PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Security Assessment of Open Source Third-parties Applications

Stanislav Dashevskyi

Advisors:

Advisor:

Prof. Fabio Massacci

Università degli Studi di Trento

Co-advisor:

Dr. Antonino Sabetta

SAP Labs France

May 2017

Acknowledgements

First of all, I would like to thank Professor Fabio Massacci (University of Trento, Italy) who had been the supervisor of my PhD studies. I would like to thank him for his support, his patience and for everything that he has taught me. It would be impossible to write this dissertation without his help and extremely valuable feedback.

I would like to thank Dr. Antonino Sabetta (SAP Labs France) for co-supervising me.

I am very thankful to Professor Achim D. Brucker (University of Sheffield, United Kingdom) for mentoring me during my stay in Germany and hosting me during my visit to the UK, for all the discussions that we had, for his invaluable contribution to my work, and for many more.

I am extremely grateful to Professor Andrea De Lucia (University of Salerno, Italy), Professor Massimiliano Di Penta (University of Sannio, Italy) and Professor Paolo Tonella (Fondazione Bruno Kessler, Italy) for dedicating their valuable time to be the members of my PhD defense committee. I am very thankful to them for providing the extremely helpful feedback on this dissertation.

I would like to thank my colleagues Katsiaryna Labunets, Luca Allodi, Viet Hung Nguyen, Avinash Sudhodanan, Nadia Metoui, Mojtaba Eskandari, and Daniel Ricardo Dos Santos. I was fortunate to meet you.

I would like to thank all the wonderful people that I met (there are too many names, but I remember you all). I thank you for the good moments that we shared together during this period of my life.

My special gratitude goes to my future wife Kateryna Tymoshenko and to my family for being with me throughout the bad moments, and for their love and support.

Abstract

Free and Open Source Software (FOSS) components are ubiquitous in both proprietary and open source applications. In this dissertation we discuss challenges that large software vendors face when they must integrate and maintain FOSS components into their software supply chain. Each time a vulnerability is disclosed in a FOSS component, a software vendor must decide whether to update the component, patch the application itself, or just do nothing as the vulnerability is not applicable to the deployed version that may be old enough to be not vulnerable. This is particularly challenging for enterprise software vendors that consume thousands of FOSS components, and offer more than a decade of support and security fixes for applications that include these components.

First, we design a framework for performing security vulnerability experimentations. In particular, for testing known exploits for publicly disclosed vulnerabilities against different versions and software configurations.

Second, we provide an automatic screening test for quickly identifying the versions of FOSS components likely affected by newly disclosed vulnerabilities: a novel method that scans across the entire repository of a FOSS component in a matter of minutes. We show that our screening test scales to large open source projects.

Finally, for facilitating the global security maintenance of a large portfolio of FOSS components, we discuss various characteristics of FOSS components and their potential impact on the security maintenance effort, and empirically identify the key drivers.

Keywords Security Vulnerabilities; Security Maintenance; Third-party Components; Free and Open Source Software; Vulnerability Screening Test

Contents

List of Tables

List of Figures

Chapter 1

Introduction

According to a recent Black Duck study [149], more than 65% proprietary applications leverage Free and Open Source Software (FOSS) components: this choice speeds up application development and flexibility [77], because FOSS components can be (and often are) used "as-is" without any modifications [90, 137]. The price to pay are security flaws being found in these components, in particular, for web applications which are the target of many well known exploits and a fertile ground for the discovery of new security vulnerabilities [142], especially when the source code is publicly available.

Since the security of a software product depends on the security of all its components, securing the whole software supply chain is of utmost importance for software vendors. Thus, FOSS components should be subject to the same security scrutiny as one's own code[1] [103].

FOSS components impose particular challenges as well as provide unique opportunities. For example, FOSS licenses contain usually a very strong "no warranty" clause and no service-level agreement. On the other hand, FOSS licenses allow to modify the source code and, thus, to fix issues without depending on (external) software vendors.

When addressing FOSS security in an academic setting, the most debated question is whether FOSS is more or less secure than proprietary software [72,75,143]. This discussion received a new impetus since Heartbleed (CVE-2014-0160), Shellshock (CVE-2014-6271), Apple's GoToFail bug (CVE-2014-1266), or Microsoft's sChannel flaw (CVE-2014-6321).

Yet, we would like to argue that the ultimate answer for this question may not be that important from the point of view of the software industry: certain FOSS components may be the de-facto standard for some applications, and certain FOSS components may offer functionalities that are very expensive to re-implement. Indeed, this debate distracts from more pressing issues that we describe below.

Consider the following (daily) scenario in the activities of software vendors: *when a new*

[1] For example, SAP, a large European software vendor, runs static code analysis tools to verify the combined code bases of its applications and FOSS components [29].

vulnerability in a FOSS component is discovered, the vendor has to verify whether it affects customers who consume software solutions into which that particular FOSS component was bundled. If the answer is positive, the vendor has to provide support (issue updates, or provide custom security fixes) to all customers that are (or may be) affected by the vulnerability.

For instance, in Enterprise resource planning (ERP) systems and industrial control systems the need for such an activity may occur years after deployment of the selected FOSS component. The more pressing issues in which the software vendors are interested in comprise of assessing not only the direct impact of vulnerabilities in third-party components on their products, but also in assessing the impact on the effort associated with the security maintenance of such components.

1.1 Problems of Secure FOSS Integration and Consumption

The above scenario includes many important aspects, which we break down into several sub-problems below:

Problem 1 : Besides the vulnerabilities discovered in a FOSS component that may affect the application that consumes it, the security of a software offering depends on the rest of its constituents as well. A successful exploitation may be possible not only because of the vulnerabilities in the source code, but also because of the environments on which applications are deployed and run: such execution environments usually consist of application servers, databases and other supporting applications (including the aforementioned FOSS components). As a part of necessary activities for securing the software supply chain, it is important to *test whether known exploits for software components can be reproduced in different settings, and understand their potential effects.*

Problem 2 : Indeed, to obtain a vulnerability proof-of-concept, a vendor may test a product that contains a potentially vulnerable FOSS component against a working exploit, but for many vulnerabilities there are no public exploits immediately available [8]. Even if such exploits exist, they must be adapted to trigger the FOSS vulnerability in the context of the consuming application, which requires significant effort. An alternative is to apply static analysis security testing tools (SAST) against the FOSS component. Unfortunately, it is difficult for vendors to locate sources of security vulnerabilities within the sheer number of FOSS components, especially when they are used as "black boxes" [90, 137]. Such an analysis requires a solid understanding of the source code of a component in question and its usage context [90]. It also requires a significant expertise in chosen SAST tools [18], as these tools can generate thousands of potentially false warnings for large projects. Further, the analysis may require days for processing even a single 'FOSS-release' 'main-application' pair [2]. All this significantly complicates the task of security analysis of third-party FOSS components and increases the chances that vulnerabilities in these components will be left unresolved by vendors [62]. Moreover, when several FOSS releases are used in many different products, the above solutions do not scale: thus, we need to understand *what could be an accurate and efficient screening test for the presence of a vulnerability within many (older) versions of a FOSS component?*

Problem 3 : When such a screening test is available, it can be used for company-wide estimates to empirically assess the likelihood that an older version of a FOSS component may be affected by a newly disclosed vulnerability, as well as the potential maintenance effort required for upgrading or fixing that version. For this task, it is important to understand which characteristics of a FOSS component (number of contributors, popularity, lines of code or choice of programming language, etc.) are likely to be sources of "troubles" for security maintenance (the number of vulnerabilities of a FOSS product is only a facet of a trouble, as a component may be used by hundreds of products). *Can different models for maintenance of FOSS components, as well as different factors that characterize these components, have significantly different impact on the global security maintenance effort of large software vendors?*

1.2 Contributions

The aim of this dissertation is to assist software vendors in identifying relevant properties of FOSS components that would facilitate their security maintenance, as well as finding evidence that can be used for identifying which versions of FOSS components across the third-party component portfolio of vendors may be affected by known or newly disclosed security vulnerabilities. This would allow vendors to quickly identify which customers may be affected and plan their security maintenance activities accordingly. We also propose models that focus on the actual effort of security maintenance required for resolving these security issues. The main contributions of this dissertation are as follows:

A case study at a large software vendor, aimed at identifying the factors to consider when evaluating the impact of FOSS selection choices on the security maintenance effort.

An open source framework for performing security vulnerability experimentations and testing, in particular, obtaining evidence that would show whether existing exploits for known vulnerabilities can be reproduced in different settings and environments, for better understanding of their technical impact, and facilitating discovery of new vulnerabilities.

An automatic scalable vulnerability screening test for estimating the likelihood of an older version of a FOSS component to be affected by a newly disclosed vulnerability, using the vulnerability fix. We also provide an insight on the empirical probability that a version of a potential vulnerable component might not actually be vulnerable if it is too old (or that its update might be likely costly). We provide a manual validation of the method, as well as an empirical analysis of the trade-offs between the likelihood that an older version is affected by a newly disclosed vulnerability and the potential maintenance effort required for upgrading to a fixed version (using popular FOSS projects), showing that the approach scales to thousands of revisions of large code bases.

A model for assessing the impact of various characteristics of FOSS components on the security maintenance effort of a large portfolio of FOSS components. We empirically test these factors, impacting the global vulnerability resolution process of third-party components of a large software vendor, on three different maintenance models:

1. The *centralized model*, where vulnerabilities of a FOSS component are fixed centrally and then pushed to all consuming products (and therefore costs scale sub-linearly in the number of products);
2. The *distributed model*, where each development team fixes its own component and effort scales linearly with usage;
3. The *hybrid model*, where only the least used FOSS components are selected and maintained by individual development teams.

The work on this dissertation was performed in the context of the European Project no. 317387 SECENTIS in collaboration with an industrial partner – SAP. This work represents the research carried out by the author, and its outcome may not necessarily represent the official position of SAP.

1.3 Dissertation Structure

This dissertation is structured as follows:

Chapter 2 provides some background on FOSS software and the broad issues of security certification of third-party components, as well as the identification of vulnerabilities. This chapter was partially published in:

> [48] S. Dashevskyi, A. D. Brucker, and F. Massacci. “On the Security Cost of Using a Free and Open Source Component in a Proprietary Product”. *In Proceedings of the 2016 Engineering Secure Software and Systems Conference*, 2016.

Chapter 3 describes a case study at a large European software vendor which integrates a large number of FOSS components into its products, and for which the present research is relevant: (1) we describe the challenges that the vendor faces when consuming third-party FOSS components; (2) discuss the process of building a theory on various FOSS maintenance aspects based on internal discussions with vendor's software developers and researchers; and (3) provide our understanding of various processes of FOSS maintenance and consumption adopted by the vendor. A part of the content of this chapter was submitted to:

> [46] S. Dashevskyi, A. D. Brucker, and F. Massacci. "On the Security Maintenance Cost of Open Source Components". *Submitted to ACM Transactions on Internet Technology (Special Issue on the Economics of Security and Privacy).*

Chapter 4 describes TestREx – a framework for repeatable exploits that allows performing penetration and security testing, in particular running and adapting exploits against potentially vulnerable web applications to identify whether they are affected by the vulnerability. This chapter is the result of a joint work with Daniel Ricardo Dos Santos, a fellow PhD student. This chapter was partially published in:

> [49] S. Dashevskyi, D. R. Dos Santos, F. Massacci, and A. Sabetta. "TestREx: a testbed for repeatable exploits". *In Proceedings of the 7th USENIX Workshop on Cyber Security Experimentation and Test*, 2014.

> [136] A. Sabetta, L. Compagna, S. Ponta, S. Dashevskyi, D.R. Dos Santos, and F. Massacci. "Multi-Context Exploit Test Management". *US Patent App. 14/692,203*, 2015.

Chapter 5 describes the screening test for estimating whether a given vulnerability is present in a version of a FOSS component. Our analysis supports software vendors in prioritizing their FOSS related maintenance and development efforts. We showed that it can scale to large open source projects. This chapter will be submitted to:

> [47] S. Dashevskyi, A. D. Brucker, and F. Massacci. "A Screening Test for Disclosed Vulnerabilities in FOSS Components". *To be submitted to ACM Transactions on Software Engineering.*

Chapter 6 illustrates our models for assessing the impact of various characteristics of FOSS components on the effort required for their security maintenance, and provides an empirical analysis of these factors. This chapter was partially published in:

> [48] S. Dashevskyi, A. D. Brucker, and F. Massacci. "On the Security Cost of Using a Free and Open Source Component in a Proprietary Product". *In Proceedings of the 2016 Engineering Secure Software and Systems Conference*, 2016.

> [46] S. Dashevskyi, A. D. Brucker, and F. Massacci. "On the Security Maintenance Cost of Open Source Components". *Submitted to ACM Transactions on Internet Technology (Special Issue on the Economics of Security and Privacy).*

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

Chapter 2

Background

This chapter aims to provide the background on the consumption of third-party FOSS components, as well as to discuss the broad issues of security certification of third-party components.

2.1 Overview of Third-party Components

Proprietary software development usually consumes several types of third-party components. The most important components (either whole sub-systems, or libraries) are:

- *Outsourced development and sub-contracting:* components are developed by a different legal entity based on a custom contract. As the software is implemented based on a customer-specific contract and is uniquely tailored to its business needs [159], the consuming party can specify the required compliance and security guidelines. Depending on the contract, such components can be either shipped in the binary or in the source form.
- *Proprietary (standard) software components:* components are licensed from a third-party. For this third-party, this is a standard offering, i.e., the same component is offered to multiple customers. Thus, there is only a very limited room for, e.g., influencing the security development processes at the supplier. Usually, such components are shipped as binaries.
- *FOSS components:* grant free access to the source code, as well as the freedom to distribute modified versions, provided that certain licensing restrictions are respected [134]. There are many open source licenses that describe different legal aspects in different ways, including the detailed conditions under which FOSS can be distributed (see [33] and [133] for a comprehensive discussion on FOSS licenses).

2.2 What Makes FOSS Special?

FOSS components share aspects with both outsourced development and subcontracting, as well as with standard proprietary software components. For example, FOSS components can be modified, adapted, and maintained by the customer (this is what they have in common with outsourcing and subcontracting).

Similarly to standard proprietary software components, FOSS components usually provide a fixed set of interfaces and functionalities that consuming products need to be adapted to (instead of having a custom made component that "just fits"). Technically, FOSS licenses are particular legal contracts that determine the rules under which a software component can be used [33, 57]. Thus, one could expect that there is no need to handle them differently from non-FOSS third-party components, but this is not the case. In practice, there are at least five aspects that are often considered to be special:

1. As FOSS components are easily available without initial costs, this might misguide developers to use them without properly assessing their licenses in detail [134]. For large software vendors, the legal check of the software license and the warranty is a part of the purchasing process: the derivative work that may be created by making custom fixes of FOSS components and re-distributing them is an important aspect. For instance, licenses such as GNU GPL[1], require any derivate work to be distributed under the same license, which may not be acceptable for proprietary software vendors. It may be also difficult to identify what is exactly the derivative work in some cases – this may create additional problems for proprietary vendors (see Carver [33] for a more comprehensive discussion on open source software licenses and potential legal issues). As FOSS components are often simply downloaded from the Internet, it may be more difficult for vendors to enforce legal checks.
2. When FOSS components are being integrated into the target application, it increases the overall costs of the resulting software product due to additional development and maintenance activities [3]. Indeed, as pointed by Ven and Mannaert [162], the most preferred way to cut these costs is contributing fixes back to the FOSS community. Unfortunately, on practice this strategy is applicable for small generic fixes, that are also useful for the community. There may be specific and more extensive modifications (including fixing newly disclosed vulnerabilities for older versions of a FOSS component) that are important for specific business users, but not for the FOSS community. The authors [162] argue that these more extensive fixes may require different strategies such as updating only over specific periods of time (for backward compatibility), or forking a component (for custom modifications). Regardless of which of the latter strategies is chosen, this often results into significant additional work due to maintenance.

[1] `https://www.gnu.org/licenses/gpl-3.0.en.html`

3. Most FOSS licenses contain rather strong “no warranty” clauses. For instance, the GNU GPL license contains the following disclaimer: “THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION”. The Mozilla Public License[2] contains similar clause. Such disclaimers often come together with the lack of a contractual binding maintenance model. Thus, when using FOSS, one needs to decide how this can be mitigated. This can be done either by entering a commercial agreement with a company that offers support for FOSS components [57], or by investing into the in-house maintenance. The lack of documentation in FOSS projects [16, 153] may prevent in-house developers from learning, thus increasing potential maintenance costs.
4. Proprietary software vendors are often unable to influence development processes and patch release policies of consumed FOSS components [153], which may complicate maintenance tasks in case vendors have to provide own support for the components. Given that vulnerability patching is prioritized among other maintenance tasks [13], it is nearly impossible to establish the equilibrium between patch releases in FOSS components and updates of versions used by proprietary vendors (see [34] for a discussion).
5. In particular, security response processes for proprietary software often try to release detailed information about a vulnerability only after a patch was released. The goals are to provide customers a safety period to patch their systems before a vulnerability gets publicly known as well as *not* publishing fixes that can be transformed into zero day vulnerabilities for the previous versions of the product. This might conflict, on one hand with FOSS licenses that require to contribute changes back to the community and, on the other hand, with security response processes set up by the FOSS projects – publishing a security patch in the source code form can be considered as making a vulnerability publicly known (see [130]).

Some of the above aspects are based on empirical studies that are already five to ten years old. As in the last decade the awareness of software security increased both in FOSS development, as well as in proprietary software industry, there is a risk that not all of the findings are still applicable. For example, many larger FOSS projects nowadays support

[2] `https://www.mozilla.org/en-US/MPL/2.0/`

confidential reporting of security issues, as well as responsible patching and disclosure processes. Thus, for such projects, the aspect (4) may be less of an issue.

There are also indirect consequences of the freedoms and additional opportunities provided by FOSS licenses. For example, if the maintenance model of a proprietary component does not fit the needs of consuming software product, one needs to negotiate a custom support contract, or search for alternative offerings. FOSS components provide at least two additional opportunities:

1. Apart from the original developers of a FOSS project, there may be other companies that can offer commercial support and bug fixes. Thus, it is possible to select between different maintenance offerings for the same component.
2. As the source code is available and modifications are allowed (under certain conditions), one can fix issues independently from the developers of a FOSS component.

Unfortunately, these opportunities also lead to certain risks (see, for example, points (3) and (4) above).

2.3 Certification and Empirical Assessment of Third-party Software

2.3.1 Selection and Integration of FOSS components

There exist numerous works [16, 45, 77, 102, 153] that investigate various aspects of open source software components lifecycle, including scenarios when these components are integrated into (or re-used by) proprietary applications, as well as discuss the importance of FOSS for the modern software development [9, 58, 89, 104].

Stol and Babar [153] perform a systematic literature review of scientific publications to identify challenges in integrating open source components into proprietary software products. The authors identified that among the main challenges there are the product selection, the lack of comprehensive information, and maintenance considerations. For instance, insufficient documentation may interfere with the developers' ability to learn how to use a component, and the lack of time for performing a thorough evaluation of a component may cause additional problems in the future (including security). Additionally, weak community and lack of support for a FOSS component may result into additional expenses for a company that is using this component. Earlier, Merilinna and Matinlassi [102] provided an overview of practices for combining open source integration techniques. The authors [102] also point out that the lack of proper documentation is a big challenge of adopting FOSS.

Ayala et al. [16] performed an interview within several software companies, and report their findings about how these companies collect information about FOSS components. They identified that they are often selected based on the previous experience, even without considering alternatives. Thus, according to the study [16], experience was one of the key factors for the component evaluation. Also, according to the results of their interview, some developers mentioned that insufficient documentation within many FOSS projects is indeed a significant problem.

Often, the decision about integration of a FOSS component is made after an appropriate candidate is evaluated against a certain set of criteria. Since every software vendor (or a development team) may have its own set of evaluation criteria, this process may be completely ad-hoc [102, 153], or being constantly revised, and, therefore, differ not only from company to company, but even from case to case.

To tackle the problem, researchers proposed various selection and evaluation models for FOSS. Ahmad and Laplante [5] proposed a systematic approach for evaluating open source software against a set of factors that include functionality, licensing, evolution speed, longevity, community, quality of documentation and support. Aiming to answer the question "which factors are considered for open source software selection?", they developed a framework that simplifies the decision making process for humans. For validating their method, the authors performed a survey with human participants, indicating that functionality was the most important selection factor, according to their responses. However, the approach does not consider any properties of FOSS projects relevant to software security.

Wheeler [166] described a generic process for evaluating open source software, which is based on identifying proper candidates, gathering information about the candidates (reading reviews), and analyzing the shortlist of candidates in more depth. The important factors to consider include functionality, market attributes, support and maintenance, various quality attributes, security and legal aspects. For assessing the security, the author proposed to use static analysis tools (such as Coverity and Fortify), vulnerability reports, as well as the common criteria evaluation. While using various tools for security code analyses is a common industrial practice [29], it may be very difficult to perform it for a large number of FOSS components (see Problem 2 in Chapter 1).

Several works [5, 14, 15, 139, 140] are focused on the overall software quality and functionality as the main selection criteria, with little or no emphasis on software security. Ardagna et al. [10] proposed FOCSE - the framework for selecting FOSS components that provides security-related functionality. This framework is based on the set of features of FOSS projects that could be aggregated and weighted, providing a unified qualitative measure. Some of these features may be only obtained having a privileged access to the development information. The approach is suitable for making a decision between several security-related FOSS projects, as well as for making a choice between other types of projects. However, it does not include any explicit security metrics that would help to reason about the security of a FOSS project itself.

Samoladas et al. [139] proposed a model that supports automated software evaluation, specifically targeted on open source. The set of metrics considered by the model is represented by the code quality metrics (including security), and community quality metrics (mailing list, documentation and developer base). While this model takes security into account, it comprises of only two variables: "null dereferences" and "undefined values". This limitation was a deliberate choice of the authors [139] in order to facilitate the automation of the metrics collection process.

Del Bianco et al. [51] developed QualiPSo - the methodology for assessing the FOSS development processes. Instead of providing measures for selecting or integrating FOSS components, it aims for providing FOSS developers with metrics that could be used to assess and improve the quality of their projects, increasing the trustworthiness between FOSS projects and their potential consumers.

Wheeler and Khakimov [167] published a white paper that describes the *Census* [3] project of Core Infrastructure Initiative. The authors describe a set of publicly available metrics that should be considered for identifying important (for their set of features) projects that have security problems due to the lack of resources (investments, developers). While this methodology cannot be directly used for FOSS project comparison, it provides an interesting set of metrics and insights that are worth to be considered when selecting open source components for consumption from the security point of view.

2.3.2 Empirical Assessment of Vulnerabilities

An extensive body of research explores the applicability of various metrics for estimating the number of bugs and security vulnerabilities of a software component. Apart from factors that characterize the overall quality, security, and liveness of software projects, software development companies that are using these projects could employ various prediction approaches from the literature for assessing the security status of a FOSS project.

The simplest such factor is time (since release), and the corresponding model is a Vulnerability Discovery Model. Massacci and Nguyen [96] provide a comprehensive survey and independent empirical validation of several vulnerability discovery models. Several other metrics have been used: code complexity metrics [113, 145, 146], developer activity metrics [25, 145], static analysis defect densities [163], frequencies of occurrence of specific programming constructs [141, 164], etc. We illustrate some representative cases with Table 2.1.

Although our focus are security vulnerabilities that may stand aside from generic software bugs (e.g., errors in functionality), Ozment [120] showed that methods for estimating trends in generic bugs used in software engineering literature can be also applied for security vulnerabilities.

The works by Ostrand et al. [118] and Bell et al. [25] aimed on predicting files in new

[3] https://www.coreinfrastructure.org/programs/census-project

Table 2.1: Vulnerability and bug prediction approaches

We provide a brief overview of various approaches for bug prediction in the existing literature (we refer the reader to [71] and [125] for a more complete discussion).

Paper	**Predictors**	**Bug data**	**Predicted vars**
Ostrand et al. [118]	Bug and change histories of files	Internal data on previous releases of a commercial system	Files with largest bug concentration
Nagappan & Ball [109]	Relative Code churn	Internal defect dataset (Windows Server 2003)	Bug density
Shin & Williams [146]	Complexity metrics	MFSA, NVD, Bugzilla	Vulnerable functions
Nguyen & Tran [113]	Member and Component dependency graphs, Complexity metrics	MFSA, NVD	Vulnerable functions
Shin et al. [145]	Complexity metrics, Code churn, Developer activity	MFSA, Red Hat Linux package manager	Vulnerable files
Walden & Doyle [163]	Static analysis vulnerability density	NVD	Number of vulnerabilities
Bell et al. [25]	Developer metrics	Internal data on previous releases of a commercial system	Files with largest bug concentration
Massacci & Nguyen [96]	Known vulnerabilities	MFSA, NVD, Bugzilla, Microsoft Security Bulletin, Apple Knowledge Base, Chrome Issue Tracker	Number of vulnerabilities
Scandriato et al. [141]	Frequencies of prog. constructs	SAST warnings (Fortify SCA)	Vulnerable files
Walden et al. [164]	Complexity metrics, Frequencies of prog. constructs	NVD, Security notes from a project	Vulnerable files

releases of software projects that may have the largest concentration of bugs, so that they can be prioritized for testing. The work by Ostrand et al. [118] considered bug modification histories of files in previous releases, while the follow-up study by Bell et al. [25] used the information about individual developers: the authors of both studies had access to the industrial systems of the same vendor that they used for evaluating their work. The authors of [25] find evidence that prediction capabilities of the previous model in [118] improve when adding the cumulative number of developers as an additional factor.

Nagappan and Ball [109] evaluated code churn metrics for predicting bug densities in software, showing that metrics taken from development history can be a good predictor for the largest clusters of generic software bugs.

Shin and Williams [146] evaluated software complexity metrics for identifying vulnerable functions. The authors collected information about vulnerabilities in Mozilla JavaScript Engine (JSE) from Mozilla Foundation Security Advisories (MFSA)[4], and showed that nesting complexity could be an important factor to consider. The authors indicate that their approach had a small number of false positives, but at the cost of having false negatives. In a follow-up work, Shin et al. [145] also analyzed several developer activity metrics showing that poor developer collaboration can potentially lead to vulnerabilities, and that complexity metrics alone are not sufficient for vulnerability prediction. Similarly to [109], the authors of [145] suggest that code churn metrics are better indicators for approximate locations of vulnerabilities than complexity.

Nguyen and Tran [113] built a vulnerability prediction model using dependency graphs as intermediate representation of software, and applying machine learning techniques to train the predictor. They used several static analysis tools for computing source code metrics, and tools for extracting dependency information from the source code, adding this information to the graphs that represent a software application. To validate the approach, the authors analyzed Mozilla JSE. In comparison to [146], the model had a slightly bigger number of false positives, but less false negatives.

Walden and Doyle [163] used static analysis for predicting web application security risks. They measured the *static analysis vulnerability density* (SAVD) metric across version histories of five PHP web applications, which is calculated as the number of warnings issued by the `Fortify SCA`[5] tool per one thousand lines of code. The authors performed multiple regression analyses using the SAVD values for different severity levels as explanatory variables, and the post-release vulnerability density as the response variable, showing that the SAVD metric could be a potential predictor for the number of new vulnerabilities.

Scandriato et al. [141] proposed to use a machine learning approach, mining source code of Android components and tracking the occurrences of specific patterns. The authors used the `Fortify SCA` tool as the source of ground truth: if the tool issues a warning about a file, this file is considered to be vulnerable. However, it may not be the case as static analysis tools can have many false positives, and authors verified manually only the alerts for 2 applications out of 20. The results show that the approach had good precision and recall when used for prediction within a single project. Walden et al. [164] confirmed that the vulnerability prediction technique based on text mining (described in [141]) could be more accurate than models based on software metrics. They have collected a dataset of PHP vulnerabilities for three open source web applications by mining the National Vulnerability Database (NVD) and security announcements of those applications. They have built two prediction models: (1) a model that predicts potentially vulnerable files

[4] `https://www.mozilla.org/en-US/security/advisories/`

[5] `http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/`

based on source code metrics; and (2) model that uses the occurrence of terms in a PHP file and machine learning. The analysis shows that their machine learning model had better precision and recall than the code metrics model, however, it is applicable only for scripting languages (and must be additionally adjusted for languages other than PHP).

