hence, we will use them to demonstrate the evaluation of our approach.

[4]OWASP Source Code Analysis Tools list: `https://www.owasp.org/index.php/Source_Code_Analysis_Tools`

[5]SAMATE Source Code Security Analyzers list: `https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html`

[6]FindBugs is integrated into Apache Tomcat build scripts. Also, Apache Tomcat is listed among the projects that use Coverity Scan service (https://scan.coverity.com/projects/apache-tomcat).

Table 3.2: The SAST tools tested for this research

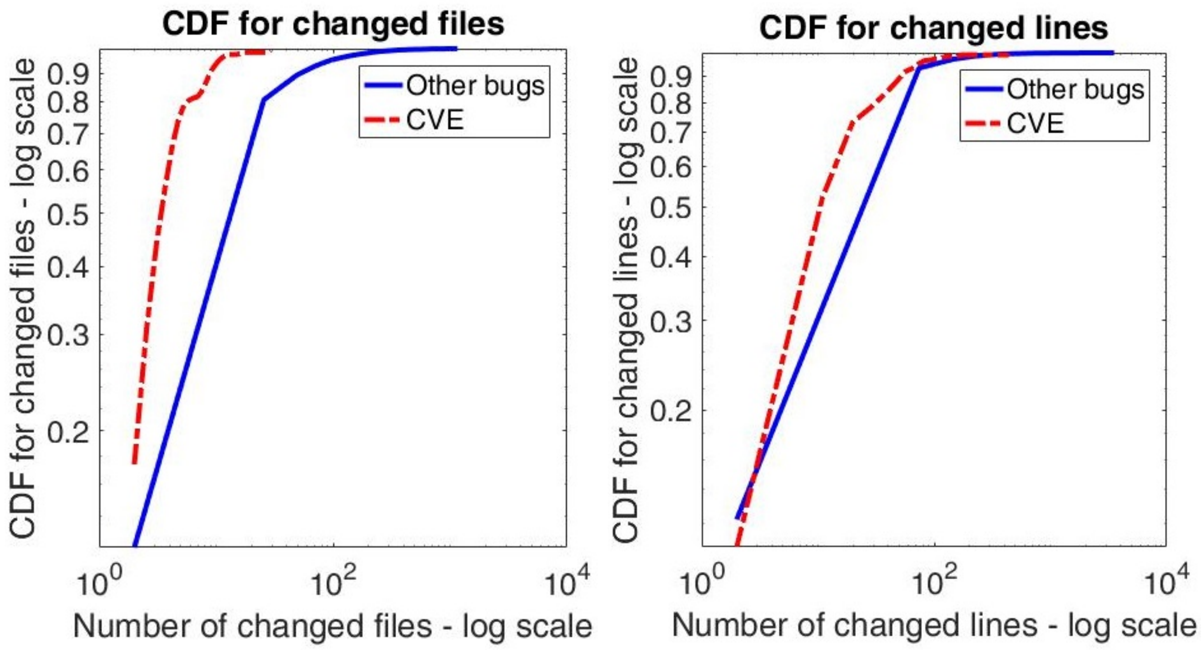
SAST	License	Version	Description
FindBugs	Free	3.01	Supports any JVM language and can detect 113 different vulnerability types.
Fortify SCA	Commercial	4.42	Supports 23 programming languages and detects over 700 vulnerabilities.
Jlint	Free	3.1.2	Works only with Java language. It helps to find more than 50 semantic and syntactic bugs.
OWASP LAPSE+	Free	2.8.1	Works only with Java language. The tool can identify 12 vulnerability types.
OWASP YASCA	Free	2.2	Supports 14 programming languages and aggregates results from 11 static analysis tools.
PMD	Free	5.5.1	Supports 20 programming languages and facilitates finding more than 25 bug types.
SonarQube	Free	5.6	Supports 20 programming languages and covers OWASP Top 10 vulnerability types.

3.4 RQ1: Can we isolate SAST tool alerts relevant to the ground truth vulnerability in a real-world software project?

Similarly to the *Defects4J* benchmark proposed by Just et al. [42], our benchmark construction methodology depends on "what else" happens during vulnerability fixes. Vulnerability fixes must not contain other changes that are not relevant to the purpose of the fix (e.g., refactorings or new features). Regular bug fixing may involve several files and several little "polishing" touches in several parts of the code base [37, 43]. If this was the case for security bugs, our ground truth could hardly be classified as such, as it would include a large amount of irrelevant changes.

Several studies observed that for disciplined projects such as Google Chrome, Mozilla's Firefox [67], and Apache Commons [53] the majority of security fixes are rather "local", which allows us to assume that in many cases the vulnerable code consists of closely related chunks located within a single file (or a handful of files).

To check if our assumption holds for Apache Tomcat, we performed a comparative



The cumulative distribution function (CDF) for CVE fixes demonstrates that CVE fixes tend to be much more local than all other fixes in terms of changed files. Also in terms of changed lines CVE fixes are more local than all other fixes: CDF for changed lines during usual fixes is sharper than CDF for changed lines during CVE fixes.

Figure 3.3: Comparing vulnerability fixes with non-security changes from the Apache Tomcat source code repository

analysis of known security fixes versus other commits not related to vulnerabilities (Figure 3.3). The distributions of the numbers of changed files and lines of code suggest that non-security changes are likely to be significantly larger (e.g., may spread to hundreds of files and involve thousands of lines), while security fixes are rather "local" (mostly a couple of files and less than 100 lines). Hence, **it is not likely that security fixes from our sample would contain irrelevant changes and we can isolate SAST tool alerts relevant to the ground truth vulnerability in a real-world software project**.

3.5 RQ2: How do the SAST tool alerts generated due to code issues not relevant for the ground truth vulnerability of a particular testcase affect the results of a benchmark?

From the perspective of "finding a vulnerable file", SAST tools are conceptually similar to defect predictors [34]. They provide the likelihood of the presence of software defects in a file (or a method) based on source code metrics (e.g., the size of the source code, cyclomatic complexity, etc.), development history (code churn, number of contributors, etc.), or other features. For example, Neuhaus et al. [66] used the information about past vulnerabilities in software components of Mozilla Firefox to identify the components that will likely cause vulnerabilities in the future; whereas, Shin et al. [93] assessed the defect prediction capabilities of traditional source code metrics versus developer activity metrics.

Whilst defect predictors mostly use statistical methods for identifying the relationships between the source code features and software defects, SAST tools use semantic-based information, and should have better performance [98].

Obviously, Tool A and Tool B (the two tools from Table 3.2 selected for evaluation) did not produce any *Background Noise* on Scanstud. Hence, we will report only one result for Scanstud, while there will be two results for Apache Tomcat.

At first we assessed whether a tool is able to identify a particular type of a security vulnerability. A tool succeeded, if there is at least one correct alert (i.e., at least one TP). Table 3.3 reports SAST tool performances by vulnerability types extracted from Scanstud and Apache Tomcat.

On Scanstud Tool B identified all 38 vulnerabilities, while Tool A found only 10 vulnerabilities (7 Cross-Site Scripting and 3 SQL injection). The analysis of SAST tool alerts in Apache Tomcat by vulnerability types showed different results. According to the Direct approach (running a tool on the vulnerable version) for alert classification, Tool A was able to find almost all vulnerabilities, while Tool B missed some of them. However, after removing the *Background Noise* the difference in tool performances changed significantly: Tool A still identified the majority of the vulnerabilities, but Tool B spotted only several of them. Some TP were made by chance, and they were filtered by Delta-Bench (removing the *Background Noise* according to the Algorithm 1). The significant change happened for Bypass, Cross-Site Scripting, Directory Traversal, and Information Disclosure vulnerability types.

Table 3.4 shows the average results for each tool. There is a significant difference in

Table 3.3: Relative Rankings by benchmark choice
The last column illustrates the relative performance of the two best tools in Table 3.2 first by running them on the vulnerable version (Direct row) versus the relative performance captured by Delta-Bench by removing the Background Noise according to the Algorithm 1 (Delta-Bench row).

Vuln type	Total vulns	Benchmark	Tool A	Tool B	Ranking
Cross-Site Scripting	35	Scanstud	7	35	$A \ll B$
SQL injection	3	Scanstud	3	3	$A = B$
Bypass	12	Direct	12	10	$A \geq B$
		Delta-Bench	12	1	$A \gg B$
Cross-Site Scripting	11	Direct	11	6	$A > B$
		Delta-Bench	10	2	$A \gg B$
Denial of Service	15	Direct	15	11	$A > B$
		Delta-Bench	13	8	$A > B$
Directory Traversal	3	Direct	3	2	$A \geq B$
		Delta-Bench	3	0	$A \gg B$
Exec code	2	Direct	2	2	$A = B$
		Delta-Bench	2	0	$A > B$
Information Disclosure	23	**Direct**	**22**	**23**	$A \leq B$
		Delta-Bench	**22**	**13**	$A \gg B$
Session Fixation	1	Direct	1	1	$A = B$
		Delta-Bench	1	0	$A \geq B$
Text Injection	3	Direct	3	3	$A = B$
		Delta-Bench	2	0	$A \geq B$

the relative ranking of the tools. On Scanstud Tool B was able to spot all the vulnerable files, while Tool A missed some of them. Hence, Tool B performs better in terms of both TP and FN. By design, synthetic benchmarks have no non-vulnerable files in C_{vuln}, hence the "n/a" for FP in Table 3.4 for Scanstud.

According to the Direct approach on Tomcat, Tool A produced more TP, less FN and FP comparing to Tool B, which shows that Tool A performs better. Delta-Bench increased the difference between the two tools, and therefore, made it possible to distinguish the two tools better. However, there is an inversion in the amount of FP: Tool B shows more FP according to the Direct approach, and Tool A shows more FP according to the Delta-Bench. This happens due to the fact, that Tool B produced much more warnings (i.e., *Background Noise*) than Tool A. Therefore, when we subtracted this *Background Noise* from the tool alerts, this eliminated the majority of FP produced by Tool B.

Table 3.4: Averages of file-level alerts

There is a difference between tool performances on Scanstud and real-world benchmarks: Tool B shows better results on Scanstud, but on real-world software Tool A performs better. Delta-Bench allows us to see this difference even better: the distance between means of tool metrics becomes bigger. In some cases there even occurs inversions, i.e., Tool B produces more false positives when executed on the vulnerable version (Direct row), while after subtracting BackgroundNoise Tool A starts to produce more False alarms (Delta-Bench row).

Metric	Benchmark	Mean of # Files		Ranking
		Tool A	Tool B	
TP	Scanstud	0.3	1.0	$A < B$
	Direct	2.2	1.7	$A > B$
	Delta-Bench	1.8	1.2	$A > B$
FN	Scanstud	0.7	0.0	$A > B$
	Direct	0.4	0.9	$A < B$
	Delta-Bench	0.7	1.4	$A < B$
FP	Scanstud	n/a	n/a	One file
	Direct	**554.0**	**677.0**	$A < B$
	Delta-Bench	**402.0**	**254.0**	$A > B$
Signal	Scanstud	0.3	1.0	$A < B$
	Delta-Bench	403.0	252.0	$A > B$
Background Noise	Scanstud	0.0	0.0	$A = B$
	Delta-Bench	152.0	426.0	$A < B$

Table 3.5: Averages of Precision, Recall and Negative Precision on file-level

Running tools on different types of benchmarks showed different performances in terms of Precision, Recall, and Negative Precision. Noise removal allows a better discrimination between tools, since the distance between metrics becomes more pronounced.

Metric	Benchmark	Tool A	Tool B	Result
Precision	Scanstud	0.3	1.0	$A < B$
	Direct	0.0039	0.0025	$A > B$
	Delta-Bench	0.0065	0.0044	$A > B$
Recall	Scanstud	0.3	1.0	$A < B$
	Direct	0.9	0.7	$A > B$
	Delta-Bench	0.7	0.4	$A > B$
Negative Precision	Scanstud	n/a	n/a	No TN
	Direct	0.9998	0.9995	$A > B$
	Delta-Bench	0.9995	0.9991	$A > B$

Table 3.5 shows the averages of *Precision*, *Recall*, and *Negative Precision*[7] for Tool

[7]Negative Precision demonstrates the ability of a tool to distinguish negative examples (i.e., testcases or parts of the code not related to the specified vulnerability) and is calculated in the same way as

A and Tool B. As it was mentioned for the average tool alerts (Table 3.4), the tools perform differently when executed on synthetic and real-world software. This is also visible for *Precision* and *Recall*. By design, there were only vulnerable files for C_{vuln} in Scanstud, and therefore, we cannot report any results for *Negative Precision*. Similarly to the observations on the average tool alerts, Delta-Bench allows tools to be better differentiated than the Direct approach.

Shaha et al. [89] in their study of bug reports showed that low-severity bugs can be very important (e.g., due to classification errors), therefore for our analysis we considered all warnings regardless of their severity, as we believe they can be also a subject to similar classification errors.

We also selected only the alerts with the top two severity levels. Both tools produced a negligible amount of TP, when limited to high severity alerts. As it was mentioned in section 3.3, Tomcat developers used other SAST tools, and therefore, they may have already fixed all the issues pointed out by the high severity alerts produced by those tools.
Claim: The improvement of the results obtained due to the application of the Delta-Bench approach allows us to conclude, that Background Noise obfuscates tool evaluations and even sometimes leads to opposite results (i.e., the inversions of the tool performance on the Information Disclosure vulnerability type in Table 3.3).

3.6 Threats to validity

Our results may be affected by errors in the data collection process, the accuracy of the information about security fixes in Apache Tomcat, and the mechanism for extracting either ground truth or code fragments pointed by alerts.

Bias in the data collection: although static analysis tools produce different kinds of output, we bring them to a common denominator by reducing the output to vulnerability warnings mapped to the source code locations. In this way we might overlook some other features of tools that, for example, can enhance user experience and may influence the selection. However, we focus on benchmarking of SAST tools from the security point of view, and therefore, we believe the actual ability of SAST tools to identify vulnerable code to be their most important feature.

Bias in the information about vulnerability fixes: there are few fixes that span over several commits (e.g., CVE-2009-3555), for which we used only the last commit that concluded the fix to reconstruct the vulnerable code fragment. It might be possible, that

Precision, where TP corresponds to absence of alerts in a safe fragment of code.

both ground truth and warning code fragments that we extract do not reflect the full vulnerable code sample. Such revisions contain partial fixes of the original vulnerability that may confuse a SAST tool, since a tool may (falsely) recognize an accepted partial fix to be complete. However, such cases are very important in industrial practice, since the incorrect decision of completeness of a vulnerability fix may leave a zero-day vulnerability. And therefore, the ability of SAST tool to distinguish partial fixes and point software developers on the part of code still affected by the vulnerability is highly important. Hence, we have used such cases in our empirical evaluation.

Bias in code base selection: Our private communications with an industrial SAST specialist suggest that such tools may be optimized towards finding vulnerabilities specific to web applications (e.g., XSS or SQLi), and therefore, some of selected SAST tools may not be effective in identification of vulnerability types extracted from Apache Tomcat. However, being a web server, Apache Tomcat still has a handful of vulnerabilities specific to web applications. Also, we have chosen the most popular SAST tools that claim to be capable of identification of vulnerability types used in our evaluation. Moreover, both tools A and B produced TP alerts for all the selected vulnerability types. Hence, we believe that this threat is limited.

Bias in SAST tool selection: we present results obtained only from two SAST tools. However, we use these tools to demonstrate the methodology without making claims about the overall performance of these tools and only show how different benchmarking methods may change the results. As for the future work we plan to use Delta-Bench for an empirical study of a large number of SAST tools on vulnerable-fixed revision pairs extracted from different real-world software projects.

3.7 Conclusions

We propose Delta-Bench – a novel approach that uses fixes of historical vulnerabilities from the existing FOSS projects as a ground-truth set of vulnerabilities to automatically construct benchmarks for SAST tools by (suitably) differencing SAST alerts from vulnerable and fixed versions. The approach allows us to evaluate SAST tools using only the alerts that a tool produced for the analyzed vulnerability (without considering the *Background Noise*). For benchmark construction Delta-Bench requires only a pair of vulnerable and fixed versions of a software code as an input.

We demonstrated Delta-Bench on a synthetic benchmark Scanstud and a set of historical vulnerabilities extracted from Apache Tomcat. Our experiments already showed significant insights between the two tools: we found that a relative tool ranking may be reverted by a different benchmarking method.

As for the future work, we plan to demonstrate Delta-Bench by using it for evaluation of different commercial and FOSS SAST tools, and on a larger set of real-world software projects as a source of historical vulnerabilities (beyond Java). In this chapter we show the results only at a file-level granularity. We have already extended Delta-Bench to work with methods, and are starting to extend it to hunks and program slices. The approach could be also applied to other types of bugs provided the assumption on the locality of fixes also applies to those bugs (as we have shown in Section 3.4 for security bugs).

By using Delta-Bench software development companies may select the most appropriate tool for their projects and tool developers improve SAST tools for sharper results.

Chapter 4

A Methodology for Counting Actually Vulnerable Dependencies

In this chapter we present the methodology for counting vulnerable dependencies, that addresses the over-inflation problem of academic and industrial approaches for reporting vulnerable dependencies in OSS software, and therefore, caters to the needs of industrial practice for correct allocation of development and audit resources. Careful analysis of deployed dependencies, aggregation of dependencies by their projects, and distinction of halted dependencies allow us to obtain a counting method that avoids over-inflation. To understand the industrial impact of a more precise approach, we considered the 200 most popular OSS Java libraries used by SAP in its own software. Our analysis included 10905 distinct GAVs (group, artifact, version) in Maven when considering all the library versions. Our study shows that the correct counting allows software development companies to receive actionable information about their library dependencies, and therefore, correctly allocate costly development and audit resources, which are spent inefficiently in case of distorted measurements.

4.1 Introduction

Current dependency analysis methodologies are based on assumptions that are not valid in an industrial context. They may not distinguish dependency scopes [49] which may lead to reporting non-exploitable vulnerabilities, or consider only direct dependencies [19] although security issues may be introduced transitively [51]. Moreover, dependency analysis methodologies 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, while 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 for a dependent project in case of a vulnerability discovery as there might be no new release that fixes the issue[1].

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 vulnerability mapping is error-prone, and therefore, not reliable;
4. *Misrepresentation as somebody's else problem* - separately considered dependencies that belong to same projects reduce 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 halted.