The biggest challenge in applying the results of empirical research in general, as well as empirical research that aims on security vulnerabilities, is the availability of the information that can be used to for evaluation of various heuristics or methods - the ground truth data. In particular, choosing the right source of vulnerability information is crucial, as any vulnerability prediction approach highly depends on the accuracy and completeness of the information in these sources. Massacci and Nguyen [95] addressed the question of selecting the right source of ground truth for vulnerability analysis. The authors of [95] show that different vulnerability features are often scattered across vulnerability databases and discuss problems that are present in these sources. Additionally, the authors provide a study on Mozilla Firefox vulnerabilities. Their example shows that if a vulnerability prediction approach is using only one source of vulnerability data (e.g., MFSA), it would actually miss an important number of vulnerabilities that are present in other sources such as the NVD. Of course, the same should be true also for the cases when only the NVD is used as the ground truth source for predicting vulnerabilities.

2.4 The Economic Impact of Security Maintenance

The cost of general software maintenance is well investigated in the literature. Banker and Slaughter investigated how software maintenance in organizations can be improved to achieve economical benefits [22]. They find support for the hypothesis that software maintenance can be characterized by scale economies, grounded on the observation that a significant part of the maintenance effort spent by developers is understanding the software to be modified (or patched) [21,59]. Several other software maintenance models considered the developers' familiarity with the software as an important factor as well [20, 21, 35].

The maintenance of software components from security economic perspective is relatively unexplored. Previous research on software economics focused on different choices such as, for example, buying a component versus building it from scratch [42], considering trade-offs between component costs and system's requirements [41], or optimizing the coupling and cohesion characteristics of component-based systems [88].

The major focus of the software engineering research so far has been on predicting software vulnerabilities (see [96]) and the security choices for different maintenance (patching) strategies. Stol and Babar described the challenges of integrating FOSS components into proprietary software, according to the past literature [153]. They identify maintenance among the most important challenges, suggesting that there may be no *immediate* costs while selecting FOSS components, but costs will eventually emerge during the consumption phase as the natural phenomenon of deteriorating software.

The consumption costs of FOSS components can be generated by delays due to software incompatibility of a newer version of a component with the target application, component failures (which imply reputation costs as well), maintenance of older versions of components, and creating patches [157]. Specifically, security patches that a proprietary vendor has to apply and distribute to end customers require significant increase in vendor's software maintenance efforts [20, 22]. There could be various other reasons why security patches provided by FOSS developers cannot be applied effortlessly: for instance, due to large number of vulnerabilities being disclosed periodically,[6] or the fact that third-party security patches should be additionally verified, or the patch has to be applied for all supported versions of a proprietary application that relies on potentially unsupported version of a third-party component being patched.

Conceptually, there is a distinction between *consumers* – parties that are using the software, and *providers* – development teams or organizations that provide the software and support it. For instance, the security patch management model by Cavusoglu et al. [34] specifies the costs of a consumer that emerge due to potential security *damage* (not applying a patch in time) and *update* (identifying, testing, and installing patches). According to the model, the providers' costs are generated by *patch release* (developing and shipping a patch) and *reputation losses* (vulnerabilities exploited before patches are released). This implies different types of costs for different parties, however, for our scenario, a proprietary software vendor would have to bear all these costs. This is because such vendors are *consumers* with respect to FOSS components, and, at the same time, they are *providers* with respect to their end customers (as FOSS components are bundled with original applications).

[6] For example, the study by Eric Rescorla [130] suggests that vulnerability discovery/fix rates for software projects do not decrease through their lifetime.

Chapter 3

An Exploratory Case Study at a Large Software Vendor

In this chapter we report on the exploratory case study that we performed at the premises of a large software vendor. The study aimed to explore the current approach and experience of our industrial partner in integrating FOSS components securely into its software supply chain, as well as to identify the most urgent problems that require attention. We describe the secure software development process used by our industrial partner and the place of FOSS components within this process, the selection and consumption of FOSS components, and discuss the relative importance of security maintenance of such components.

3.1 Introduction

The integration of FOSS components into products of proprietary software vendors is a complex problem that spans different issues at development and maintenance steps. For instance, at development time, development teams must ensure that FOSS components adhere to the same standard as a typical vendor's product.

To this end, our aim was to understand the role of FOSS components within the software supply chain of a large proprietary software vendor. We followed Yin [169] as a guidance on conducting case studies for performing our study. Various techniques exist for knowledge elicitation [76], and structured and semi-structured interviews are considered to be among the most important sources of information [169]. We used purposive sampling [70] while performing informal discussions with developers, and members of Security Testing and Maintenance teams. We conducted a case study at the premises of our industrial partner for exploring the following questions:

1. *What is the actual secure software development process of an industrial company, how is it managed, and what is the place of FOSS components within this process?*

2. *How are FOSS components selected for consumption, and which are the roles and activities involved in the choice and integration of FOSS components?*
3. *How is security maintenance of FOSS components managed, and what is its relative importance for the software supply chain of our industrial partner?*

A total period of 15 months was spent by the author of this dissertation on the premises of our industrial partner, including 12 months at the Research Lab, and 3 months with the Security Testing team at the main headquarters. During the latter time period, the author of this dissertation worked closely with Dr. Achim D. Brucker, who at that time had been a member of the Central Security Testing team for 8 years.

We collected notes, memos, and emails. We could not hold formally recorded interviews, as they would require an extremely heavy and lengthy authorization process through the legal department.

We were also making internal presentations to validate our insights, and to capture the variety of roles and activities related to the secure integration and consumption of FOSS components for our industrial partner at that time. Dr. Brucker, being the "key informant" in Yin's terminology [169], suggested the participants of these meetings, and provided the necessary introductions and background details to the participants. The set of participants consisted of interested software developers and security researchers, employed by our industrial partner. During that time period, we also had an opportunity to present parts of this work to a much broader audience of software developers at the yearly development kick-off meeting, organized by our industrial partner internally. During this meeting, we had in-depth discussions with software developers who confirmed our understanding of the FOSS integration and maintenance problems of our industrial partner, and allowed us to define our further steps.

3.2 Secure Software Development Lifecycle

The first finding concerns the *secure software development process* and the place of FOSS components within it. From what we understood during our meetings with members of the Central Security Team, our industrial partner follows a Security Development Lifecycle (*SDL*) process[1], the main steps of which are split into the following phases:

- *Preparation:* this phase consists of activities related to security awareness trainings for developers and team managers. One of the purposes of these activities is to raise awareness for security implications of using third-party FOSS components during product development.

- *Risk Identification*: development teams, together with the local security experts,

[1] This *SDL* is only one example of a security development lifecycle, and our study is not specific to this particular security development process. For example, it is similarly applicable to Microsoft's *SDL* [78].

organize various threat modeling activities. The goal of these activities is to identify potential application-specific risks of all third-party components (including FOSS) and their attack surfaces.

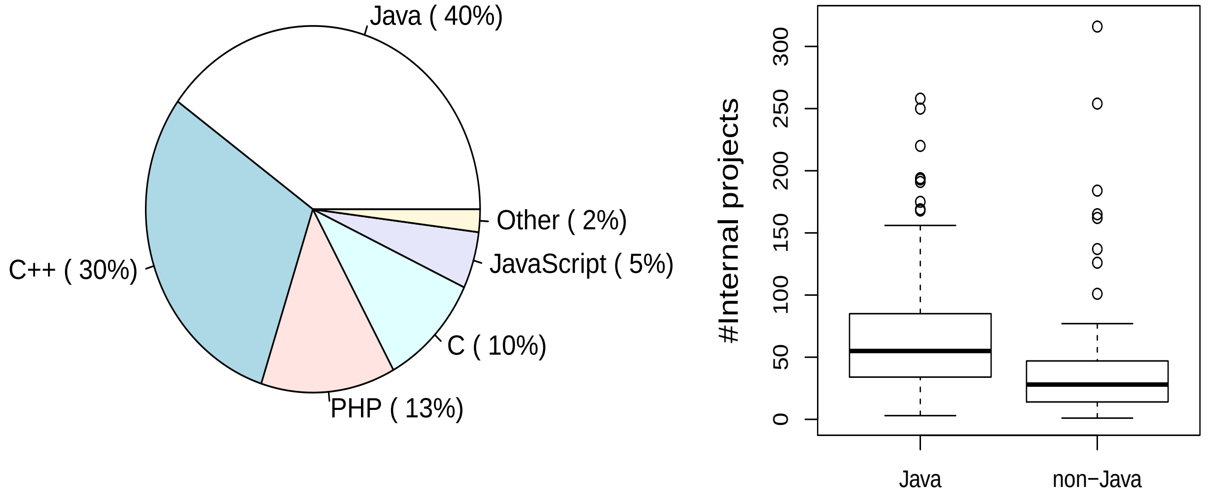
- *Development:* this phase includes activities of planning the development of a new product (or a new product version). In particular, it covers:
  - *Planning of Security Measures*, which describes the mitigation of the previously identified security risks. For example, it also includes the security testing plan that describes a product specific, risk-based security testing strategy that ensures that the security measures are implemented correctly. This security measures plan needs to cover both in-house development and all kinds of third-party components.
  - *Secure Development*, using defensive implementation strategies such as secure code guidelines and implementation best practices. While this step is mostly focused on the in-house development, third-party components might influence it as well (e.g., developers might consider to limit the usage of potentially insecure interfaces of third-party components). Moreover, for FOSS components the same techniques (e.g., static code analysis) could be used for the security assessment.
  - *Security Testing*, which ensures that the planned security measures (including defensive implementation strategies) are implemented and are effective in preventing security threats.
- *Transition:* this phase is performed by the *Security Validation* team, which is an independent control entity that acts like the first customer, and performs security assessment of the final product. Depending on the previous risk assessment, this may include architectural security analyses, code reviews, or penetration testing. Any security issues found during this step, regardless whether they are in own coding or third-party components, need to be fixed before the actual shipment.
- *Utilization:* during this phase, the *Security Response* team handles the communication with customers and external security researchers about reported vulnerabilities, as well as ensures that development and maintenance teams fix the reported issues (including down-ports to all supported releases and all their third-party components as required by the support agreements).

According to the *SDL* process defined by our industrial partner, the secure consumption of FOSS components requires attention in all its phases. However, applying standard secure development procedures to all FOSS components (for instance, performing static code analyses) requires solid understanding of the source code, the architecture, and the use case of each FOSS component – which may be costly for a large number of

FOSS components (see [29] for further details on applying static analysis in the industry). Therefore, our industrial partner is exploring risk-based security assessment approach as a part of the secure development activities (see [19] for example), which motivated the work carried out in this dissertation. The risk-based approach would, for instance, favor the search for the factors that help to estimate the security risk and the maintenance effort associated with the consumption of particular FOSS components, that are easier to obtain and to assess.

Figure 3.1: Descriptive statistics of FOSS components used or requested by internal projects

The two figures characterize the sample of the most popular 166 FOSS projects used and requested by different internal projects of our industrial partner: the figure on the left illustrates the sample in terms of the size of the code base implemented in a specific programming language, while the figure on the right illustrates the distribution of the number of usages/requests of FOSS components.


(a) The prevalence of programming languages in components

(b) The number of integrations of a component

In order to understand the role of FOSS components in the development process of our industrial partner, we collected data for 166 most popular FOSS projects that were requested by developers of internal projects as components during the last five years[2].

We learned that the number of FOSS components per product may vary: for example, while traditional ERP systems written in proprietary languages (e.g., ABAP or PeopleCode) usually do not contain many FOSS components, the situation is quite the opposite for recent cloud offerings, such as the ones based on OpenStack[3] or Cloud Foundry[4]. As we can see from Figure 3.1, FOSS components are integrated into (or requested for integration by) a large number of projects of our industrial partner. Figure 3.1a illustrates

[2] This information is publicly available and can be reconstructed from the bill of materials of individual projects found on the web community of the vendor (although, it was significantly easier to collect this information using the internal sources).

[3] `https://www.openstack.org/`

[4] `https://www.cloudfoundry.org/`

the cumulative size of the code bases of the components in the sample broken down by different programming languages in which they were implemented: the distribution suggests that the largest code base corresponds to Java.

Figure 3.1b shows the distributions of the number of internal projects that are using (or have requested) a FOSS component from the sample, divided by Java and non-Java components: these distributions also suggest the prevalence of Java-based components in comparison to non-Java components. To verify this difference, we used non-parametric Wilcoxon test, since the data that we collected is not normally distributed (Shapiro-Wilk test returned $p < 0.05$), and it contains unpaired samples. The results of Wilcoxon test confirmed that Java-based components are indeed more used (or requested) by the developers of our industrial partner in comparison to the others: the two distributions have small-to-medium and statistically significant difference ($p < 0.05$, Cohen's $d = 0.44$).

Table 3.1: Popular Java projects used by our industrial partner

Our communications with our industrial partner allowed us to identify several Java projects that we felt to be among the most interesting and challenging ones when they are to be integrated (or are already integrated) as components.

Project	**Total commits**	**Age (years)**	**Avg. commits (per year)**	**Total contributors**	**Current size (KLoC)**	**Total CVEs**
Apache Tomcat (v6-9)	15730	10.0	1784	30	883	65
Apache ActiveMQ	9264	10.3	896	96	1151	15
Apache Camel	22815	9.0	2551	398	959	7
Apache Cxf	11965	8.0	1500	107	657	16
Spring Framework	12558	7.6	1646	416	997	8
Jenkins	23531	7.4	2493	1665	505	56
Apache Derby	7940	10.7	742	36	689	4

Table 3.1 describes several popular (both externally and internally) Java projects that we are allowed to directly disclose. Our communications with developers of our industrial partner suggested that these projects are among the most interesting and challenging ones when they are to be integrated (or are already integrated) as components. For each project, the table lists various characteristics that describe its popularity and size, as well as the number of historical vulnerabilities that affect different versions of Java sources.

3.3 FOSS Components Approval Processes

To address the second question that concerns the processes for selection of FOSS components, we identified how this selection is managed, and which are the critical roles and activities connected which the selection process.

Our industrial partner has formal processes in place for integration of third-party FOSS components into its products (inbound approval), as well as for releasing their own software products as FOSS, and contributing to already existing FOSS projects (outbound approval). These processes are similar to the inbound and outbound approval processes described by Goldman and Gabriel [65, Chapter 7]: they also start with legal and business case checks, as well as identification and assessment of various risks. These risks may include potential intellectual property infringement, possible lack of support from FOSS communities, and the quality of the source code and the corresponding documentation that is intended as a contribution to FOSS communities. Additionally, our industrial partner has implemented checks for potential security risks for both processes.

Typically, both inbound and outbound approval processes require vast expert knowledge, therefore for different phases of these processes different experts may be involved: for instance, the security checks are mostly carried out by the Central Security Team, while legal checks are performed by the legal department. However, all phases of both inbound and outbound processes may be carried out by the same experts (or teams of experts), as there is no strict requirement that they should be separated.

3.3.1 Outbound FOSS Approval Process

The *outbound* process is started when a product group either wants to release a component using a FOSS license, or to contribute to a FOSS project. This process includes, among others:

- *Legal and license check:* is the license chosen to publish the product compatible with the dependencies and/or the license of the project to which the contribution will be made?
- *Business case check:* can this contribution under a FOSS license be justified from a business perspective? Are the interactions with the FOSS communities well defined? Are there internal resources for the maintenance and support?
- *Security check:* does this contribution comply to the Product Security Standard[5]? It must have the same level of security assurance as the rest of the products, since its quality directly reflects on the company.

3.3.2 Inbound FOSS Approval Process

The inbound process is started by software product groups that intend to integrate a FOSS component in the product they are developing. They must perform a request and specify how the component will be integrated, as well as which functionality of the component will be used. Among other activities, this process includes the following:

[5] See Brucker and Sodan [29] for more information about the Product Security Standard used by our industrial partner.

- *Legal and license check:* is the license of a FOSS component compliant to the license of a product into which it is integrated? Are all requirements of the license (e.g., contributing patches back to the community) understood and documented?
- *Business case check:* what is the significance of the technology that will be integrated? Are there viable alternatives? How this technology will be distributed and maintained? In comparison to the *outbound* approval process, the business case check here is more lightweight.
- *Security check:* does the component comply to the Product Security Standard? Are there unpatched security vulnerabilities that may affect customers? Are the security patches provided by the FOSS developers in a timely manner?

Yet, with respect to the *inbound* FOSS approval process, security checks are an ingredient, but not the main decision point. First, certain FOSS components may be the de-facto standard (e.g., Apache Hadoop[6] for big data), so that the end customers of our industrial partner may expect them to be used. Second, FOSS components may offer functionalities that are very expensive to re-implement, so FOSS it is the most economical choice [90], and there might be only one FOSS component with the desired functionality to choose from. Finally, a component that is better in terms of security in comparison to the similar ones, may not fit because of a restrictive license.

3.3.3 Security Checks Revisited

For both inbound and outbound approval processes, it is required to perform security checks for ensuring that the FOSS component adheres to the same standard as the rest of the products of our industrial partner. This is rather easy to achieve for FOSS contributions (outbound), as this code was being developed by the internal developers using the *SDL*. In contrast, for the consumed third-party FOSS components, assessing the compliance to the Product Security Standard is difficult, as development teams usually have very limited knowledge about the (secure) development process used to develop a FOSS component and, moreover, often lack detailed knowledge about the actual implementation. Still, to ensure the security of the products offered by our industrial partner, including consumed FOSS components, the general guidance is to treat third-party FOSS components as own coding with respect to security.

Since 2010, static application security testing (SAST) is widely used by the developers of our industrial partner (we refer to Brucker and Sodan [29] for more details), and there is a number of SAST tools used in the industry as a whole (such as HP Fortify[7], Synopsis Coverity[8], or Checkmarx[9]) as a part of the inbound approval process by development

[6] `https://hadoop.apache.org/`

[7] `http://www.fortify.com`

[8] `http://www.coverity.com`

[9] `http://www.checkmarx.com`

teams that request FOSS components. If the analysis of findings of SAST tools shows that there are exploitable vulnerabilities, mitigation measures are put in place. The latter could, for instance, result in developing a fix for the FOSS component, or implementing certain usage restrictions (or workarounds) such as white-listing of user input before it reaches the FOSS component. As the analysis of SAST results requires a solid understanding of the source code, the architecture and the use case of the FOSS component under analysis, the requirement to statically analyze all FOSS components is a big burden to the development teams.

3.4 FOSS Maintenance And Response

After we identified how the choice of FOSS components is carried out, our final task was to understand the relative importance of the security maintenance of chosen FOSS components, as well as how the maintenance is managed.

The maintenance activities have a significant economic impact that is often not perceived by the "lay users", as they are used to the "monthly upgrade" process of web browsers and their plug-ins. Large-scale enterprise software, such as ERP systems, or industrial control systems are the back-bone of the businesses and, thus, enterprise software customers are often rather conservative in upgrading (or replacing) their on-premise software solutions.

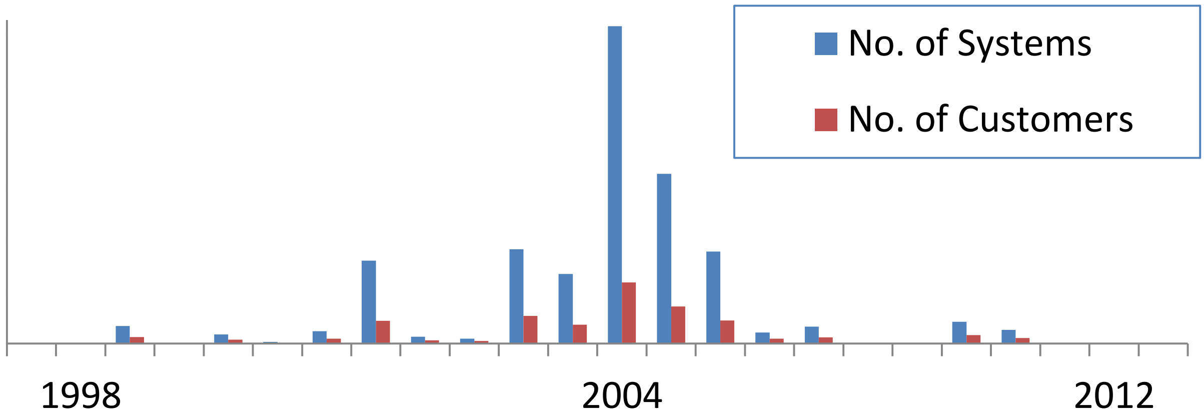
Figure 3.2 shows the distribution of customers, as of 2014, of a large on-premise proprietary product: the y-axis shows the number of customers (systems), and the x-axis shows the year in which a certain version of the software was released: most customers were using systems that are between eleven and nine years old. To meet the demands of the customers, our industrial partner offers support and maintenance, and for selected products – mainstream support for a number of years and, additional, customer-specific extended maintenance for several additional years. Thus, all third-party components (including FOSS) must be as well supported with security fixes for the same amount of time.

As customers expect support and maintenance for the complete software solution, our industrial partner must also ensure maintenance for all integrated third-party components. This includes security fixes for all such components that require to upgrade or modify the product (resulting in a security upgrade or a patch that fixes, e.g., Heartbleed[10] or POODLE[11]), but also issuing articles and security notes that inform customers about fixing security issues in the environment their system is operated on (e.g., recommending upgrades of a Linux distribution that customers might use to operate the system).

For pure cloud offerings (e.g., Software-as-a-Service), the situation can be the opposite:

[10] http://heartbleed.com/

[11] https://www.oracle.com/technetwork/topics/security/poodlecve-2014-3566-2339408.html



The distribution (as of 2014) shows #systems (blue) and #customers (red) using the releases of an ERP application that are released in the year x (values on the y-axis are omitted for confidentiality). In 2014, most customers used versions (and corresponding FOSS components) that were between 9 and 11 years old.

Figure 3.2: ERP application releases and #customers still using them

cloud offerings usually have rapid release-cycle and, thus, do not require a long maintenance phase – it is more important here whether a consumed third-party component can be upgraded easily.

As integrated FOSS components may be heavily modified or merged into the code base of the in-house software products (or integrated prior to the existence of a software inventory), there exists a problem of identifying the FOSS components that are used across the software portfolio. For instance, the study by Davies et al. [50] discusses the importance of identification of open source Java components, as well as proposes effective heuristics to identify them.

To mitigate this problem, our industrial partner has a FOSS component inventory, which was created using the Black Duck[12] solution. We learned that developers of our industrial partner use the high-level information provided by this solution and similar sources to learn about characteristics of FOSS components and make decisions about them. This information is easily available (at least internally), and contains data such as the age of a FOSS project, the information about its historical vulnerabilities (mostly taken from the NVD), and various cumulative data that can be extracted (not without an effort) from the source code repositories of these projects: the current size of their code bases, the number of contributors, commits, and similar. We as well used this software inventory to extract the data on the most popular FOSS projects that we discussed in Section 3.2.

Table 3.2 summarizes the vulnerability types reported for these FOSS components in the National Vulnerability Database (NVD). This distribution suggests that the most prevalent historical vulnerability type is denial of service – the absence of such vulnera-

[12] `http://www.blackduck.com`

Table 3.2: Historical vulnerabilities of 166 FOSS components

The table shows the distribution of historical vulnerability types in the sample of the most popular FOSS projects used or requested by our industrial partner. The distribution suggests that denial of service was the most prevalent vulnerability – the absence of such vulnerabilities is critical for business software solutions.

Vulnerability type	Portion	Vulnerability type	Portion
Denial of Service	30.8%	Gain Privileges	3.1%
Code execution	20.3%	Directory Traversal	2.4%
Overflow	16.6%	Memory Corruption	2.2%
Bypass Something	10.3%	CSRF	0.9%
Gain Information	7.1%	HTTP response splitting	0.3%
XSS	5.9%	SQL injection	0.1%

bilities is critical for business software solutions that must be constantly available online. Also, vulnerabilities of this type may be particularly hard to identify with conventional static analysis [36]. Given the numbers of internal projects that use and request FOSS components, the problem of their security maintenance becomes of great importance.

3.5 Preliminary Findings

Our discussions with software developers, security and maintenance experts of our industrial partner allowed us to identify the heuristics and best practices that development teams and product owners are following to support secure integration of FOSS components into their software supply chain. We base our further line of work on some of these insights, however we do not deal with most of the organizational measures.

We split these findings into the following two parts: (1) a checklist for product owners and developers that they follow both when selecting a FOSS component, and when integrating the component into a software product, and (2) organizational and process improvements that provide the overall environment for using FOSS securely.

3.5.1 FOSS Integration and Maintenance Checklist

We have identified the following points that our industrial partner considers for secure selection of FOSS components (*inbound* FOSS approval process that we describe in Section 3.3.2), as well as for integration with new or existing software products when considering future maintenance (the *utilization* phase of the SDL process that we describe in Section 3.2):

- *How widely is a component used within the software portfolio of our industrial partner?* Those components that have been already used in some products require lower effort, as licensing checks are already done, and internal technical expertise can be tapped. Thus, the effort for fixing issues or integrating new versions can be shared across multiple development teams.

- *Are the technologies used in a FOSS project familiar to the development teams?* Similarly to the above point, if there exists an internal technical expertise in certain programming languages, frameworks, and development processes, it can be re-used for lowering the effort of integration and future support of a FOSS component.
- *What is the maintenance lifecycle used by the FOSS components?* It may be important to consider the planned support for FOSS components provided by their own developers. For example, if the security maintenance support provided by the FOSS community "outlives" the planned maintenance lifecycle of the consuming proprietary product, only the integration of minor releases into proprietary releases would be necessary.
- *How active is the FOSS community?* Consuming FOSS from active and well-known FOSS communities (e.g., Apache) should allow development teams to leverage external expertise, as well as to benefit from externally provided security fixes.
- *Are FOSS maintainers providing information about security issues and secure development processes?* Explicit and up-to-date information about security issues in FOSS projects provided by FOSS maintainers facilitates timely issue resolution for all consumers. The availability of information about security testing processes within FOSS projects may facilitate external security assessment by consumers: for instance, FOSS project maintainers may explicitly state that its developers are using certain security testing tools, such as public offerings from Coverity (`http://scan.coverity.com`).
- *Does a FOSS component have a large number of dependencies?* This is a factor that becomes even more important after a FOSS component was already selected and integrated, since *all* such dependencies have to be maintained. Ideally, all the above considerations should also apply to the *transitive closure* of dependencies of a FOSS component that is to be integrated.

Additional elements, such as license compatibility, or requests from customers that need integration with their code are also important. However, we do not consider them in this study. Licenses follow different principles and can be a separate subject for a dissertation in business and law. For further discussion on alliances for software production see [159], or [134] on legal issues and licensing models.

3.5.2 FOSS Organizational and Process Measures

From an organizational perspective, we have identified the following processes and campaigns that are used by our industrial partner, and are relevant to the goal of secure integration of FOSS components into the software supply chain:

- *FOSS approval processes are defined and used.* Clear definition of the approval processes that describe the necessary steps for using FOSS components, as well as contributing to FOSS, help to make informed decisions about FOSS components, to avoid unnecessary risks (e.g., intellectual property, or security), as well as to maintain the software inventory that mitigates risks that usually may in the consumption phase (e.g., security maintenance). These approval processes typically cover at least the license checks, as well as the security and maintenance checks (inbound), and a compliance check of the security patch strategy (outbound).

- *Software inventory is implemented and is regularly updated.* A software inventory that contains information about FOSS components used and their corresponding versions allows to track the usage of FOSS components through the entire portfolio of software products. On the other hand, the absence of such an inventory may lead to the problems of identifying the components and corresponding versions of these components that constitute a software product – this significantly complicates timely resolution of security issues. Ideally, this inventory should be managed automatically, by analyzing the build system or the binaries of the final product (for instance, using approaches similar to the work of Davies et al. [50]).
- *Vulnerability databases are monitored for new vulnerabilities in used FOSS components.* Regular monitoring of public vulnerability databases and project-specific vulnerability data sources for newly disclosed security vulnerabilities in the consumed FOSS components allows to identify the information that can be used by developers to timely fix security issues. This also allows to timely inform customers about relevant security issues in third-party FOSS components, so that they can plan their actions.
- *A maintenance strategy that fits the current FOSS usage model is defined and used.* This could range from buying maintenance and support for FOSS components from third-party vendors, to applying local maintenance when a team that is consuming a FOSS component is responsible for resolving security issues in them, or to introducing a centralized maintenance model for the entire company, or a mixture of the aforementioned options. Having a clear maintenance strategy helps to save precious development resources. We base the maintenance models that we discuss in Chapter 6 on this insight. Additionally, a maintenance strategy needs to have particular focus on applying security fixes in timely manner (e.g., down-porting fixes, or upgrading), as discovery of new vulnerabilities is close to impossible to plan beforehand. Another important issue that emerged is that there exists a possibility that FOSS components will be maintained and supported in-house exclusively (e.g., due to the lack of support from FOSS developers). Therefore, accessing these components in the source code form may be more preferable.
- *An awareness campaign about FOSS components is run.* Regular internal seminars or trainings for developers and product owners on FOSS components, that explain

the open source software licenses, as well as the associated implications for the effort of secure integration and maintenance may help to ensure that the above processes are running as effective as possible.