In this chapter we make the following contributions:

- The Vuln4Real methodology, that caters to the needs of industrial practice, for reliable measurement of vulnerable dependencies in free open-source software;
- A tool to perform large-scale studies of (Maven-based) FOSS libraries and to determine whether any of their dependencies are affected by known vulnerabilities;
- An empirical study of 10905 library instances of the 200 Java Maven-based FOSS libraries that are most frequently used in SAP software. The study is designed to offer two views of the FOSS ecosystem: the traditional academic view and the industrial view of a project developer. This allows us to present the Vuln4Real impact from both academic and industrial perspectives.

We found that Vuln4Real changes the academic perception of the situation regarding software dependencies, by showing that the developers of the analysed libraries can (and should) fix 80% of vulnerable dependencies by simply updating the direct dependencies of their projects, in contrast to the state-of-the-art approaches (by updating direct

[1]For example, there is no fixed version available for the dead 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).

dependencies, developers can fix only 37% of vulnerable dependencies)[2]. The proposed methodology also identifies and removes alerts for 27% of vulnerable direct and 21% of vulnerable transitive dependencies, that could not be exploited. Our study indicates that, under a conservative model to characterize dead dependencies, 13.2% of the total number of dependencies are dead, and therefore, may not receive updates (including security fixes). Such dependencies should be used with caution, since mitigations of their vulnerabilities may be costly.

The simulation of the proposed methodology on an individual software library shows that an industrial developers benefits from the correct resolution of the dependency analysis results: in average Vuln4Real allows an individual developer to receive 27% less false alerts. Also, it 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 complicated mitigations should be considered.

4.2 Terminology

In this chapter we rely on the terminology established among practitioners and used in well-known dependency management tools such as Apache Ivy[3] and Apache Maven[4]:

- 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*[5] 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*[6] is a representation of a software library instance and its dependencies where each node is a library instance and edges connect dependent library

[2]The developers are capable of fixing all the vulnerable dependencies of their projects, although mitigation of a vulnerability in a transitive dependency may require its transformation into a direct dependency, which is against the core idea of the automated dependency management approach.

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

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

[5]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.

[6]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.

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 *dead* 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 4.4).

To illustrate how this terminology is used in practice, we refer to Figure 4.1, 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.
- 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.

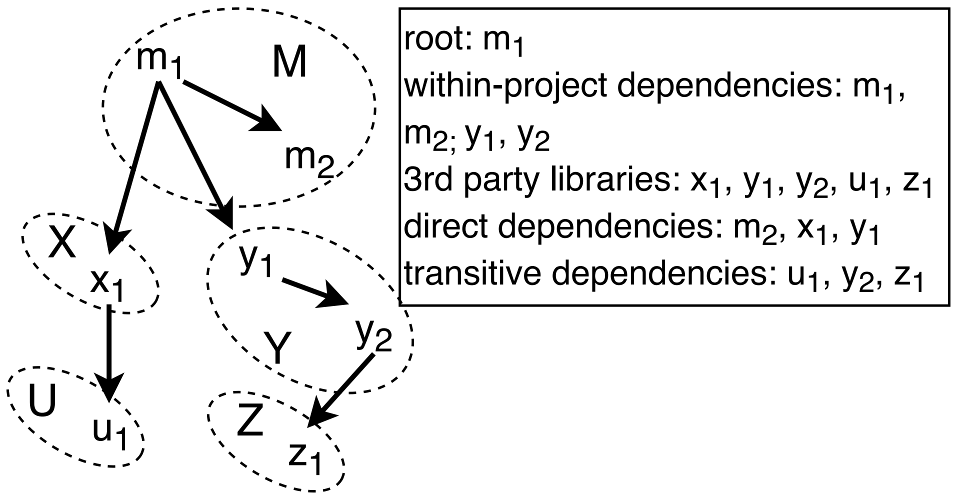
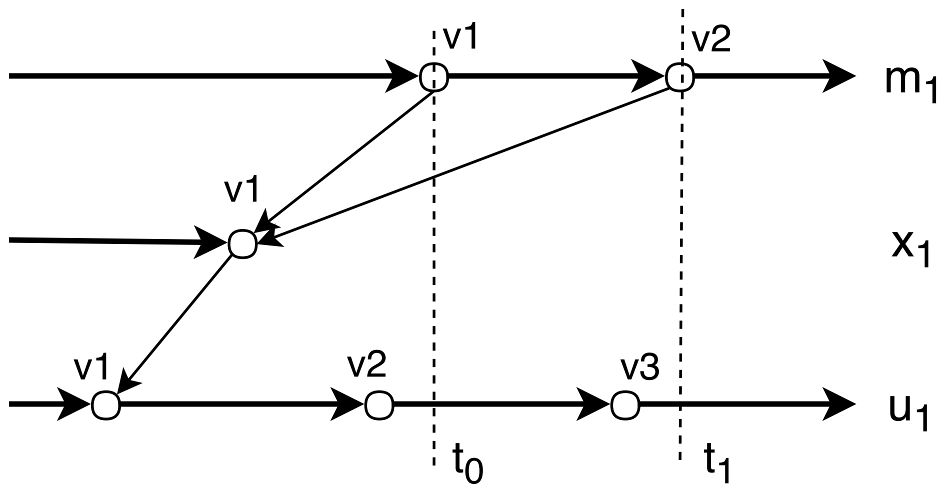

m1
M
m2
X
x1
y1
y2
Y
U
u1
Z
z1
root: m1
within-project dependencies: m1, m2; y1, y2
3rd party libraries: x1, y1, y2, u1, z1
direct dependencies: m2, x1, y1
transitive dependencies: u1, y2, z1

Figure 4.1: Dependency tree

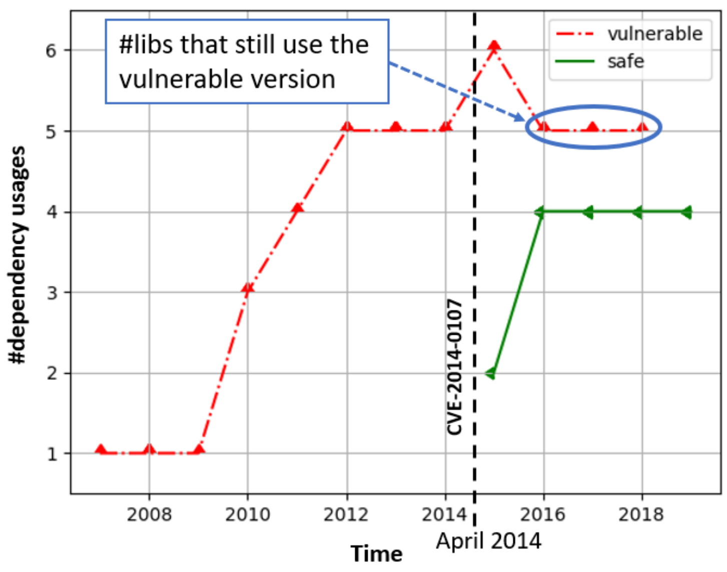
Even if library dependencies are not affected by known vulnerabilities, presence of dead dependencies may lead to costly mitigations in future: if a security vulnerability is discovered in a library that is no longer actively developed, there may be no version of this library that fixes the vulnerability[7]. Hence, being a dependency, this library will introduce the vulnerability to all its dependents.

Additionally, a dead dependency may transitively introduce outdated dependencies and expose the root library instance to bugs and security vulnerabilities (Figure 4.2): the root library instance m_1 depends on the last version of dead 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.



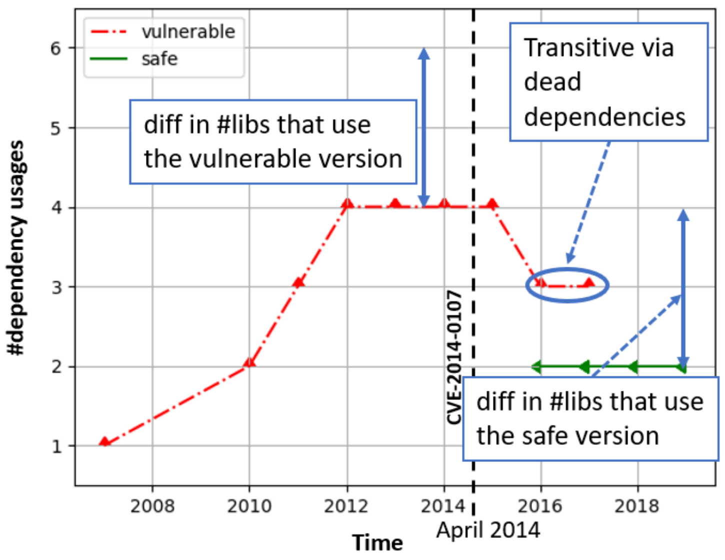
Library m_1 has a halted dependency x_1. In case a vulnerability is discovered in x_1 or its dependency u_1, there would be no version of x_1 that fixes such a vulnerability or adopts a fixed version of u_1.

Figure 4.2: Dead dependency



The figure presents the Apache xalan:xalan dependency migration plot calculated according to the approach presented in [49]. 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 development and test libraries. Hence, they may present a misleading picture of a dependency usage.

Figure 4.3: What Appears with State of the Art Methods



The figure shows the changed dependency usage plot for xalan:xalan library after removing test usages of this library. Three libraries did not adopt the safe version of the analyzed dependency for the simple reason that maintenance and development of those libraries halted.

Figure 4.4: Reality With Proper Processing

4.3 Motivating Example

Figure 4.3 shows the dependency migration analysis [49] 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 4.5). 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.

[7]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 abandonded 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).

Figure 4.4 shows the dependency migration plot after considering the five issues of the current state-of-the-art dependency analysis approaches (See Section 4.1). 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 4.3).

The presence of dead dependencies has a major impact on a library maintenance strategy. Indeed, in Figure 4.4, 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 vulnerable version of `xalan:xalan` via direct dead dependencies. In this case, a different mitigation strategy might be needed: (i) contribute to the dead library, i.e., to develop its new release; or (ii) fork the dead library and continue its maintenance as part of the dependent library.

4.4 Methodology

Table 4.4 overviews the Vuln4Real methodology for counting vulnerable dependencies.

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 of the library, that are used only during development of this library and 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 dead dependencies

INPUT	Resolved dependency tree for an analysed library
OUTPUT	The set of dead 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 maintained, otherwise count it as dead

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: • Analyse the patches that fix vulnerabilities in open-source software dependencies according to the code-base approach introduced by Plate et al. [82] • Use this to compare the nodes of the dependency tree to check whether one of them is affected by a known vulnerability

Step 6: Path extraction

INPUT	The dependency tree of an analysed library, the sets of development-only dependencies, groups of within-project dependencies, dead 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 dead 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[8] and the *dependency:resolve* goal to have the version conflicts resolved. The JavaScript packet manager *npm* provides the *npm ls <package-name>* command to display the dependency tree of a specified package[9]. Alternatively, there exists the *dependency-tree* plug-in[10], 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*[11] or *pipenv*[12] 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[13].

[8]https://maven.apache.org/plugins/maven-dependency-plugin/index.html
[9]https://docs.npmjs.com/cli/ls.html
[10]https://www.npmjs.com/package/dependency-tree
[11]https://pypi.org/project/pipdeptree/
[12]https://pypi.org/project/pipenv/
[13]https://github.com/pypa/pip/issues/988

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[14];
- 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[15], 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[16]. 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 dead 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:

[14]Such information is always available, albeit in possibly different formats.

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

[16]The monorepo development strategy is widely adopted by large software development companies, such as Google [85], Facebook [29], and Microsoft [35]

at earlier stages of development it needs to have more updates than an established library with a long development history. An example of a mature library is the Apache commons-logging package. Released on 2007-11-26 version 1.1.1 was the latest available version for more than 5 years till the release of version 1.1.2 on 2013-03-16.

Since the time difference between recent releases should have bigger impact on the *Last release interval* comparing to the time difference between older releases, the typical statistical model that describes such a process is a simple Exponential Smoothing model [13]:

$$\begin{aligned} \text{Release interval} = & \quad \alpha \sum_{i=0}^{n} \left\{ (1-\alpha)^i * \text{Release time}_{n-i} \right\} \\ \text{Expected release date} = & \quad \text{Last release} + \text{Release interval} \end{aligned}$$

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 time intervals decreases[17]. We estimate the *Expected release date* for a library by adding the *Last release interval* to the release date of the latest available version of the library. Then we determine the status of the library as follows:

- *next release date* $<$ *TIME*: the library is *dead*
- *next release date* $\geq$ *TIME*: the library is *alive*

TIME represents the date, for which the library status is calculated. In our study, for each analyzed library instance we will identify its release date and use it to calculate whether any of the dependencies were dead. To know the current status, *TIME* should be equal to the current date.

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

Step 5: Identification of dependencies with known vulnerabilities

To avoid the FP and FN inflation introduced by name-based vulnerability matching ([4, 14, 57]), we leverage on precise code-based approaches to vulnerability detection such as

[17]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 chapter we count $\alpha = 0.6$. For libraries with less than 3 releases, we take the *Last release interval* equal 3 months.

Ponta et al. [84] and Dashevskyi et al. [24]. Starting from known vulnerabilities from the NVD, advisories, bug tracking systems, etc., the commit fixing the vulnerability is identified manually and analyzed resulting in a list of code changes. All software constructs (e.g., constructors, methods) included in such list are the so-called *vulnerable code*. The creation of such knowledge is a one-time effort for each vulnerability. Then, for every analyzed project, the list of all 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 [84]. Whenever the vulnerable code fragment is contained within a dependency, the corresponding vulnerability is automatically reported for our analysis.

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;
- dead 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 4.1: 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).

Table 4.2: Descriptive statistics of the library sample
We considered the 200 most popular FOSS Java libraries used by SAP in its own software, which resulted in 10905 distinct GAVs when considering all library versions.

		μ	σ	min	max	Q1	Q2	Q3
#GAVs		54.52	49.24	1.0	248	15.0	35.0	87.0
#deps		10.91	16.98	0.0	127	0.0	3.0	15.0
	#direct	4.26	6.80	0.0	51	0.0	2.0	6.0
	#trans	6.65	12.01	0.0	82	0.0	1.0	8.0
#vuln deps		1.46	2.86	0.0	26	0.0	0.0	2.0
	#direct	0.54	1.32	0.0	9	0.0	0.0	0.0
	#trans	0.92	2.16	0.0	17	0.0	0.0	1.0
#usages		55.96	508.41	1.0	29 472	1.0	5.0	23.0

4.5 Data collection

Considering the popularity and industrial relevance of Java[18], in the following we demonstrate the proposed methodology on Java projects.

Over the past decade, Apache Maven established itself as a standard solution in the Java ecosystem for dependency management and other tasks related to build processes. Other solutions exist, such as Apache Ivy and Gradle (which is gaining popularity)[19], however Maven still has the largest share of users[20]. Hence, we use it to demonstrate the proposed mitigations for each problem described in Section 4.3.

In Maven the name of a component is standardized[21] 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

[18]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.
[19]`https://gradle.org/`
[20]`https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/`
[21]`https://maven.apache.org/guides/mini/guide-naming-conventions.html`

impractical and especially would include artifacts of no relevance in industrial practice. Hence, for this chapter we take a sample from Maven Central, as explained below.

Library selection - incorrect way. Initially, we followed the approach of [90] and selected the number of library instance usages as a proxy for its popularity. By usage we understood the number of direct dependent library instances of a library instance of interest[22].

However, 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.

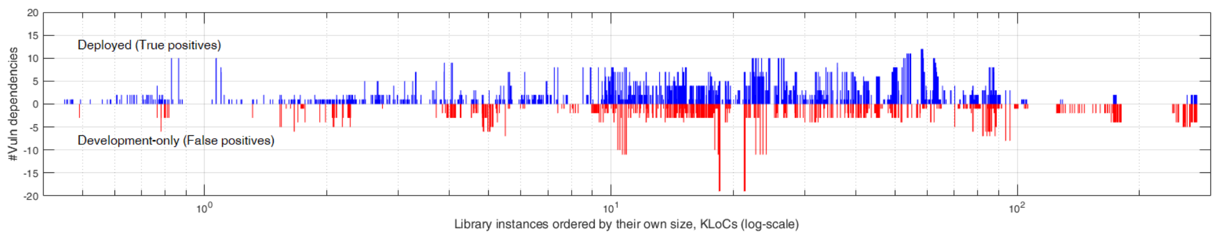
This approach may have potentially allowed us to receive a "good" list of libraries, if as a proxy for popularity we used the number of dependent projects. However, such information is not easily available (to obtain it, we would have to build dependency trees for all library instances in Maven Central),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

Library selection - the way we followed. To ensure industrial relevance of our study, we selected the top 200 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 10905 library instances when considering all versions (see Table 5.1 for descriptive statistics of the selected 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 [84] 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.

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



The number of deployed vulnerable dependencies (the true positives) per library are shown above the X-axis, while the number of development-only vulnerable dependencies (the false positives) are shown below the X-axis.

Figure 4.5: Deployed (the true positives) vs Development-only (the false positives) vulnerable dependencies per library instance

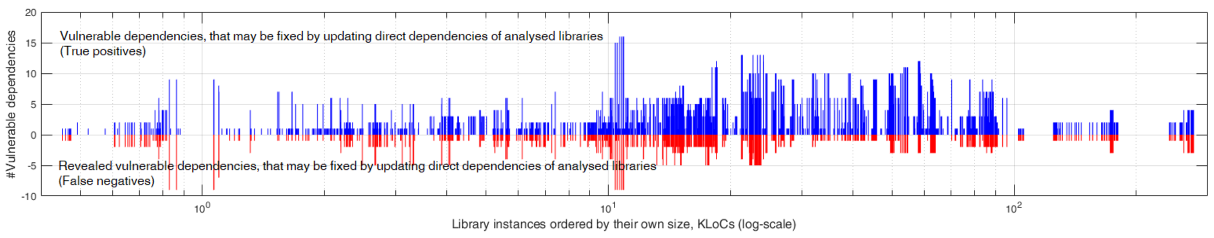
- applies path simplifications and produces the results in the form of a human-readable report.

4.6 Evaluation: Ecosystem View

In this Section we present how Vuln4Real influences the results of a dependency study for the complete sample of selected libraries by answering the following research questions:

- RQ1: How effective is the proposed methodology in removing false positive alerts?
- RQ2: How effective is Vuln4Real in revealing direct within-project dependencies of the analysed libraries?
- RQ3: How different is the view of dependencies provided by the proposed methodology in comparison to the state-of-the-art approach (SoA)?
- RQ4: How effective is the proposed methodology in finding dead dependencies?
- RQ5: Can the number of dependencies be used as a predictor for a number of vulnerabilities in a library?

To answer the research questions 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 4.1), which corresponds to approaches presented in, for example, [49] or [19]. Then we applied Vuln4Real and compared the number of vulnerable/non-vulnerable dependencies calculated according to both approaches.



Above the X-axis we show the total number of direct vulnerable dependencies and their within-project dependencies. They are directly introduced into the analysed libraries by importing the functionality from vulnerable projects. Below the X-axis we show the fraction of vulnerable within-project dependencies of the direct dependencies of the analysed libraries, i.e., the false negative alerts revealed by Vuln4Real.

Figure 4.6: The comparison of the actual number of vulnerable dependencies that could be directly fixed by developers of the analysed libraries (true positives) and the fraction of such dependencies revealed by Vuln4Real (the false negatives)

RQ1: How effective is the proposed methodology in removing false positive alerts?

Vulnerabilities in development-only dependencies cannot be exploited, and therefore, the alerts reporting them are not useful for software developers of the analysed libraries, so we count them as false positives. Figure 4.5 shows the per library instance comparison between the number of deployed vulnerable dependencies (true positives) and development-only[23] vulnerable dependencies (false positives). The results are ordered by the own size (number of lines of code) of the analysed library instances.

We observe, that development-only vulnerable dependencies are widely used within the analysed libraries and for some library instances their amount exceeds the number of deployed dependencies. Hence, following the SoA approach, software developers would have to face a big number of false alerts (for some library instances their amount exceeds the actual true positive alerts by up to 3 times). Their analysis may require significant amount of highly expensive developers' time and, as a result, decrease the value and trust in the dependency analysis findings. Instead, the proposed methodology allows software developers to receive trustworthy dependency analysis reports.

When considering only deployed dependencies, the number of both non-vulnerable and vulnerable direct dependencies decreased by 45% and 27%, respectively[24]. At the same

[23]To identify development-only dependencies in Maven we have used both *test* and *provided* scopes, since dependencies in both scopes do not appear as transitive dependencies in the dependency trees of the dependent libraries.

[24]Such difference may be due to the fact, that development-only dependencies are of less interest to both security researchers and hackers, and therefore, there are less vulnerabilities discovered in such

time, our methodology being applied to transitive dependencies allowed us to reveal 21% of development-only vulnerable transitive dependencies.

RQ2: How effective is Vuln4Real in revealing direct within-project dependencies of the analysed libraries?

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 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 analysed libraries, while some vulnerable transitive dependencies appear to be direct dependencies of the analysed libraries.

Vulnerable within-project dependencies. Since the analysed library instances may as well be parts of multimodule 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 4.7 presents the difference between the within-project dependencies and the third-party direct dependencies of the analysed libraries. Although the mean number of third-party direct dependencies ($Direct_{3rdPty} = 4,38$) is almost two times bigger than the mean number of within-project dependencies ($Direct_{within-project} = 2,53$), we observe, that within-project dependencies introduce as much as 21% of direct vulnerabilities.

Vulnerable direct dependencies. Figure 4.6 shows the per library instance comparison of the number of direct and the number of revealed vulnerable dependencies.

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 is affected by FN. For several libraries the amount of FN alerts equal to the number of TPs. Hence, Vuln4Real allows us to reveal direct vulnerable dependencies, which were falsely hided by the SoA approach.

libraries.

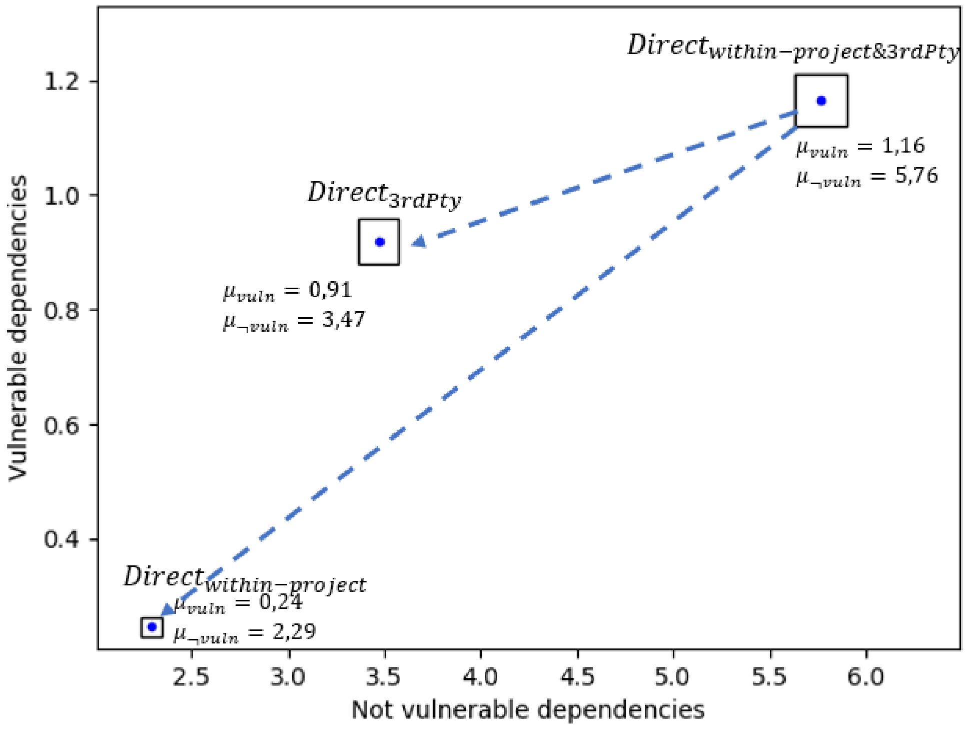


Figure 4.7: Own and third-party dependencies of the analysed libraries

After grouping within-project dependencies, we observe, that 80% of vulnerable third-party dependencies are direct (in contrast with 37% reported by SoA), and therefore, could (and should) be fixed by the developers of the analysed library instances.

RQ3: How different is the view of dependencies provided by the proposed methodology in comparison to SoA?

Figure 4.8 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 ($\mu = 0.78$), rather than direct dependencies ($\mu = 0.73$) per library instance.

In contrast, our methodology dramatically changes this picture. Filtering out deployed dependencies decreased the mean number of non vulnerable dependencies from $\mu_{SoA} = 12.26$ to $\mu_{deployed} = 6.20$ and the mean number of vulnerable dependencies withing the

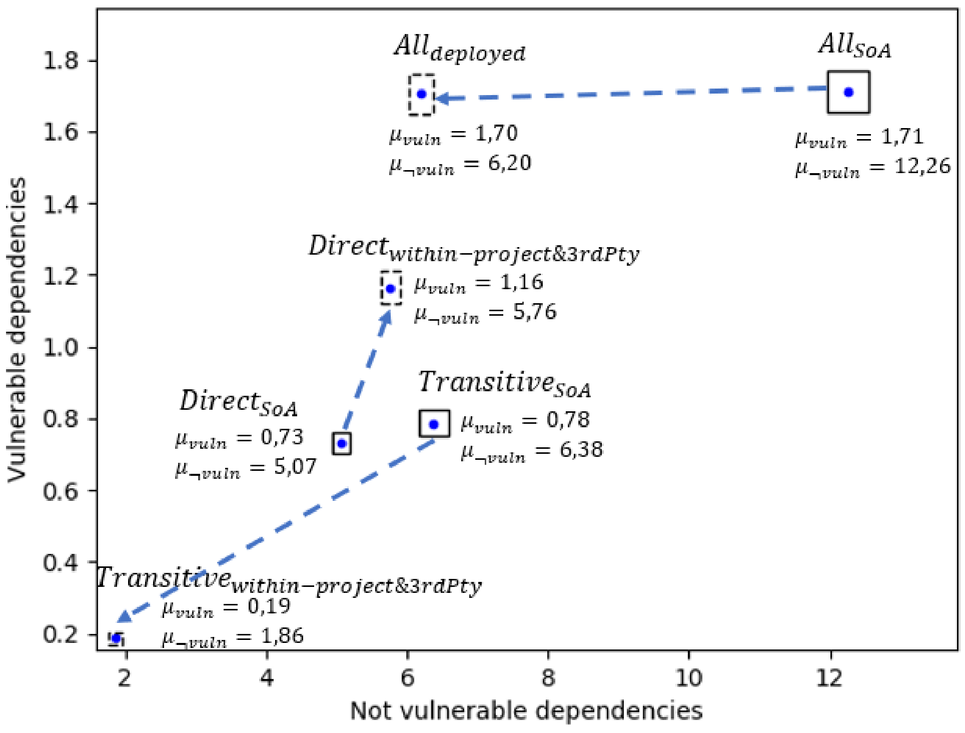


Figure 4.8: Direct and transitive dependencies according to SoA and Vuln4Real

analysed library sample from $\mu_{SoA} = 1.71$ to $\mu_{deployed} = 1.70$. Furthermore, after the grouping procedure, the mean number of both vulnerable ($\mu = 0.73$) and non vulnerable ($\mu = 5.07$) direct dependencies has become bigger than the mean number of vulnerable ($\mu = 1.16$) and non vulnerable ($\mu = 5.76$) transitive dependencies.

Table 4.4: The effect of direct dead dependencies (RQ3)

13.2 % of dependencies of the analyzed library instances are dead, while 2 % of them are affected by known vulnerabilities.

	deployed & controlled & 3rdPty			dead		
	total	μ	CI	total	μ	CI
vuln	10038	1.48	[1.42, 1.54]	88	0.013	[0.010, 0.016]
¬vuln	35389	5.23	[5.10, 5.36]	5350	0.79	[0.760, 0.821]

Table 4.5: The effect of transitive dead dependencies (RQ3)
Direct dead dependencies introduced 12 vulnerable dependencies transitively.

	deployed & controlled & 3rdPty			dead			transitive via dead		
	total	μ	CI	total	μ	CI	total	μ	CI
vuln	1975	0.29	[0.27, 0.32]	77	0.011	[0.009, 0.014]	12	$17.7E^{-4}$	[$6.1E^{-4}$, $29.3E^{-4}$]
¬vuln	17013	2.51	[2.39, 2.64]	3006	0.444	[0.417, 0.471]	645	0.09	[0.08, 0.11]

RQ4: How effective is the proposed methodology in finding dead dependencies?

To answer RQ4 we considered only deployed dependencies, grouped according to the software projects they belong to (Tables 4.4 and 4.5). We found that 12,0% of the overall number of direct dependencies and 16.2% of transitive dependencies in our sample are dead. Some of them (88 direct and 77 transitive out of 8521 dead dependencies) are affected by known security vulnerabilities. Although this number is not big, each case of a dead dependency is very important. Such dependencies do not have a fixed version, and therefore, a costly mitigation is needed to fix such vulnerabilities.

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

Vuln4Real allowed us to identify that 13.2% of the dependencies in our sample are dead, while 2% of them are affected by known vulnerabilities. Moreover, direct dead dependencies also transitively introduced 645 dependencies, 12 of which are vulnerable.

RQ5: 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 [36, 51], Vuln4Real changes the distributions of vulnerabilities between direct and transitive dependencies. Hence, we have been 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*

Table 4.6: The influence of quantity of software dependencies on the number of vulnerabilities in the analysed libraries

	#vulns SoA		#vulns direct		#vulns trans	
	estimate	std error	estimate	std error	estimate	std error
root	0.0147	0.027				
direct deps	0.068	0.003	Not Applicable			
transitive deps	0.161	0.002				
within-project deps			0.159	0.002	-0.020	0.001
3rdPty direct deps	Not Applicable		-0.023	0.003	0.084	0.001
transitive			-0.003	0.005	0.024	0.002

Table 4.7: The influence of dependency sizes on the number of vulnerabilities in the analysed libraries

	#vulns SoA		#vulns direct		#vulns trans	
	estimate	std error	estimate	std error	estimate	std error
root size	0.106	0.004				
direct dep size	0.364	0.003	Not Applicable			
transitive dep size	-0.010	0.018				
within-project dep size			-0.166	0.011	-0.192	0.006
3rdPty direct dep size	Not Applicable		0.318	0.003	-0.012	0.002
transitive dep size			-0.068	0.005	0.356	0.003

dependencies:

$$V \sim \text{own code} + \text{within-project dep} + \text{direct dep} + \text{trans dep} \tag{4.1}$$

Then we compute the linear model for (4.1) and estimate coefficients for each of the supposed 'predictors'. Table 4.6 presents the estimated coefficients and their descriptive statistics, when we have considered the quantity 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 (4.1). 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 $R^2_{SoA} = 0.604$ and stochastically distributed residual errors, hence, is appropriate for 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*, however, the root estimates for both models (direct and transitive vulnerabilities) have p-value > 0.05, and therefore, we have excluded it. Other predictors are significant (p-values $\ll 0.05$). The models have $R^2_{\#\text{ vulns direct}} = 0.571$ and $R^2_{\#\text{ vulns trans}} = 0.475$, residual errors are stochastically distributed. Therefore, the number of dependencies can be used for predicting both the number of direct and transitive dependencies.

However, considering only the quantity of dependencies may not reflect the fact that various dependencies bring different values to the analysed libraries. Hence, we have also considered the dependency sizes. For this purpose we extracted the number of lines of code in the java files of involved library instances. We referred to Maven Central to extract the source code of the involved library instances. In case the source code of a particular dependency was not available in Maven Cenral, we ignored such dependency and its dependent library. Hence, 8275 library instances (instead of 10905) left for the regression analysis. The results are presented in Table 4.7.

The regression analysis suggests, that the number of dependencies have a significant impact on the number of vulnerabilities in a software library.

Table 4.8: Ecosystem view - results

RQ	Finding
RQ1	The Vuln4Real methodology has allowed us to remove alerts about 27% of direct and 21% of transitive dependencies with known vulnerabilities, that cannot be exploited.
RQ2	Vuln4Real reveals that as much as 21% of direct dependencies affected by known security vulnerabilities are within-project dependencies of the analysed libraries, hence, developers of analysed libraries should fix them directly by changing their code. Also, according to the SoA approach, it may seem that developers of the analysed libraries have direct control of only 37% of the vulnerable dependencies, while in reality they are responsible for fixing 80% of the deployed vulnerable dependencies.
RQ3	The Vuln4Real methodology removes the appalling feeling of an unmanageable 'dependency hell'.
RQ4	Analysis according to Vuln4Real shows, that 13.2% of the dependencies in our sample are dead, while 2% of them are affected by known security vulnerabilities. Direct dead dependencies also transitively introduced 645 dependencies, 12 of which are vulnerable.
RQ5	The number of dependencies have a significant impact on the number of vulnerabilities in a software library, hence, it can be used to model and predict the probability of a dependency to introduce a security vulnerability into a dependent library instance.

4.7 Evaluation: Developer View

4.7.1 Requirements for an Industrial Practice

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

As part of SAP's secure development life-cycle, all development projects go through several validation steps and each single finding has to be audited, assessed, and mitigated. After the product is released to customers, and for its entire operational lifetime, its own security and the security of its third-party dependencies are continuously monitored. When a vulnerability is detected in one of the dependencies, timely mitigations need to be developed and deployed to all affected systems. In the case of FOSS dependencies, these mitigations may consist of dependency updates, or in ad-hoc fixes in the product that relies on the affected library or in the dependency itself (through a company-internal fork that can be temporary or persistent). When the product portfolio of a company includes thousands of products, whose support period can extend to decades, wrong assessments lead to inadequate risk management and inefficient allocation of resources, which ultimately translate to increased chances of security incidents and financial loss.

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

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

Finally, determining precisely whether a dependency could be upgraded to a non-vulnerable version or not (because such a version does not exist, and perhaps will never exist, if the dependency is no longer maintained) is the key to choosing the correct mitigation strategy. Addressing vulnerabilities in 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.

4.7.2 Simulation of the Vuln4Real methodology on an individual software library

To identify a typical industrial library, we have extracted the number of direct dependencies for each SAP software project in the proprietary Nexus 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 11. Then we have artificially constructed dependency trees for 100 software projects according to Algorithm 2.

Table 4.9 shows the effect of Vuln4Real for a typical industrial library. We observe that the mean number of vulnerable dependencies decreased from 11 to 8. This corresponds to the effect of the proposed methodology observed on the ecosystem level, i.e., the reported number of vulnerable dependencies becomes smaller due to filtering out (falsely reported) findings of development-only dependencies with known security vulnerabilities.

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 (6 out of 10). However, our methodology changes this view: only two vulnerabilities are introduced by transitive dependencies, while six are coming from direct dependencies. Moreover, one out of the six vulnerable direct dependencies is the within-project dependency of the simulated library. Additionally, Vuln4Real reports presence of seven dead dependencies.

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.

4.8 Threats to Validity

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 selection criterion may yield a sample not representative of what libraries are most

Algorithm 2: Effect of the proposed methodology on an individual library

```
input : Sample of analysed libraries $AnalysedLibs$, sets of deployed dependencies, grouped
        dependencies, and dead dependencies
output: Effect of the Vuln4Real methodology on a software library
1  $Vuln\_Paths \leftarrow \emptyset$ // Output according to the standard approach
2  $Vuln\_Paths\_filtered \leftarrow \emptyset$ // Output according to Vuln4Real
3  $Dead\_deps \leftarrow \emptyset$ ;
4  $i = 0$ ;
5  while $i < 100$ do
       // Random selection of 12 libraries
6      $l = 0$;
7      $Libs \leftarrow \emptyset$ ;
8      while $l < 12$ do
9          $lib \leftarrow \{Random(lib) | lib \in AnalysedLibs)\}$ // random selection of a library
10         $lib\_version \leftarrow \{Random(version) | version \in lib\}$ // random selection of a library
              version
11         $Libs \leftarrow Libs \cup lib\_version$ ;
12         $l = l + 1$;
13     end
       // Calculation of the results according to the standard approach
14     $Vuln\_Paths \leftarrow VulnPaths(Libs)$ ;
       // Calculation of the results according to the proposed methodology
15     $Vuln\_Paths\_filtered \leftarrow DeployedOnly(Vuln\_Paths)$ // Leave only deployed deps
16     $Vuln\_Paths\_filtered \leftarrow Group(Vuln\_Paths\_filtered)$ // Group coupled deps
17     $Dead\_deps \leftarrow Dead(Vuln\_Paths\_filtered)$ // Get dead deps
18     $i = i + 1$ ;
19 end
```

Table 4.9: Impact of the proposed methodology on the view of a single developer

Issues	SoA	Ours	
	CI	CI	$\Delta\mu$ SoA
No problem	[103, 123]	[106, 126]	+3
in dead deps	–	[6, 8]	+7
Problem			
total	[10, 12]	[7, 9]	-3
in your code	[0.3, 0.6]	[0.3, 0.6]	0
in your within-project deps	–	[1, 2]	+1
in your direct deps	[4, 5]	[4, 5]	+1
in your transitive deps	[5, 7]	[1, 2]	-4
in dead deps	–	[0.1, 0.3]	+0.2

Table 4.10: 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 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 do 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.
5	Identification of dependencies with known vulnerabilities			Generally, Vuln4Real is not affected by any errors at this step. However, the code-centric vulnerability mapping approach [84] used in this study may not be applied for some vulnerable library instances (for example, 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).