3.6 Conclusions

The main goal of this study was to identify the current status of handling FOSS components within the software supply chain of our industrial partner, as well as to identify the most important problems in securing these components that require attention the most.

From our communications with software developers and security experts, we understood that the SDL process of our industrial partner consists of several phases, and that the security of FOSS components is considered throughout the most of them. While the SDL process dictates that FOSS components should receive the same treatment in all these phases with no difference to the in-house coding, this rule is difficult to enforce due to the lack of in-depth knowledge about every component by the in-house software developers, and the large number of integrated components and their versions.

We observed that our industrial partner is using *inbound* and *outbound* FOSS approval processes that aim to identify various risks that are relevant to either contributing to FOSS projects (or releasing in-house coding under an open source software license), or to integrating FOSS components into own products. These two processes consist mainly of assessing legal, business, and security risks, and may be carried out by different experts within the company. Note that for the *inbound* approval process, the security checks may be only a part of the FOSS selection problem besides legal and business considerations.

However, the importance of the security characteristics of FOSS components may become more apparent at the *utilization* phase of the SDL process, specifically, during the maintenance period of a software product. From what we understood by questioning developers and software experts, security maintenance of these components (updating different versions of a product because of security issues in FOSS, or providing a custom fix) generates a significant amount of effort for developers due to the sheer number of integrated FOSS components, and their different versions. This is also aggravated by the potential lack of expertise from the in-house developers on every FOSS component that is being used.

As we looked at the software maintenance processes of our industrial partner in general, we understood the utmost importance of the security maintenance of third-party FOSS components, which can potentially generate significant amount of effort for the in-house developers and software maintenance experts. Therefore, the goal of minimizing these efforts throughout the entire software product lifecycle (FOSS approval process, SDL process, as well as maintenance and support processes) motivated the main problems that we tackle in this dissertation (we outline them in Chapter 1).

Chapter 4

TestREx: a Testbed for Repeatable Exploits[1]

In this chapter we describe our solution to Problem 1 that lies in understanding whether existing exploits for disclosed security vulnerabilities can be reproduced in different settings and environments.

4.1 Introduction

Web applications are nowadays one of the preferred ways of providing services to users and customers. Modern application platforms provide a great deal of flexibility, including portability of applications between different types of execution environments, e.g., in order to meet specific cost, performance, and technical needs. However, they are known to suffer from potentially devastating vulnerabilities, such as flaws in the application design or code, which allow attackers to compromise data and functionality (for instance, see [156, 170]). Vulnerable web applications are a major target for hackers and cyber attackers [142], while vulnerabilities are hard to identify by traditional black-box approaches for security testing [44, 94, 161].

A key difficulty is that web applications are deployed and run in many different execution environments, consisting of operating systems, web servers, database engines, and other sorts of supporting applications in the backend, as well as different configurations in the frontend [94]. This difficulty can be illustrated with typical exploits for the two types of web application security vulnerabilities: SQL injection exploits (the success depends on the capabilities of the underlying database and the authorizations of the user who runs it [156, Chapter 9]), and Cross-site Scripting (XSS) exploits (the success depends on a specific web browser being used and its rules for executing or blocking JavaScript

[1] This chapter is the result of a joint work with Daniel Ricardo Dos Santos, a fellow PhD student in the SECENTIS project.

Table 4.1: Available exploits in TestREx corpus

Language	Exploits	Source
PHP	83	BugBox [114]
Java	10	WebGoat [119]
Server-side JavaScript	7	Own

code [170, Chapter 14]). Such differences in software environments may transform failed exploitation attempts into successful ones, and vice versa.

Industrial approaches to black-box application security testing (e.g., IBM AppScan[2]) or academic ones (e.g., Secubat [83] and BugBox [114]) require security researchers to write down a number of specific exploits that can demonstrate the (un)desired behavior. Information about the configuration is an intrinsic part of the vulnerability description. Since the operating system and supporting applications in the environment can also have different versions, this easily escalates to a huge number of combinations which can be hard to manually deploy and test.

We need a way to automatically switch configurations and re-test exploits to check whether they work with a different configuration. Such data should also be automatically collected, so that a researcher can see how different exploits work once the configuration changes. Such automatic process of "set-up configuration, run exploit, measure result" was proposed by Allodi et al. [7] for testing exploit kits, but it is not available for testing web applications.

Our proposed solution, TestREx[3], combines packing applications and execution environments that can be easily and rapidly deployed, scripted exploits that can be automatically injected, useful reporting and an isolation between running instances of applications to provide a real "playground" and an experimental setup where security testers and researchers can perform their tests and experiments, and get reports at various levels of detail.

We also provide a corpus of vulnerable web applications to illustrate the usage of TestREx over a variety of web programming languages. The exploit corpus is summarized in Table 4.1. Some of the exploits are taken from existing sources (e.g., BugBox [114] and WebGoat [119]), while others are developed by us. For the latter category, we focused on server-side JavaScript, because of its growing popularity in both open source and industrial usage (e.g., Node.js[4] and SAP HANA[5]) and, to the best of our knowledge, the lack of vulnerability benchmarks.

[2] http://www.ibm.com/software/products/en/appscan

[3] http://securitylab.disi.unitn.it/doku.php?id=testrex

[4] http://nodejs.org/

[5] https://help.sap.com/hana

4.2 Related Work

Empirical security research has been recognized as very important in recent years [32, 54, 97]. However, a number of issues should be tackled in order to correctly provide security experimentation setups. These issues include isolation of the experimental environment [7, 27, 30, 114], repeatability of individual experiments [7, 54], collection of experimental results, and justification of collected data [97].

The use of a structured testbed can help in achieving greater control over the execution environment, isolation among experiments, and reproducibility. Most proposals for security research testbeds focus on the network level (e.g., DETER [27], ViSe [12], and vGrounds [81]). A comparison of network-based experimental security testbeds can be found in the Master's thesis by Stoner [154]. On the application level there are significantly less experimental frameworks. The BugBox framework [114] is one of them. It provides the infrastructure for deploying vulnerable PHP-MySQL web applications, creating exploits and running these exploits against applications in an isolated and easily customizable environment. As in BugBox, we use the concepts of execution isolation and environment flexibility. However, we needed to have more variety in software configurations and process those configurations automatically. We have broaden the configurations scope by implementing software containers for different kinds of web applications, and automatically deploy them.

The idea of automatically loading a series of clean configurations every time before an exploit is launched was also proposed by Allodi et al. in their MalwareLab [7]. They load snapshots of virtual machines that contain clean software environment and then "spoil" the environment by running exploit kits. This eliminates the undesired cross-influence between separate experiments and enforces repeatability, so we have incorporated it into TestREx. For certain scenarios, cross-influence might be a desired behavior, therefore TestREx makes it possible to run an experiment suite in which the experimenter can choose to start from a clean environment for each individual exploit/configuration pair or to reuse the same environment for a group of related exploits.

Maxion and Killourhy [97] have shown the importance of comparative experiments for software security. It is not enough to just collect the data once, it is also important to have the possibility to assess the results of the experiment. Therefore, TestREx includes functionalities for automatically collecting raw statistics on successes and failures of exploits. We summarize the discussed tools and approaches in Table 4.2.

4.3 Overview of TestREx

TestREx was designed to provide testers with a convenient environment for automated, large-scale experiments. We believe that TestREx is useful for developers as well. To

Table 4.2: Security testing and experimentation tools

The existing tools and approaches provide various functionalities with respect to deployment (e.g., from running on a local virtual machine to providing controlled environments on real hardware). Most of the security research testbeds focus on the network level, while on the application level there are significantly less experimental frameworks.

Tool	**Description**	**Exploit types**
BugBox [114]	A corpus and exploit simulation environment for PHP web application vulnerabilities.	Selenium and Metasploit scripts in Python that exploit PHP application vulnerabilities.
MalwareLab [7]	A controlled environment for experimenting with malicious software.	Programs that exploit various software vulnerabilities or malware kits.
MINESTRONE [56]	A software vulnerability testing framework for C/C++ programs. The applications are deployed in virtualized environments via Linux Containers	Programs that exploit memory corruption, null pointer, number handling and resource leak vulnerabilities in C/C++ software.
DETER [27]	A testbed facility that consists of a large set (around 400) of real machines. The resources infrastructure can be reconfigured on-the-fly upon request.	Programs that exploit various software vulnerabilities or malware kits.
ViSe [12]	A virtual testbed for reproducing and collecting the evidence of security attacks that is based on VMWare virtualization environment.	Multi-level attacks that include network tampering and software vulnerability exploitation.
SecuBat [83]	Web vulnerability scanner, that automatically scans live web sites for vulnerabilities using a web crawler infrastructure.	Specially crafted HTTP requests that exploit SQLi and XSS vulnerabilities.
vGround [81]	A virtual playground for malware assessment, that is created on top of a physical infrastructure - a machine, a cluster or a multi-domain overlay infrastructure.	Malicious software such as virtual worms or malware kits.

support this claim, we give an example of a possible loophole in a bug fixing workflow of a hypothetical company:

- A tester finds a bug and opens a new issue in a bug tracking system. She submits it as a test case described in natural language, explaining all preconditions and steps needed to reproduce the bug.

- A manager assigns the issue to a developer. In order to pinpoint the source of the bug and understand how to fix it, the developer must reproduce the test case in his own setting. If the tester makes a mistake while creating the test case, the developer will be unable to trigger the bug. As a consequence, the developer rejects the fix request.

- In the worst case, it might take a long time before the bug will be re-discovered and eventually fixed. In a better case, more resources are wasted if the tester has to re-describe the bug, and a manager has to re-assign the bug to a developer.

Using TestREx, the tester could create an "executable description" of a bug in the form of a script, and a packed execution environment that allows to instantly replay the situation that triggered the bug. Despite taking longer for the tester to initially describe the bug this way, it has many advantages over the natural language approach. First, the tester and the developer are ensured that the bug can be reproduced. Second, the test case can be kept as a template on which future tests can be developed, i.e., the first test is harder to describe, but future tests can reuse parts of the first one. Third, the test can be automatically added to a library of regression tests, to ensure that the same bug will be detected if reinserted in future versions of the application.

4.3.1 Terminology

Before we proceed, we introduce several concepts that we use for further discussion[6]:

- `Image` – a snapshot of an application configured to run in a certain software environment (e.g., an operating system, a web server, and a database engine) that includes the software environment as well. An image can be instantiated into a container that a tester can interact with.

- `Configuration` – Configurations are used for creating images. We use this term to denote a particular setup for an application and its supporting software components with particular values of setup parameters (configuration files, packages, etc.), as well as a set of instructions that are automatically executed in order to create an image in which these applications and components are "deployed".

- `Container` – an instance of an image. This instance represents a certain state of an application and its software environment, that can be "run" for testing, and dismissed when testing is over. It can be either started using the pristine state of its base image (creating a new container, i.e., instance), or resumed from a certain existing state (re-using a container, that was already instantiated).

[6] Technically, these concepts are implemented using Docker (`https://www.docker.io/`) – we describe the implementation in Section 4.4. However, a different implementation may be obtained using traditional virtual machines to which these general concepts can be applied as well.

4.3.2 Typical workflow

An automated testbed should help security researchers in answering (semi) automatically a number of security questions. Given an exploit X that successfully subverts an application A running on an environment E:

1. Will X be successful on application A running on a new environment E'?
2. Will X be successful on a new version of the application, A', running on the same environment?
3. Will X also be successful on a new version of the application, A', running on a new environment E'?

These questions can be exemplified in the following situation:

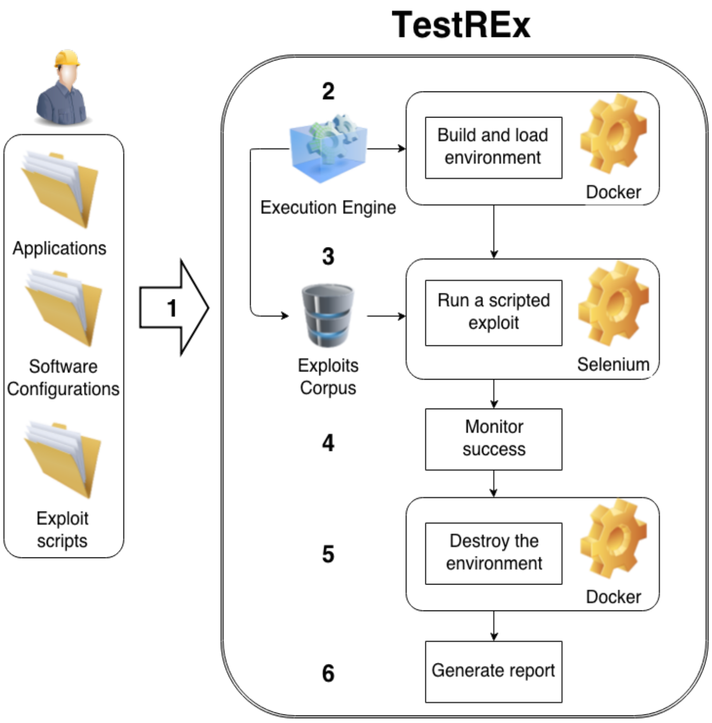
Example 1 *We have a working SQL injection exploit for the WordPress 3.2 application running with MySQL and, we would like to know whether (1) the same exploit works for WordPress 3.2 running with PostgreSQL; (2) the same exploit works for WordPress 3.3 running with MySQL; and (3) the same exploit works for WordPress 3.3 and PostgreSQL.*

We use this example throughout the chapter to illustrate the concepts and components used in the framework.

A key feature that we have sought to implement, is that the architecture of TestREx should be easily extensible to allow for the inclusion of new exploits, applications, and execution environments. Figure 4.1 shows a typical workflow when an application and the corresponding scripted exploits are deployed and run within TestREx:

1. A tester provides the necessary `configuration` for a specific `image`, including the application and software component files, and the scripted exploits to be executed (the latter is optional, as TestREx also supports manual testing).
2. The `Execution Engine` component of TestREx builds the `image` and instantiates the corresponding `container`.
3. The `Execution Engine` runs corresponding exploit(s) against the application container,
4. and monitors whether the exploit execution was successful.
5. After the exploit(s) are executed, the `Execution Engine` dismisses the corresponding container (optionally, further exploits may reuse the same container when the tester wishes to observe the cumulative effect of several exploits) and cleans up the environment.
6. The exploit(s) execution report is generated.

One of the main goals of TestREx is to make the testing process as automated as possible. Another important task is to make it possible to run applications and exploits in a clean and isolated environment. This is why we included the option of resetting the state of an application before running a test – this allows to run tests in parallel (see the point 5 above).



The workflow of TestREx *is straightforward: a tester provides configuration details of an application, its deployment environment, as well as the exploit scripts;* TestREx *automates the remaining actions, such as building and loading the environment, and running and monitoring the exploit.*

Figure 4.1: TestREx workflow

TestREx also includes some additional utilities. For instance, the `Packing Module` allows to package configurations in compressed archive files that can be easily deployed in another system running TestREx. Also, the `Utilities` module includes a collection of scripts to import applications and exploits from other sources, such as BugBox, and to manage the containers.

Example 2 *The inputs for Example 1 are instantiated as follows:*

- ***Application:*** *There are two applications of interest, each one is a set of* `.html`*,* `.php` *and* `.js` *files in a* `Wordpress` *folder.*
- ***Configuration:*** *There are four configurations of interest, one for WP3.2 with MySQL, one for WP3.3 with MySQL, one for WP3.2 with PostgreSQL, and one for WP3.3 with PostgreSQL.*
- ***Image:*** *There are two possible images, one with Ubuntu Linux distribution, Apache web server and MySQL database engine, and one with Ubuntu, Apache and PostgreSQL.*
- ***Exploit(s):*** *There is only one exploit – a script that navigates to the vulnerable web page, interacts with it and injects a payload, simulating the actions of an attacker.*

In our setting, exploits are unit tests: (1) every exploit is self-contained and can be executed independently; and (2) every exploit is targeted to take advantage of a specific vulnerability in a given application.

When using the framework in a specific application, the exploit can be written by the tester or taken from a public source. In any case, the exploit code must be compliant with what we expect from an exploit, e.g., it must be a subclass of the BasicExploit class provided with TestREx, and contain metadata that specifies the target image and describes the exploit script (more details are in Section 4.6.3).

4.4 Implementation

TestREx is implemented in Python, mainly because it allows fast and easy prototyping and because of the availability of libraries and frameworks, such as `docker-py` to interface it with Docker (see below). Below we describe in details the implementation of each component of the framework.

4.4.1 Execution Engine

The `Execution Engine` is the main TestREx module that binds all its features together. It supports three modes of operation: *single*, *batch* and *manual*.

The *single mode* allows testers to specify and run a desired exploit against a container that corresponds to the chosen application image just once. This is useful when the tester wants to quickly check whether the same exploit works for a few different applications, different versions of the same application or the same application deployed in different software environments. A "`.csv`" report is generated at the end of the run.

To run applications and exploits in the *batch mode*, TestREx loops through a folder containing exploit files, and runs them against respective containers, generating a summary "`.csv`" report in the end. In this mode, the `Execution Engine` maps exploits to application images by scanning the metadata in each exploit, where appropriate target images are specified by the tester.

For *manual* testing, the `Execution Engine` instantiates a container based on the chosen application image, and returns the control to the tester (e.g., by opening a web browser and navigating to the application, or returning a shell). No report is generated in this case.

The `Execution Engine` contains an additional setting for handing containers when chosen exploits are executed: it is possible to either destroy a particular container after the execution, in order to start with a "fresh" instance of the image for each exploit run; or to reuse the same container when its state has to be preserved, so that further exploits may have a cumulative effect that the tester wishes to observe.

4.4.2 Applications

Applications are packaged as "`.zip`" files containing all their necessary code and other supporting files, such as database dumps. Unpacked applications must be located under the "`<testbed_root>/data/targets/applications`" folder to be accessible by the `Execution Engine`.

Table 4.3: Applications in the corpus

The table shows the applications (real-world and artificial ones) that TESTREX *currently includes. The "Containers" column specifies a generic container upon which a specific application image is created, and the "Source" column specifies the source from which we adapted exploits for these applications.*

Language	Applications	Containers	Source
PHP	WordPress, CuteFlow, Horde, PHP Address Book, Drupal, Proplayer, Family Connections, AjaXplorer, Gigpress, Relevanssi, PhotoSmash, WP DS FAQ, SH Slideshow, yolink search, CMS Tree page view, TinyCMS, Store Locator Plus, phpAccounts, Schreikasten, eXtplorer, Glossword, Pretty Link	ubuntu-apache-mysql	BugBox
Java	WebGoat	ubuntu-tomcat-java	WebGoat
Server-side JavaScript	CoreApp, JS-YAML, NoSQLInjection, ODataApp, SQLInjection, ST, WordPress3.2, XSSReflected, XSS-Stored	ubuntu-node, ubuntu-node-mongo, ubuntu-node-mysql	Our examples

As an example, we provide some applications with known vulnerabilities (listed in Table 4.3) most of which are known real-world applications, only some of them being small artificial examples developed by us to explore security vulnerabilities typical for server-side JavaScript applications.

4.4.3 Images and Containers

Ideally, security testers should have the possibility of using various types of computing components and platforms, regardless of the type of underlying hardware and software that may be available.

To provide testers with the possibility of running applications in various environments in a flexible, scalable, and cost-effective manner, we employ software images (that are, implementation-wise, Docker images). Every such image represents a data storage for virtualized computing components or platforms, e.g., operating systems, application servers, database management systems, and other types of supporting applications.

Instead of creating virtual machines for applications and their software environments, we instantiate and run containers from corresponding images. These containers are based

Table 4.4: Software components for generic images currently provided with TestREx

Web server	DB engine	OS
Apache	MySQL	Ubuntu
Node.js	MySQL	Ubuntu
Node.js	MongoDB	Ubuntu
Tomcat	MySQL	Ubuntu

on the OCI[7] standards, which are nowadays widely accepted in industry as a form of "lightweight virtualization" at the operating system level. They are sandboxed filesystems that reuse the same operating system kernel, but have no access to the actual operating system where they are deployed.

Some initial developments in this area were FreeBSD Jails[8], Solaris Zones[9], and Linux Containers[10]. Currently, Docker is the *de facto* standard for containers. Docker provides a format for packing and running applications within lightweight file repositories that are called Docker containers. We use Docker to create images and instantiate containers.

Images are specified in Dockerfiles (a format defined by the Docker project) – these files represent *configurations* to which we refer in Section 4.3.1. Downloading generic software components and re-creating a Docker container from a corresponding image every time an application has to be run might be resouce- and time-consuming. Therefore, we use image inheritance supported for Dockerfiles, creating several images for containers that hold generic software components, and can be reused by certain types of web applications. For instance, such images may encapsulate an operating system, a web server and a database engine, and their corresponding containers are instantiated only once. We provide some predefined images for common environments, using software components shown in Table 4.4. We use the following naming convention for such images: "`<operating_system> -<webserver>-<database>-<others>`". In contrast, for images which actually contain an application to be tested (apart from generic software components) we use a different naming convention: "`<application-name>__[software-image-name]`".

When the `Execution Engine` invokes an application image, the corresponding container will be instantiated and run using Docker. Then, depending on the run setting (see Section 4.4.1), the container will be handled correspondingly when chosen exploits are executed (either destroyed, or reused for further exploit runs).

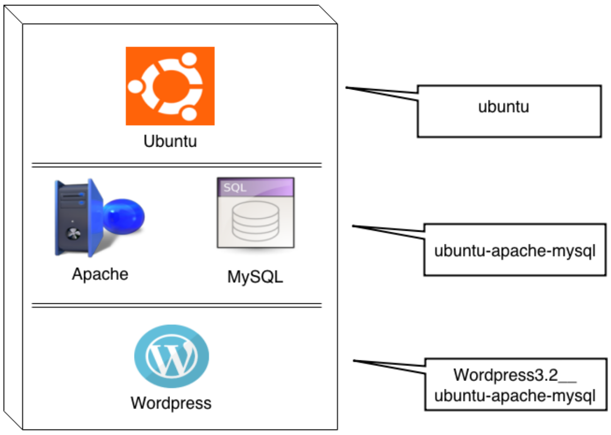
Figure 4.2 gives an intuition on how an image for the *WordPress 3.2* application can be composed with Dockerfiles: the image is created on the basis of two images combining Ubuntu OS with Apache web server and MySQL database.

[7] `https://www.opencontainers.org/`

[8] `https://www.freebsd.org/doc/handbook/jails.html`

[9] `https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm`

[10] `https://linuxcontainers.org/`



Application images are composed of several "layers": an operating system, a web server, and a database engine – the application itself is deployed on top. These components can be combined in all possible configurations supported by the application.

Figure 4.2: *Wordpress3.2__ubuntu-apache-mysql* image

4.4.4 Configurations

Implementation-wise, configurations correspond to the contents of Dockerfiles and supporting scripts that specify how an application can be installed and run in a container, including, e.g., prerequisites such as preloading certain data to a database, creating users, and starting a server. Additionally, configuration data for applications may include databases and application data.

The configuration files must be placed in a separate folder under the configurations root folder ("`<testbed_root>/data/targets/configurations`"). We use the following naming convention to simplify matching configuration files with images that can be created using them: "`<app-name>__<app-container-name>`".

Example 3 *A configuration folder for the application "`Wordpress_3.2`", might have the names "`Wordpress_3.2__ubuntu-apache-mysql`" or "`Wordpress_3.2__ubuntu-apache-postgresql`", depending on the image that is intended for it.*

Listings 4.1 and 4.2 present an example of a Dockerfile and a "`run.sh`" file, used to configure a *WordPress 3.2* application within the "`ubuntu-apache-mysql`" image.

In Listing 4.1, line 1 specifies that the image for this application is built on top of the "`ubuntu-apache-mysql`" image. In lines 2 and 3, the application files are copied to the "`/var/www/wordpress`" folder in the image and in lines 4 and 5, the "`run.sh`" script is invoked inside the container.

```
1 FROM ubuntu-apache-mysql
2 RUN mkdir /var/www/wordpress
```

```
ADD . /var/www/wordpress
RUN chmod +x /var/www/wordpress/run.sh
CMD cd /var/www/wordpress && ./run.sh
```

Listing 4.1: Dockerfile example

```
1 #!/bin/bash
2 mysqld_safe &
3 sleep 5
4 mysql < database.sql
5 mysqladmin -u root password toor
6 apache2ctl start
```

Listing 4.2: Shell script file example

In Listing 4.2, lines 2-5 are used to start the database server and pre-load the database with application data. Line 6 starts the Apache web server.

4.4.5 Exploits

Table 4.5 shows the classification of typical security flaws that might be present in both client- and server-side parts of a web application[11], which can be tested with TestREx.

In TestREx, exploits may be any executable file that, when executed in a specified context, provide testers with unauthorized access to, or use of functionality or data within that context. Exploits include any sequence of steps that must be taken in order to cause unintended behavior through taking advantage of a vulnerability in an application and/or surrounding environment. For example, exploits may be used to provide access to sensitive data, such as financial data or personal data. Exploits may hijack capabilities or other functionalities of applications and cause the applications to perform tasks that are not desired by authorized users, such as tracking user activities and reporting on these to the unauthorized user of the exploit. Other types of exploits may allow unauthorized users to impersonate authorized users.

Still, the above description of an exploit is quite vague, and may lead to having many automated exploit scripts that are not compatible due to various differences in their implementation (e.g., as a consequence it may be difficult to run them in a batch, and/or use them to produce a unified testing report). To avoid these potential problems, we implemented exploits as a hierarchy of Python classes that have the following minimal set of properties: (1) every exploit contains metadata describing its characteristics such as name, description, type, target application and container; (2) exploit classes must pass logging information and results of the run to the `Execution Engine`, providing a way for the `Execution Engine` to know that the exploit execution was successful.

[11] This classification is according to the OWASP TOP 10: `https://www.owasp.org/index.php/Top_10_2013-Top_10`

Table 4.5: Security flaws of web applications

The following security flaws may be present in web applications regardless of their implementation and deployment details. Yet, their successful exploitation strongly depends on the actual variant of deployment (e.g., MongoDB versus MySQL database, type and version of the web server, etc.).

Security flaw	**Description**	**Tech. impact**
SQL/NoSQL injection (SQLi/NoSQLi)	User input is used to construct a database query and is not properly sanitized, allowing a malicious user to change the intended database query into an arbitrary one. **Threats: Information Disclosure, Data Integrity, Elevation of Privileges**.	Severe
Code injection	Similar to SQLi/NoSQLi, however, instead of a database, user input is executed by a code/command interpreter. Malicious payload can be executed on both client and server, and may result into a complete takeover of the host machine on which the vulnerable application runs. **Threats: Information Disclosure, Data Integrity, Elevation of Privileges, Host Takeover**.	Severe
Cross-site scripting (XSS)	Each time a user-supplied data is being displayed in a web browser, there is a risk of XSS attacks: attacker can supply JavaScript code that either gets executed in a victim's browser and stealing victim's credentials or making actions on her behalf. Almost any source of data can be an attack vector (e.g., direct user input, data coming from a database, etc.). **Threats: Information Disclosure, Elevation of Privileges**.	Moderate
Cross-site request forgery (CSRF)	CSRF attacks take advantage of benign applications that allow attackers to act on their behalf: user is secretly redirected from a trusted page to attacker's page, and user's authentication information is used by an attacker. Applications that allow manipulations with DOM container of its pages are vulnerable. **Threats: Session/Credentials Hijacking**.	Moderate
Unvalidated URL redirects	URL redirects instruct the web browser to navigate to a certain page. While this feature can be useful in many different contexts, developers should be careful and restrict user manipulations with a destination page: an attacker may conduct phishing attacks using a trustworthy website that has this vulnerability. **Threats: Open Redirect**.	Moderate
Sensitive data disclosure	Sensitive/Personal data is attractive for attackers per definition, therefore the goal of most of attacks is to get a piece of such data. Since personal data is usually protected by law regulations, every such data flow in a web application must be protected against injection and interception attacks, as well as overly detailed error messages and application logic flaws that disclose context information to potential attackers. **Threats: Information Disclosure**.	Severe
Test code leftovers	A tester may insert a piece of testing code into the application and forget to remove it upon release. This can lead to any kind of unexpected behavior: for example, anyone could get access to the application with a login 'Bob' and a password '123' gaining full administrator access. Such forgotten pieces of test code are indistinguishable from maliciously crafted backdoors per se. **Threats: Backdoor**.	Severe
Using known vulnerable components	If vulnerable versions of third-party components are used (e.g., an open source library) in a web application, an attacker can identify known vulnerabilities and perform a successful attack. In many cases, developers are not aware of all components they are using for their application. Vulnerable component dependencies aggravate the problem. **Threats: Potentially all of the above**.	May vary

We also incorporate the *Selenium Web Driver*[12] for implementing exploits, as it can be used to simulate user/attacker actions in a web browser, and provides all necessary means to automate them. Additionally, it supports JavaScript execution and DOM interaction [114]. Every Selenium-based exploit in the framework is a subclass of the `BasicExploit` class, which encapsulates basic Selenium functionality to automate the web browser (e.g., "`setUp()`" and "`tearDown()`" routines, logging and reporting, etc.). To create a new exploit, the tester has to create a new exploit class, specify the exploit-specific metadata and override the "`runExploit()`" method by adding a set of actions required to perform an exploit. The success of an exploit run is also verified within the "`runExploit()`" method - this might be different for every exploit – this allows us to handle complex exploits that are not always deterministic. For such cases, the exploit can be specified to run a certain number of times until it is considered a success or a failure.

The current implementation of TestREx exploits allows testers to interact with a vulnerable web application via a web browser, and is targeted at attacking (and changing the state of) that web application only. However, it is possible to develop other classes of exploits for exploring more complex scenarios that allow to go beyond testing for the presence of a vulnerability using a web browser. For instance, a vulnerability in a web application may be used as an entry point for installing a backdoor to the machine where this application is deployed (e.g., chaining the path traversal vulnerability in Apache Tomcat *CVE-2008-2938* to obtain a local web server's account, and obtain the root shell using another vulnerability of Apache Tomcat – *CVE-2016-1240*).

4.4.6 Report

Different context conditions may transform failed exploit attempts into successful ones, and vice versa. A given exploit test may include a number of possible combinations of applications, execution environments, and exploits, each of which may be configured in various ways. For example, an exploit that may be successful in exploiting a first application in a first environment may not be successful in exploiting the same application in a second environment, but may be successful in exploiting a second application in the second environment. Moreover, upon determining a success of a given exploit, it will be necessary to make some change to the application and/or execution environment, which will necessitate yet another testing (re-testing) of the previously successful exploit to ensure that the change will prevent future successful exploits.

Therefore, we include reporting functionality: whenever TestREx runs an exploit, it generates a report that contains the information about its execution. A report is a "`.csv`" file that the `Execution Engine` creates or updates every time it runs an exploit. Every report contains one line per exploit that was executed. This line consists of the exploit

[12] `http://docs.seleniumhq.org/projects/webdriver/`

and the target application names, identifier of an application-specific container, the type of the exploit, the exploit start-up status, the exploit execution result, and a comment field that may include other information that might be exploit-specific. Along with this report, the `Execution Engine` maintains a log file that contains information which can be used to debug exploits.

Example 4 *The listing below shows a single entry from the* Wordpress_3_2_XSS *exploit that was run against the* WordPress 3.2 *application.*

```
Wordpress_3_2_XSS, Wordpress3.2, ubuntu−apache
−mysql, XSS, CLEAN, SUCCESS, SUCCESS, 30.345,
Exploits for ”XSS vulnerability in WordPress app”
```

Listing 4.3: An example of the report file entry after the exploit run

4.5 Evaluation

As a starting point in evaluation our framework, we successfully integrated 10 examples from WebGoat [119], as well as the corresponding vulnerable web application. We have also developed exploits for 7 specially crafted vulnerable applications, in order to demonstrate different types of exploits for SQL injection, NoSQL injection, stored and reflected XSS, path traversal and code injection vulnerabilities in server-side JavaScript applications. The path traversal and the code injection examples take advantage of vulnerabilities discovered in Node.js modules [115, 116].

Table 4.6: Number of exploits in the corpus

The table lists the number of exploits in the current corpus of TestREx*, broken down by a vulnerability type and a programming language of the vulnerable portion of the source code that makes the exploitation possible.*

Exploit	**#PHP**	**#Java**	**#Server JS**
Cross-site scripting	46	2	3
SQL injection	17	2	1
Code injection	7	-	1
Authentication flaw	4	3	-
Information disclosure	2	-	-
Local file inclusion	2	-	-
Cross-site request forgery	2	-	-
Denial of service	1	-	-
Database backdoor	-	1	-
Parameter tampering	-	2	-
Path traversal	-	-	1

The framework also supports the possibility of importing applications and exploits from BugBox, and similar testbeds. An automated script copies the applications and exploits into the corresponding folders under the framework, and creates identical configuration files for every application, using Apache as a web server and MySQL as a database server. We are able to run most of the BugBox native exploits and collect statistics without modifying their source code.

Table 4.6 summarizes the types of exploits that we tested in various applications using TestREx: we can see that TestREx supports a variety of typical web application security flaws.

4.6 Contributing to TestREx

Here we describe in more detail the steps needed to add an experiment to TestREx, given an existing application. These steps consist of: adding an application; creating configuration files for images; instantiating containers; creating and running exploits. Again, we use *WordPress 3.2* as the example application.

4.6.1 Deploying an Application

We again use *Wordpress 3.2* as an example. The code of the application must be copied into a separate folder under the applications root "`<testbed_root>/data/targets/applications`". The folder name must correspond to a chosen name of the application in the testbed.

To deploy the *WordPress 3.2* application, copy all of its files to the folder "`<testbed_root>/data/targets/applications/WordPress_3_2`".

4.6.2 Creating Configuration Files and Building Containers

If there are no generic images that might be reused for creating a new image for the application set up, this image must be created in the first place. Configuration files for generic images are located under the "`<testbed_root>/data/targets/containers`" folder.

In our example, we create a generic image with the `ubuntu-apache-mysql` name, since the application requires *Apache* as a web server and *MySQL* as a database engine. To do this, we create a Dockerfile under "`<testbed_root>/data/targets/containers/ubuntu-apache-mysql`" that contains the code shown in Listing 4.4, and build it with the script located under "`<testbed_root>/util/build-images.py`".

```
FROM ubuntu:raring
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get -y install mysql-client mysql-server
    apache2 libapache2-mod-php5 php5-mysql php5-ldap
RUN chown -R www-data:www-data /var/www/
EXPOSE 80 3306
```

```
CMD ["mysqld"]
```

Listing 4.4: The Dockerfile for creating the `ubuntu-apache-mysql` generic image

As a next step, we create configuration files for the image that will hold the application, extending the above generic image. We create a new Dockerfile and a shell script file under the "`<testbed_root>/data/targets/configurations/Wordpress_3_2__ubuntu-apache-mysql`" folder (see Listings 4.1 and 4.2 in the Section 4.4.4 for the code examples).

There is no need to manually invoke Docker for instantiating a container based on this image for running exploits or manual testing, as `Execution Engine` does it automatically.

4.6.3 Creating and Running an Exploit

Finally, we create an exploit for the *Wordpress 3.2* application by creating a Python class under the "`<testbed_root>/data/exploits`" folder. As mentioned in the previous sections, to ensure integration with the `Execution Engine`, the new exploit class must be a subclass of the already existing `BasicExploit` class. As a last step, we specify the exploit's metadata using the `attributes` dictionary, and specify the steps required to run the exploit within the "`runExploit()`" method (see Listing 4.5).

```
from BasicExploit import BasicExploit
class Exploit(BasicExploit):
  attributes = {
      'Name' :          'Wordpress_3_2_XSS',
      'Description' : "XSS attack in Wordpress 3.2",
      'Target' :        "Wordpress3.2",
      'Container': 'ubuntu-apache-mysql',
      'Type' : 'XSS'
   }

   def runExploit(self):
      w = self.wrapper
      w.navigate("http://localhost:49160/wordpress/wp-admin/post-new.php?post_type
   =page")
      ‘‘‘
      ‘‘‘
      content_elt = w.find("content").clear()
      content_elt.keys("<script>alert(\"XSS!!\")</script>")
      w.find("publish").click()

      w.navigate("http://localhost:49160/wordpress/?page_id=23")
      alert_text = w.catch_alert()
      self.assertIn("XSS", alert_text, "XSS")
```

Listing 4.5: Wordpress_3_2_Exploit.py file contents

Listing 4.5 shows the stored XSS exploit for the *Wodpress 3.2* application. The script navigates to the login page of the *Wordpress* application, logs in as the administrator (the full list of steps is shortened in the listing for the sake of brevity), and creates a new post putting the `<script>alert('XSS')</script>` string as the content. To verify whether the exploitation was successful, the script navigates to the newly created post and checks if an alert box with the “XSS” message is present.

```
#1: Manual mode
sudo python run.py --manual --image
    [app-name]__[image-name]

#2: Single exploit mode
sudo python run.py --exploit [exploit-name].py
    --image [app-name]__[image-name]

#3: Batch mode for a single application
sudo python run.py --batch
    --image [app-name]__[image-name]

#4: Batch mode for all applications
sudo python run.py --batch
```

Listing 4.6: Running modes in TestREx

Listing 4.6 shows the list of commands for different running modes in TestREx:

1. To run the application container for *manual* testing, a tester has to use the “–*manual*” flag and the corresponding *application* image. TestREx will run the container and halt, waiting for the interrupt signal from the tester. In this mode, when the container is up, the application can be accessed from a web browser by navigating to “http://localhost:49160”.
2. In the *single mode* a tester can select a specific exploit and run it against a specific *application* image.
3. In the *batch mode for a single application*, a tester has to specify the running mode as “*–batch*”, and select the desired *application* image. TestREx will invoke a Docker container for the image, search for the exploits that are assigned to the application (through exploits' metadata), and run all of them one by one.
4. Finally, if a tester specifies nothing but the “*–batch*” running mode, TestREx will invoke containers for all *application* images that are currently in the corpus, and run all corresponding exploits against them.

By default, the exploit execution report is saved into the “`<testbed_root>/reports/ExploitResults.csv`” file. In order to specify a different location for the results, the tester may add an additional parameter to the `run` command:
`--results new/location/path.csv`.

4.7 Potential Industrial Application

There are several uses of TestREx that we are exploring in an industrial setting, covering different phases of the software development lifecycle and fulfilling the needs of different stakeholders. In the following we summarize the directions that we deem more promising.

4.7.1 Support for Internal Security Testing and Validation

Automated validation and regression testing. As a part of the software development lifecycle, TestREx can be used to check for the absence of known vulnerabilities or to perform regression tests (for instance, as a part of automated Continuous Integration processes) to verify that a previously fixed vulnerability is not introduced again. In large corporations, the results of these tests are part of the evidence needed in order to pass quality assurance gates. Currently, the process of producing such evidence is mostly relying on manual work, which increases the costs, potential errors, and decreases the predictability of the final result. To this end, TestREx can be used to accelerate and improve the effectiveness and the predictability of quality assurance processes: a company can maintain a corpus of exploits and configurations stored in a corporate-wide repository (updating configurations and exploits on a periodic basis if necessary), and use it to perform automated tests all along the development cycle.

Support for penetration testing. An important problem arising in penetration testing of large systems is the complexity of setting-up and reproducing the conditions of the target system – typically involving many hosts and software components, each of which may need to be configured in a specific way. A key strength of our framework is the ability to capture these configurations as reusable scripts; this requires a non-negligible effort, but the results can be reused across different pen-testing sessions. This has the advantage of providing automation, reproducibility, and the ability to proceed stepwise in the exploration of the effect of different configurations and versions of the software elements on the presence (or absence) of vulnerabilities in the system.

4.7.2 Support for Testing of Third-parties Applications

Security testing of cloud-based applications. One valuable use of TestREx is for cloud-based applications. In this scenario, a Cloud Service Provider (CSP) provides the platform on which an Application Provider (AP) may run their applications. CSPs allow the same application to be provided in different platforms. However, such variations in context correspond to potential difficulties in ensuring reliable and complete security testing, because successful protection against an exploit in one context may prove unsuccessful in another context. In this setting, TestREx can provide highly adaptable, flexible, efficient, and reliable testing for different configurations, without requiring highly-specialized knowledge or abilities on the part of the security tester. For example, the security tester may be an employee of a CSP, which must provide evidence to the AP that the CSP platform is secure. In turn, the security tester may be a member of the AP, who needsto perform an independent testing of one or more platforms or platform providers.

"Executable documentation" of vulnerability findings. When a vulnerability is found in a product, the ability to reproduce an attack is a key to the investigation of the root cause of the issue and to providing a timely solution. The current practice is to use a combination of natural language and scripting to describe the process and the configuration necessary to reproduce an attack. Unfortunately, the results may be erratic and may complicate the security response. TestREx exploit scripts and configurations can be seen as "executable descriptions" of an attack. The production of exploits and configurations could not just be the task of the security validation department, but also of external security researchers, for which the company might set up a bounty program requiring that vulnerabilities are reported in the form of TestREx scripts.

4.7.3 Analysis and Training

Malware analysis. Malicious applications, also known as malware, are applications intentionally designed to harm their victims, by, e.g., stealing information or taking control of the victim's computer. Malware in general, and especially web malware, are known to react differently to different environments (usually to avoid detection) [38,93]. Containers provide safe and repeatable environments for malware analysts to run their experiments. One possible use of TestREx is as a highly configurable sandboxing environment, where malware analysts can run potentially malicious applications in different configurations of an application to study its behavior. Another possible use is as a honeypot generator. Honeypots [37] are intentionally vulnerable applications deployed on a network to capture and study attacks.

Part of a training toolkit. Security awareness campaigns, especially secure coding training, are commonly conducted in large enterprises, also in response to requirements from certification standards. From our own experience with TestREx, we believe that writing exploits may be an effective way to acquire hands-on knowledge of how security vulnerabilities work in practice, and how to code defensively in order to prevent them: we successfully used TestREx for teaching a Master course[13] on security vulnerabilities in web applications. To quickly create a large corpus of artificially vulnerable applications for training purposes, it is possible to start from well-known applications and use vulnerability injection, as done in [60, 122]. This way, we can easily create multiple examples for each category of vulnerabilities, with different levels of complexity for detection or exploitation.

[13] A past edition of the "Offensive Technologies" course provided by University of Trento (`http://securitylab.disi.unitn.it/doku.php?id=course_on_offensive_technologies`).

4.8 Conclusions

In this chapter, we presented TestREx, a Framework for Repeatable Exploits that combines a way of packing applications and execution environments, automatic execution of scripted exploits, and automatic reporting, providing an experimental setup for security testers and researchers. TestREx provides means for the evaluation of exploits, as the exploits are reproduced in a number of different contexts, and facilitates understanding of effects of the exploits in each context, as well as discovery of potential new vulnerabilities.

We also provide a corpus of applications and exploits, either adapted from the existing works, or developed by us – we collected it to test the variety of applications and exploits that can be handled by TestREx.

4.8.1 Lessons learned

We can summarize the key lessons learned during the design and development of TestREx as follows: (1) build on top of existing approaches; (2) have a simple and modular architecture; (3) find reliable information on applications, exploits and execution environments in order to replicate them.

Building on top of the existing work, like we did with BugBox [114] for the format of our exploits, and MalwareLab [7] for the vulnerability experimentation design, was extremely valuable. This simplified our design and development time, and allowed us to quickly add a large corpus of applications and exploits on which we could test our implementation.

Having a simple and easily extensible architecture was crucial in the development of TestREx, because this allows replacing the supporting frameworks (such as Selenium and Docker), selecting those that best fit the tester's purposes.

When adding the experiments to TestREx corpus, we soon learned that writing configuration files for application containers required the most effort. This is partially because the information on how to configure an application for a certain environment may be not detailed enough and may require additional knowledge from a tester. Also, in many cases, publicly available exploit descriptions are vague, limited to a proof-of-concept (which may not necessarily work), and often lack information on how to reproduce them in a specific software environment. This makes replication of the exploits a time consuming problem, even for experienced developers – which is one of suitable scenarios for which TestREx is intended.

4.8.2 Future work

When stating Problem 1 we discussed that successful exploitation may depend on specific software environments in which a vulnerable application is deployed. A possible extension of the approach may be in including additional tools (for instance, tools that allow to instrument the code of an application) that would allow to test whether a certain exploit failed because of the aspects of the software environment (e.g., the database engine is MongoDB instead of MySQL), or the exploit failed because of some other reasons (e.g., there is a bug in the exploit itself, or the application is already protected against the exploit).

It is also possible that other types of exploits, such as malware, may run successfully on a physical machine, but not in virtual environments [126]. Our current focus is on checking whether atomic exploits are successfully executed against particular vulnerabilities of web applications deployed in particular software environments. Still, checking the success of malware that implements detection evasion mechanisms may be an interesting line of future work.

Another potential line of future work is expanding the current vulnerability corpus by taking public exploits from, e.g., Exploit-DB and reconstructing the corresponding vulnerable environments in TestREx. This activity will be useful for performing an evaluation of the framework in order to identify its interesting potential extensions (for instance, introducing new classes of exploits, or configurations).

We also plan to extend the architecture of TestREx to add support for plugins. Plugins (e.g., proxy tools, vulnerability scanners) could be used to facilitate activities such as penetration testing, vulnerability analysis, and malware analysis, mentioned in Section 4.7.

Chapter 5

A Screening Test for Disclosed Vulnerabilities in FOSS Components

This chapter describes our solution to Problem 2 of secure FOSS consumption, which concerns fast and efficient identification of FOSS versions that are likely affected by a newly disclosed vulnerability. We provide a screening test: a novel, automatic method based on thin slicing, for quickly estimating whether a given vulnerability is present in a consumed FOSS component by looking across its entire repository in a matter of minutes. We show that our screening test scales to large open source projects that are routinely used by large software vendors, scanning thousands of commits and hundred thousands lines of code. Further, we provide insights on the empirical probability that for the sample of projects on which we ran our test a potentially vulnerable component might not actually be vulnerable after all.

5.1 Introduction

Software vendors that ship FOSS components to their customers as parts of their software applications must provide maintenance support for the software as a whole, including the consumed FOSS libraries. Therefore, if a vulnerability is reported, e.g., in the NVD (`https://nvd.nist.gov/`) about those FOSS libraries, the vendor is called to make a decision: (1) the upgrade requires minimal effort as the FOSS methods, and its APIs have not changed between the old deployed version and the new fixed version; (2) the consuming application is not using the vulnerable part of the FOSS component; (3) the application may already have code that protects itself against the vulnerability; (4) the old version of the FOSS component may be not affected at all because the vulnerable code was not yet introduced[1].

[1] Vulnerability databases (e.g., NVD) over-approximate the classification by using a default claim “version x and all prior versions”, so public data is not reliable for old components.

If a new vulnerability is disclosed and fixed in the *current* version of a FOSS component, the vendor of the consuming application must assess *(i) which (often more than 5 years old) releases are vulnerable* and *(ii) what actions should be taken to resolve issues for its different customers.* One simple solution would be to just update the used version of a component as soon as a fix is available. This is the model many free, no-warranty, software clients (e.g. Google Chrome and Mozilla Firefox), but it is often not applicable to components used in business software, control systems, etc. Economic theory [55, 79, 107] dictates that if updates are costly (e.g., regression testing, re-licensing licensing fees, re-certification of compliance, users' training, etc.) and problematic [131], and new features are not needed, customers will stay with old but perfectly functioning versions of the main application – we illustrated this empirically for an ERP product with Figure 3.2 in Chapter 3: significant number of customers used applications which were between five and nine years old. These applications included FOSS components that were "new" at the deployment time, but are now several years old. As a result, an enterprise system may be bundled with a FOSS release which is several years older than the currently available FOSS version. Thus, for an enterprise software vendor, addressing the above questions (i) and (ii) is far from trivial [90].

Unfortunately, is difficult for vendors to locate defects in a FOSS component used as a "black box" [90, 137], and therefore these vulnerabilities have higher chances of being left unresolved [62]. To identify the above cases, a vendor may test its application against a working exploit, but for many vulnerabilities there are no public exploits [8]. Even if published exploits exist, they must be adapted to trigger the FOSS vulnerability in the context of the consuming application. An alternative is to apply static application security testing tools (SAST) against the FOSS component. Such analysis requires a solid understanding of the FOSS source code [90], as well as expertise in SAST tools [18], as they can generate thousands of potentially false warnings for large projects. Further, the analysis may require days for processing even a single 'FOSS-release' 'main-application' pair [2]. If several FOSS releases are used in many different products [48] the above solutions do not scale.

This chapter presents our solution to the difficulties of identifying which older versions of FOSS components that software vendors ship to their customers are likely to be affected by newly disclosed vulnerabilities. We build on the intuition by Hindle et al. [74]: "*semantic properties of software are manifest in artificial ways that are computationally cheap to detect automatically, in particular when compared to the cost [...] of determining these properties by sound (or complete) static analysis*" in the direction of "soundiness" [92] to provide an automatic scalable method for performing such estimates using vulnerability fixes. Our solution provides an insight on the empirical probability that a potential vulnerable component might not actually be vulnerable if it is too old (or its update might be likely costly).

Table 5.1: Maintenance Cycles of Enterprise Software

Maintenance cycles of ten years or more are common for software used in enterprises. During this period, vendors need to fix security issues without changing either functionality or dependencies of the software.

Product	Release	EoLife	ext. EoL
Microsoft Windows XP	2001	2009	2014
Microsoft Windows 8	2012	2018	2023
Apache Tomcat	2007	2016	n/a
Red Hat Ent. Linux	2012	2020	2023
SAP SRM 6.0	2006	2013	2016
Siemens WinCC V4.0	1997	2004	n/a
Symantec Altiris	2008	2013	2016

5.2 Research Questions

The presence of a long lifecycle is not a characteristic that is inherent only to ERP software. Table 5.1 provides an illustrative example of the life-cycle of several products with respect to the maintenance, from operating systems to web servers, from industrial control software to security products. For example, Red Hat Enterprise Linux released in 2012 has an extended support for 11 years (until 2023). Siemens WinCC v4.0 (software for industrial control systems) had a lifetime of 7 years, and Symantec Altiris (service-oriented management software) released in 2008 has an extended lifetime of 8 years.

Security experts, developers and customers have, naturally, different priorities when deciding whether a component should be upgraded, fixed or left alone: *security experts* want to minimize the attack surface and, thus, prefer upgrades of potential vulnerable components over staying with old versions. *Developers and customers* try to minimize maintenance and operational risks of changes and, thus, prefer staying with an old version if the security risk in doing so is low.

Either way, we need to allocate resources to either port each application release, or audit their security. For example, developers could use Wala [150] or Soot [160] to construct a program slice on the vulnerable release, focusing only on the relevant subset of the vulnerable component. This slice could be later used by a sound SAST tool to ascertain that the vulnerability is indeed not present. Unfortunately, neither precise program slicing, nor a precise SAST tool scales well to large programs: tools providing a precise analysis can take days for one version of a component [2] or generate too many false alarms [17, 135] – this approach would not make it possible to manage the situation depicted in Figure 3.2 (Chapter 3), where we must assess several FOSS versions at once.

To focus our efforts on the actual vulnerable products, we must tentatively identify within minutes (not hours or days) all products that are likely affected by the vulnerability. We need the software equivalent of a clinical *screening test* [68]: something that may be neither (formally) sound, nor complete, but works well enough for practical problem instances and is fast and inexpensive[2]. Therefore, our first research question is as follows:

RQ1: *Given a disclosed security vulnerability in a FOSS component, what could be an accurate and efficient screening test for its presence in previous revisions of the component?*

Once we have such a screening test, we may use it for company-wise estimates to empirically assess the likelihood that an older version of a FOSS component may be affected by a newly disclosed vulnerability, as well as the potential maintenance effort required for upgrading or fixing that version:

RQ2: *For how long the vulnerable coding is persistent in FOSS code bases since its introduction? What are the overall security maintenance recommendations for such components?*

5.3 Related Work

5.3.1 Identifying the vulnerable coding

As our **RQ1** is concerned with finding an appropriate technique for capturing a vulnerable code fragment using vulnerability fixes, we build upon Fonseca and Vieira [61] as the basis for our idea of using an intra-procedural fix dependency sphere that we introduce in Section 5.5.4. The authors of [61] compared a large sample of fixes for injection vulnerabilities to various types of software faults in order to identify whether security faults follow the same patterns as general software faults: their results show that only a small subset of software faults are related to injection vulnerabilities, also suggesting that faults that correspond to this vulnerability type are rather simple and do not require a complex fix. Also, the work by Thome et al. [158] shows that sound slices for this type of vulnerabilities are significantly smaller than traditional program slices, and that control flow statements should be included into slices. Therefore, we collect control-flow statements as well, in contrast to the original approach of *thin slicing* [150] on which we build our implementation for capturing vulnerable code fragments using vulnerability fixes.

Modern static analysis tools such as *Wala*[3], and *Soot*[4] can be used for extracting the vulnerable coding using security fixes. These tools implement different slicing algorithms that work over byte code, offering various features and trade-offs such as redefining the notion of relevance of statements to the seeds [150] or improving the precision of intermediate program representation [66]; simplifying the notion of inter-procedural dependencies

[2] A sound and complete solution is formally impossible to achieve: Rice's theorem states that no recursive program can take a non-trivial set of programs (e.g., all past releases of a FOSS component) and produce the subset of programs satisfying a non-trivial property [147, Proof 5.28, pp243] (e.g., containing a semantically equivalent fragment of the vulnerable code fragment).

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

[4] `https://sable.github.io/soot/`

for efficient slicing of concurrent programs [127]; and defining slices that capture specific classes of security vulnerabilities [158]. Acharya and Robinson [2] evaluated the applicability of static slicing to identifying the impact of software changes. Their findings suggest that for small programs (and change-sets) static slicing can be used effectively, but it faces serious challenges when applied routinely against large systems: they report that the build time for the intermediate representation of one version of a project took about four days and observed that one must investigate and resolve various accuracy trade-offs in order to make large-scale analysis possible.

Thome et al. [158] implemented a lightweight program slicer that operates on the bytecode of a Java program and allows to extract all sources and sinks in the program for computing a program chop that would help software developers to perform faster audits of potential XML, XPath, and SQL injection vulnerabilities. It runs significantly faster than traditional slicing evaluated by Acharya and Robinson [2], however, still, it was close to impossible for our scenario (assessing thousands of revisions within seconds) to use precise tools based on byte code, as they require to build source code and resolve all dependencies as well. We found that for versions of Java projects which are older than five years from now, the latter could be very challenging. Moreover, we are interested in particular vulnerable code fragments that correspond to confirmed vulnerability fixes, but not in the whole set of slices that may contain all possible potentially vulnerable code fragments. Still, the approach by Thome et al. [158] can be used as a second-level test after our screening.

Considering the above, we have reverted to *thin slicing* [150] and modified the original algorithm to include the control flow statements, and limit the scope of slicing to the methods, where a security vulnerability was fixed.

To identify whether the library is called within the context of an application that consumes it, the approach by Plate et al. [123] can be also used as an additional test after our screening. However, the approach [123] cannot replace our own test as it requires to call a fully-fledged static analyzer to extract the call graph and fail our requirement of being inexpensive.

5.3.2 The SZZ approach: tracking the origin of the vulnerable coding

It is well known that to manually identify when exactly a certain vulnerability is introduced into a software component is a long process. For example, Meneely et al. [101] studied properties of source code repository commits that introduce vulnerabilities – the authors manually explored 68 vulnerabilities of Apache HTTPD[5], and they took six months to finish their analysis.

[5] We did not include this project to our sample as it is written in C, while our current implementation supports only Java.

Many studies on mining software repositories aim at solving the problem of manual analysis [86, 112, 148], allowing to automate this tedious task. The seminal work by Sliwerski, Zimmermann, and Zeller, widely known as SZZ [148], provided an empirical study on the introduction of bugs in software repositories, showing how to locate bug fixes in commit logs and how to identify their root causes. Their method had inspired the work by Nguyen et al. [112] on which we also build our approach. Unfortunately, the original SZZ approach has several limitations [86]: for instance, SZZ identifies the origin of a line of code with the "annotate" feature of the version control system, therefore it could fail to identify the true origin of that line of code when the code base is massively refactored (e.g., the line of code is moved to another position within its containing method). In our case, such a limitation would be a problem, since the code of the projects that we considered has been massively changed over the course of time (for example, see Figure 5.4 in Section 5.7). Therefore, we adopted the heuristics by Kim et al. [86]: we perform cross-revision mapping of individual lines from the initial vulnerability evidence and associate them with their containing files and methods. This allows us to track the origin of lines of code even if they are moved, or their containing file or method is renamed, or they are moved to another location within the code base.