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[25]. 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.

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

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

Currently we considered only Maven based projects. We used Apache Maven, because it provides 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 Apache Maven, the methodology that we present in this chapter is language independent and it only relies on the availability of a dependency management mechanism, such as those provided for Java (Maven, Gradle), Javascript (npm), Python (pip), PHP (pear), and so forth.

We use Maven groupIds as an approximation for a project. This may potentially lead to an incorrect grouping of libraries because some projects may use the same cross-project groupIds, or conversely, different groupIds to identify their components. The former threat has a minimal impact, since the Maven naming convention of assigning different group identifiers to distinct projects is quite well established. We observed the latter case for test

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

or example libraries, e.g., org.apache.activemq has a subgroup org.apache.activemq.tooling. We considered two groupIds as equal if one of the two includes the other groupId (as in the activemq example). The projects that cannot be distinguished only by groupId could be distinguished using additional atributes, such as *Repository*, *ProjectID*, and others (which might be specific to certain language ecosystems).

Table 4.10 shows the potential impact of the threats to validity discussed above on each step of Vuln4Real.

4.9 Conclusions

In this chapter 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 200 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 changes the view on the situation regarding software dependencies:

- it removes alerts about 27% of direct and 21% of transitive dependencies with known vulnerabilities, that cannot be exploited;
- the proposed methodology reveals that as much as 21% of direct dependencies affected by known security vulnerabilities are within-project dependencies of the analysed libraries, hence, their developers should directly fix such vulnerabilities in the code of their projects.
- according to the SoA approach, it may seem that developers of the analysed libraries have direct control of only 37% of the vulnerable dependencies, while in reality they are responsible for fixing 80% of the deployed vulnerable dependencies.
- the results of the dependency study suggest that 13.2% of the total number of dependencies are not receiving updates, and therefore, may not have a fixed version if a security issue is discovered. Such dependencies should be used with caution, since mitigations of their bugs and bugs of their dependencies may be costly;

- the library simulation shows that Vuln4Real has a positive impact on the correct resolution of dependency analysis results of a single industrial library.

As future directions of our research we plan to identify a precise model for automatic identification of whether a certain library is dead and to complement the existing studies on the reasons why developers do not update dependencies with an investigation of developers' behavior with regard to security-related updates.

Chapter 5

Technical Debt and the Risk of Leverage in the Free Open Source Software Ecosystem

5.1 Introduction

The notion of technical debt [20] captures the cost of reworks caused by a quick inclusion of a functionality into a software project, instead of its proper (but often time-consuming) quality assessment. Such a concept is widely adopted by both commercial and Open Source Software (FOSS) that import functionality from third-party components as dependencies of their projects.

Dependency management systems, like Apache Maven[1] or Gradle[2], make the procedure of managing software dependencies fully automated, and therefore, extremely easy to use. However, as reported by several studies [49, 78], the black-box approach to dependency management also hides bugs and security vulnerabilities introduced via dependencies, and therefore, prevents developers from proper maintenance (i.e., timely updates) of their dependencies. This may end up in severe security incidents, as the one occurred with *Equifax*, where over 100'000 credit card records were leaked due to the vulnerability introduced into the project by an outdated software dependency[3].

As Allman [3] drew parallels between technical and monetary debts, one may relate dependencies in FOSS to the well-studied financial leverage instruments whose excessive

[1]`https://maven.apache.org/`

[2]`https://gradle.org/`

[3]`https://blogs.apache.org/foundation/entry/media-alert-the-apache-software`

use might cause a financial crisis (for example, in [52,92]).

In this chapter, we show the level of technical debt and leverage in the FOSS ecosystem on the sample of 200 most used FOSS Java Maven-based libraries by the proprietary projects of a multinational software company, which correspond to 10905 distinct library instances, when considering all the library versions. Our analysis of the code development process suggests the possibility of applying financial models to the ecosystem of FOSS projects to estimate the risks of technical bankruptcy, and therefore, avoid (or at least minimize) negative consequences.

5.2 Terminology

In this paper we rely on the terminology established among practitioners (e.g., the users of Apache Maven) and consolidated by Pashchenko et al. [78]:

- 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* is a library instance, some functionality of which is used by another library instance (the *dependent* library instance).
- A *dependency tree* 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 *direct* dependency is *directly* invoked from the dependent library instance, while a *transitive dependency* is connected to the root library instance of a dependency tree through a path with more than one edge.
- A *project* is a set of libraries developed and/or maintained together by a group of developers. Dependencies belonging to the same project of the dependent library instance are *own dependencies*, while library instances maintained by other projects are *third-party dependencies.*

Additionally, for each library instance in our sample we identify the following dimensions that characterize a library

- *Own size* (ℓ_{own}) as the number of lines of own code in the files of a library.

- *Dependency size* (ℓ_{dep}) as the sum of the lines of code of (third-party) direct and transitive dependencies of a library.
- *Total size* (ℓ_{total}) as the sum of own and dependency sizes of a library.
- *Leverage* (ℓ_{dep}/ℓ_{own}) as the relation between own and third parties' code.

5.3 Technical Debt in the FOSS ecosystem

Technical and monetary debts share several common concepts [3]:

- the debt has to be eventually paid;
- there exists some interest (an extra cost), when the debt is paid back;
- in case a debt cannot be turned back, there is a very high cost that exposes the borrower to bankruptcy, i.e., to abandon the affected activity.

According to the recent literature review [5], most of the academic studies consider the not well-written own code of a project as a source of a technical debt for a software project. However, current projects actively rely on software dependencies: the dependency size of a project may exceed its own size by as much as 400% [81].

Since developers are only responsible for fixing bugs and security vulnerabilities within own code of their project, (FOSS) dependencies seem to be very attractive: while software projects import functionality of their dependencies, the technical debt of this code stays with the developers of those projects. This may create a misleading feeling of a 'freebie' functionality both in terms of technical debt and development cost.

However, real-world software projects face the issue of keeping their dependencies up-to-date [19, 78]. This issue is due to the cost of upgrading project dependencies, that may introduce incompatible (breaking [36]) changes in its newer version. In this case developers of a dependent project have to refactor its own code and perform a thorough testing, i.e., spend some development effort (or, in other words, pay the cost of upgrading). If developers decide to postpone the upgrade of a dependency, it exposes the dependent projects to bugs and vulnerabilities fixed in the dependency from the time of the release of the adopted version[4]. Hence, the technical debt of the dependent project increases.

Even if a software project keeps its dependencies up-to-date, this still does not eliminate the technical debt completely. Since software developers want to have as less bugs

[4]In some cases, dependent projects keep using outdated components for more than 10 years [23].

Table 5.1: Descriptive statistics of the library sample
We considered the 200 most popular FOSS Java libraries used by a multinational software company in its own software, which resulted in 10905 distinct library versions.

	mean	median	st.dev	min	max	Q_1	Q_3
#lib versions	54.52	35.00	49.24	1.0	248	15.00	87.00
lib version size (KLoC)	47.20	17.92	75.41	2.0	496	5.73	44.43
#direct deps	4.26	2.00	6.80	0.0	51	0.00	6.00
dep size (KLoC)	262.80	164.30	314.70	2.0	2338	62.29	309.60

as possible, they logically select stable versions of software dependencies, i.e., released versions[5]. In this case the technical debt connected with the dependency code exists due to intermediate commits introduced by the dependency contributors from the time of the last release. Although for most cases, the relatively short size of a time interval between releases allows software developers to ignore such technical debt, some FOSS libraries have a release interval spanning over several years (e.g., the updated version 1.1.2 of the library commons-logging.commons-logging.1.1.1 was released after 6 years). Such technical debt cannot be safely ignored by developers of the dependent projects.

5.4 Data selection

For the study of the FOSS ecosystem, we have selected the top 200 FOSS Maven based libraries used by a set of over 500 Java projects (actual products and software developed by the company for internal use) developed at a large software manufacturer. The resulted set corresponds to 10905 library instances when considering all versions and includes such widely used libraries, like org.slf4j:slf4j-api and org.apache.httpcomponents:httpclient. Table 5.1 presents the descriptive statistics of the selected library sample.

We use the information directly available from the dependency management system. The FOSS libraries distributed via Apache Maven are published on the Maven Central Software Repository[6], that keeps all the publicly released versions of its libraries, i.e., their packages (for example, jar), project object model files (pom-files), and, often, some extra information, such as source code of a library or its documentation (JavaDoc). Maven also provides a Dependency plug-in, that allows us to retrieve a list of dependencies of a particular library instance. In this study, we use only direct dependencies, since we want to focus on the dependencies directly controlled by developers of the analyzed libraries.

[5]This option is also supported by dependency management systems.

[6]`http://central.maven.org/maven2/`

Algorithm 3: Extract own size of a library version

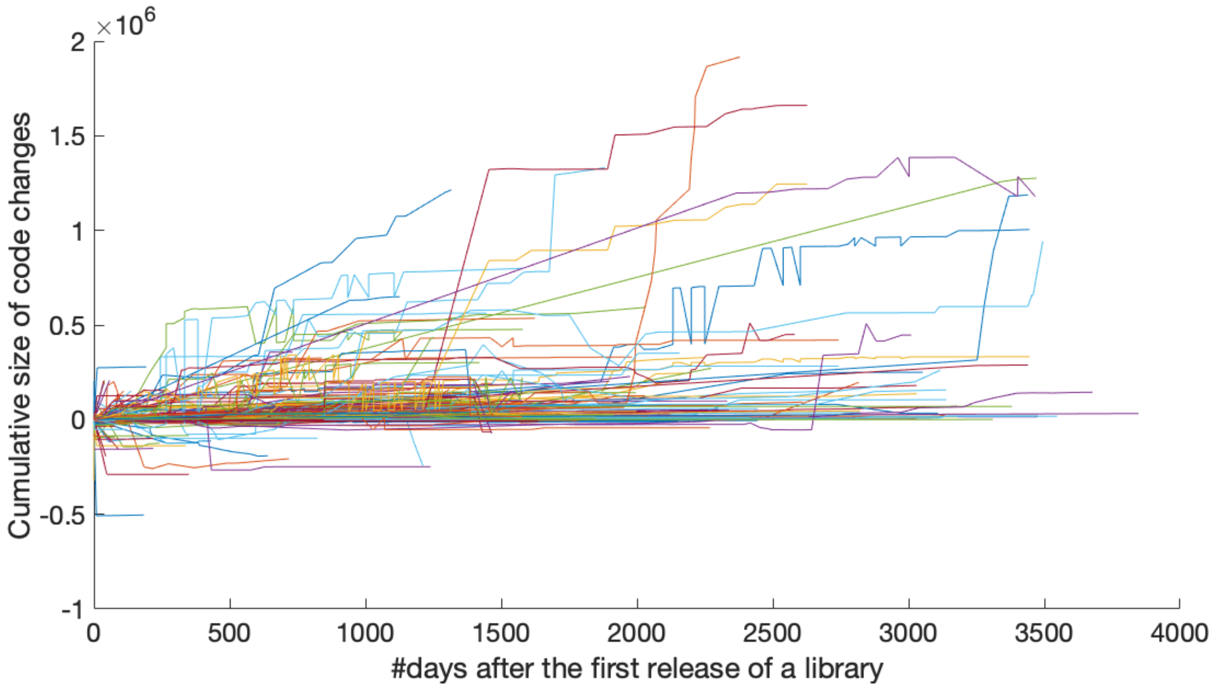
```
input  : A folder $dir$ with the source code of a library
output: The number of lines of code in a library $num\_locs$
$file\_list \leftarrow getAllFileNames(dir)$ // Get the list of all file names in the folder $dir$
$num\_locs \leftarrow 0$;
for $file | file \in file\_list$ do
    // Counting the number of lines in a file
    $lines \leftarrow readAllLines(file)$ // Load content of a file
    if $isCodeFile(file)$ then
        // Including only code containing files
        for $line \in lines$ do
            // Counting only lines that are not empty and are not comments
            if $line <> \emptyset$ and $isNotComment(line)$ then
                $num\_locs \leftarrow num\_locs + 1$;
            end
        end
    end
end
```

We follow Algorithm 3 to identify the own size of a library instance. To calculate the size of library dependencies, we recursively apply the Algorithm 3 to each dependency and then sum the resulting number of lines of code (LoCs). For several library instances (or their dependencies) there was no source code available, so we had to remove them from our analysis. The final list comprises 8464 library instances.

5.5 Code Changes in FOSS Ecosystem: Random Process

After an FOSS project is started and code changes (development cost) are required, developers can abandon development and maintenance if their (security) technical debt is too high, fixing the bugs is too complex and prospects do not look promising (or developers simply lose interest in such development). Mirroring a duality of what happens in the financial literature with dropping prices (or raising supply costs) [52], developers will suffer the equivalent of a technical bankruptcy when the required code changes will raise sufficiently. At technical bankruptcy, a library value becomes negligible[7]. De facto, the

[7]This is a simplification as the code can be reused in other projects. We do not introduce such option here to keep the mathematical model simple.



Each line in the figure shows the evolution of the cumulative size of code changes (both in own and dependency sizes) for a library in our sample from the time of the first release of this library. The majority of libraries have small deviations of total library size, while some libraries either increase or decrease their total size.

Figure 5.1: Cumulative size of code changes in time

library's ownership is left to the library's users who will incur the bankruptcy costs[8]. If development cost continues to raise, at some point the library will be abandoned, which would eventually expose the dependents of such libraries to a high number of bugs and security vulnerabilities.

To understand the nature of the FOSS code development process, we have studied how the total size of the libraries in our sample changes between releases.

Developers of some libraries maintain several versions of the library at the same time. For example, the developers of Apache Tomcat[9] project supported four versions (7.0.x, 8.0.x, 8.5.x, and 9.0.x) of org.apache.tomcat:tomcat-catalina library for the last three years (starting from March, 2016). Hence, we constructed chains of consecutive releases for libraries in our sample according to Algorithm 4.

Figure 5.1 presents the evolution of cumulative changes of total sizes for the FOSS

[8]For example, SAP would have to compensate damage to the affected customers, in case of a security breach due to a vulnerability exploitation in a third-party component of a company product. Hence, SAP has to provide appropriate maintenance (including upgrades and patch creation) of all the third-party components shipped along with its proprietary projects [23].

[9]`http://tomcat.apache.org/`

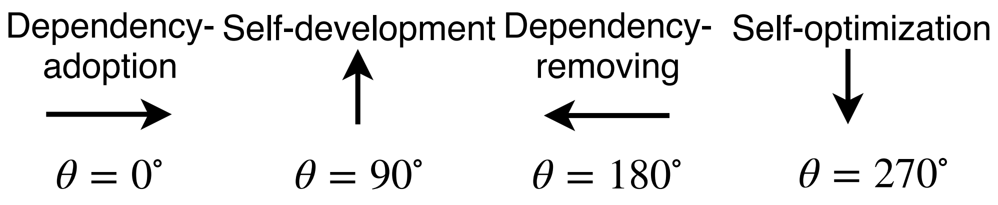
Algorithm 4: Extract consecutive release chains from a library set

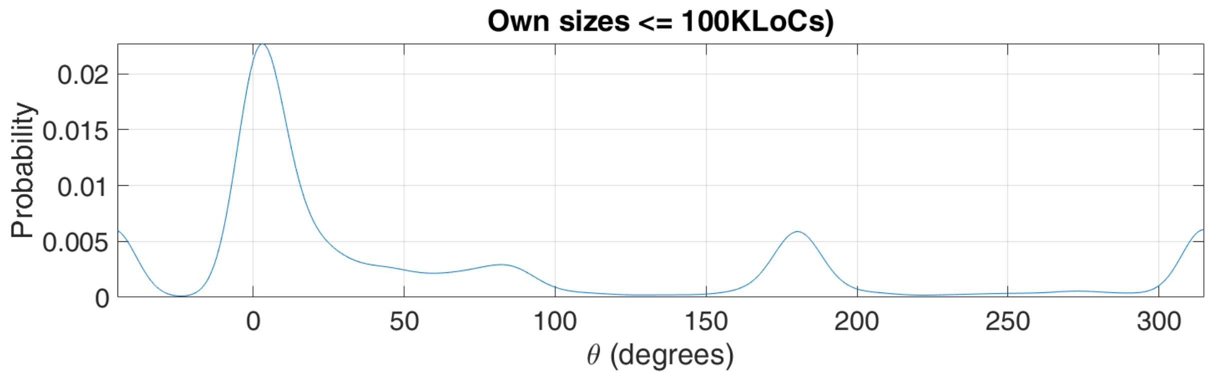
```
input  : A set of library names $libraries$
output: A set of lists of consecutive releases $releases$
$releases \leftarrow []$;
for $library \in libraries$ do
    $cur\_lib = library.getGA()$ // Use groupId:artifactId as identificator for the
        current library
    $releases[cur\_lib] \leftarrow []$;
    $branches \leftarrow []$// Prepare a list for storing library branches
    for $i \in range(0, len(library))$ do
        $lib\_version \leftarrow library[i]$ // get i-th library instance of a library
        if $releases[cur\_lib] == \emptyset$ then
            $releases[cur\_lib] \leftarrow [lib\_version]$;
        end
        $lib\_v\_id = cur\_lib + lib\_version[0]$ // Calculate id of a library version
        if $lib\_v\_id \in branches$ then
            $releases[lib\_v\_id].append(lib\_version)$;
        else
            if $lib\_version < releases[cur\_lib][-1]$ then
                $branches.append(lib\_v\_id)$;
                $releases[lib\_v\_id] = [lib\_version]$;
            else
                $releases[cur\_lib].append(lib\_version)$;
            end
        end
    end
end
```

libraries in our sample. Although several libraries significantly increased their total size (up to 2 million LoCs), most of the observed libraries tend to have small deviations of the total size (<20K LoCs). Also, developers of several libraries constantly consolidated code of their libraries, which resulted in the reduction of the total sizes of up to 50K LoCs.

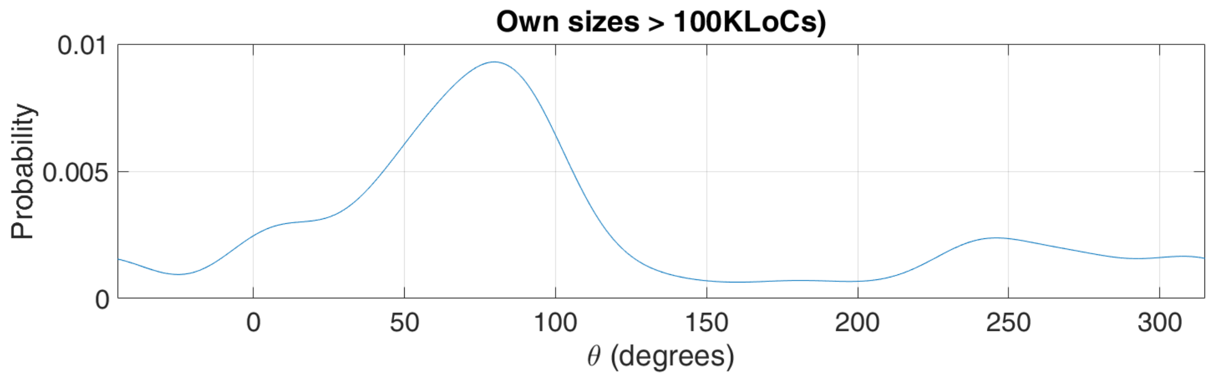
This observation suggests that applicability of financial models [8, 52, 62, 92, 96] that describe ecosystems, where participants exchange financial assets to leverage their benefits, is promising to be investigated for FOSS projects. These models suggest that if such leverage exceeds a certain threshold there might be a crisis for the whole ecosystem.

5.6 Changes in FOSS Ecosystem: Dependency Adoption





The picks at the KDE for the angles of library evolution plots suggest that developers of libraries with own code size smaller than 100 KLoCs tend to operate with their dependencies: they mostly adopt new dependencies and sometimes consolidate them.



The KDE of the library evolution vectors for the libraries bigger than 100 KLoCs suggest, that developers of such libraries tend to increase the size of own code of their libraries while importing some functionality from new dependencies (both adopting new dependencies and upgrading currently used ones).

Figure 5.2: Kernel density estimation plots for angles of library evolution vectors

To understand which libraries are most vulnerable to leverage risk, we propose to use change velocity vectors (see Section 4.2) to characterize changes between two consecutive releases of a library:

Definition 1

Table 5.2: Linear model fit to check the correlation between ρ and release time
Regression fit parameters for libraries with own_size ≤ 100 KLoCs: root mean squared error = 95.7; $R^2 = 0.15$; $R^2_{adjusted} = 0.15$. *Regression fit parameters for libraries with own_size > 100 KLoCs: root mean squared error = 29.5;* $R^2 = 0.24$; $R^2_{adjusted} = 0.24$.

$$time\_rel \sim \frac{\rho}{own\_size+dep\_size}$$

	estimate	stand.err.	t-stat	p-value
$own\_size \leq 100KLoCs$	26.51	2.50	10.62	$5.61*10^{-26}$
$own\_size > 100KLoCs$	44.38	6.80	6.52	$2.26*10^{-10}$

Change Velocity Vector $\langle \Delta\ell_{dep}, \Delta\ell_{own} \rangle$ *characterizes how a library changes between releases* r_0 *and* r_1 *with respect to its own and dependency sizes:*

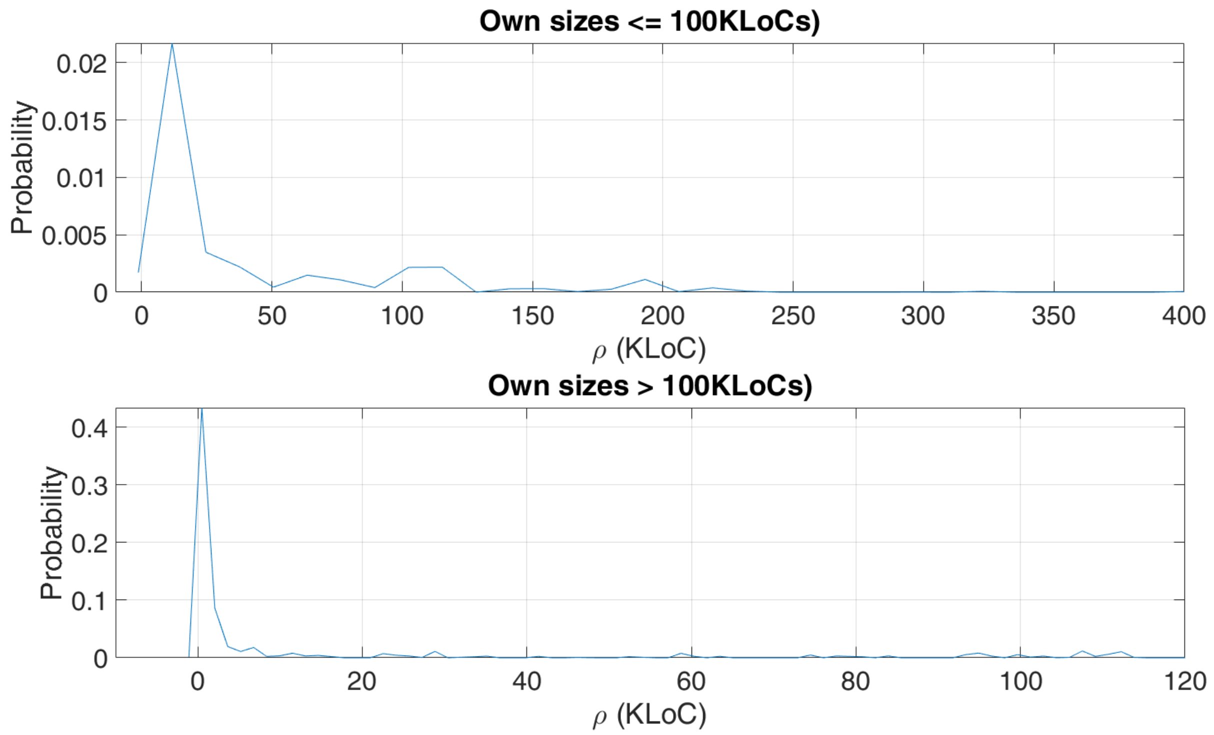
$$\langle \Delta\ell_{dep}, \Delta\ell_{own} \rangle = \langle \ell_{dep}(r_1) - \ell_{dep}(r_0), \ell_{own}(r_1) - \ell_{own}(r_0) \rangle \tag{5.1}$$

In particular, the library development behavior can be described using polar coordinates of the change velocity vector. I.e., considering different values of the change velocity angle θ (the angle between the change velocity vector and the x-axis), we identify four main directions of a library evolution as shown in the top of Figure 5.2:

- Dependency adoption ($\theta = 0^o$) - software developers increase the size of library dependencies, while not changing its own size: $\Delta\ell_{dep} > 0, \Delta\ell_{own} \to 0$
- Self-development ($\theta = 90^o$) - developers do not change the dependency size, while increasing its own size: $\Delta\ell_{dep} \to 0, \Delta\ell_{own} > 0$
- Dependency removing ($\theta = 180^o$) - software developers decrease the dependency size, while not changing its own size: $\Delta\ell_{dep} < 0, \Delta\ell_{own} \to 0$
- Self-optimization ($\theta = 270^o$) - developers do not change the dependency size, while decreasing its own size: $\Delta\ell_{dep} \to 0, \Delta\ell_{own} < 0$

Combination of these library evolution directions can describe every change of a library. For example, if both own and dependency sizes increase between two consecutive releases of a library ($\theta \in (0^o, 90^o)$), one may say, that its developers both adopt new dependencies and perform self-development of the library.

Intuitively, the radial distance ρ of the change velocity vector may serve as an indicator for a distance of a library from the point of becoming halted. To check the correlation between ρ and the time that developers need to release a library (rel_time), we have used the linear regression model. The direct fit of the linear model $rel\_time \sim \rho$ did not



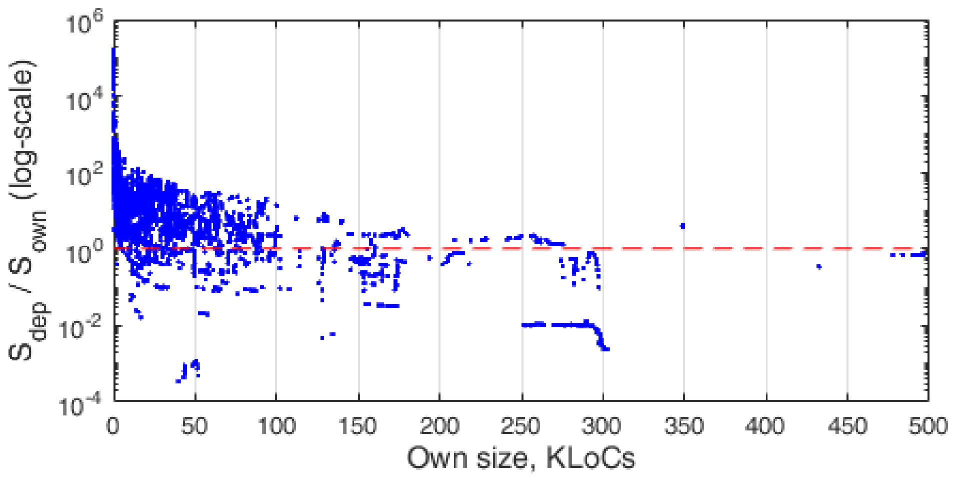
We observe, that libraries with smaller own size (<100 KLoCs) have almost 4 times longer library evolution vectors than the libraries with big code base (>100 KLoCs), and therefore, have a higher probability of becoming halted.

Figure 5.3: Kernel density estimation plots for lengths of library evolution vectors

show significant correlation between the variables ($p - value \gg 0.05$). However, we found that the release time of a library and ρ has correlation, when the total size of the library is considered: $rel\_time \sim \frac{\rho}{own\_size+dep\_size}$. Table 5.2 shows the fitting results of linear models for libraries with $own\_size \leq 100KLoCs$ and $own\_size > 100KLoCs$.

The R^2 of the linear model fit does not allow us to conclude that ρ could be used as a single predictor of rel_time. However, the results of the fit suggest that there is a positive relation between these variables (both estimates are positive): as ρ increases, so increases the probability for a library to become halted. Depending on such a library may have unpleasant consequences [78]: in case a vulnerability is discovered in a halted dependency, there would be no new version that fixes it, hence the developers of the dependent project would have to apply a costly mitigation strategy. Additionally, a halted library may transitively introduce security vulnerability into its dependent library as there would be no version of a library, that adopts the fixed version of the transitive dependency.

Using the change velocity vectors for the libraries in our sample, we observe the following:



The developers of relatively small software libraries (with own size smaller than 100 KLoCs) operate with a huge leverage: sometimes, the dependency size for such libraries is four orders of magnitude *greater than their own code size. In other words, they ship mostly somebody else's code. The leverage of large libraries (> 100 KLoCs) does not exceed their size of more than an order of magnitude (in few cases) and is typically much smaller than the size of their own code.*

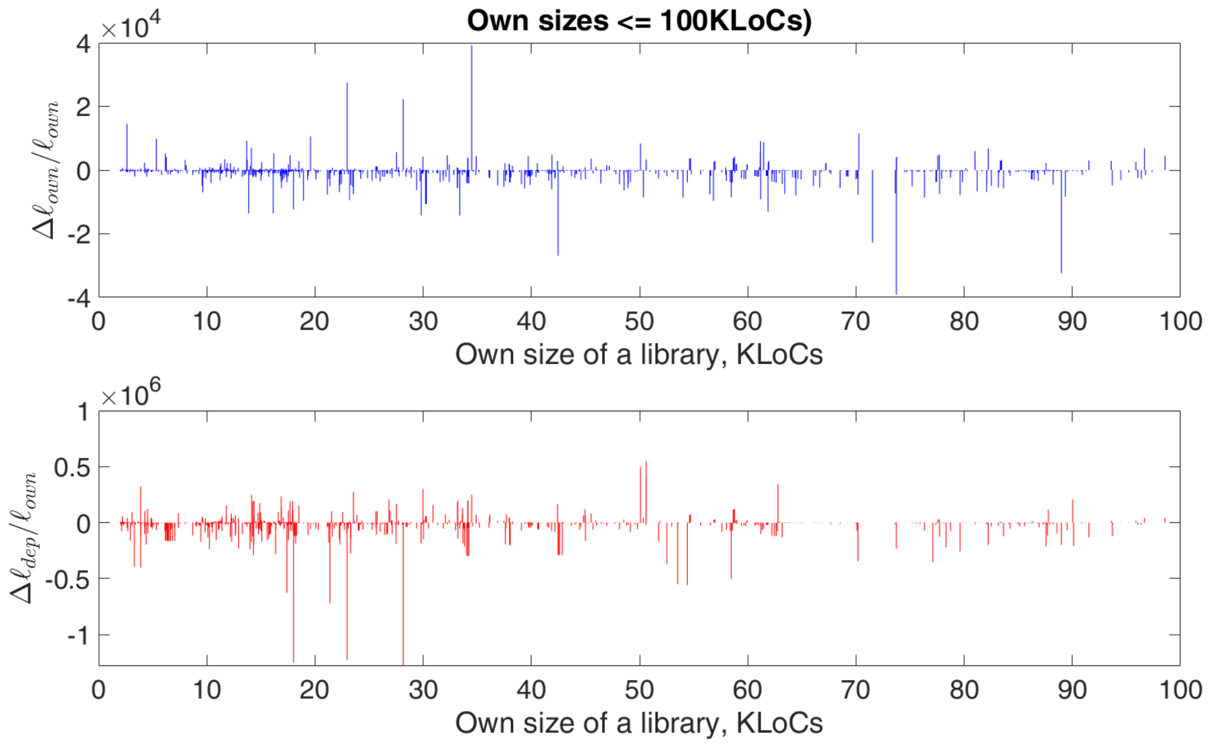
Figure 5.4: The risk of leverage in comparison to the own size of a library

- *Analysis of the change velocity angles θ* (Figure 5.2): the developers of small libraries ($\leq 100 KLoCs$) either adopt or remove software dependencies. The developers of big libraries ($> 100 KLoCs$) also change software dependencies, but they often modify own code of their libraries at the same time.

- *Analysis of the change velocity radial distances ρ* (Figure 5.3): small software libraries ($\leq 100 KLoCs$) have three times longer change velocity vectors comparing to the big libraries ($> 100 KLoCs$), and therefore, have higher probability of becoming halted.

5.7 Code Changes in FOSS Ecosystem: Leverage

As we observe from Table 5.1, FOSS developers widely adopt dependencies to reduce their development effort. Figure 5.4 shows the comparison of own and dependency sizes in the analyzed sample of software libraries. We observe that library instances use a large code base of dependencies, that may 10000 times exceed their own size. This especially applies to the software libraries with a relatively small code base (less than 100 KLoCs). The increase of the own size of a library leads to the decrease of the relative size of their dependencies. For the selected library sample, the libraries with the own size bigger than 150 KLoCs have 100 times smaller size of software dependencies.



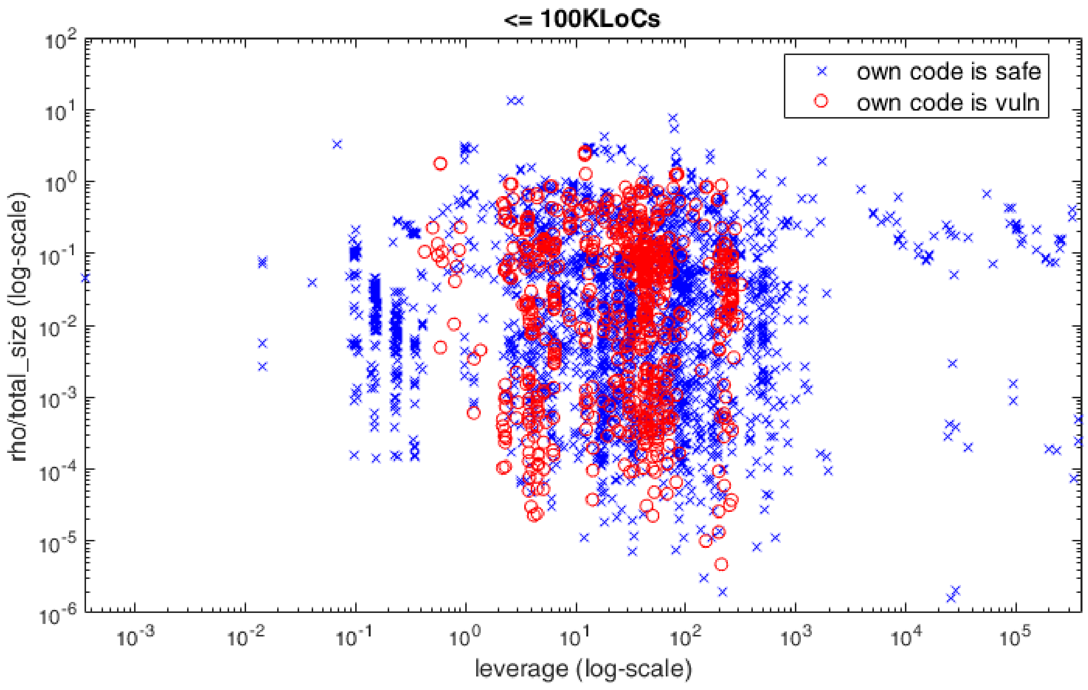
Velocity plots for the changes of size in a software library support our finding regarding the randomness and the riskiness of the FOSS code development process: for small libraries with less than 100KLoCs both own and dependency size changes may have various values from less than 1% up to 100% of a code base.

Figure 5.5: Velocity vector plots for the changes of size in a software library

Figure 5.5 shows the velocity plots of changes in own size of software libraries from our sample. We observe that library instances may have a variety of changes in code sizes from less than 1% of the own code base up to the size of a change equal to the own size of a library instance with an average of $\Delta\ell_{own} = 234.76$ LoCs ($st.dev = 3.02$ KLoCs) for libraries with own sizes smaller than 100 KLoCs and an average of $\Delta\ell_{own} = 598.91$ LoCs ($st.dev = 28.79$ KLoCs) for bigger libraries (>100 KLoCs).