5.3.3 Empirical studies on trade-offs between the security risk posed by the presence of the vulnerable coding and the maintainability

Di Penta et al. [52] performed an empirical study analyzing the decay of vulnerabilities in the source code as detected by static analysis tools, using three open source software systems. The decay likelihood observed by the authors [52] showed that most of potential vulnerabilities tend to be removed from the system before major releases (shortly after their introduction), which implies that developers may prioritize security issues resolution over regular code changes. One of the questions that the authors in [52] aimed to answer is similar to the first part of our **RQ2**, however we use a different measure of the vulnerable coding: the lines of code relevant to a security fix as opposed to the lines of code relevant to a static analysis warning. Moreover, our main focus is on distinct vulnerabilities that already have evaded static analysis scans and testing by developers, therefore they will likely show different decay.

For assessing various "global" trade-offs between a vulnerability risk that a component (or a set of components) imposes and its maintainability, one feasible option is to employ various risk estimation models. Samoladas et al. [139] proposed a model that supports automated software evaluation, and specifically targets open source products. The set of metrics considered by the model is represented by various code quality metrics (including security), and community quality metrics (e.g., mailing lists, the quality of documentation and developer base). While this model takes security aspects into account, they are represented only by two source code metrics: “null dereferences” and “undefined values”, which is largely insufficient to cover the vulnerability fixes in our sample (see Table 5.4).

Zhang et al. [173] proposed an approach for estimating the security risk for a software project by considering known security vulnerabilities in its dependencies, however the approach does not consider any evidence for the presence of a vulnerability. Dumitras et al. [53] discussed a risk model for managing software upgrades in enterprise systems. The model considers the number of bugs addressed by an update and the probability of breaking changes, but cannot be applied to assess individual components. As such approaches would not allow to answer the second part of our **RQ2**, we resort to the code-base evidence for telling whether it is likely that a certain version of a component imposes security risk.

5.4 Terminology and Definitions

In this Section we briefly introduce the terminology used in the chapter:

Fixed revision r_1: the revision (commit) in which certain vulnerability was fixed.

Last vulnerable revision r_0: the last revision in a source code repository that contained a specific vulnerability, which was eventually fixed by r_1.

Initial vulnerability evidence $E[r_0]$: the set of lines of code that correspond to the vulnerable source code fragment in r_0, obtained using changes between r_0 and r_1.

Vulnerability evidence $E[r_{-i}]$: the set of lines of code from the initial vulnerability evidence, that are still present in some revision r_{-i} that precedes r_0.

Repository difference $\mathrm{diff}(r_{-i}, r_{-i+1})$: the set of lines of code changed (deleted and added) when changes from r_{-i} to r_{-i+1} were made.

Deleted lines $\mathrm{del}(r_{-i}, r_{i+1})$: the set of lines of code deleted when changes from r_{-i} to r_{i+1} were made, s.t. $\mathrm{del}(r_{-i}, r_{i+1}) \subseteq \mathrm{diff}(r_{-i}, r_{i+1})$.

Added lines $\mathrm{add}(r_{-i}, r_{i+1})$: the set of lines of code added when changes from r_{-i} to r_{i+1} were made, s.t. $\mathrm{add}(r_{-i}, r_{i+1}) \subseteq \mathrm{diff}(r_{-i}, r_{i+1})$.

Source code of a revision $\mathrm{code}(r_{-i})$: the set of lines of code that belong to the source code of r_{-i}.

Set of relevant methods $\mathrm{methods}(locs, \mathrm{code}(r_{-i}))$: set of methods to which certain lines of code $locs \subseteq \mathrm{code}(r_{-i})$ belong.

Set of lines of code relevant to a set of methods $\mathrm{code}(\mathrm{methods}_{-i})$: the set of lines of code that belong to the specified set of methods, s.t. $\mathrm{code}(\mathrm{methods}_{-i}) \subseteq \mathrm{code}(r_{-i})$.

Set of defined variables $\mathrm{def}(s)$: the function that returns a set of variables which values are defined or re-defined in a statement s.

Set of referenced variables $\mathrm{ref}(s)$: the function that returns a set of variables which values are used in s.

Statement predicate $\mathrm{isPredicateOf}(s1, s2)$: this function indicates whether a statement $s1$ is a conditional statement, and a statement $s2$ is statement execution of which depends upon $s1$ (e.g., is a part of *then* or *else* branches of a conditional expression in Java).

5.5 Vulnerability Screening

We start answering **RQ1** by discussing several alternative techniques for performing screening tests for the likelihood of vulnerability presence.

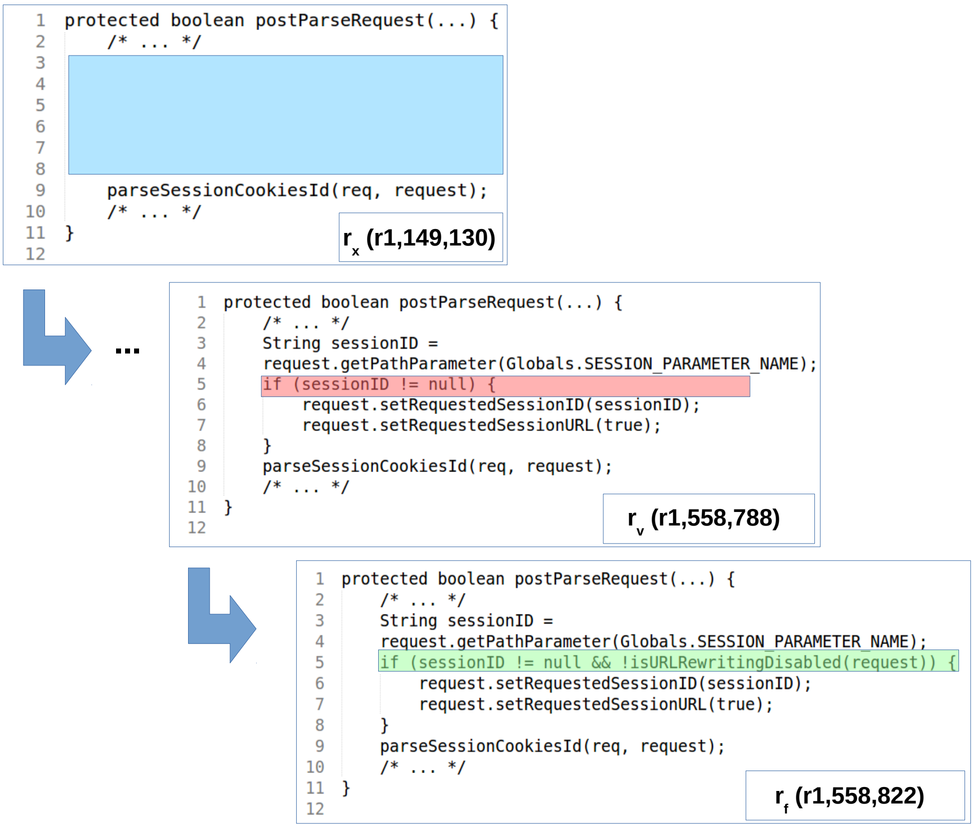
As the fixed version r_1 of a FOSS component is usually made available when a vulnerability is publicly disclosed, the information about source code modifications for implementing the fix transforming the last vulnerable version r_0 to r_1 can be used to understand where the vulnerable part in the source code is located [111, 148, 176]. Then, the approximate code fragment that is responsible for a vulnerability can be identified and tracked backwards in the source code repository history to identify a version that is not yet vulnerable [112].

Figure 5.1 illustrates an example for the vulnerability evolution in Apache Tomcat 6 (CVE-2014-0033): developers prohibited rewriting the URL string while handling session identifiers but a flag was not checked correctly and attackers could bypass the check and conduct session fixation attacks. The vulnerability was fixed in revision r_1(1558822) (on 16.01.2014) by modifying the incorrect check (line 5) in r_0(1558788)[6], making it impossible to set a different session identifier and rewrite the URL (lines 6 and 7). Searching for the presence of these lines in previous versions, reveals that in r_{1-i}(1149130) (on 21.07.2011) neither the check nor the session fixation lines are present. At that point in time, the URL rewriting set-up was not yet introduced by developers, and hence the code base is not yet vulnerable (to this attack).

Indeed, the absence of the vulnerable code fragment in some version r_{-i} that is older than the fixed r_1 is an ***evidence***, as opposed to a ***proof***, that this version is potentially not vulnerable: the vulnerable lines of code might be present in a different form or even in completely different, refactored files. If security fixes are rather local [112], these code lines constitute a *prima facie* evidence that we should allocate SAST, testing, or code auditing resources to analyze in depth the versions that correspond to the revisions where the vulnerable coding is still present, whilst having a more relaxed attitude on those versions preceding r_{-i}.

Let code(r_0) be a source code fragment that represents a vulnerable version of software application r_0 that also contains a vulnerability $V \subseteq \text{code}(r_0)$, which is responsible for an unwanted behavior. What is currently known, is that r_0 contains the vulnerability, and the next revision of this program r_1 is fixed. It is unknown, however, whether an older

[6] Changes in one file may correspond to ordered but not necessarily consecutive revisions, because Subversion uses repository global commit IDs.



```
protected boolean postParseRequest(...) {
    /* ... */






    parseSessionCookiesId(req, request);
    /* ... */
}
```

```
protected boolean postParseRequest(...) {
    /* ... */
    String sessionID =
    request.getPathParameter(Globals.SESSION_PARAMETER_NAME);
    if (sessionID != null) {
        request.setRequestedSessionID(sessionID);
        request.setRequestedSessionURL(true);
    }
    parseSessionCookiesId(req, request);
    /* ... */
}
```

```
protected boolean postParseRequest(...) {
    /* ... */
    String sessionID =
    request.getPathParameter(Globals.SESSION_PARAMETER_NAME);
    if (sessionID != null && !isURLRewritingDisabled(request)) {
        request.setRequestedSessionID(sessionID);
        request.setRequestedSessionURL(true);
    }
    parseSessionCookiesId(req, request);
    /* ... */
}
```

In Apache Tomcat 6, CVE-2014-0033 is fixed at revision 1558822 (=r_1) on 16/01/2014. Revision 1558788 (=r_1) is the last vulnerable revision that lacks a check on whether the URL rewriting setting is disabled. The revisions prior and including 1149130 (=r_i) from 21/07/2011 and earlier are not vulnerable to CVE-2014-0033, as the vulnerable feature is not present in these revisions.

Figure 5.1: Not all code declared vulnerable is actually so (CVE-2014-0033).

variant of the program r_{-i} where $i \geq 1$ contains this vulnerability as well.

This problem is similar to the screening tests used by clinicians to identify a possible presence of a disease in individuals [68]. In our case, we treat all revisions prior to r_0 (which is surely vulnerable) as those that potentially have the vulnerability, while different vulnerability evidences obtained from the fix are the metric that we use to separate the vulnerable part of the population from the non-vulnerable one.

Algorithm 1 illustrates a generic screening method for the potential presence of the vulnerable coding:

1. Init(r_0, r_1) is an abstract function that, using diff(r_0, r_1) operation from the source code repository, retrieves the changes made during the fix and infers the code fragment responsible for the vulnerability – the initial vulnerability evidence $E[r_0]$. An example of such evidence can be the source code lines that were directly modified during a fix (such evidence is considered by the original SZZ approach by Sliwerski et al. [148], as well as by the method proposed by Nguyen et al. [112]). However,

these modified lines of code may be not the ones actually responsible for the vulnerability, therefore we consider several other alternatives which we discuss in the next subsections.

2. for each revision r_{-i}, where $i \geq 1$, the current vulnerability evidence is represented by the lines of code from the initial vulnerability evidence that are still present in r_{-i}. We use the function $\text{Track}(r_{-i}, r_{-i+1}, E[r_{-i}])$ that keeps track of these lines of code individually, as suggested by Kim et al. [86].
3. for each revision r_{-i}, where $i \geq 1$, there is a test $\text{Test}(r_{-i})$ which is essentially a binary classifier that tells whether r_{-i} is likely vulnerable, based on the current vulnerability evidence and a reliability parameter δ. This parameter can be different for actual screening methods that use different vulnerability evidence extraction techniques.

Algorithm 1 Generic screening test using vulnerability evidence

Extract the vulnerability evidence using the last vulnerable revision r_0 and the fixed revision r_1:

$$E[r_0] \quad \leftarrow \quad \text{Init}(r_0, r_1) \tag{5.1}$$

For each revision r_{-i}, where $i \geq 1$, the evidence is computed as follows:

$$E[r_{-i}] \quad \leftarrow \quad \text{Track}(r_{-i}, r_{-i+1}, E[r_{-i+1}]) \tag{5.2}$$

Check, whether the source code of r_{-i} is still likely to be vulnerable:

$$\text{Test}(r_{-i}) = \begin{cases} r_{-i} \text{ is vuln.} & \text{if } \frac{|E[r_{-i}]|}{|E[r_0]|} > \delta \\ r_{-i} \text{ is not vuln.} & \text{otherwise} \end{cases} \tag{5.3}$$

The key question, however, is how to identify the right $\text{Init}(r_0, r_1)$ function for the test? As this is the primary concern of our **RQ1**, we start off with describing several candidates and explaining how each of them works.

5.5.1 Deletion Screening

A prior work by Nguyen et al. [112] (inspired by the work of Sliwerski et al. [148]) has shown that the presence of the lines of code deleted during a security fix for a browser component may be a good indicator on the likelihood that older software versions are still vulnerable: if at least one line of the initial evidence is present in a certain revision, this revision is considered to be still vulnerable.

The results in [112] suggest that the source code of the files and methods in which a security vulnerability was fixed may be not yet vulnerable at the point where they were first introduced into the code base of a project, so that the portion of the source code in these files and methods that is actually responsible for a vulnerability was added later during further development.

The approach works as follows:

1. It starts by collecting the *deleted* lines of code from a vulnerability fix – *deletion vulnerability evidence*;
2. Then, it goes iteratively over older commits/revisions in the source code repository and checks for the presence of these lines;
3. Finally, it stops either when none of the lines from the initial evidence are present, or when all commits/revisions are processed. When a vulnerability is fixed by only adding lines of code, there will be no evidence to track, and the authors in [112] conservatively assume that in such cases the whole version prior the fix (namely, code(r_0)) is vulnerable. This screening test was appropriate for the empirical analysis of Vulnerability Discovery Models, which are typically based on NVD and its cautious assumption "r_0 is vulnerable and so are all its previous versions" (see [112]), as this would create a consistent approximation of the NVD.

Essentially, the overall approach can be seen as an instance of the generic screening test that we defined in Algorithm 1. In this particular case, threshold $\delta = 0$, and our functions are instantiated as follows:

$$\mathrm{Init}_d(r_0, r_1) \quad = \quad \begin{cases} \mathrm{code}(r_0) & \text{if } \mathrm{del}(r_0, r_1) = \emptyset \\ \mathrm{del}(r_0, r_1) & \text{otherwise} \end{cases} \tag{5.4}$$

$$\mathrm{Track}(r_{-i}, r_{-i+1}, E[r_{-i}]) \quad = \quad E[r_{-i+1}] \cap \mathrm{code}(r_{-i}) \tag{5.5}$$

$$\mathrm{Test}(r_{-i}) \quad = \quad |E[r_{-i}]| > \delta = 0 \tag{5.6}$$

For security management this may be at the same time overly conservative and too liberal as the presence of the deleted lines may not be necessary for the vulnerability to exist (see [112] for a discussion on such cases).

5.5.2 Method Screening

An alternative simple heuristic is the following one: "if a method that was changed during a security fix is still present in an older version of a software product, this version is still vulnerable", under the conservative assumption the methods modified during the fix are responsible for a vulnerability. Again, this rule is likely imprecise but fast and inexpensive.

We instantiate the screening test for this heuristic as follows:

$$\mathit{methods}_1 \quad \leftarrow \quad \mathrm{methods}(\mathrm{add}(r_0, r_1), \mathrm{code}(r_1)) \tag{5.7}$$

$$\mathit{methods}_0 \leftarrow \text{methods}(\text{del}(r_0, r_1), \text{code}(r_0)) \tag{5.8}$$

$$\text{Init}_m(r_0, r_1) = (\text{code}(r_0) \cap \text{code}(\text{methods}_1)) \cup \text{code}(\text{methods}_0) \tag{5.9}$$

For *Track* and *Test* we use the same functions as for the deletion screening. However, tracking the presence/absence of vulnerable methods (or a change in their size) may be still overly conservative, because for cases when a method did not contain vulnerable code since it was first introduced, it may be still reported as vulnerable.

5.5.3 "Combined" Deletion Screening

For the original deletion screening test (see Section 5.5.1), if lines were only added during a fix, there are no cues on where vulnerable code could be located. Therefore, we can combine the original test with the method tracking: when a vulnerability was fixed only adding lines of code, we assume that the whole method (or methods) where these lines were added are responsible, otherwise, the technique works exactly as the original one (as before, $\delta = 0$)

$$\text{Init}_{ed}(r_0, r_1) = \begin{cases} \text{Init}_m(r_0, r_1) & \text{if del}(r_0, r_1) = \emptyset \\ \text{del}(r_0, r_1) & \text{otherwise} \end{cases} \tag{5.10}$$

$$\text{Track}(r_{-i}, r_{-i+1}, E[r_{-i}]) = E[r_{-i+1}] \cap \text{code}(r_{-i}) \tag{5.11}$$

$$\text{Test}(r_{-i}) = |E[r_{-i}]| > 0 \tag{5.12}$$

5.5.4 Fix Dependency Screening

Finally, we assume that not always the entire source code of fixed methods is responsible for the occurrence of a vulnerability. For instance, Fonseca and Vieira [61] empirically show that most of injection vulnerabilities may be due to a missing call to a sanitizer function, which is typically located at methods where user input is processed. Therefore, we need to devise a better approximation of the vulnerability evidence.

Let F be the fixed lines of code obtained with diff(r_0, r_1). In order to fix the lines of code $F \subseteq \text{code}(r_0)$, a developer might need to consider several other lines of code related to F – the actual vulnerable code fragment F'. Such expansion from F to F' can be progressively scaled by a parameter k: an expansion $D_k(\text{code}(r_0), F)$ that, given a code fragment code(r_0) and the fixed lines of code F, returns the lines of code that F depends-on or that are dependent-on F for k steps according to some criteria for the notion of dependency. By $D_*(\text{code}(r_0)\,, F)$ we identify the transitive closure of these dependencies, such that $F' \subseteq D_*(\text{code}(r_0)\,, F) \subseteq \text{code}(r_0)$ – the *fix dependency sphere*[7] of the code fragment F.

Therefore, we instantiate another screening test that considers the source code dependencies of the fixed source code fragment as follows:

$$\text{Init}_{fd}(r_0, r_1) \;=\; D_*(\text{code}(r_1)\,, \text{add}(r_0, r_1)) \cap D_*(\text{code}(r_0)\,, \text{del}(r_0, r_1)) \tag{5.13}$$

$$\text{Track}(r_{-i}, r_{-i+1}, E[r_{-i}]) \;= E[r_{-i+1}] \cap \text{code}(r_{-i}) \tag{5.14}$$

$$\text{Test}(r_{-i}) \;=\; \frac{|E[r_{-i}]|}{|E[r_0]|} > \delta \tag{5.15}$$

5.6 Implementing the Fix Dependency Sphere

We implemented the dependency expansion D_* as a generalized intra-procedural version of *thin slicing* introduced by Sridharan et al. [150], which improves over the notion of statement relevance of the original slicing algorithm by Weiser [165] to avoid collecting overly large slices. However, our implementation is intra-procedural and operates directly on the source code. It is possible to use precise tools such as Wala and Soot, but we have already noted that they would take too long and require too much expertise (see [2, 18]). They should be run after the screening test on the candidate vulnerable versions.

In our case, the lines modified during a vulnerability fix are *seeds*, and, similarly to [150], a slice includes a set of *producer statements* for the seeds. To identify simple dependencies between statements we look for relevance relations between variables in them. We also include a set of *explainer statements* that are relevant to the *seeds*. These are the following types of statements:

1. **Producer statements**: "[...] statement s is a producer for statement t if s is a part of a chain of assignments that computes and copies a value to t" [150]. This is an assignment of a value to a certain variable.
2. We distinguish the following types of **explainer statements**:
   (a) **Control flow statements**: the statements that represent the expressions in the condition branches under which a producer statement will be executed (this concept is taken from [150] as well). A statement s is control-dependent on a conditional expression e if e can affect whether s is executed. A statement s is

[7] This concept is similar to the notion of *k-dependency sphere* introduced by Renieris and Reiss [129] for dependencies in fault localization.

flow-dependent on a statement t if it reads from some variable v that is defined or changed at t, or there exists a control flow path from t to s on which v is not re-defined.

(b) **Sink statements** – represents a statement that corresponds to a method call which has a parameter to which a value flows from a producer statement. A statement k is a relevant sink of the statement s if k is a procedure call and k is flow-dependent on s.

Algorithm 2 Forward Slices of Relevant Variables

```
Set $Rel_f(s) \leftarrow \text{def}(s)$ if any of the following holds
  1. $s \in Seeds \wedge \text{def}(s) \neq \emptyset$
  2. there exists a preceding $t$ such that:
      (a) $\text{ref}(s) \cap Rel_f(t) \neq \emptyset$, or
      (b) $\text{isPredicateOf}(t, s) \wedge Rel_f(t) \neq \emptyset$
Set $Rel_f(s) \leftarrow \text{ref}(s)$ if any of the following holds
  1. $s \in Seeds \wedge \text{def}(s) = \emptyset$
  2. there exists a preceding $t$ s.t. $\text{ref}(s) \cap Rel_f(t) \neq \emptyset$
Otherwise $Rel_f(s) \leftarrow \emptyset$
```

Algorithm 3 Backward Slices of Relevant Variables

```
Set $Rel_b(s) \leftarrow \text{ref}(s)$ if any of the following holds:
  1. $s \in Seeds$
  2. there exists a preceding line $t$ s.t.
     $\text{def}(t) \cap Rel_b(s) \neq \emptyset$
     // conservative: ignore step (3) for “light” slicing
  3. there exists a preceding line $t$ s.t.
     $t \in Sinks \wedge \text{ref}(t) \cap Rel_b(s) \neq \emptyset$
  4. there exists a succeeding $t$ s.t.
     $\text{isPredicateOf}(s, t) \wedge Rel_b(t) \neq \emptyset$
Otherwise set $Rel_b(s) \leftarrow \emptyset$
```

Therefore, our implementation proceeds as follows:

1. We start with the set of seed statements $Seeds$ used as the slicing criteria, where every criterion can be represented as a tuple $\langle s, V \rangle$ (similarly to Weiser's slicing criterion [165]) where s is the seed statement, and V is the set of variables of interest in that statement.
2. Then, for every statement in the slicing criteria s, we recursively identify the set of relevant variables that are dependent-on or influence the relevant variables in s (Rel_f and Rel_b) using Algorithms 2 and 3.
3. The final slice will include all statements in the method, for which there is at least one variable that is relevant to the seeds.

When a statement s is a sink of the form $s(x, y, z)$, and we collect this statement because the parameter x is a variable that is relevant at some other statement t, we conservatively consider that x becomes relevant at s. However, we also consider the parameters y and z to become relevant at s since x may be changed inside of s, as well as its value may be passed to y and/or z.

Since this may be too conservative, as we may end up collecting too many statements that are not actually relevant to the seeds, we also implemented a *light* variant of this slicing that ignores the effect of the parameters: if a sink statement s of the form $s(x, y, z)$ will be included into a slice because of the parameter x, then we assume that neither x, nor other parameters are changed inside of s, therefore their relevance will not be propagated further (we empirically compare these two variants and discuss their performance in Section 5.8).

5.7 Data Selection

During an empirical study on the drivers for security maintenance effort that we describe throughout Chapters 3 and 6, we had informal discussions with software developers and maintenance experts of our industrial partner. We understood that maintenance decisions over a potentially vulnerable FOSS component can be taken on ad-hoc, component-by-component basis. For example, a component may be forked due to porting, changing a subset of features, or performing custom fixes for security bugs (as well as other technical modifications) [117, 132, 151].

The unlikely, but not a rare decision to down-port a security fix[8] may happen due to a combination of reasons:

1. The newer version of a FOSS component that provides the fix is largely incompatible with the coding of the application that consumes it, thus there is significant effort involved in migrating the application;
2. The internal changes of the library are of limited concern for the developers of the consuming application unless the functionality have been changed – the latter change is often being captured by a change in the APIs [23, 131];
3. The community that maintains the component is not likely to provide the solution for a specific security problem with an outdated version[9].

Our selection of FOSS projects is based the case study that we describe in Chapter 3, and on what we felt to be the most challenging and interesting projects. They are typical, popular, and large Java-based components of similar size that have been actively maintained for more than seven years. Table 5.2 lists the sample of FOSS projects on

[8] An example of forking and long-term maintenance is SAP's decision to provide its own Java Virtual Machine for several years "because of end of support for the partner JDK 1.4.2 solutions". See `http://docplayer.net/22056023-Sap-jvm-4-as-replacement-for-partner-jdks-1-4-2.html`.

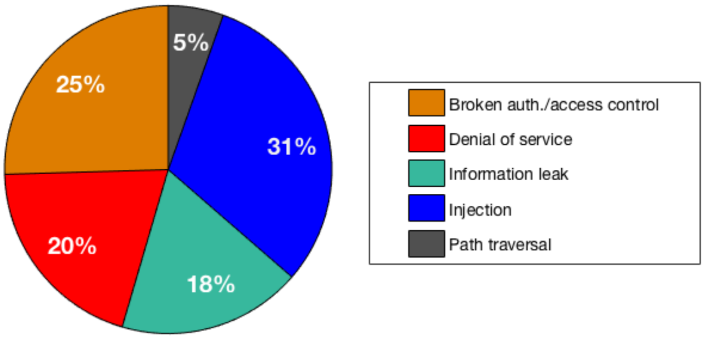
[9] This could happen when the old version of a FOSS component is affected by a vulnerability but it is not supported by its developers (e.g., EOL of Tomcat 5.5), or it is not actively maintained at the moment.

Table 5.2: The sample of FOSS projects used in this chapter

The FOSS projects from our sample have been actively developed over several years (e.g., the commit speed is between 742 and 2551 commits per year). For each component, the table lists the number of CVEs that affect Java sources that we analyzed, and the total number of CVEs. While Apache Tomcat is older than 10 years, its current trunk in the source code repository starts with version 6.0.0.

Project	**Total commits**	**Age (years)**	**Avg. commits (per year)**	**Total contributors**	**Current size (KLoC)**	**Total CVEs**	**Processed CVEs**	**μ files touched per fix**
Apache Tomcat (v6-9)	15730	10.0	1784	30	883	65	22	1.5
Apache ActiveMQ	9264	10.3	896	96	1151	15	3	1.5
Apache Camel	22815	9.0	2551	398	959	7	3	1.0
Apache Cxf	11965	8.0	1500	107	657	16	10	2.0
Spring Framework	12558	7.6	1646	416	997	8	5	1.6
Jenkins	23531	7.4	2493	1665	505	56	9	1.9
Apache Derby	7940	10.7	742	36	689	4	3	2.7

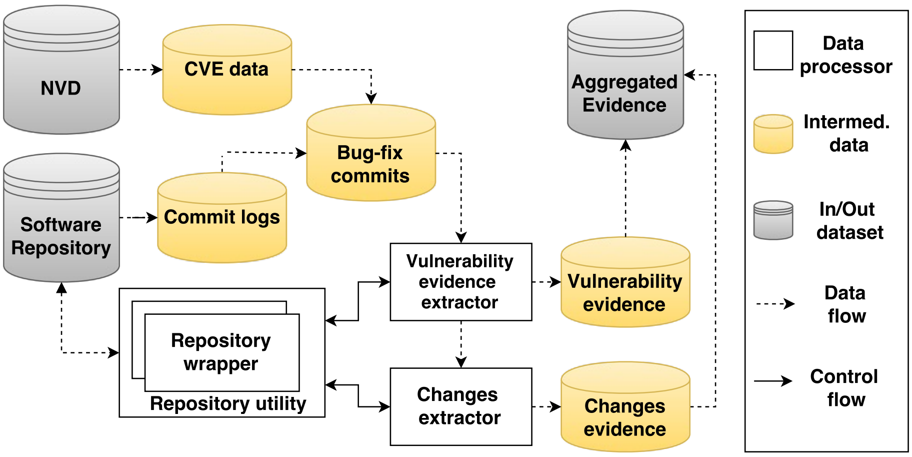
which we ran our screening tests, including the total numbers of CVEs for each project that exist in the NVD, as well as the number of CVEs that we analyzed (for which we could identify the corresponding fix commits in their source code repositories). Figure 5.2 shows the distribution of vulnerability types from our analyzed sample of 55 CVEs.



We identify vulnerability types of every CVE from our sample according to their descriptions in the NVD. This distribution suggests that the most prevalent type of vulnerabilities of the sample is Injection (including XSS, CSRF, command/code execution), however it does not significantly outnumber other types (except Path traversal).

Figure 5.2: The distribution of vulnerability types

From our communications with developers we also understood that a simple metric for change, the number of changed API, is of greater interest for developers, as their focus is to use the FOSS component as a (black box) library.



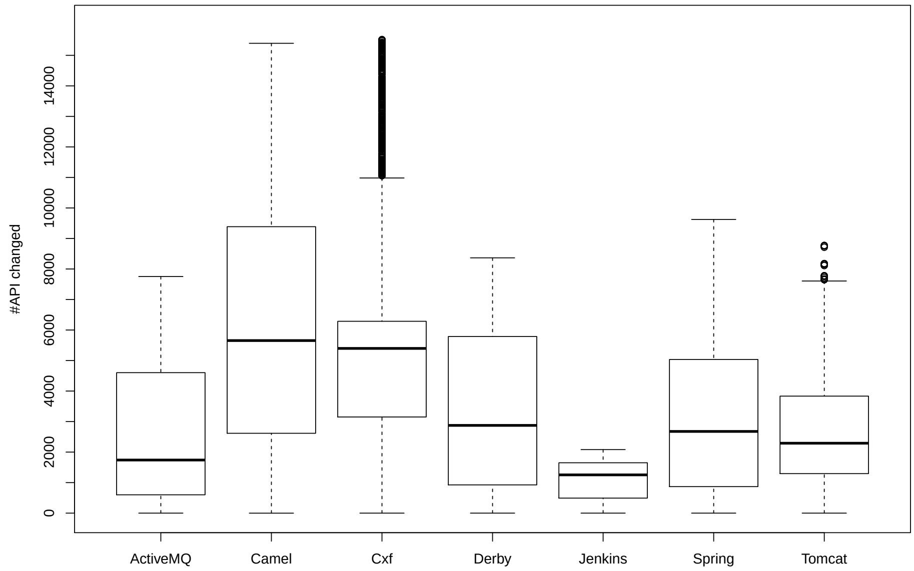
To collect the data, we examine the textual description of vulnerabilities from public vulnerability databases (e.g., the NVD) and search for references to the actual commits fixing the vulnerability. This information can be present in the NVD, commit logs of the projects, or security notes.

Figure 5.3: Software infrastructure for obtaining the aggregated evidence

Figure 5.3 describes the software infrastructure we set up for collecting the data. We first take the textual description of vulnerabilities from public vulnerability databases such as the NVD, and search for the references to the actual commits that fixed the vulnerability: this information can be either present in the vulnerability description, or in the "references" section, or mentioned in the security notes related to the vulnerability. Alternatively, it can be also present in the commit logs of the project's source code repository. While this activity can be automated, as it was done by Nguyen et al. [112], we chose to perform it manually. It was shown by Bird et al. [28] that automatic collection of such information may be biased, moreover, apart from looking at the NVD and commit logs (this process can be easily automated), we had to resort to looking into various security notes, bug trackers, and other third-party sources that do not belong to the actual projects (this process is difficult to automate).

After the vulnerability fix commit information is consolidated, we invoke the *Repository utility* component that automates various operations over source code repositories: it instantiates a particular *Repository wrapper* that corresponds to a certain repository type (currently we support Git and Apache Subversion). For every vulnerability, *Repository wrapper* iterates backwards over the set of commits starting from a vulnerability fix commit, and invokes the *Vulnerability evidence extractor* component for obtaining the various types of vulnerability evidences: at present we implemented all algorithms discussed in Section 5.5.

To provide also the demographics on the "number of API changes" that can cause



For every commit in which we tracked the vulnerable coding, we collected the number of public methods that were changed with respect to the public methods in corresponding fixes. The only exception was Jenkins – for this project we measured the number of changed methods in all commits (not only in those in which the vulnerable coding was present), as we found out that the repository history of this project was malformed (see Section 5.8 for an explanation). This distribution gives an intuition on the amount of such changes within each project.

Figure 5.4: API changes statistics per project

problems when updating older vulnerable versions of FOSS components (shown on Figure 5.4), we include the *Changes extractor* component:

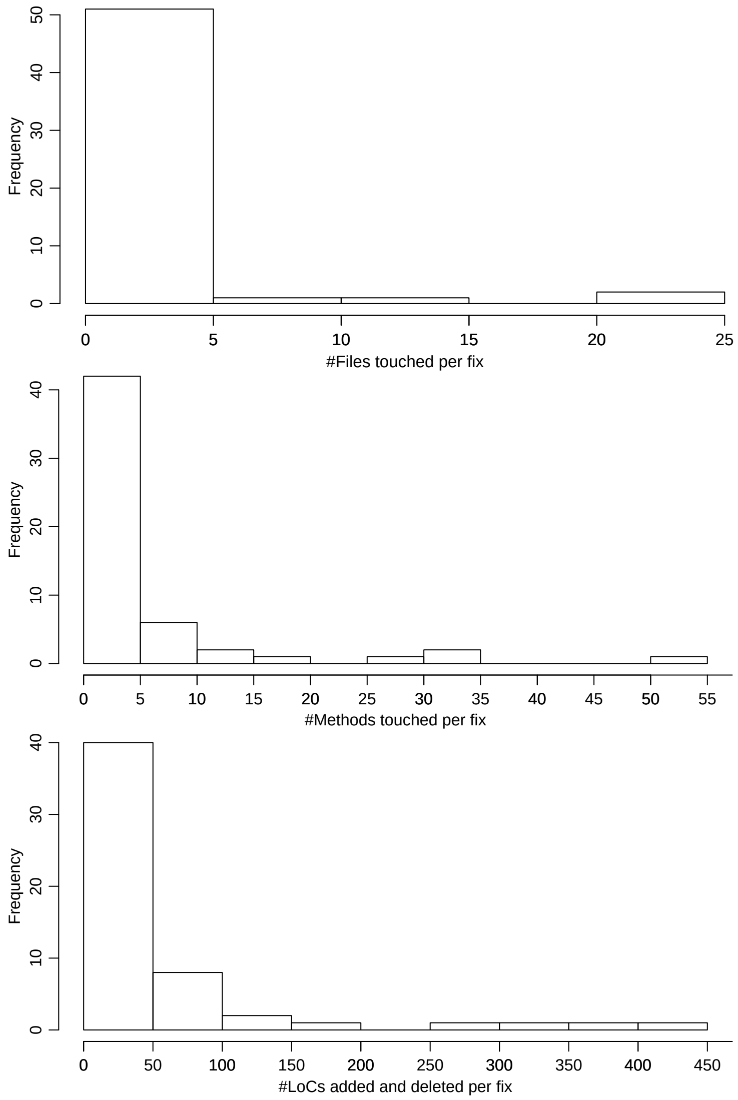
1. For each commit in which we track vulnerable coding, we identify all Java files; for each Java file, we count and sum the number of public methods;
2. Then, we take the difference between each such commit and the commit of the corresponding fix and count the number of public methods that are not present in the fix, or could have been changed (looking at method signatures);
3. We record the number of changed methods in the current commit with respect to the fix using the above two numbers.

After the vulnerability data is processed and all evidences are extracted, they are aggregated and stored in a CSV file or a MongoDB database which can be used for further analysis.

Figure 5.5 shows the distributions of the number of files[10] and methods that were modified for fixing CVEs from our sample, as well as the code churn[11]. In most cases (51 out of 55), at most 5 files were modified, while in 29 cases only 1 file was modified. Additionally, in 40 cases the code churn was at most 50 lines of code. This gives us an

[10] Here we count only Java files, excluding unit tests.

[11] It is difficult to automatically calculate modified lines using the *diff* tool, therefore we calculate code churn as the sum of deleted and added lines as a superset that contains changed lines as well.



The majority of security fixes from our sample were rather "local", not spanning across many files, methods, and lines of code.

Figure 5.5: The distribution of files and methods changed during a security fix

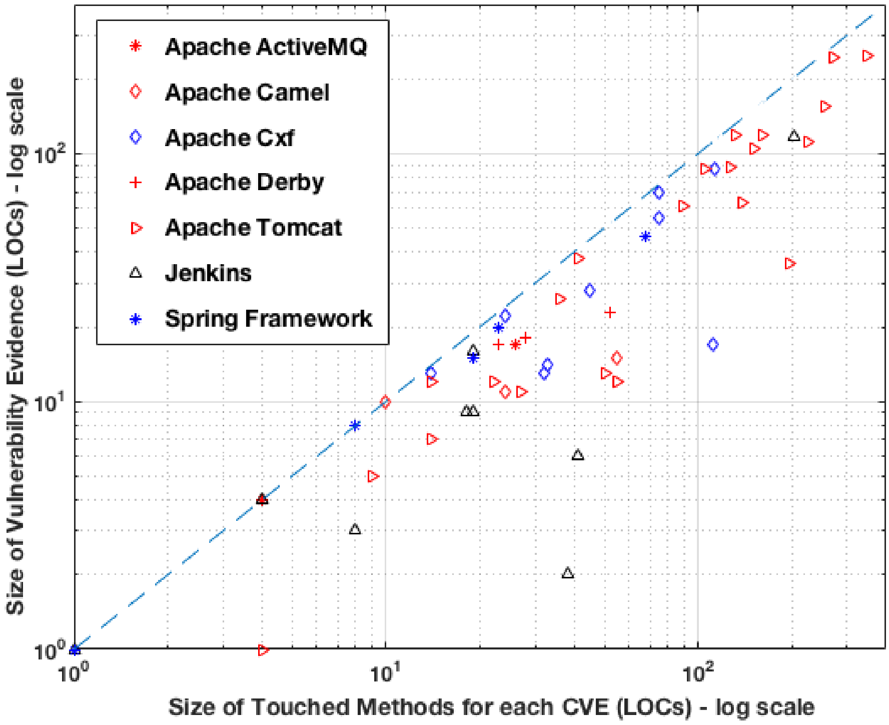
intuition that the majority of fixes were rather "local" and not spanning across multiple files, methods, and commits (which would possibly require a more complex evidence extraction mechanism).

For 49 out of 55 CVEs a fix was performed with a single commit. For every of the other 6 CVEs that were fixed with several commits, we used an ad-hoc procedure:

1. For each such commit, we extracted and tracked the initial vulnerability evidence independently (with (5.1) and (5.2) from Algorithm 1);
2. We aggregated the vulnerability evidences chronologically (with revision numbers) by CVE identifiers, in particular making sure that if the evidence overlaps (e.g., the

same method was changed with different commits), we do not include the same lines of code as the evidence again;

3. For such aggregated evidence, we applied the test (5.3) from Algorithm 1.



A possible phenomenon that we feared was that our implementation of fix dependency sphere (see Section 5.6) could have saturated the analysis by including the whole method where the vulnerable code was present. This is not happening in our sample of projects, as the vulnerable code is not totally mingled with other functionalities present in the method.

Figure 5.6: Comparing the initial amount of lines of code obtained with conservative fix dependency screening versus the initial size of the entire fixed method

5.8 Validation

In this Section we describe validation process that we performed to answer the part of **RQ1** about the accuracy and performance of the proposed vulnerability screening test, and to assess the overall usefulness of the approach for the problems outlined in Section 5.2. The empirical evaluation of the lightweight slicer for finding security features inherent for injection vulnerabilities by Thome et al. [158] reports the running time between 50 seconds to 2 minutes on a project that has 28 KLoC on average. Table 5.3 reports the runtime of our approach over the entire repository: while we cannot directly compare the running time of our implementation of extracting the fix dependency sphere and the slicer

Table 5.3: Runtime performance of fix dependency evidence

The vulnerability screening test can provide an approximate evidence (based on actual code) about the presence of the newly discovered vulnerabilities by scanning the entire lifetime of a FOSS project in matter of minutes. Precise (but costly) static analyses can be deployed after that step, in surgical fashion.

Project	Analysed Data		Time (in sec)	
	#Commits	#MLoCs	mean	(std)
Apache Tomcat	141016	186.331	35	(19)
Apache ActiveMQ	11598	27.904	28	(21)
Apache Camel	8892	4.706	16	(7)
Apace Cxf	53822	28.525	49	(33)
Spring Framework	17520	3.854	44	(35)
Jenkins	8039	9.416	16	(10)
Apache Derby	7588	5.597	17	(11)

by Thome et al. [158][12], the running time of our entire approach is comparable, which shows that it is practical.

Next, we review the vulnerabilities in our data set, and analyze their fixes to understand whether the fix dependency sphere would capture them. The results of this analysis for the conservative fix dependency screening are summarized in Table 5.4: it lists descriptions of vulnerability types (taken from OWASP Top 10[13]), as well as descriptions of typical fixes for these vulnerabilities. The "completeness" column describes the dependencies of a fix that will be captured by the fix dependency sphere $D_*(\text{code}(r_0)\,, F)$. We claim that for these vulnerability types and fixes $D_*(\text{code}(r_0)\,, F)$ includes the fragment of the code responsible for the vulnerability. Therefore, tracking the evolution of $D_*(\text{code}(r_0)\,, F)$ from r_0 and downwards may be a satisfying indicator for the presence of a vulnerability.

[12] As we only extract the relevant code within a set of methods (it takes less than a millisecond), while the slicer by Thome et al. [158] extracts all potentially relevant sources and sinks.

[13] `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

Table 5.4: Construction of a fix dependency sphere

Vuln. type	**Description**	**Fix**	**Completeness**
Injection (SQLi, XSS, code execution)	There exists a flow of data where a value coming from an insecure *source* (user input) can reach a secure *sink* (database, command interpreter, web browser) and be executed (e.g., CVE-2008-1947, CVE-2014-0075).	*The fix may:* 1. break such a flow (delete the source/sink statements), or insert a sanitizer between the source and the sink (add new method call inside of the method where the user input gets to the sensitive sink); 2. fix an incorrect sanitizer (the fix is inside the sanitizer method).	*The fix dependency sphere includes:* 1. statements that capture the faulty part of the flow between the source and the sink. 2. statements that capture the faulty part of the flow flow within the sanitizer.
Path traversal	There exists a flow of data from an insecure source, which is used for constructing path names intended to identify a resource located underneath a restricted parent location (e.g., CVE-2008-2370).	*The fix may:* 1. insert a sanitizer for the input used to construct a path (add new method call inside of the method where the path is constructed); 2. fix the sanitizer, e.g., add missing character encoding (the fix is inside the sanitizer method); 3. in case of URL construction, remove the query string from the path before the path is sanitized (re-arrange some of the statements inside of the method where the path is constructed);	*The fix dependency sphere includes:* 1. statements that capture the faulty flow between the source and the sink; 2. statements that capture the faulty flow within the sanitizer; 3. statements that capture faulty flow between the source and the sink as well as the statements that represent the altered control flow.

Continued on next page...

Table 5.4: Construction of a fix dependency sphere

Vuln. type	Description	Fix	Completeness
Information leak (configuration, username, passwords)	Incorrectly implemented or extensive error messages allow attackers to identify configuration details of an application and use it as a leverage for exploiting other vulnerabilities (e.g., CVE-2010-1157). Simple errors, such as passing a wrong variable into a method call (e.g., CVE-2014-0035) can lead to sending sensitive data over the network unencrypted. In case, when authentication functionality reveals too much context information, this can be used for enumeration of existing users and password guessing (e.g., CVE-2009-0580, CVE-2014-2064).	*The fix may:* 1. replace an incorrect message; 2. change the control flow upon which the message is shown; 3. neutralize an unhandled exception that is triggered under specific error conditions (add catch block); 4. replace an incorrect parameter passed to a method call.	*The fix dependency sphere includes:* 1. the modified error message, as well as control flow statements upon which the message is generated; 2. the altered control statements as well as the corresponding error message; 3. the whole “try” block that corresponds to the added “catch” block, as well as the relevant error message and it’s control flow statements; 4. the faulty method call (sink) as well as all statements that capture the data flow of the replaced parameter (source).
Cross-site request forgery (CSRF)	An application accepts web requests from an authenticated user, but fails to verify whether these requests correspond to the user’s session (are actually sent from am user’s browser).	*The fix may:* 1. Implement a protection mechanism by adding specific tokens and cookies (place a token into a response body); 2. As most protection mechanisms for this vuln. can be bypassed if there exists related cross-site scripting vulnerability, a potential fix may actually be equivalent to *Injection* vulnerability fixes.	*The fix dependency sphere includes:* 1. statements that capture a control flow in which the protection mechanism is inserted and under which this protection mechanism was lacking (e.g., a token is placed into the response body); 2. same as the fix dependency sphere for *Injection* vulnerabilities.

Continued on next page...

Table 5.4: Construction of a fix dependency sphere

Vuln. type	Description	Fix	Completeness
Broken authentication, access control flaws	Application fails to ensure the access control of resources (e.g., CVE-2006-7216, CVE-2012-5633), or user accounts (e.g., CVE-2014-0033, CVE-2013-0239).	*The fix may:* 1. Add (or replace) an ad-hoc access control rule to the resource (alter the control flow); 2. Add new method that specifies explicit permissions of an object (e.g., for serialization and deserialization); add corresponding method call to a method where corresponding object was created;	*The fix dependency sphere includes:* 1. the statements that correspond to the resource referenced at the point where the new ad-hoc access control rule was added by the fix (control-dependency of the newly added rule), as well as some additional control and data flow dependencies; 2. the statements that correspond to the control/data flows under which the newly added statement is being called.
Denial of service	Application becomes unavailable to users due to the errors in resource management that are exploited by an attacker. This vulnerability may exist either due to insufficient input validation (e.g., CVE-2011-0534), logical flaws (e.g., CVE-2014-0230, CVE-2012-2733), or the combination of both (e.g.., CVE-2014-0095).	*The fix may:* 1. add or change existing conditions that control the "expensive" resource operation (e.g., buffer limits, thread numbers, etc.); 2. move the condition under which an "expensive" resource operation is invoked (e.g., under some conditions, an operation may be invoked before the check is performed), or add additional checks; 3. alternatively, add a sanitizer that does not allow user input to trigger the fault (e.g.., size of the data to be cached, data validity checks, etc.).	*The fix dependency sphere includes:* 1. the statements that correspond to the resource operation itself, as well as control and data flows relevant to this operation; 2. same as the above; 3. same as the fix dependency sphere for *Injection* vulnerabilities.

Initially, we selected 25 vulnerabilities for a manual validation, however, after initial inspection we decided to exclude four vulnerabilities of the Jenkins project, as we observed that the whole repository layout was deleted and copied over several times – this would make our current automatic analysis incomplete (it stops when there is no more evidence, and currently we did not implement heuristics that would allow to identify such corner cases), and significantly complicate the manual analysis.

For each of the selected vulnerabilities, we identified the set of ground truth values as follows:

1. We performed source code audits starting from the last vulnerable revision r_0, moving backwards through the repository history.

2. When we observed that any of the files initially modified to fix a vulnerability had some changes in an earlier revision, we manually checked whether the vulnerability in that revision was still present.

3. We stopped the analysis either on a revision that we find to be not yet vulnerable (this implies that all earlier revisions are not vulnerable as well – we did several spot checks going further past the first non-vulnerable revision an that was indeed the case), or until we reached the initial revision of the repository.

The final sample for the manual assessment consisted of 126 data points across the total of 126193 revisions, which corresponds to histories of 21 CVEs: we went backwards iteratively, and for many revisions the vulnerability evidence did not change. Therefore, we had to check only those points where it did actually change. The manual assessment was carried out by three experts, who were cross-checking and discussing the results among them before arriving at the final conclusion.

In this way, we manually annotated every revision from r_0 and backwards with ground truth values, obtaining the *ground truth* binary classifier:

$$Test_{gt}(r_i) = \begin{cases} 1 & \text{if } r_i \text{ is still vulnerable} \\ 0 & \text{otherwise} \end{cases}$$

Then, we ran every variant of the vulnerability screening test described in Section 5.5, and compared the results with the ground truth. For every revision $r_i < r_0$ (where $i < 0$), this comparison had the following result:

1. *True positive:* a revision was correctly classified as *vulnerable* (e.g., a test marks the revision as vulnerable, and we identified that it is indeed vulnerable with our ground truth analysis);

2. *False positive:* a revision was incorrectly classified as *vulnerable* (type I error of a classifier);

3. *True negative:* a revision was correctly classified as *non-vulnerable*;

4. *False negative:* a revision was incorrectly classified as *non-vulnerable* (type II error of a classifier).

As a part of the answer to **RQ1**, we wanted to understand whether our fix dependency variants of the vulnerability screening test (see Section 5.5.4) show results that are significantly different in comparison to the existing work of Nguyen et al. [112] and the simplest possible heuristic that can be expressed as "if the vulnerable piece of code (methods that were fixed) does not exist yet, the vulnerability does not exist as well".

Figure 5.7 shows the performance of the variants of the vulnerability screening test in terms of true positive (Sensitivity) and false positive (1-Specificity) rates, when compared to the ground truth classifier. We discuss the results below.

The *"method screening"* test did not show very high performance with most values of the threshold δ. However, when δ is set to 0, the classifier is marking a revision vulnerable based on just the presence or the absence of these methods, which may be a good vulnerability indicator when security is the only factor that matters, as its Sensitivity is equal to 1. Still, this strategy may have too many false positives in case affected methods were not vulnerable right from the point when they were introduced (Specificity = 0.002). This may result in potentially high security maintenance effort.

The *"combined deletion screening"* test showed similar performance to the above variant of the test, however it has slightly smaller Sensitivity (which does not contradict with the false negative error rate reported by Nguyen et al. [112]), as in several cases the deleted lines disappear before the actual vulnerable part of a method is gone.

The *"light fix dependency screening"* test shows significantly better performance when the threshold δ is set to 0.5 and 0.2. With $\delta = 0.5$, Sensitivity = 0.863, with Specificity = 1.0 (no false positives); while with $\delta = 0.2$, Sensitivity equals to 1.0. However, in the latter there are much more false positives (Specificity = 0.218). The amount of false positive results may be not important for a security assurance team, as long as Sensitivity is close to 1.0 [18]. On the other hand, for making quick estimates, significantly cutting down the number of false positives may be more preferable. Thus, the above threshold values may represent the trade-offs between the two conflicting goals: (1) the limited amount of development resources that dictates to prioritize only the work that is necessary, and (2) the requirement to provide maximum security assurance regardless the cost. In the first case, most of vulnerable revisions will be recognized correctly so that the appropriate action can be taken immediately, but there is still a small chance that some significantly older vulnerable revisions will be marked as safe. In the second case, no revisions will be incorrectly classified as non-vulnerable, but developers may spend a lot of additional work on false positives – this case is still better than looking at the presence of a vulnerable method, as it provides the same level of assurance with significantly smaller number of false positives.

On the other hand, the *"conservative fix dependency screening"* test yields more false positives after $\delta > 0.5$, however, for $\delta > 0.2$ it is the same as the *light* test. This is because for some of the vulnerabilities from our manual sample, the *conservative* test yields a larger initial vulnerability evidence fragment capturing more lines of code within a method that are not relevant to the vulnerable code fragment. Therefore, in such cases initial vulnerability evidence decays slower than the initial vulnerability evidence for the *light* test, showing different results at certain thresholds.

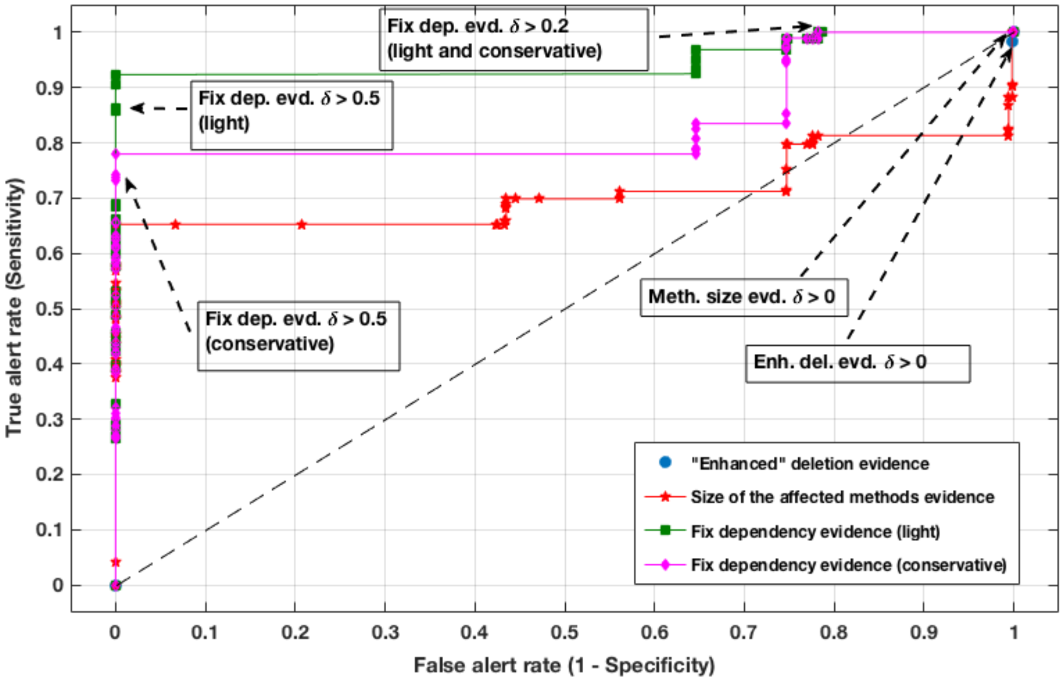
Table 5.5: Performance of the screening tests

The Precision for each test reflects the likelihood that a vulnerable revision will be correctly identified as such by the test, while the Negative Precision suggests the opposite – the likelihood of a non-vulnerable revision to be correctly identified as non-vulnerable. The results show that either variant of the fix dependency screening has better discriminative capabilities than the variants of the test based on the presence of deleted lines, or the size the affected methods.

Screening test	Threshold	Sensitivity	Specificity	Precision	Neg. Precision
Method screening (Section 5.5.2)	$\delta > 0.0$	1.000	0.002	0.927	1.000
	$\delta > 0.2$	0.905	0.002	0.920	0.002
	$\delta > 0.5$	0.801	0.224	0.929	0.082
	$\delta > 0.8$	0.653	1.000	1.000	0.186
"Combined" deletion screening (Section 5.5.3)	$\delta > 0.0$	0.982	0.002	0.925	0.010
Light fix dependency screening (Section 5.5.4)	$\delta > 0.2$	1.000	0.218	0.941	1.000
	$\delta > 0.5$	0.863	1.000	1.000	0.367
	$\delta > 0.8$	0.457	1.000	1.000	0.128
Conservative fix dependency screening (Section 5.5.4)	$\delta > 0.2$	1.000	0.218	0.941	1.000
	$\delta > 0.5$	0.742	1.000	1.000	0.235
	$\delta > 0.8$	0.458	1.000	1.000	0.128

However, Sensitivity and Specificity are the general characteristics of a test where the population does not affect the result. To account for the prevalence [68] of the vulnerable revisions we also calculate Precision and Negative Precision of the tests, which account for the test predictive capabilities. These values are shown in Table 5.5 alongside Specificity and Sensitivity. The values of these metrics show that the "fix dependency" variants of the screening test have better discriminative capabilities than other variants of the test we tried.

As can be seen from Table 5.5, the *light fix dependency test* ($\delta > 0.5$) had no false positives, but had false negatives; in contrast, the *conservative fix dependency test* ($\delta > 0.2$) had no false negatives, but had false positives. Therefore, we approximate the potential



The ***"combined" deletion screening*** *test could almost always identify a vulnerable revision (Sensitivity = 0.982), but almost always failed to distinguish a revision that is not yet vulnerable. The* ***method screening*** *test with* $\delta = 0$ *(a revision is classified as vulnerable when affected methods are present) could always identify a vulnerable revision (Sensitivity = 1.0), but had the same problem as the deletion screening (Specificity = 0.002). At the same time, both light and conservative fix dependency screening tests show significantly better performance than just looking at the deleted lines or the method(s) size: both in terms of true positive and false positive rates.*

Figure 5.7: ROC curves for different variants of the vulnerability screening test

error rates for both tests – we use the Agresti–Coull confidence interval [4], that requires to solve for p the following formula:

$$|\hat{p} - p| = z \cdot \sqrt{p \cdot (1-p)/n}, \tag{5.16}$$

where p – is the estimated proportion of vulnerable (non-vulnerable) revisions; $\hat{p}$ – is the sample size proportion of vulnerable (non-vulnerable) revisions over the total sample of revisions n; and $z = 1.96$ – is the coefficient for the 95% confidence interval. We have chosen a large sample of CVEs for manual verification since it corresponds to a large sample of revisions n, which ensures small margin of error. Thus, we have a potential error rate for the tests as follows:

- The *light fix dependency test* with $\delta > 0.5$ had the 0% error rate when classifying *non-vulnerable* revisions (no false positives), and 13.7%±0.2% error rate when classifying vulnerable revisions (few false negatives);
- The *conservative fix dependency test* with $\delta > 0.2$ had the 78.3% ± 0.8% error rate

when classifying *non-vulnerable* revisions (significant number of false positives), and 0% error rate when classifying *vulnerable* revisions (no false negatives).