Much larger changes (several magnitude larger than their own code base in several cases) are due to changes in their dependencies. We observe that huge changes in size of dependencies are typical for library instances with own size less than 100 KLoCs (average $\Delta\ell_{dep} = 505.96$ KLoCs, $st.dev = 62.06$ KLoCs), while bigger libraries do not have changes in software dependencies that exceed their own size (average $\Delta\ell_{dep} = 1.65$ KLoCs, $st.dev = 25.68$ KLoCs).

Since ρ and θ characterize the size and type of change of a library between two consecutive releases and leverage is a measure of the size of "borrowed" functionality, we are interested to know whether these metrics could be used to assess the quality of the own code of the library from the security point of view. As a criteria for such an assessment,



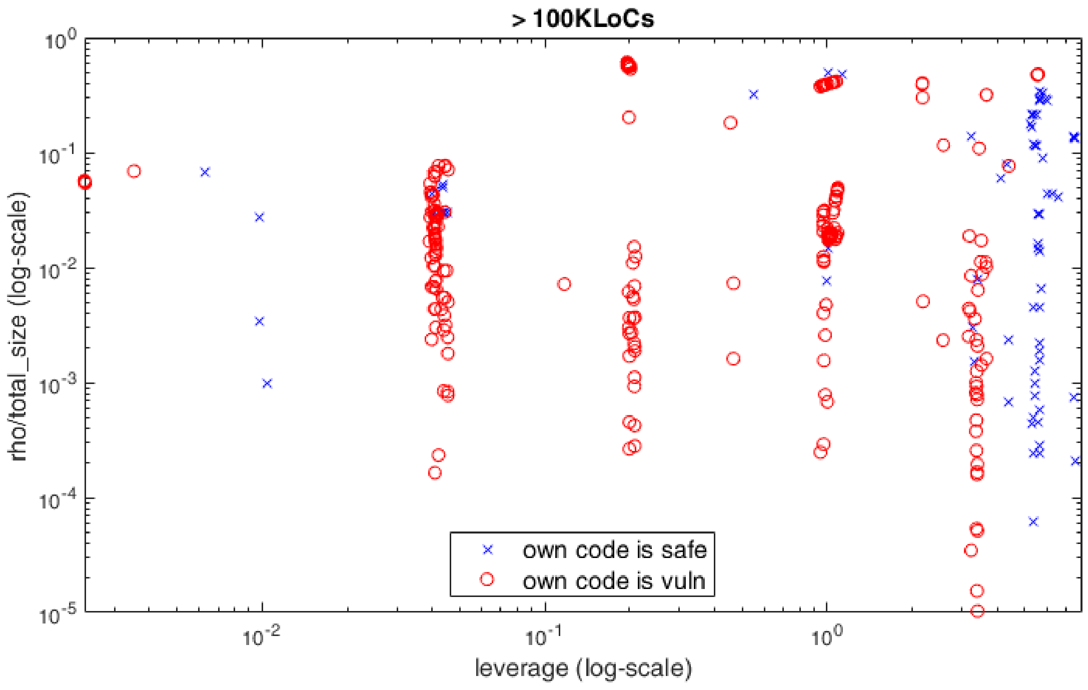


Figure 5.6: Leverage vs ρ

we use presence of a security vulnerability that affects the analysed library. However, a vulnerability may originate both from library dependencies and its own code. Since many software dependencies are affected by security vulnerabilities (see Chapter 4), they may bias the assessment of the quality of the own code of a library. Hence, we will consider only vulnerabilities that affect own code of analysed libraries for the quality assessment.

To identify whether own code of an analysed library is affected by security vulnerability, we used the publicly available data of the Snyk database[10] that is constantly updated and for July 2019 contains data about more than 2100 vulnerabilities in the libraries distributed via Apache Maven dependency management system. Each entry in the database contains the information about a security vulnerability; the library own code of which is affected by the vulnerability; and the range of affected library versions.

Figure 5.6 shows the leverage-ρ ratio for the libraries in our sample. We observe, that small libraries (own code $\leq$ 100 KLoCs) with both high (leverage $>$ 300) and low (leverage $<$ 1) leverage, as well as big libraries (own code $>$ 100) with high leverage (more than 3) have safe own code.

Figure 5.7 shows the relation between leverage and type of library changes (θ). We observe, that small libraries (own code $\leq$ 100 KLoCs) with $\theta \in [-45; 225]$ are more likely to be vulnerable. Such libraries either include/remove functionality from software dependencies or increase their own code base, and therefore, are likely to be under active development. In contract, there are less vulnerable small libraries with $\theta \in (225; 315)$. Such libraries decrease the size of their own code, and therefore, they are likely to review the already developed functionality instead of developing new features (i.e., to be mature). Visual analysis of leverage–θ relation plots for libraries with own code $>$ 100 KLoCs (Figure 5.8) suggests that in case of a big library there always exists a chance that its own code is affected by a security vulnerability.

5.8 Threats to Validity

The internal validity may be influenced by the fact that we have based the FOSS library selection for this study on their popularity from within a company. We surveyed the usage data of the selected sample from MVNRepository.com[11] and the number of users from BlackDuck Openhub[12]. Since both sources showed that libraries in our sample are also popular among the FOSS developers, we believe, the internal validity threat of our study

[10] `https://snyk.io/vuln`
[11] `https://mvnrepository.com/`
[12] `https://www.openhub.net/`

is minimal.

The generalization of the presented results may be exposed to an external validity threat since we considered only Maven based libraries. In this study, we aimed at creating awareness regarding the effects of technical debt and leverage within the software ecosystem. Since Maven has the largest share of users between the developers in the Java ecosystem[13], our results reflect the practice of the majority of Java developers. Also, our study is easy to replicate for other dependency management systems.

5.9 Conclusions and Future Work

In this chapter we have used the 200 industry-relevant FOSS libraries, that resulted in 10905 library instances when considered all library versions, to show the level of leverage and technical debt in the ecosystem of FOSS projects. Our analysis of the code development process suggests that application of financial models for the FOSS ecosystem may be promising, since such models may help estimation of risks of technical bankruptcy, i.e., a crisis situation when FOSS developers halt maintenance of software libraries and dependent projects loose interest in using FOSS dependencies.

As a future work, we plan to adjust the existing financial models to the FOSS ecosystem and use the data from the real-world FOSS software libraries to calculate the thresholds for software developers to work on new functionality for their libraries, to reduce their technical debt, or to quit the development process.

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

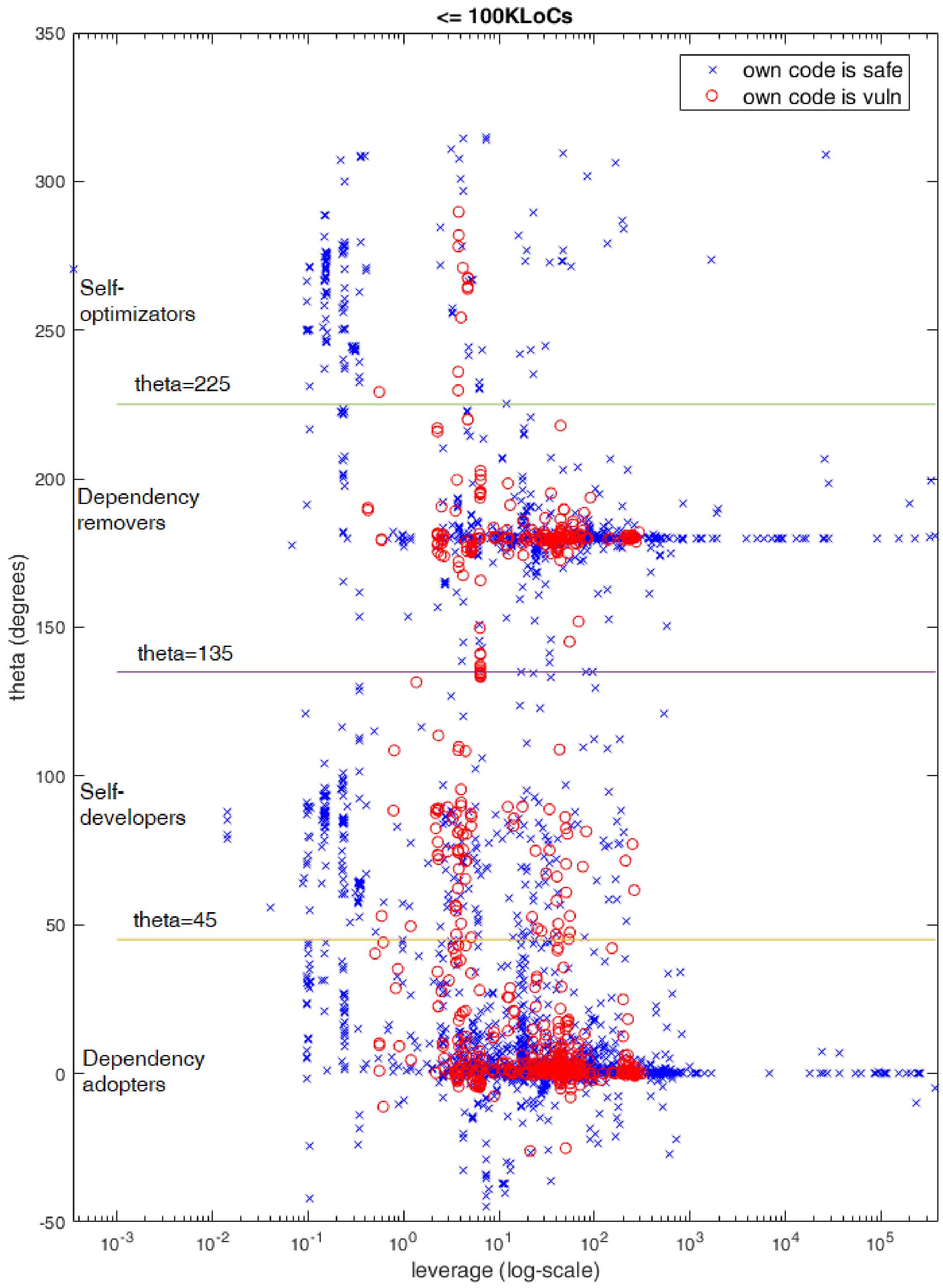


Figure 5.7: Leverage vs θ for libraries with own_size $\leq$ 100 KLoCs

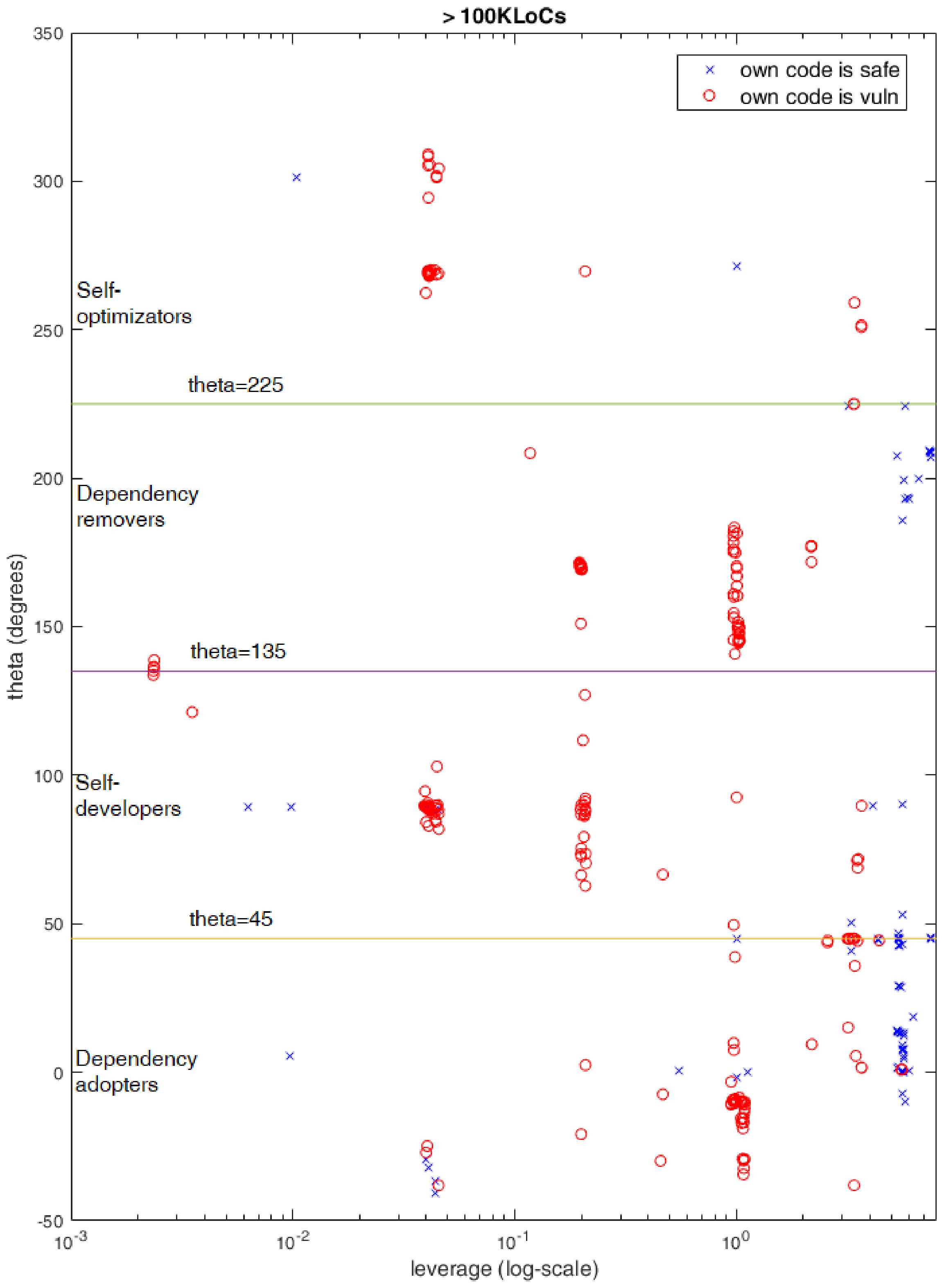


Figure 5.8: Leverage vs θ for libraries with own_size > 100 KLoCs

Chapter 6

Developers' perception of software dependencies

In this chapter we present the results of our qualitative study of the FOSS ecosystem. For the study we have run 15 semi-structured interviews with software developers of 4 most popular programming languages (C/C++, Java, JavaScript, Python) to understand their perception of software dependencies.

6.1 Introduction

Vulnerable dependencies are a known problem in the nowadays software ecosystems [49, 78]. Usually software developers introduce dependencies, when they need to add a new functionality into their projects. And then they have to maintain those dependencies as a part of the software development life cycle of their projects. However, current empirical studies provide some limited insights on the developers' strategies for selection and management of FOSS dependencies: they either report some developers' feedback as a support to the proposed metrics [19, 49] or focus the data collection within one software development company [9, 80]. Moreover, we do not find any empirical study, that reports the influence of security concerns on the developers' decisions.

Hence, we were interested in finding the decision making strategies that software developers follow for selecting and updating software dependencies, as well as the influence of security concerns on their decisions:

- RQ1: How do security concerns influence developer strategies for selecting software dependencies to include in their projects?

- RQ2: How do security concerns influence developer decisions of updating dependencies versus using the same versions?

Additionally, we have investigated the questions of whether software developers use any automatic tools to support the dependency management process:

- RQ3: Which methods and/or techniques do developers apply, while managing software dependencies?

- RQ4: How do developers mitigate bugs and vulnerabilities in dependencies that do not have fixed versions?

In this chapter we present the empirical study based on the semi-structured interviews with 15 professional software developers. The interviewees have at least 3 years of professional experience at various positions spanning from regular software developers to company CTOs. They come from 15 companies located in 7 different countries, such as Germany, Italy, The Netherlands, Russia, Slovenia, United Kingdom, and Vietnam.

We have found, that developers are more worried about security, rather than functionality issues introduced by software dependencies. They report, that dependency management requires many resources and the current dependency analysis tools are not usable, since the tools do not suit developers' work flow. The interviewed professionals consider code analysis tools to be similar to the dependency analysis tools. Two developers even expressed the desire to have the results of static analysis and dependency analysis tools to be combined into one report.

6.2 Qualitative Theory Construction

In this study we have adopted the Grounded Theory approach proposed by Glaser and Strauss [32, Ch.6]. Figure 6.1 summarizes the approach we used. It follows the principle of emergence [31], according to which data gain their relevance in the analysis through a systematic generation and iterative conceptualization of codes, concepts and categories. Data is analyzed, broken into manageable pieces (codes) and compared for similarities and differences. Similar concepts are grouped together under the same conceptual heading (category). Categories are developed in terms of their properties and dimensions and finally they provide the structure of the theory [97].

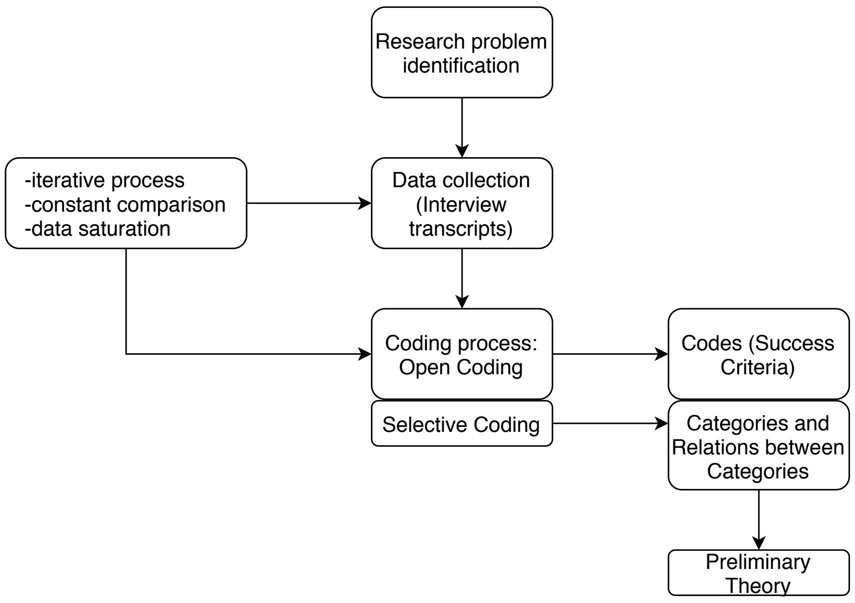

Research problem identification
-iterative process
-constant comparison
-data saturation
Data collection (Interview transcripts)
Coding process: Open Coding
Selective Coding
Codes (Success Criteria)
Categories and Relations between Categories
Preliminary Theory

Figure 6.1: Research Stream

6.2.1 Data Collection

The core part of this study is the information received from software developers on their perception of software dependencies. We could involve various techniques for collection of such knowledge, such as online surveys or controlled experiments. However, these methods imply forcing the investigator point of view, and therefore, may blur the real developers' opinions. Since in our study we were looking for the actual (and sometimes hidden) perceptions of industrial professionals on software dependencies, the method of semi-structured interviews suited best our goals [103].