These results allow us to provide an answer to **RQ1**: tracking the presence/absence of the vulnerable methods or lines of code deleted during a security fix may be not sufficient from the security maintenance management perspective. Still, fairly simple heuristics that capture the lines of code that are potentially relevant to the vulnerable part of a method can be more suitable for this task.

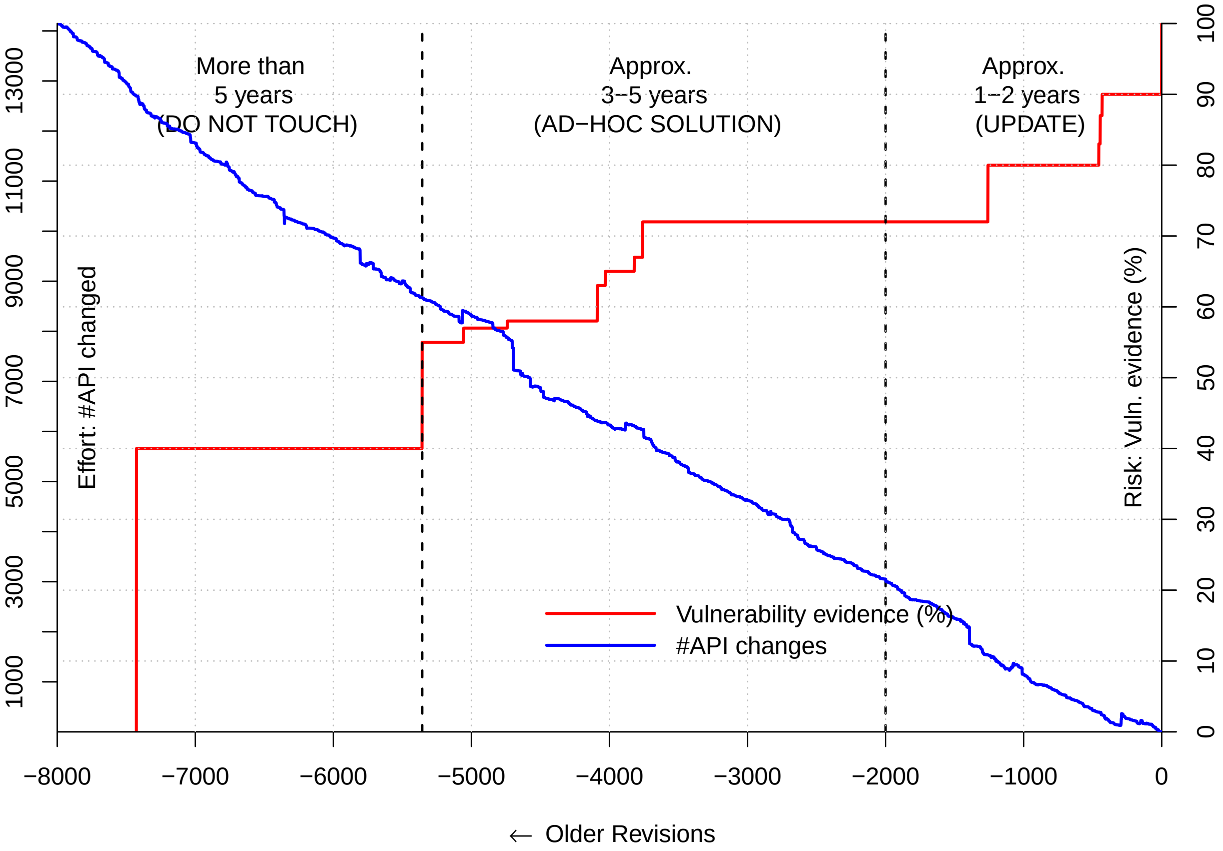
5.9 Decision Support for Security Maintenance

For those FOSS components, where upgrading to the latest version would likely require a low effort, we just might want to update them – even if the security risk is comparatively low. For components where the upgrade (or fixing) effort is high, we still can do a more expensive and more precise analysis. Still, getting an immediate estimate on the trade-offs between the upgrade effort and the likelihood of the security risk is the key for *not* wasting the (limited) available resources on FOSS components that are unlikely to be vulnerable, or are likely easy to upgrade.

Therefore, to answer **RQ2**, and provide an insight on whether developers could extract quick indicators for security maintenance decisions on FOSS components they consume, we performed an empirical analysis of the persistence of potentially vulnerable coding in source code repositories of the chosen projects. We also extracted the amount of changes between each revision and the fix in terms of changed public API, which we use as a proxy for the overall changes that may complicate component updates, increasing maintenance costs (see Section 5.7).

First of all, upon disclosure of a new vulnerability, developers could use a "local" decision support that would allow them to identify the vulnerability risk for a version of a FOSS component in question, as well as the likelihood that the component can be updated without any major efforts. If an easy update is not possible (and for considerably older versions of software components this is rarely the case), the value of the vulnerability risk indicated by the presence of the vulnerable coding may be a useful indicator for the maintenance planning. With Figure 5.8, we illustrate such a decision support for developers: this information is generated by running the *conservative fix dependency screening* test for CVE-2014-0035 (Apache CXF). We take the absolute value of the vulnerability evidence as the potential security risk, and measure the changes in the API between each revision and the fix for this CVE as a proxy for the upgrade effort. If a version of a FOSS component is not older than 2000 revisions back from the fix (approx. 1-2 years), it may be preferable to update the component, as most of the vulnerable coding is present, and difference in the API with respect to the fix is only starting to accumulate.

On the other hand, if it is older than 5000 revisions back from the fix (more than 5 years), it may be more preferable to take no action, as most of the potentially vulnerable coding is gone, and changes accumulated between that point in time and the fix are too many. For cases when the version of interest lies somewhere between these two areas, a custom fix may be implemented.



As we move backward from the fix in the revision history, the coding that is responsible for a vulnerability possibly disappears (red curve, shows the absolute value of evidence in LoC), whereas other changes in the code base start to accumulate (blue curve is the amount of API that changed in a certain revision with respect to the fix that represents the effort of upgrading from that point to the fix). A very old version may require to change 13000+ public methods for a vulnerability that may be very unlikely to be there (85% chances, see Figure 5.9). Thus, the position of the revision of interest in this diagram provides developers with a good insight on what decision to make.

Figure 5.8: Trade-off curves for one vulnerability of Apache CXF (CVE-2014-0035)

To sketch a trade-off model that would allow to perform a retrospective analysis for "global" security maintenance of the whole FOSS component, we attempt to generalize the above "local" decision support. Similarly to Nappa et al. [110], who employed *survival analysis* to analyze the time after a security patch is applied to a vulnerable host, we used it to analyze the persistence of vulnerable coding that we extracted from the sample of FOSS projects (shown in Table 5.2) with our screening tests. Survival analysis is the field of statistics that analyzes the expected duration of time before an event of interest occurs [106], it is being widely used in biological and medical studies.

In our scenario, time goes backwards (from the fix), and we identify the following event affecting every pair of (CVE, FOSS) as an individual entity, depending on one's considerations:

Security Risk: the event whose probability we measure is "the ratio of the vulnerability evidence $E[r_{i-1}]/E[r_0]$ in a screening test falls below δ".

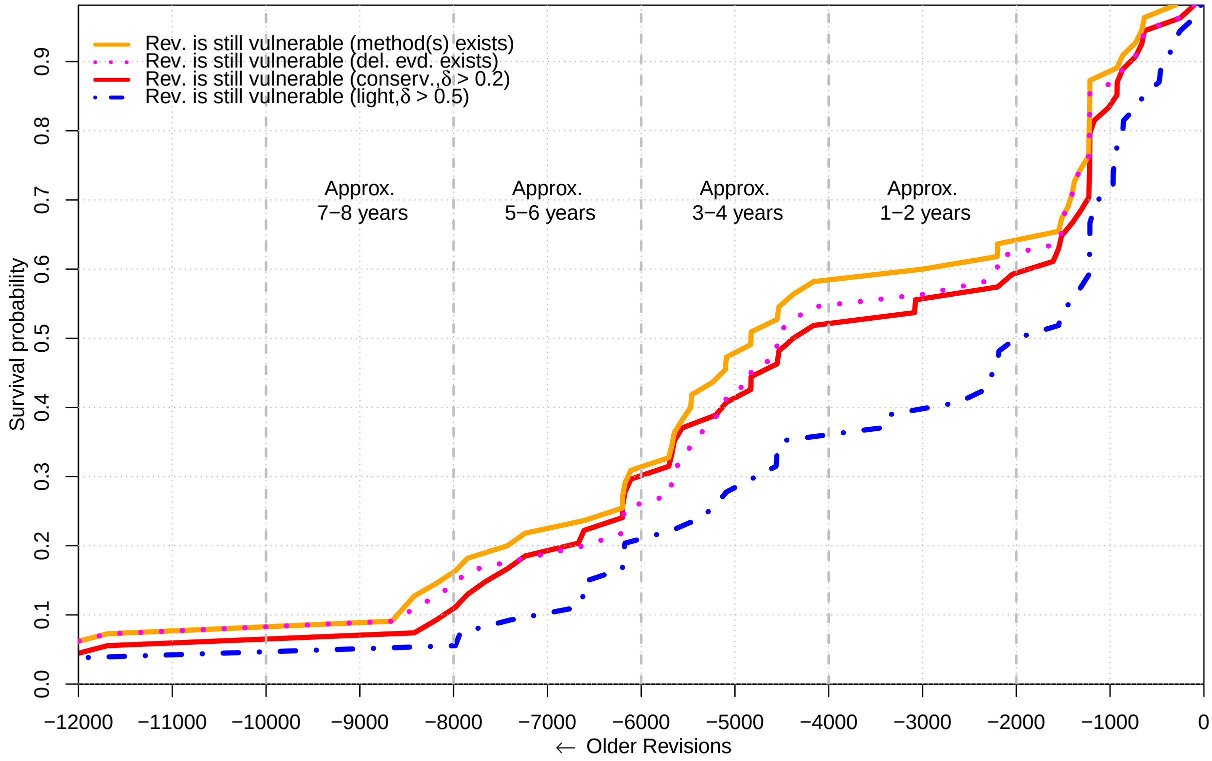
This event corresponds to the likelihood of the presence of the coding that is responsible for the vulnerability. To identify how this security risk may change over time, which is the concern of our **RQ2**, we computed the survival probabilities of vulnerable code fragments using the *light fix dependency screening* with $\delta > 0.5$, *conservative fix dependency screening* with $\delta > 0.2$, *method screening* with $\delta > 0$, and *"combined" deletion screening* tests (the variants of the screening test which performance we show in Figure 5.7). We performed survival analysis using the *survfit*[14] package in R, fitting the Kaplan-Meier non-parametric model (The Nelson-Aalen model gives the same qualitative result).

Figure 5.9 shows these survival probabilities: the vulnerable coding tends to start disappearing after 1000 commits (approximately 1 year preceding the fix), as already at 2000 revisions back there are 60% chances that the vulnerable coding is still there according to the evidence collected by *conservative fix dependency screening* (red curve). At 6000 revisions back (approx. 4 years) there is only 30% chance that the vulnerable coding survived, according to the same evidence. The curve that represents the probability of being vulnerable according to the evidence obtained with *light fix dependency screening* (blue curve) decays even faster. While the difference between the *conservative fix dependency screening* and method/deletion evidence presence is not that obvious on this figure, it is still significant (recall Figure 5.7).

Finally, we sketch the "global" decision support that represents the trade-offs that can be considered for the security maintenance of a FOSS project (**RQ2**), we further combine the survival curves for vulnerability evidences obtained with *light* and *conservative fix screening* tests over the set of vulnerabilities for the Apache Tomcat project, using the average values of API changes per project. Figure 5.10 represents the "global" trade-off decision support for the Apache Tomcat project, that consists of the following elements:

1. The dashed red line corresponds to the *conservative* probability that the vulnerable coding has survived at a certain point in time – this is based on the *conservative fix dependency screening* with $\delta > 0.2$ (our manual assessment for this test in Section 5.8 showed no false negatives, but a considerable amount of false positives).
2. The solid red line corresponds to the *lightweight* probability that the vulnerable coding is still there – this is based on the *light fix dependency screening* with $\delta > 0.5$ (our manual assessment for this test in Section 5.8 showed no false positives and a

[14] `https://cran.r-project.org/web/packages/survival/survival.pdf`



The vulnerable coding tends to start disappearing after 1000 commits (<= 1 year preceding the fix), as already at 2000 revisions back there are 60% chances that the vulnerable coding is still there according to the evidence collected by conservative fix dependency screening (red curve). Further back (after approx. 6 years), there is only a small probability that a component is vulnerable.

Figure 5.9: Survival probabilities of the vulnerable coding with respect to different variants of the screening test

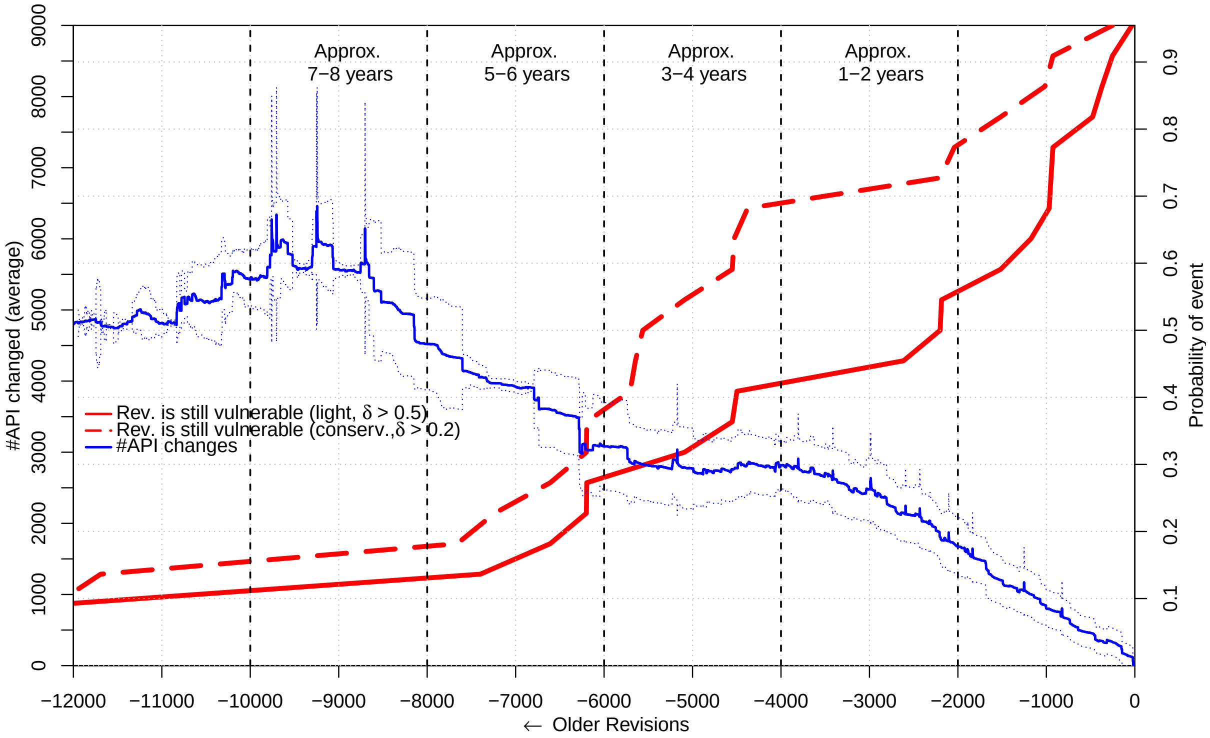
low number of false negatives).

3. Each point on the solid blue line corresponds to the number of the API changed in a certain revision in comparison to the fix: these are the aggregated average numbers taken for the whole project sample (the two dashed lines are the .95% confidence interval).

Figure 5.10 gives a recommendation to developers to update their versions of a component on a yearly basis, as after that time the vulnerability risk is likely to be still high, and the API changes tend to grow fast. The average amount of API changes[15], as well as both risk values, suggest that the security assessment should be performed when a version of interest lags for around 3-4 years behind the fix (between 4000 and 6000 commits). Here the down-port decision could be evaluated, considering that the *conservative* risk estimate is still high at this point. Alternatively, if the *lightweight* risk estimate

[15] A certain older revision r_y may actually have less API changes with respect to the fix than a certain newer r_x for a simple reason, that r_y has less functionality than r_x – this may be the reason why the amount of API changes that we observe in Figure 5.10 is not as "linear" as in Figure 5.8.

is tolerable, developers may already prefer to take no action at this point. Looking at both *conservative* and *lightweight* probabilities for the vulnerability risk and the average amount of the API changes, the point after 8000 commits could be the one at which the "do not touch" decision might be the only reasonable choice.



As we move back from the fix in the revision history, the probability that a revision is still vulnerable (red solid and dashed curves) holds high within the first 2 years before the fix (around 4000 revisions back). At the same time, the average amount of API changes (blue curve, the two dashed blue curves are the .95% confidence interval) accumulates fast – this may be the right time for an update. Further back, between approx. 3 and 4 years before the fix, the amount of changes does not grow significantly, but the vulnerability risk is still relatively high – this may be the time frame for a thorough security assessment of a version in question. Further back (after approx. 4 years before the fix), the vulnerability risk falls down, and changes begin to accumulate even more – here the "do not touch" decision might be the only reasonable choice.

Figure 5.10: "Global" trade-off curves for 22 vulnerabilities of Apache Tomcat

5.10 Threats to Validity

In our approach the *construct validity* may potentially be affected by the means of data collection and preparation, the selected sample of FOSS projects, and the accuracy of the information about security fixes in them:

- *Misleading commit messages.* As pointed by Bird et al. [28] (and from our own experience), linking CVE identifiers to specific commits in source code repositories is not trivial: developers may not mention fix commits in the NVD and security notes, and they may not mention CVE identifiers within commit logs. Also, automatic extraction of bug fix commits may introduce bias due to misclassification (e.g., a developer mentions a CVE identifier in a commit that is not fixing this CVE). To minimize such bias, we collected this data manually, using several data sources, including third-party sources that do not belong to the actual projects. Manual data collection allowed us to additionally verify that every vulnerability fix commit that we collected is indeed a fix for a particular CVE, therefore we do not have the latter bias of misclassification.

- *Tangled code changes in vulnerability fixes.* There is a potential bias in bug-fix commits, such that along with fixing relevant parts of the functionality, developers may introduce irrelevant changes (e.g., refactor some unrelated code). Kawrykow and Robillard [85], and Herzig et al. [73] explored to what extent bug-fix commits may include changes irrelevant to the original purpose of the fix: while they show that there may be significant amount of irrelevant changes for general bugs, Nguyen et al. [112] observed that for the majority of security fixes this was not the case – this is also supported by our findings of very "local" changes (Figure 5.5). The subset of vulnerabilities that we checked manually did not contain such refactorings.

- *Incomplete or broken histories of source code repositories.* The commit history of FOSS projects may be incomplete (e.g., migrating to different types of version control systems, archiving or refactoring), limiting the analysis capabilities. We checked the repository histories of all seven projects in our sample finding them all to be complete, except for Jenkins. In case of Jenkins, at one point in time the whole repository layout was deleted, and then re-created again. Our current implementation does not handle such cases, as it works under the assumption that repositories are complete and well-structured. Still, such cases (and similar ones) can be handled automatically, extending the current implementation with more heuristics.

- *Existence of complex "architectural" vulnerabilities.* We improved over the work by Nguyen et al. [112] by using slicing over the source code albeit limiting the scope of the slice to distinct Java methods. This may be not adequate for sophisticated, "architectural", vulnerabilities. Nguyen et al. [112] have reported only a handful of vulnerabilities in the Firefox and Chrome browsers that required to look at many files and therefore called for inter-procedural slicing analysis (we also found few of such vulnerabilities in our study). Hence, a *prima facie* evidence is that such complex and rare vulnerabilities can be considered as outliers from the perspective of our methodology. In such complex cases, additional analysis would be anyhow needed.

- *Human error.* Our manual validation of the screening tests over the subset of vulnerabilities might be biased due to human errors and wrong judgement. In order to minimize such bias, manual checks were performed by three different experts, who were cross-checking and discussing the results of each other.

The *internal validity* of the results depends on our interpretation of the collected data with respect to the analysis that we performed. We carefully vetted the data to minimize the risk of wrong interpretations. We did not create exploits to test the actual behavior of various versions against selected vulnerabilities (as it is close to impossible), performing manual code audits instead.

The *external validity* of our analysis lies in generalizing to other FOSS components. It depends on the representativeness of our sample of FOSS applications from Table 5.2, and the corresponding CVEs. As the FOSS projects that we considered are widely popular, have been developed for several years, and have a significant number of CVEs, those threats are limited for FOSS using the same language (Java), and having the same popularity. Generalization to other languages (such as C/C++) should be done with care, looking at Table 5.4.

5.11 Conclusions

We presented an automated, effective, and scalable approach for historical vulnerability screening in large FOSS components, consumed by proprietary applications. Our approach represents an enhancement of the original SZZ approach [148] and its successors (e.g., Nguyen et al. [112]), and can be applied to identify changes inducing generic software bugs. However, the fixes of such bugs should have similar properties as the security vulnerabilities that we discuss in this chapter (see Table 5.4), and should be “local”. Otherwise, different heuristics for extracting the evidence may be needed.

While our current prototype is limited to vulnerabilities in Java source code, the approach can be extended to other programming languages and configurations. In practice, it depends on the availability of a program slicer for a particular programming language.

Currently, our experimental validation has focused on a selection of software components motivated by the needs of the security team at large enterprise software vendor. It can be adapted to support other scenarios: e.g., for development teams to assess whether the vulnerable functionality is actually invoked by a consuming application (as in Plate et al. [123]), or for security researchers to improve the quality of vulnerability database entries (as in Nguyen et al. [112]).

Chapter 6

Effort Models for Security Maintenance of FOSS Components

This chapter is motivated by the need of our industrial partner to estimate the effort for security maintenance of FOSS components within its software supply chain. We aim to assist in resolving Problem 3 of secure FOSS consumption (see Chapter 1) by understanding whether various characteristics of a FOSS component (e.g., the number of contributors, the popularity, the size of the code base, etc.) may indicate the potential sources of "troubles" when it comes to the security of an application that consumes the component. We investigated the publicly available factors to identify which ones may impact the security maintenance effort.

6.1 Introduction

For most software vendors, FOSS components are an integral part of their software supply chain. Initially, we started to work with the Central Security Team of our industrial partner in order to validate whether static analysis (which was already used successfully in the industry [19, 29]) can be used for assessing the security of FOSS components. We realized that, while being the original motivation, it may be not the most urgent question to answer. A more important task may be to allow the development teams to plan the *future* security maintenance effort due to FOSS components.

In case of our industrial partner, who ships the software that is used over very long time periods (i.e., decades) by its customers, it is very common that this software contains old versions of FOSS components, which are not necessarily supported by the open source community. In this scenario, it becomes very important to be able to estimate the required security maintenance effort, which may be either caused by down-porting a fix to the actual consumed version, or by upgrading it.

During the course of the case study that we describe in Chapter 3, we identified that the developers of our industrial partner use a software inventory that contains high-level information about FOSS components. After one of the internal presentations that we held on the premises of our industrial partner at that time, we asked the following question to the audience: *"Think of different characteristics of FOSS projects for a moment. If you are selecting one as a component for your own application, which of its characteristics would you consider important from the security point of view?"*. The majority of responses were almost equally divided by the community factors (e.g., *is the community alive?*, *does the community have good a reputation?*), the popularity of a project (e.g., the *number of users* and the *age*), as well as security-related characteristics (e.g., the *history of past vulnerabilities*, and whether the developers of a project are using *security code analysis tools*). The information about most of these characteristics can be found by looking into this software inventory (which mostly contains publicly available information about FOSS projects collected from different sources), and we were not surprised that the developers were mentioning them.

As our goal is to identify which of the factors the developers can take into account, when considering the security maintenance of projects that include FOSS components, we had deliberately chosen to use the data that was present in that software inventory (and similar data sources), as it is readily available to the developers, and this is what they use when dealing with FOSS components. We formulate our research question as follows:

RQ3 *Which factors have a significant impact on the security effort to manage a FOSS component in different maintenance models?*

We concentrated our collaboration on performing a study over the 166 FOSS components that we identified during our exploratory case study (see Section 3.2 in Chapter 3). We extracted different publicly available characteristics of these components, and used them as factors. We sketched different security maintenance models for assessing whether these factors may impact the future security maintenance of the internal projects that use these components.

The case study described in Chapter 3 also helped us to delineate the key features that a model for capturing the economics of software maintenance should have for extending it to security. We build on the seminal study of Banker et al. [20] on the variables that impact the software maintenance as a whole. One of the first challenges we have to face, in comparison to Banker et al. [20], is that we cannot measure the software vendor working hours on security maintenance alone (as this data is not separable from the "functional" maintenance), nor we can measure them on each consumed FOSS component (as this data is simply not available). Hence, we need to identify suitable proxies.

In the present study we adopted several elements of the *Grounded Theory* approach initially proposed by Glaser and Strauss [64]. The goal of this approach is to construct a theory based on a phenomenon that can be explained with data [69]. The approach follows the principle of emergence [67]: the data gain their relevance within the analysis through the systematic generation and interactive conceptualization of codes, concepts and categories. Data that are similar in nature 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 [155].

To identify the right factors to consider when evaluating the impact of FOSS components on the security maintenance effort, we incorporate several maintenance models: (1) `distributed`, where each development team fixes its own set of components, and the security maintenance effort scales linearly with the number of used components; (2) `centralized`, where the vulnerabilities of a FOSS component are fixed centrally and then pushed to all consuming products (and therefore the effort scales sub-linearly in the number of products); and (3) `hybrid`, in which only the least used FOSS components are selected and maintained by individual development teams.

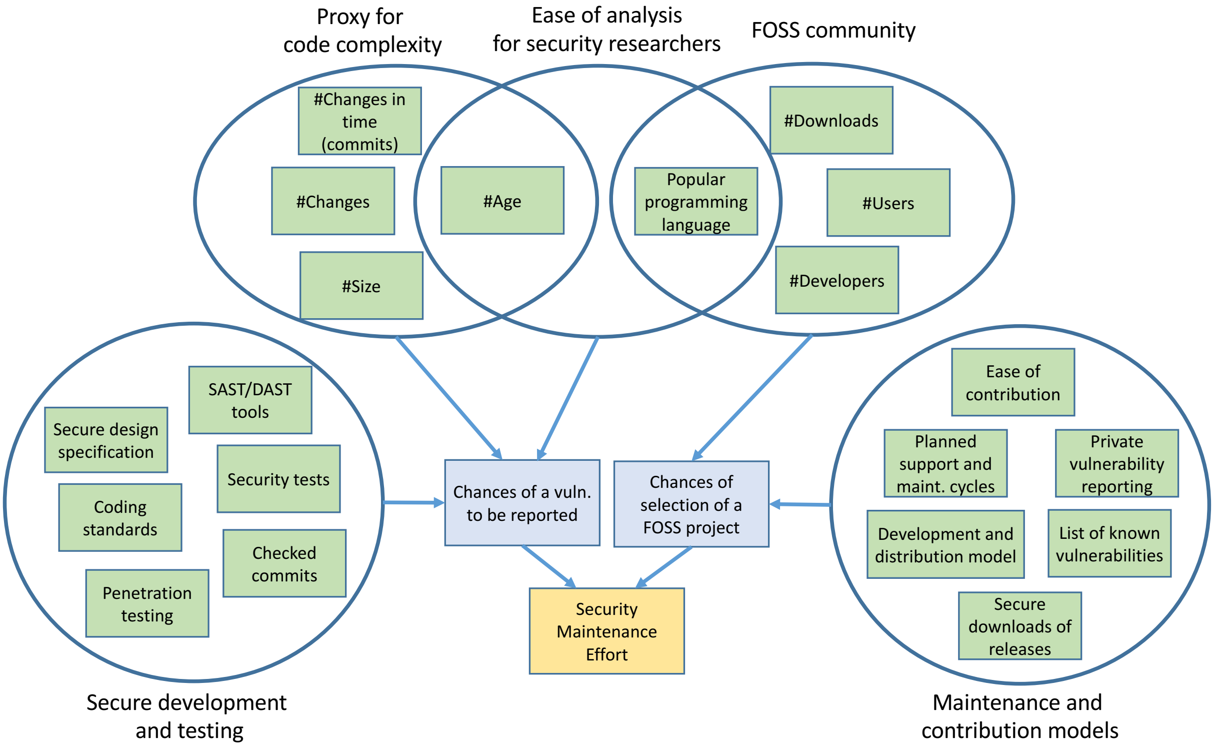
We consider these distinctions between the models, as we try to take into account the software familiarity aspect discussed in studies by Banker et al. and Chan et al. in [20, 21, 35]: in `centralized` and `hybrid` security maintenance models, a proprietary software vendor could have a centralized team of experts that consolidates the necessary knowledge about the population of FOSS projects used internally. In these models, the "bulk" security issue resolution may be beneficial when the number of usages of a component is high. These models are further discussed in Section 6.3.