Interviewee selection - failed attempt. First, to invite software developers for the interviews, we decided to reach developers of the most popular open-source Java projects. For this purpose we created a search on Github by the key word "Java" and selected the top 20 most starred projects. Then we used our tool for the dependency study (See §6.2.1 for details) to generate dependency analysis reports for those projects. We sent these reports to the main contributors (or owners) of the selected projects and asked them to provide their feedback on the reports as well as to dedicate some time for an interview. Unfortunately, this activity did not provide us with the sufficient number of interviewees, because there was only one response.

As suggested by B.Adams [2], the most likely reason for the fact, that software developers of the most popular Github projects ignored us, is that they may be overloaded by the various research studies. I.e., the developer selection approach we followed is very tempting for researchers. Hence, developers of popular research projects may receive many emails with different requests for participation in various scientific studies. So, they treat such kind of requests as spam and ignore it. In our case the request for the study contained also an attachment. And in the light of constantly increasing threat of ransomware, the such kind of emails looked very suspicious. So, we had to select a different strategy for hiring interviewees.

Table 6.1: Interviewees in our sample
The interviewed developers come from 15 countries and various company types, i.e., large enterprises (LE), small/medium enterprises (SME), programming language user groups (UG), and FOSS projects.

#	position	company	location	professional experience, years	primary languages
I1	CTO	SME	Germany	3+	Python, JavaScript
I2	Moderator	UG	Italy	10+	Java
I3	Developer	LE	Italy	10+	Java, JavaScript
I4	CEO	SME	Slovenia	7+	Python, JavaScript
I5	Developer	SME	The Netherlands	3+	Python
I6	Freelancer	SME	Russia	3+	Python, JavaScript
I7	Developer	SME	Germany	5+	Python, JavaScript
I8	Developer	LE	Russia	4+	Python, JavaScript
I9	CTO	SME	Italy	4+	JavaScript
I10	Developer	LE	Germany	10+	C/C++
I11	Developer	LE	Vietnam	5+	C/C++
I12	Developer	SME	Germany	4+	Java, Python
I13	Team leader	LE	Russia	10+	C#, JavaScript
I14	Developer	SME	Russia	4+	Java, C#
I15	FOSS Project Leader	FOSS project	UK	10+	Python, C/C++

Interviewee selection - the way we followed. As an alternative source for finding software developers we referred to local developers' communities, in particular Speck&Tech[1], Java Virtual Machine User Group in Trentino Alto Adige Südtirol[2], EIT Digital Alumni[3].

[1]`https://speckand.tech/`
[2]`http://www.jugtaas.org/`
[3]`https://alumni.eitdigital.eu/`

We used the public channels for these groups to post our call for interviews. Additionally we applied the snowball sampling approach [30] to increase the number of interviewees, by asking the respondents to also invite their friends for the interviews.

Table 6.1 presents the interviewees in our sample. For our study we recruited software developers, that are professionals in one of the following programming languages: C/C++, C#, Java, JavaScript, or Python. The interviewees have at least 3 years of professional working experience (with more than 10 years for 4 developers) and hold various positions, spanning from regular and senior developers to team leaders and CTOs. They work either as freelancers or come from both small companies and large corporations. In total we interviewed 15 software developers coming from 15 companies located in 7 countries. Hence, the subjects can be seen as a small, but representative selection of software developers that work with software dependencies.

Although all interviewed software developers use software dependencies in their projects, different development communities may influence the way how developers perceive software dependencies. One may argue that the interviewees in our sample have very different perceptions on dependency management question, since we have interviewed software developers of four programming languages (See Table 6.1) and there exist several dependency managers for one language: for example, Apache Maven, Gradle, Apache Ant, and Apache Ivy are the dependency managers used by Java projects. However, Java developers in our sample use Maven, which corresponds with the recent study by S.Maple [61] that reports Maven to have the largest share of users. JavaScript and Python have default dependency managers *npm* and *pip* consequently. C/C++ languages are a bit different, since they do not have a preferred dependency manager. Although there exist several alternatives, C/C++ developers mostly import third-party code by simply copying it into their projects. Hence, although there exist many various dependency management alternatives for each programming language, in practice one dependency management strategy is used by the majority of developers of each programming language.

6.2.2 Interview process

To collect primary data, we had interview sessions with software developers lasting approximately 30 minutes. We met personally the interviewees who reside in Trento, while we scheduled remote meetings via Skype[4] or Webex[5]. Although we did not specify any fixed structure for the interviews and allowed software developers to define the flow of the

[4]`https://www.skype.com/en/`

[5]`https://www.webex.com/`

discussion, all our interviews included the following parts:

Introduction - an interviewer describes the context and mentions some details regarding the background and motivation for the interviews. I.e., we used the development of the dependency analysis tool [78] as the background and the goal "to make it suitable for the needs of real-world developers" as the motivation. We have also specified that we are looking for the **personal experience of a software developer** on the topic of software dependencies;

Developer's self presentation - a developer describes us her professional experience and the context of her current activities;

Selection of new dependencies - a developer tells us about the process of selection and including new dependencies into her software projects;

Updating of software dependencies - an interviewee explains the motivations and insights of updating software dependencies in her projects. I.e., when it is the right time to update, how often she updates software dependencies, and if there is any routine or regulation regarding the dependency update process in her organisation;

Usage of some automatic tool for dependency analysis - a developer describes an automatic tool (if it is used), that facilitates dependency analysis process in her projects, and provides some general details about the integration of this tool into her development process;

Mitigation of issues introduced by software dependencies - an interviewee describes how she addresses issues, like bugs or security vulnerabilities, introduced by software dependencies;

Other general comments regarding dependency management - at this stage we ask for some general perceptions, comments or recommendations, that a developer may give regarding the process of dependency management and, in particular, about the security issues introduced by software dependencies.

6.3 Data analysis

6.3.1 Interview coding

The first phase of analysis (open coding) consists of collating the key point statements from each focus group transcript; a code summarizing the key points in a few words is assigned to each key point statement.

The author of this dissertation and one other PhD student from the security research group of the University of Trento independently followed the "iterative process" described by Saldaña [91] to code the transcribed interviews[6]. Then they looked together at the resulted codes and agreed on the common code structure, which was reviewed by the PhD students supervisor not involved into the preliminary coding process. The final set of codes comprises 25 items and is listed in Table 6.3. We have grouped the codes on their topics, which resulted in 6 categories: developers' perception, programming language, issues, process, dependencies, and general topic.

Table 6.3: List of codes

category	code	code meaning
developers' perception	it was a problem	Developer perceives something as a problem
	not a problem	Developer assumes something not to be a problem
	recommendation	Developer extremely likes something and recommends this for others
programming language	C/C++	Developer talks about C or C++
	Java	Developer talks about Java
	JavaScript	Developer talks about JavaScript
	Python	Developer talks about Python
issues	broken	A dependency breaks the build of software project or causes compatibility issues
	bugs	Unexpected software behavior, that causes crashes, failures, errors, etc.
	lack of resource	Lack of resource (human, time, information, cost)
	license issues	Issues related to license of software (e.g., license compatibility)
	no fix available	There is no fixed version of a dependency or there is even no guideline on how to fix bug/vulnerability
process	automation	Substitution of a manual task execution with a script or a tool
	code analysis tool	An automated tool, that checks for the issues in the code of a software project (e.g., static analysis tools)
	company integration process	Description of a company strategy for releasing/development/continuous integration process

[6]We used *Atlas.ti* software to perform interview coding.

Table 6.3: List of codes

category	code	code meaning
	dependency tool	An automated tool, that facilitates dependency management
	manual	A task performance without applying automated tools
dependencies	dependency maintenance	Fixing bugs/vulnerabilities via changing code of a software project and/or its dependencies
	dependency management	Including/Analyzing/Removing/Testing/Updating software dependencies and/or manipulating with their versions via changing configurations of the project
	direct dependencies	Dependencies directly referenced from within the project
	seeking for information	Sources of information for dependency selection, vulnerable dependency identification, dependency internals, technical explanation about how to fix dependency problems, dependency characteristics (e.g., its reputation), supported by community of developers (e.g., contributors in Github), official software websites.
	transitive dependencies	Dependencies indirectly referenced from within the project
general topic	functionality	Developers are talking about adding/deleting/improving some functionality
	requirements	Requirements of users of a software project
	security	Developers are talking about conditions or tasks related to information security

6.3.2 Evidence from the Interviews

We start analysis of the interview data by examining the popularity of the topics touched by developers during the interviews. We observe, that the developers in our sample mentioned more often (hence, more interested in) the following topics: *dependency man-*

agement (506 mentions), *security* (360 mentions), *it was a problem* (339 mentions), and *bugs* (267 mentions). At the same time, the least important topics for them are *direct dependencies* (23 mentions), *requirements* and *license issues* (38 mentions each), *code analysis tools* (47 mentions), and *transitive dependencies* (66 mentions). This suggests us, that software developers are aware about the possible issues (including security bugs) that dependencies may introduce into their projects. On the other hand, the small number of occurrences of the codes, like direct or transitive dependencies, may signalise that developers do not consider all the details of the dependency management process:

> "That's our not so well secure approach to the problem. If there's something we really know to be broken – we fix it. Otherwise it's kind of left to itself."

Then we analyse the codes, that are mentioned together. For this purpose we have extracted the co-occurrence table of the interview codes [48]: each column and row of the table corresponds to an interview code, while each cell contains the number of code co-occurrences. To identify the cells with significantly high number of co-occurrences, we calculated the mean and standard deviation for the code co-occurrences ($\mu = 5.95$, $\sigma = 8.41$) in the table and highlighted with red the cells, where the number exceeds μ by at least the value of σ (highlighted if the value in the cell exceeds 14.36).

RQ1: How do security concerns influence developer strategies for selecting software dependencies to include in their projects?

Table 6.4 shows the fragment of the co-occurrence table for the developers' perception category.

Table 6.4: A fragment of the co-occurrence table for the "developers' perception" category.

	C/C++	Java	JS	Python	broken	bugs	lack of resource	dep maintenance	dep mgmnt	seeking for info	trans deps	functionality	security
it was a problem	15	18	27	39	12	23	17	8	62	10	11	15	30
not a problem	2	16	22	30	4	20	3	6	34	5	1	2	29
recommendation	3	4	6	4	0	3	1	2	11	3	4	7	6

The developers consider dependency management process as problematic: they have mentioned issues during the dependency management process twice more often (62 times), rather than concluding, that it was not a problem (34 times). This observation remains stable, when considering developers of each programming language we have studied.

> "There are a lot of dependencies in Java, especially the transitive dependencies. And this is the principal problem – because in Java there's a building system

> used, like Maven or Gradle. And they bring a lot of transitive dependencies into your project. In principle, you may have used just 2 dependencies on the first level, but you end up with 20, 30 or even more indirect dependencies."

The developers told us their perception about the security topic 2.7 times more often rather than about functionality (65 vs 24 mentions), which means, they are aware about security vulnerabilities that may be introduced by dependencies. Considering the developers' perception regarding the *security* code, the developers in our sample have almost equally treated security as a problem and not as a problem (30 and 29 times respectively). Similar proportion of mentions we observe regarding the code *bugs* (23 and 20 times respectively).

An unexpected issue regarding software dependencies, that we discovered during the interviews, was the license issues. We observe, that this issue is important for C/C++ and Java developers of medium and big companies. Surprisingly, JavaScript and Python developers do not have any concerns regarding the licensing issues of dependencies.

> "From our perspective – the dependency issues are not only the security issues, but also license issues. Because it's not only for the vulnerabilities of the code, but also the license issues. In Enterprise, if you sell some software, and inside your software you have an restricted license, like GPLv3, you could have a lot of legal issues, because of the owner of the library may discover that and you may have a lot of legal problems."

While selecting new dependencies for their software projects, developers look at various sources of information and take into account several different metrics, such as (i) how well this dependency is known and used; (ii) how fast library maintainers fix issues; (iii) understandability of the code of the library:

> "First, I am looking for something trustable, that everybody knows."
>
> "From the security side, the first question for the code – when I see it, I need to understand it. Because if I do not understand it (and, unfortunately, majority of libraries are written in this way). So the question – if I understand the logic. If it is a kind of a "spaghetti code" – this decreases chances for a library to be used."
>
> "If I see some bug, that, I think, the developers should have been fixed, but the bug still remains open for one year, or developers do not respond – this means, developers do not support the project. And I would have to face the problems alone."

RQ2: How do security concerns influence developer decisions of updating dependencies versus using the same versions?

The fragment of the co-occurrence table for the issues category is shown in Figure 6.5.

Table 6.5: A fragment of the co-occurrence table for the "issues" category.

	dep maintenance	dep mgmnt	direct deps	seeking for info	trans deps	functionality	requirements	security
broken	2	15	0	0	2	3	0	3
bugs	8	35	2	15	3	8	3	34
lack of resource	6	24	1	0	0	1	0	15
license issues	0	4	0	3	1	5	0	1
no fix available	9	3	0	2	0	2	0	9

The most important and discussed issue for the developers in our sample were *bugs*. Most of mentions of this code we received from the Python developers (29 mentions):

> "When we see that a new library has a bug. On the example of our service, we contact developers, say that there is a bug. They fix it. And we update version."

When the developers spoke about bugs, quite often they discussed security bugs or vulnerabilities (34 mentions):

> "Most of the time the most important reason for that is security. Software sometime some kind of vulnerability and they have a security fix in a later version. When the customer, they are saying - wow, this software we are using, they have something like a security fix. They do not care what the security fix will be. It will be back to the developer to increase the software version to fix that."

We observe, that when software is developed for internal usage, developers are more concerned about functionality, rather than security vulnerabilities:

> "Honestly, since we are developing the software for the internal usage, there is no access for users from outside. So, for the security, for the bugs related to security, i.e., unauthorized access or something else, we do not have serious requirements. We do not pay so much attention. We pay attention more to the stability of operations and solving business tasks."

Another very important problem for software developers was the lack of resources. The interview analysis shows, that developers experience it, when they are discussing dependency management or security topics:

> "...Because our project is huge. We tried once, and 1000 tests became down. To fix it - we just do not have time for that."

Often, developers mention, that they do not want to upgrade dependencies in their projects due to the possibility of breaking their projects:

> "Developers usually underline in a separate item breaking changes. Some changes, that are breaking accustomed way of working with this library. If they are present, then we do not install it. Because the simple update should not force us to change our code significantly."

RQ3: Which methods and/or techniques do developers apply, while managing software dependencies?

Figure 6.6 shows the fragment of the co-occurrence table for the 'process' category.

Table 6.6: A fragment of the co-occurrence table for the "process" category.

	automation	code analysis tool	company integration process	dep tool	manual	dep maintenance	dep mgmnt	seeking for info	functionality	requirements	security
automation	0	6	9	10	3	1	15	5	0	0	14
code analysis tool	6	0	6	3	0	0	5	2	0	0	9
company integration process	9	6	0	7	3	1	23	8	6	4	19
dep tool	10	3	7	0	5	3	23	7	2	0	13
manual	3	0	3	2	0	5	10	4	0	0	6

Describing process, the developers paid a lot of attention to dependency management and security of their *company integration process*:

> "How it works. We have, let's say, we don't have any regulations. We have a rule. Well, not a rule, but general understanding, that packets should be up-to-date. And this is how we work."

For the big companies there is a division between third-party libraries and external FOSS libraries. Usually, the developers of big companies first try to use the internal libraries:

> "...when the company is big there are many teams and each of them may develop some functionality, which you can then just connect."

The interviewees have mentioned, that they tried to use some dependency analysis tools. Although sometimes developers mentioned and even recommended us to check some dependency analysis tools, very often, those tools did not suit their needs:

> "I just don't use them, because they just do not fit my work flow. It does not just fit what I need to be doing."

Hence, we find several cases, when developers complained about the automation part in respect to both collection of the information regarding software dependencies and creation of false alerts from the automatic dependency analysis tools:

> "I had one and it tend to spamming and I turned it off. For example, reporting minor vulnerabilities, so I was kind of annoyed of them."

Regarding the automatic tools, we discovered an interesting correlation: some developers considered code analysis tools (i.e., static or dynamic analysis tools) to be similar to dependency analysis tools.

> "Python has, at least what I know, two libraries. First is bandit. It follows for the vulnerabilities in your own project. . . . Second library is called Safety. It checks for dependencies."

Several developers even gave us a surprising recommendation to augment the reports from a code analysis tool (for example, SonarQube) with alerts generated by a code analysis tool:

> "Maybe it's possible to plug the results of dependency analysis to SonarQube? So we can use it later on in our continuous integration and we would be able to do continuous code analysis. It would be cool to have this."

RQ4: How do developers mitigate bugs and vulnerabilities in dependencies that do not have fixed versions?

The co-occurrence table for the 'dependencies' category is shown in Figure 6.7.

Table 6.7: A fragment of the co-occurrence table for the "dependencies" category.

	it was a problem	not a problem	recommendation	broken	bugs	lack of resource	automation	company integration process	dep tool	functionality	requirements	security
dep maintenace	8	6	2	2	8	6	1	1	3	2	0	6
dep mgmnt	62	34	11	15	35	24	15	23	23	21	9	47
direct deps	1	0	0	0	2	1	0	0	0	0	0	3
seeking for info	10	5	3	0	15	0	5	8	7	14	2	19
trans deps	11	1	4	2	3	0	2	0	1	3	0	5

Besides, the widely mentioned topic of dependency management, several experienced developers mentioned, that they actually participate in the dependency maintenance process, i.e., they actually contribute to the dependencies of their software projects:

> "So if we can fix it by ourselves, then we, for sure, fix it and publish it. This is also one of our rules. An unspoken one."

However, the developers of small companies do not feel themselves to be skilled enough to fix security vulnerabilities:

> "It's a crucial fix. If you do not do it properly, then people think that it's fixed and maybe you introduce some other bugs and stuff like that. I'd say that it's really the last resort if you have to."

6.4 Analysis summary

Selection of software dependencies:

- While selecting a new dependency, software developers are more concerned about security, rather than functionality. However, they have mentioned almost equal number of times potential security issues and functionality bugs as a problem and not as a problem.
- The developers of each programming languages considered in this study (C/C++, Java, JavaScript, or Python) treat the dependency management process as a problem due to[7] (i) the potentially introduced security vulnerabilities and bugs, (ii) lack of resource for proper dependency management, (iii) high number of transitive dependencies, and the (iv) lack of proper dependency information sources.
- When talking about issues, the C/C++ and Java developers of medium and big companies concern about licensing issues. They do not select software dependencies that violate license policies of their companies.
- When selecting new dependencies to include into software projects, developers (i) consider how well this dependency is known and used; (ii) check the repository of the software library; (iii) check how fast library maintainers fix issues; (iv) check the release notes of the library; (v) check if the code of the library is understandable.

[7]We report the reasons in decreasing order according to the number of mentions.

Updating software dependencies:

- Software developers in our sample count *bugs* (including security bugs and vulnerabilities) to be the most important issue that motivates them to update dependencies.
- However, security concerns are not important for software developers in case they are developing software for internal purposes.
- The *breaking changes* in the dependencies and *lack of resources* were the main reason for developers to postpone the update of their dependencies.

Automatic tools for dependency management:

- The developers count currently existing dependency analysis tools being not satisfactory for their needs, although they are trying to use them.
- Software developers perceive code and dependency analysis tools to be similar. Some of the developers shared with us their will to have code and dependency analysis results to be combined together in one report.

Mitigation of bugs and vulnerabilities that do not have fixes:

- Experienced software developers occasionally contribute to their dependencies by fixing bugs or vulnerabilities, however, the less skilled developers do not feel themselves comfortable enough for doing this.

6.5 Threats to Validity

The internal validity of our study may be influenced by the fact, that *we recruited software developers for our study without using any material rewards, only on the basis of their interest to the topic.* In our study we aimed to receive information from professional industrial specialist, who have good salaries and solid social allowance. Hence, we could not think of any better reward for them, than a possibility to improve the development practice by sharing their experience and to tell us their opinions on their problems. Moreover, very often, the developers were motivated by the fact, that we had already had a prototype of a tool, that we could use to produce some dependency analysis reports for their projects. We believe, that this strategy allowed us to receive the especially valuable feedback from the field specialists, who have appropriate level of knowledge of the topic.

The generalization of the results of our study may be affected by the fact, that *we considered developers of only four programming languages: C/C++, Java, JavaScript,* *and Python.* However, according to both the Tiobe index[8], which combines data about search queries from 25 most popular websites of Alexa and the PYPL index[9], which uses Google search queries, these programming languages are the most popular ones. Hence, we believe, that the experience of developers from our sample can give an intuition of the developers' perception of software dependencies.

6.6 Conclusions

In this chapter we have presented the results of the qualitative empirical study of the developers' perception of software dependencies. For the study we have run 15 semi-structured interviews, each lasted 30', with software developers from 15 companies located in 7 different countries.

To analyse the interviews, the author of this thesis and the other PhD student from the security research group of the University of Trento independently performed the "coding" of the interviews, and then they agreed on the resulted set of codes. The final set of codes was reviewed and approved by their supervisor, not involved into the coding process. This activity resulted in 25 codes, that were grouped into 6 categories: developers' perception, programming language, issues, process, dependencies,and general topic.

We summarise the main findings of our qualitative study as follows:

- The developers are more concerned about security rather than functionality issues, when selecting and managing software dependencies. On the other hand, developers, that create projects only for internal use, are not concern about security issues.
- Bugs (including security bugs and vulnerabilities) in software dependencies are the biggest motivator for software developers to update dependencies of their projects. While breaking changes and lack of resources are the top reasons to postpone the adoption of the fixed versions.
- The developers count currently existing dependency analysis tools being not satisfactory for their needs, although they are trying to use them.

[8]`http://www.tiobe.com/tiobe-index/`
[9]http://pypl.github.io/PYPL.html

Chapter 7

Conclusions and Future Work

This dissertation has shown the way to facilitate the security assessment process of both own code and third-party components of software projects in industrial context.

For the part connected with the own code we propose Delta-Bench, an automatic approach for benchmarking static analysis security testing tools. Delta-Bench allows software developers to select the SAST tool, that works best for their project. To achieve this purpose, the proposed benchmarking approach uses the set of historical vulnerability fixes (possibly, extracted from the development history of the project under analysis) to extract the ground-truth for tool evaluation (Chapter 3).

Regarding the security of third-party components used by software projects, we designed a methodology for extracting third-party components used within the project, identifying dependencies affected by known vulnerabilities, and post-processing the results in order to report only the findings relevant to the developers of the analysed projects (Chapter 4). In this way, we use the information of the known vulnerabilities that affect FOSS components, to provide the actionable reports for software development companies to plan the correct allocation of resources for fixing issues coming from the dependencies of their projects.

The dependency analysis methodology raised the interest of SAP, our industrial partner. To facilitate the dependency analysis, we have developed a tool driven by the ideas of the proposed methodology. We plan to structure the code of the tool and release it as an open source project. Research-wise, we also continue our collaboration to extend our ideas from identification of security issues to their automatic mitigation.

Our empirical study of the FOSS ecosystem allowed us to estimate the levels of technical debt and leverage in the FOSS projects. Our preliminary analysis of the code development process suggests us the possibility to apply the current financial models to estimate the risks (and predict) of technical bankruptcy - a situation, when FOSS developers loose interest in maintaining their software libraries, which may eventually lead projects, that depend on such libraries, to become exposed to bugs and security vulnerabilities.

As a future work, we may try to adjust the currently existing financial models to estimate the thresholds for developers to develop new functionality in their projects, to update dependencies of their projects (i.e., reduce the technical debt), or to abandon maintenance of their software libraries.

To validate the industrial relevance of the proposed approaches, we have performed an empirical study, where we interviewed software developers on their perception of software dependencies. The results of this study have demonstrated, that developers consider code and dependency analysis processes to be related. Two developers even told us, that they would like to see the results of the code and dependency analysis to be joined within one report. We have found, that the interviewed developers are aware of the dependency analysis tools, however, these tools do not always suit their work flow.

We have presented the results from 15 highly experienced software developers that work in 15 companies in 7 different countries, which is already a fair number to make some conclusions. However, we plan to include the opinions of at least 10 more developers to make the results of the study even more credible. We have recruited additional developers, while presenting our research at the Java User Group (JUG) meeting in Bolzano as well as at the JUG meeting and the Speck&Tech event in Trento in February, 2019.

Bibliography

[1] National Security Agency Center for Assured Software (NSA CAS). Juliet Test Suite v1.2 for Java user guide, 2012.

[2] B. Adams. Developers of popular software projects are overloaded by the requests from academic researchers. Suggested during a personal communication with the authors at ESEM'2018, 2018.

[3] Eric Allman. Managing technical debt. *Commun. ACM*, 55(5), 2012.

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

[5] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. The financial aspect of managing technical debt: A systematic literature review. *Inf. and Softw. Tech. Journ.*, 64:52–73, 2015.

[6] Nuno Antunes and Marco Vieira. Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples. *Trans. Serv. Comput.*, 8(2):269–283, 2015.

[7] Muhammad Asaduzzamad, Roy K. Chanchal, Kevin A. Schneider, and Massimiliano Di Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. *Proc. of ICSME'13*, 2013.

[8] Avner Bar-Ilan and William C Strange. The timing and intensity of investment. *Journ. of Macroeconomics*, 21(1), 1999.

[9] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proc. of ICSE'10*, volume 1, pages 125–134. IEEE, 2010.

[10] Paul E. Black and Athos Ribeiro. SATE V Ockham sound analysis criteria. Technical report, NIST SP, 2016.

[11] Andrea Bonaccorsi and Cristina Rossi. Why open source software can succeed. *RP*, 32(7), 2003.

[12] Vasilis Boucharas, Slinger Jansen, and Sjaak Brinkkemper. Formalizing software ecosystem modeling. In *In Proc. of IWOCE'09*, pages 41–50, New York, NY, USA, 2009. ACM.

[13] Robert Goodell Brown. *Statistical forecasting for inventory control.* McGraw/Hill, 1959.

[14] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *Proc. of SANER'15*, pages 516–519. IEEE, 2015.

[15] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, HKUST-CS98-01, Hong Kong University of Science and Technology, 1998.

[16] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proc. of ASE'16*, 2016.

[17] Jailton Coelho and Marco Tulio Valente. Why modern open source projects fail. In *Proc. of FSE'17*, pages 186–196. ACM, 2017.

[18] Jailton Coelho, Marco Tulio Valente, Luciana L Silva, and Emad Shihab. Identifying unmaintained projects in github. In *Proc. of ESEM'18*, page 15. ACM, 2018.

[19] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *Proc. of ICSE'15*, ICSE '15, pages 109–118, Piscataway, NJ, USA, 2015. IEEE Press.

[20] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2), 1993.

[21] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *Proc. of USENIX'14*, 2014.

[22] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proc. of ASE'07*, pages 433–436. ACM, 2007.

[23] Stanislav Dashevskyi, Achim D Brucker, and Fabio Massacci. On the effort for security maintenance of free and open source components. *Proc. of WEIS'18*, 2018.

[24] Stanislav Dashevskyi, Achim D Brucker, and Fabio Massacci. A screening test for disclosed vulnerabilities in foss components. *TSE*, 2018.

[25] Aurelien Delaitre, Vadim Okun, and Erin Fong. Of massive static analysis data. In *Proc. of SERE'13*, 2013.

[26] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Emp. Soft. Eng. Journ.*, 10(4):405–435, 2005.

[27] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Proc. of SSP'16*, 2016.

[28] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *ENTCS*, 216:5–21, 2008.

[29] Durham Goode. Scaling mercurial at facebook. *Online: https://code.fb.com/core-data/scaling-mercurial-at-facebook/*, 2014.

[30] Leo A Goodman. Snowball sampling. *AOMS*, pages 148–170, 1961.

[31] Robert Wayne Gregory, Mark Keil, Jan Muntermann, and Magnus Mähring. Paradoxes and the nature of ambidexterity in it transformation programs. *ISR*, 26(1):57–80, 2015.

[32] Greg Guest, Kathleen M MacQueen, and Emily E Namey. *Applied thematic analysis.* sage, 2011.

[33] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proc. of WEA'13*, pages 1–5. ACM, 2013.

[34] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *TSE*, 38(6):1276–1304, 2012.

[35] Rain Brian Harrys. The largest git repo on the planet. *Online: https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/*, 2017.

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

[37] Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Emp. Soft. Eng.*, 21(2):303–336, 2016.

[38] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proc. of ICSE'94*, pages 191–200. IEEE Computer Society Press, 1994.

[39] Javier Luis Cánovas Izquierdo, Valerio Cosentino, and Jordi Cabot. An empirical study on the maturity of the eclipse modeling ecosystem. In *Proc. of MODELS'17*, pages 292–302. IEEE, 2017.

[40] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678, 2011.

[41] Martin Johns and Moritz Jodeit. Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools. In *Proc. of ICSTW'11*, 2011.

[42] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. of ISSTA'14*, 2014.

[43] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proc. of ICSE'11*, 2011.

[44] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. Is it all lost? a study of inactive open source projects. In *In Proc. of IFIP OSS'13*, pages 61–79. Springer, 2013.

[45] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proc. of MSR'17*, pages 102–112. IEEE, 2017.

[46] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proc. of ICSE'07*, pages 344–353. IEEE Press, 2007.

[47] Kendra J Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code. Technical report, DTIC Document, 2005.

[48] Paul R Kroeger. *Analyzing grammar: An introduction.* Cambridge University Press, 2005.

[49] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Emp. Soft. Eng. Journ.*, May 2017.

[50] James A Kupsch and Barton P Miller. Manual vs. automated vulnerability assessment: A case study. In *Proc. of MIST'09*, pages 83–97, 2009.

[51] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *Proc. of NDSS'17*, 2017.

[52] Hayne E Leland. Corporate debt value, bond covenants, and optimal capital structure. *Journ. of Finance*, 49(4), 1994.

[53] Daoyuan Li, Li Li, Dongsun Kim, Tegawendé F Bissyandé, David Lo, and Yves Le Traon. Watch out for this commit! a study of influential software changes. *arXiv preprint arXiv:1606.03266*, 2016.

[54] Peng Li and Baojiang Cui. A comparative study on software vulnerability static analysis techniques and tools. In *Proc. of ICITIS'10*, 2010.

[55] Benjamin Livshits. Stanford securibench. *Online: http://suif. stanford. edu/livshits/securibench*, 2005.

[56] Benjamin V. Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. of USENIX'13*, 2005.

[57] SS Jeremy Long. Owasp dependency check, 2015.

[58] Konstantinos Manikas. Revisiting software ecosystems research: A longitudinal literature study. *Journ. of Sys. and Soft.*, 117:84–103, 2016.

[59] Konstantinos Manikas. Supporting the evolution of research in software ecosystems: reviewing the empirical literature. In *Proc. of ICSOB'16*, pages 63–78. Springer, 2016.

[60] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems–a systematic literature review. *Journ. of Sys. and Soft.*, 86(5):1294–1306, 2013.

[61] Simon Maple. Java tools and technologies landscape report 2016. *Online: https://jrebel.com/rebellabs/java-tools-and-technologies-landscape-2016/*, 2016.

[62] David C Mauer and Sudipto Sarkar. Real options, agency conflicts, and optimal capital structure. *JBF*, 29(6), 2005.

[63] Tom Mens, Mathieu Goeminne, Uzma Raja, and Alexander Serebrenik. Survivability of software projects in gnome–a replication study. *Proc. of SATToSE'14*, page 79, 2014.

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

[65] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Emp. Soft. Eng. Journ.*, 12(5):471–516, 2007.

[66] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proc. of CCS'07*, 2007.

[67] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. An automatic method for assessing the versions affected by a vulnerability. *Emp. Soft. Eng.*, 21(6):2268–2297, 2015.

[68] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. An automatic method for assessing the versions affected by a vulnerability. *Emp. Soft. Eng. Journ.*, 21(6):2268–2297, 2016.

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

[70] Vadim Okun, Aurelien Delaitre, and Paul E. Black. The second static analysis tool exposition (SATE) 2009. *NIST SP*, pages 500–287, 2010.

[71] Vadim Okun, Aurelien Delaitre, and Paul E. Black. Report on the third static analysis tool exposition (SATE 2010). *NIST SP*, pages 500–283, 2011.

[72] Vadim Okun, Aurelien Delaitre, and Paul E. Black. Report on the static analysis tool exposition (SATE) IV. *NIST SP*, 500:297, 2013.

[73] Vadim Okun, Romain Gaucher, and Paul E. Black. Static analysis tool exposition (SATE) 2008. *NIST SP*, 5(00-2):79, 2009.

[74] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. Myths and facts about static application security testing tools: An action research at telenor digital. In *Proc. of XP'18*, pages 86–103. Springer, 2018.

[75] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[76] Ivan Pashchenko. Foss version differentiation as a benchmark for static analysis security testing tools. In *Proc. of FSE'17*, pages 1056–1058. ACM, 2017.

[77] Ivan Pashchenko, Stanislav Dashevskyi, and Fabio Massacci. Delta-bench: differential benchmark for static analysis security testing tools. In *Proc. of ESEM'17*, pages 163–168. IEEE Press, 2017.

[78] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: counting those that matter. In *Proc. of ESEM'18*, page 42. ACM, 2018.

[79] Jannik Pewny and Thorsten Holz. Evilcoder: automated bug insertion. In *Proc. of ACSAC'16*, pages 214–225. ACM, 2016.

[80] Shaun Phillips, Guenther Ruhe, and Jonathan Sillito. Information needs for integration decisions in the release process of large-scale parallel development. In *Proc. of CSCW'12*, pages 1371–1380. ACM, 2012.

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

[82] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *Proc. of ICSME'15*, pages 411–420. IEEE, 2015.

[83] L Ponemon. Ponemon institute cost of a data breach study, 2016.

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

[85] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, 2016.

[86] Latifa Rabai, Arfa Ben, Barry Cohen, and Ali Mili. Programming language use in us academia and industry. *Inf. in Education*, 14(2):143, 2015.

[87] Donald J Reifer, Victor R Basili, Barry W Boehm, and Betsy Clark. Eight lessons learned during cots-based systems maintenance. *IEEE Softw. Journ.*, 20(5):94–96, 2003.

[88] Sofia Reis and Rui Abreu. Secbench: A database of real security vulnerabilities. In *SecSE@ ESORICS*, pages 69–85, 2017.

[89] Ripon K. Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E. Perry. Are these bugs really normal? In *Proc. of MSR'15*, 2015.

[90] Hitesh Sajnani, Vaibhav Saini, Joel Ossher, and Cristina V Lopes. Is popularity a measure of quality? an analysis of maven components. In *Proc. of ICSME'14*, pages 231–240. IEEE, 2014.

[91] Johnny Saldaña. *The coding manual for qualitative researchers.* Sage, 2015.

[92] Sudipto Sarkar. Optimal size, optimal timing and optimal financing of an investment. *Journ. of Macroeconomics*, 33(4), 2011.

[93] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *TSE*, 37(6):772–787, 2011.

[94] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. Test suites for benchmarks of static analysis tools. In *Proc. of ISSREW'15*, pages 12–15. IEEE, 2015.

[95] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proc. of FSE'14*, pages 23–34. ACM, 2006.

[96] Nancy L Stokey. *The Economics of Inaction: Stochastic Control models with fixed costs.* Princeton University Press, 2008.

[97] Anselm Strauss and Juliet Corbin. *Basics of qualitative research.* sage, 1990.

[98] Hao Tang, Tian Lan, Dan Hao, and Lu Zhang. Enhancing defect prediction with static defect analysis. In *Proc. of INTERNETWARE'15*, 2015.

[99] Andreas Wagner and Johannes Sametinger. Using the juliet test suite to compare static security scanners. In *Proc. of SECRYPT'14*, pages 1–9. IEEE, 2014.

[100] John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. 2002.

[101] Jeff Williams and Arshan Dabirsiaghi. The unfortunate reality of insecure libraries. *Asp. Sec.*, pages 1–26, 2012.

[102] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proc. of MSR'16*, pages 351–361. IEEE, 2016.

[103] Robert K Yin. *Qualitative research from start to finish.* Guilford Publications, 2015.