6.2 A Conceptual Model of Business And Technical Drivers

A key question is how to capture the security maintenance effort in broad terms. We identified four main areas that impact the maintenance effort due to security. Figure 6.1 summarizes the relationships between them. The main areas are as follows:

1. *Proxy for code complexity:* this area of factors comprises of various quantitative characteristics of FOSS projects that represent their overall complexity. This complexity may have an impact on the number of disclosed vulnerabilities, thus affecting the maintenance effort of resolving them.

2. *FOSS community:* this area includes both quantitative and qualitative factors that reflect characteristics of community around a FOSS project, being a function of the project's general popularity and the appeal to contributors. This area also reflects the chances that a project will be selected as a component by external developers. Some of the factors from the previous category and the present one may belong to both areas at the same time (e.g., the popularity of used technologies, such as the programming language used for implementation), therefore we group them into the sub-area that represents the *ease of analysis for security researchers.*


Proxy for code complexity
Ease of analysis for security researchers
FOSS community
#Changes in time (commits)
#Changes
#Size
#Age
Popular programming language
#Downloads
#Users
#Developers
SAST/DAST tools
Secure design specification
Security tests
Coding standards
Checked commits
Penetration testing
Secure development and testing
Chances of a vuln. to be reported
Chances of selection of a FOSS project
Security Maintenance Effort
Ease of contribution
Planned support and maint. cycles
Private vulnerability reporting
Development and distribution model
List of known vulnerabilities
Secure downloads of releases
Maintenance and contribution models

The key drivers for the security maintenance effort are represented by suitable proxies (defined above), each of them corresponds to an area of factors that may impact the global security maintenance.

Figure 6.1: The model for the impact of various factors on the security maintenance

3. *Secure development and testing:* the factors that characterize how well secure development and testing activities are built into the lifecycle of a FOSS project. These factors may influence the potential security issues that the consumers of FOSS projects may face.

4. *Maintenance and contribution models:* the factors that identify the response, maintenance and support processes within a FOSS project. This particular area represents the appeal of a FOSS project for potential consumers with respect to the maintainability and support in general, as well as the availability of security-related information about the project.

Although many of the factors from the above areas are not specific to security and are used as general predictors for software bugs, we show in Section 2.3.2 (Chapter 2) that there are works such as Ozment [120], Shin and Williams [146], Shin et al. [145], that support their relevance to the security vulnerabilities.

In Section 6.4 we discuss the potential data sources from which various factors can be collected. Below we describe each area of impact in detail.

Table 6.1: Proxy for code complexity drivers

Factor	Source	Collection method	Description	References
Size	Open Hub, code repository	Automatic	The total size of the code base of a project (LoC).	[31], [63], [87], [24], [172], [20]
Changes	Open Hub, code repository	Automatic	The development activity of a project (e.g., the added/deleted lines of code)	[109] [63] [175] [145] [84] [171]
Commits	Open Hub, code repository	Automatic	The total number of the source code commits.	[109] [145] [171]
Age	Open Hub, code repository	Automatic	The age of a project (years).	[121] [171] [84]

6.2.1 Proxy For Code Complexity Drivers

The age of a project (*#Age*), its size (*#Size*), and the number of changes (*#Changes*) are traditionally used in various studies that investigate defects and vulnerabilities in software [63, 87], software evolution [24, 31] and maintenance [171]. For example, the study by Koru et al. [87] demonstrated a positive relationship between the size of a code base (LoC) of a project and its defect-proneness. Zhang [172] evaluated the LoC metric for the software defect prediction and concluded that larger modules tend to have more defects.

The number of security bugs can grow significantly over time [91], and many works (see [63, 84, 109, 145, 171, 175]) suggest a positive relation between the number and the frequency of changes in the source code (*#Changes in time*: e.g., commits, added/deleted lines of code), and the number of software defects[1].

Table 6.1 summarizes the factors that we identified for this area of impact.

6.2.2 FOSS Community Drivers

Several studies considered the popularity of FOSS projects as being relevant to their quality and maintenance [128,138,171]. It is a folk knowledge that “Given enough eyeballs, all bugs are shallow” [128], meaning that FOSS projects have the unique opportunity to be tested and scrutinized not only by their developers, but by their user community as well. In our case, we also assume that the overall popularity (the number of users, developers, etc.) will impact the chances that a particular FOSS component will be selected, thus, the user count (*#Users*) impacts the overall number of vendor’s products that consume the component. The number of downloads (*#Downloads*) can serve as another alternative

[1] We consider security vulnerabilities to be particular software defects (see [168]), which may be impacted by these factors.

Table 6.2: FOSS community drivers

Factor	Source	Collection method	Description	References
`Popular programming language`	Project website, Open Hub, code repository	Automatic	The project is mostly written in Java, C, C++, PHP, JavaScript, SQL, etc.	[98] [118]
`Developers`	Project website, Open Hub, code repository	Automatic	The number of unique developers.	[145] [171] [25]
`Users`	Project website, Open Hub	Automatic	The user count of a project (Open Hub).	[128] [124] [1] [171] [138] [167]
`Downloads`	Project website, CII Census	Semi-automatic	The number of downloads of project releases or packages.	[43]

measure for popularity [43]. Apart from the popularity measure, the number of developers (*#Developers*) of a software product may serve as an independent factor that impacts the number of bugs or vulnerabilities in that product (for instance, see [25]).

Understanding how software works is a necessary prerequisite for successful software maintenance and development (see [21, 22, 59]). Ostrand et al. [118] observed that in multi-language projects the files that are implemented in certain programming languages may contain more bugs than the others. While the authors [118] did not suggest that certain languages may be more prone to bugs, they stress the importance of considering various programming languages in connection to the number of bugs.

Also, according to the vulnerability discovery process model described by Alhazmi et al. [6], the longer is the active phase of a software the more attention it will attract, and the more malicious users will get familiar with it to “break” it (as an additional side-effect of using a *popular programming language* or a popular technology).

Table 6.2 summarizes the factors that we identified for this area of impact.

6.2.3 Secure Development, Testing, Maintenance And Contribution Drivers

Secure design specifications help the developers (especially the less experienced ones) to build a more secure product [99]. The availability of such documentation for external reviewers helps to eliminate the security defects at the early stage of product development. Additionally, the practice of internal reviews (e.g., the *source code commits are checked* before the code is pushed into production) improves the overall quality and the security of the product [100]. Finally, the presence of the secure *coding standards* as a taxonomy of common programming errors [82, 144] (which had led to security vulnerabilities in the past) reduces the amount of future vulnerabilities and the efforts for security maintenance.

Wheeler [166] suggested that successful FOSS projects should use static analysis security testing *(SAST) tools*, which should at least reduce the amount of trivial vulnerabilities [39] (see [40] for the examples of such vulnerabilities). *Penetration testing* and dynamic analysis security testing *(DAST) tools* facilitate the early discovery of security vulnerabilities [11], while maintaining the *security regression tests* for past vulnerabilities, and tests for the security-critical functionality ensure that the same (or similar) security issues are not re-introduced, thus lowering the security maintenance effort.

Planned maintenance and support for FOSS projects can be a critical factor that influences the choice of a project as a component: this will enable consumers to plan ahead when they should make a decision on whether to select a replacement project when the maintenance becomes too costly, or fork a FOSS project and provide support internally. The *ease of contribution* to the project (such as clear guidelines for new developers and transparent contribution processes) may prevent the necessity of forking a project.

The same is true for *development and distribution models* of a FOSS project: if these models do not match the models used by their consumers (or are not taken into account), this may bring disturbances to the software maintenance processes of consumers, significantly increasing the corresponding efforts.

There are several security related-factors that may not impact the security maintenance effort significantly, but have a direct effect on the reputation a project, making it more or less appealing for selection. For instance, a project that provides means for *downloading its source and binary packages securely* (via https, cryptographically signed or hashed) protects its users from the malware that malicious third parties could inject into downloads. Additionally, the *private vulnerability reporting* process allows them to issue a fix in a timely manner and notify their commercial partners before the vulnerability will be publicly known.

Finally, the presence of a *list of known past vulnerabilities* maintained by the project could as well simplify the task of identifying whether certain versions of that project are vulnerable.

Table 6.3 summarizes the factors that we identified for this area of impact.

6.3 From Drivers to Effort Model for FOSS Maintenance

In Chapter 3 we have sketched some of the security activities that a development team must perform during the maintenance phase. Unfortunately, a team is normally assigned to several tasks, with security maintenance being only one of them. Therefore, it is close to impossible to get analytical accounting for security maintenance to the level of individual vulnerabilities. Furthermore, when a FOSS component is shared across different consuming applications, each development team can differ significantly in the choice of the solution and hence in the effort to implement it.

Table 6.3: Secure development and testing, maintenance and contribution model drivers

Factor	Source	Collection method	Description	References
`Security tests`	Project website, code repository	Manual	The test suite contains tests for past vulnerabilities (regression) and/or tests for security functionality.	[166], [167], [1], [40], [20]
`Private vuln. reporting`	Project website	Manual	There is a possibility to report security issues privately.	[105]
`SAST/DAST tools`	Project website, code repository, Coverity website	Manual	A project is using security code analysis tools during development.	[166], [167], [1], [39]
`Secure design specs`	Project website, documentation	Manual	The secure design specification of the project is documented.	[99]
`Penetration testing`	Project website, documentation	Manual	The penetration testing is performed regularly by the project developers.	[11]
`Coding standards`	Project website, documentation, code repository	Manual	Secure coding standards are documented.	[20], [82], [144]
`List of known vulnerabilities`	Project website, vulnerability databases	Manual	Past security vulnerabilities of the project are documented and are publicly available.	[167]
`Devel. & distr. model`	Project website, documentation	Manual	The patch and release cycles are documented.	[174]
`Planned support & maintenance`	Project website, documentation	Manual	The maintenance roadmap and support cycles for different versions of a project are documented.	[166]
`Ease of contribution`	Project website, documentation, code repository	Manual	Clear guidelines for new developers or potential contributors are present.	
`Checked commits`	Project website, documentation, code repository	Manual	There exists a review process for new contributions, including security code reviews.	[100], [167]
`Secure downloads of releases`	Project website, package distribution stats., CII Census	Manual	The project provides means for downloading the source code or binaries securely (e.g., https protocol, cryptographic signatures).	

Banker et al. [20] have already shown that maintenance models do not scale linearly, and that software maintenance may be affected by economies of scale. Preliminary discussions with developers and researchers of our industrial partner suggested the combination of vulnerabilities of the FOSS component itself and the number of company's products using it can be a satisfactory proxy for the security maintenance effort. A large number of vulnerabilities may be the sign of either a sloppy process or a significant attention by hackers and may warrant a deeper analysis during the selection phase or a significant response during the maintenance phase. This effort is amplified when several development teams are asking to use a FOSS component, as a vulnerability which eschewed detection may impact several hundred products and may lead to several security patches for different products.

We also considered the option of using the number of exploits from the Offensive Security database (`http://www.offensive-security.com`) as an alternative metric. Numbers of vulnerabilities and exploits have a strong correlation in our sample of projects (Spearman's rho = 0.71, $p < 0.01$) – it could be because security researchers can create exploits to test published vulnerabilities and, alternatively, they can create exploits to test a vulnerability they have just found (for which a CVE entry does not yet exist). We tested both values without finding significant differences and, for simplicity, we use the number of vulnerabilities as the proxy for effort since this was considered by developers a "standardized" information available from known trusted sources, whereas exploits would come from less neutral sources.

We assume that the effort structure has the following form:

$$e = e_{\text{fixed}} + \sum_{i=1}^{m} e_i \tag{6.1}$$

where e_i is a variable effort that depends on the i-th FOSS component, and e_{fixed} is a fixed effort that depends on the security maintenance model (e.g., the initial set up costs). For example, with a distributed security maintenance approach an organization will have less communication overhead and more freedom for developers in distinct product teams, but only if a small number of teams are using a component.

Let $|vulns_i|$ be the number of vulnerabilities that have been cumulatively fixed for the i-th FOSS component and let $|products_i|$ be the number of proprietary products that use the component:

1. In `centralized` model a security fix for all instances of a FOSS component is issued once by the Central Security team of the company and then distributed between all products that are using it. This may happen when, as a part of FOSS selection process, development teams must choose only components that have been already used by other teams and are supported by the company. To reflect this case, the effort for security maintenance in this model scales logarithmically with the number

of products using a FOSS component.

$$e_i \propto \log\big(|vulns_i| \cdot |products_i|\big) \tag{6.2}$$

2. `Distributed` model covers the case when security fixes are not centralized within a company, so each development team has to take care of security issues in FOSS components that they use. In this scenario the effort for security maintenance increases linearly with the number of products using a FOSS component.

$$e_i \propto |vulns_i| \cdot |products_i| \tag{6.3}$$

3. `Hybrid` model combines the two previous models: security issues in the least consumed FOSS components (e.g., used only by lowest quartile of products consuming FOSS) are not fixed centrally. After this threshold is reached and some effort linearly proportional to the threshold of products to be considered has been invested, the company fixes them centrally, pushing the changes to the remaining products.

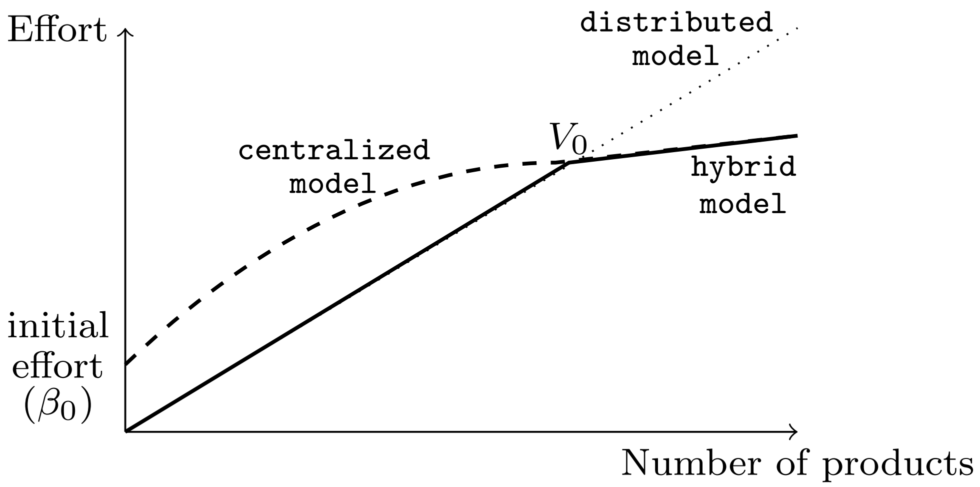
$$e_i \propto \begin{cases} |vulns_i| \cdot |products_i| & \text{if } |products_i| \leq p_0 \\ p_0 \cdot |vulns_i| + \log\Big(|vulns_i| \cdot \big(|products_i| - p_0\big)\Big) & \text{otherwise} \end{cases} \tag{6.4}$$

As shown in Figure 6.2, `hybrid` model is a combination of `distributed` and `centralized` models, when centralization has a steeper initial effort. The point V_0 is the switching point where the company is indifferent between `centralized` and `distributed` effort models. `Hybrid` model captures the possibility of a company to switch models after (or before) the indifference point. The fixed effort of `centralized` model is obviously higher than the one of `distributed` model (e.g., setting up a centralized team for fixing vulnerabilities, establishing and communicating a fixing process, etc.).

Hence, we extend the initial function after the threshold number of products p_0 is reached, so that only a logarithmic effort is paid on the *remaining* products. This has the advantage of making the effort e_i continuous in $|products_i|$. An alternative would be to make the effort logarithmic in the overall number of products after $|products_i| > p_0$. This would create a sharp drop in the effort for the security maintenance of FOSS components used by several products after p_0 is reached. This phenomenon is neither justified on the field, nor by the economic theory. In the sequel, we have used for p_0 the lowest quartile of the distribution of the selected products.

We are not aiming to select a particular model – we consider them as equally possible scenarios. Our goal is to see which of the FOSS characteristics can have an impact on the security maintenance effort when such models are in place, keeping in mind that this impact could differ from one model to another.

We now define the impact that the characteristics of the i-th FOSS component have on the expected effort e_i as a (not necessarily linear) function f_i of several variables and



`Centralized` model has initial set up costs β_0 (setting up and training the team, communications overhead, etc.), and the effort scales logarithmically with the number of applications where components are used, and the number of vulnerabilities in them. `Distributed` model does not suffer from β_0, but the growth in effort is linear, which generates additional costs when the number of usages of a component is large. `Hybrid` model is a combination of `distributed` and `centralized` models, when centralization has a steeper initial cost. V_0 is the switching point in the number of usages where a company is indifferent between the previous two models.

Figure 6.2: Illustration of the three effort models

a stochastic error term ϵ_i:

$$e_i = f(x_{i1}, \ldots, x_{il}, y_{il+1}, \ldots, y_{im}, d_{im+1}, \ldots, d_n) + \epsilon_i \tag{6.5}$$

The variables $x_{ij}, j \in [1, l]$ impact the effort as scaling factors, so that a percentage change in them also implies a percentage change in the expected effort. The variables $y_{ij}, j \in [l+1, m]$ directly impact the value of the effort. Finally, the dummy variables $d_{ij}, j \in [m+1, n]$ denote qualitative properties of the code captured by a binary classification in $\{0, 1\}$.

For example, in our sample of FOSS components, the 36-th component is "Apache CXF", and the first scaling factor for effort is the size of the code base of the component (LoC), so that $x_{i,1} \doteq locs_i$, and $x_{36,1} = 868,183$.

Given the above classification, we can further specify the impact equation for the i-th component as follows

$$\log(e_i) = \beta_0 + \log\left(\prod_{j=1}^{l}(x_{ij}+1)^{\beta_j}\right) + \sum_{j=l+1}^{m} \beta_i \cdot e_{ij}^{y} + \sum_{j=m+1}^{n} \beta_i \cdot d_{ij} + \epsilon_i \tag{6.6}$$

where β_0 is the initial fixed effort for a specific security maintenance model.

All three models reflect to a certain extent on the experience of our industrial partner in managing the security maintenance of FOSS components. These models focus on technical aspects of security maintenance of consumed FOSS components, putting aside all organizational aspects[2] such as the overhead of communications between different development and maintenance teams, and interactions with customers. The models aim to reflect (on the abstract level) how the cumulative maintenance effort will look like when all FOSS components are fixed by a single team, many independent teams, or a mixture of both.

Currently, we do not consider how exactly the knowledge about software security, domain-specific knowledge about FOSS components, and other types of expertise may be distributed within various teams in the three models (for example, `centralized` model could have specialists that have in-depth knowledge in software security, but only high-level knowledge about every FOSS component used company-wise, while in `distributed` model the members of independent teams could have less knowledge in software security, but much more knowledge of the FOSS components that they actually use). This may affect, for example, how much effort does it take to perform fixes in certain locations of FOSS components. These knowledge distributions may vary from company to company and from team to team. It would be interesting to study how various differences in expertise may affect the security maintenance effort with actual development teams in several large software development companies – this could be a potential line for future work.

6.4 Identification of Empirical Data

We considered the following public data sources to obtain the factors of FOSS projects that could impact the security effort in maintaining them:

1. **National Vulnerability Database (NVD)** – the US government public vulnerability database. We use it as the main source of public vulnerabilities in FOSS components (`https://nvd.nist.gov/`).
2. **Open Sourced Vulnerability Database (OSVDB)** – an independent public vulnerability database. We use it as the secondary source of public vulnerabilities to complement the data that we obtain from the NVD (`http://osvdb.org`).
3. **Black Duck Code Center** – a commercial platform for the open source governance that can be used within an organization for the approval of the usage of FOSS components by identifying legal, operational and security risks that can be caused by these components. We use the installation of our industrial partner to identify the most "interesting" FOSS components.
4. **Open Hub (formerly Ohloh)** – a free offering from Black Duck supported by an online community that maintains statistics which represent various properties of FOSS projects. Additionally, Open Hub retrieves data from the source code repositories of FOSS projects (`https://www.openhub.net/`).

[2] For organizational aspects see [26].

5. **Coverity Scan Service website** – in 2006 Coverity started the initiative of providing free static security testing code scans for FOSS projects, and many of the projects have registered since that time. We use this website as one of the sources that can help to infer whether a FOSS project is using SAST tools (`https://scan.coverity.com/projects`)
6. **Core Infrastructure Initiative (CII) Census** – the experimental methodology for parsing through the data of open source projects to help identify projects that need some external funding in order to improve their security. We use a part of their data[3] to obtain information about Debian installations which is used within Census as the measure of popularity of a project, and which we use as an additional measure of popularity as well.

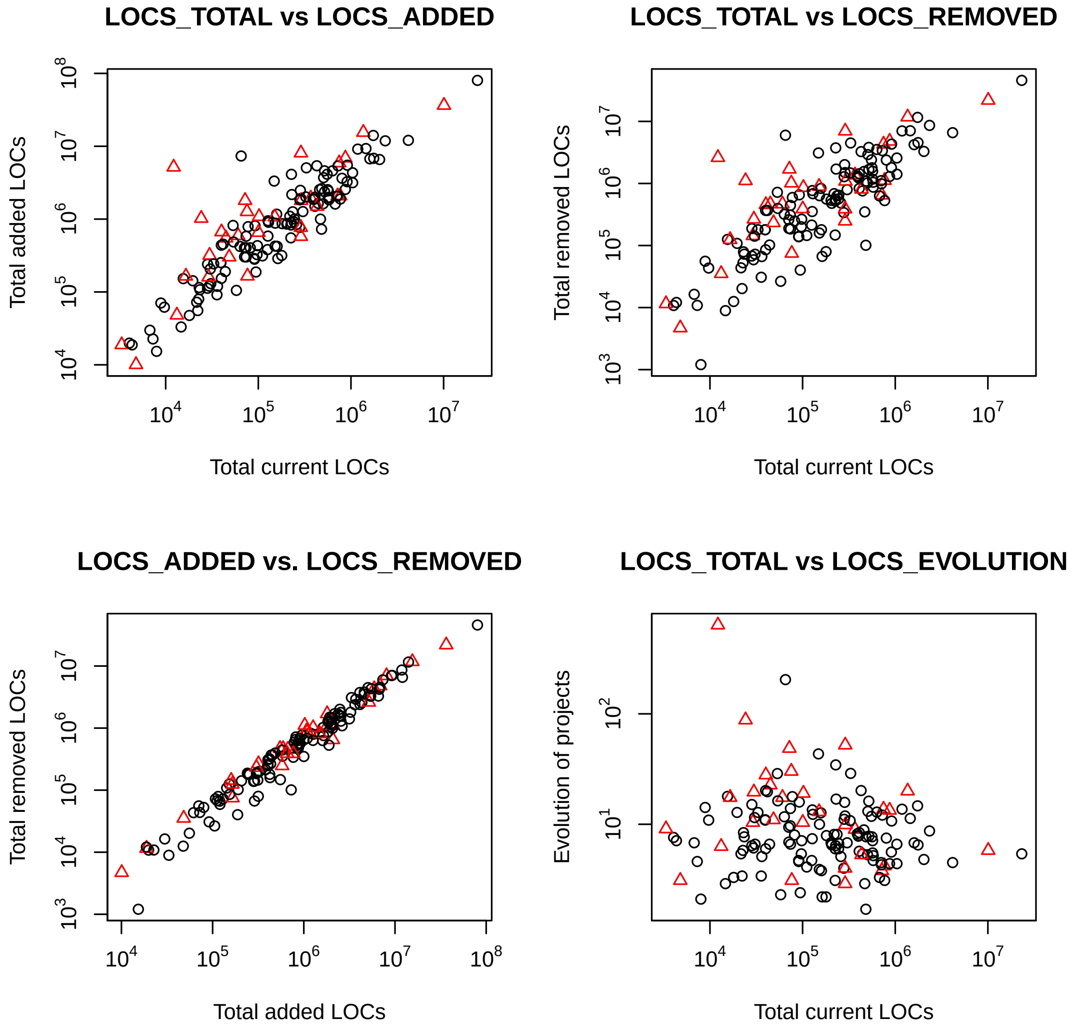
After searching for factors in a smaller sample of 50 projects we understood that only variables that could be extracted automatically and semi-automatically are interesting for the maintenance phase. Gathering the data manually introduces bias and limits the size of a data set that we can analyze, and, therefore, the validity of the analysis at all. Further, from the software vendor's perspective, this data cannot be effectively collected and monitored on a periodic basis. Thus, we removed the manual variables and expanded the initial data set up to 166 projects (projects that are consumed by at least 5 products as indicated in the Black Duck repository of our industrial partne): we showed the descriptive statistics of these projects earlier in Table 3.2 and Figure 3.1 (Chapter 3). Moreover, in spite of their intuitive appeal, we excluded dummy variables related to the programming languages[4] from the data analysis, because we realized that almost all projects have components of both, so these variables would not be discriminating. We also had to exclude other dummy variables (e.g., related to the presence of security tests, or usages of SAST/DAST tools), as we realized that, given the amount of projects that we considered, the most accurate way to extract this information is manual exploration. However, even manual work would not be able to resolve the bias: when we could not answer the question *"are the developers of a project using SAST/DAST tools?"* by looking at various sources, it could only mean that we failed to find this information, and that the answer to this question is unknown.

We also tried to find commonalities between FOSS projects in order to cluster them. However, this process would introduce significant human bias. For example, the "Apache Struts 2" FOSS component is used by the vendor as a library in one project, and as a development framework in another one (indeed, it can be considered to be both a framework and a set of libraries). If we "split" the "Apache Struts 2" data point into another two instances marked as "library" and "framework", this would introduce dependency relations

[3] `https://www.coreinfrastructure.org/programs/census-project`

[4] Such as variables that indicate whether there are parts of the code base written in programming languages without a built-in memory management, or in scripting languages that could be prone to code injection vulnerabilities (see [167]).

between these data points. Assigning arbitrarily only one category to such data points would also be inappropriate. A comprehensive classification of FOSS projects would require to perform a large number of interviews with developers to understand the exact nature of the usage of a component and the security risk. However, it is unclear what would be the added value to *developers* of this classification and the time spent for the interviews.



The number of added and deleted lines of code could not be used as independent variables, since they have a strong correlation with each other (as well as with the total number of lines of code, and other variables). On the other hand, the fraction of changed lines of code by the total size (locsEvolution) can be used as an independent predictor. The red triangles show the fraction of lines of code in scripting languages, the black circles show other languages. The axes are logarithmic.

Figure 6.3: The rationale for using the **locsEvolution** metric

Table 6.4: Cross correlations of the explanatory variables

(a) Spearman correlation coefficients

	userCount	debianInst	contribs	locs	locsEvolution
years	0.47	0.47	0.09	0.33	0.04
userCount		0.45	0.34	0.35	0.19
debianInst			-0.01	-0.01	-0.15
contribs				0.39	0.31
locs					-0.12

(b) Variance inflation factors

Variable	VIF
years	1.27
userCount	1.11
debianInst	1.18
contribs	1.12
locs	1.14
locsEvolution	1.02

As we are interested in performing a regression analysis to assess the impact of individual explanatory variables on the dependent variable – the maintenance effort, we had to alter the number of variables in the analysis, as some of the initial variables that we considered have strong correlations with each other: for instance, we had to remove the number of commits as it strongly correlates with the size of the code base and the number of developers. To assess the potential impact of the popularity of a programming language, we tried to divide the size of the code base into two different variables: one of them showing the size of the code base written in popular languages (e.g., Java, C/C++, PHP, JavaScript), and the other showing the size of the code base implemented in other less popular languages (e.g., Lisp, Scala). Eventually, we understood that it would introduce the same multi-collinearity problem. Additionally, the numbers of deleted and added lines of code has a strong correlation with the numbers of commits and developers, and the size of the code base.

For the latter, we had to come up with another metric that would capture the changes to the source code – **locsEvolution**. Figure 6.3 illustrates why we could not use the added and deleted lines of code as factors, as they correlate with each other and with the total lines of code **locs**. As **locsEvolution** does not have a strong correlation with **locs**, it can be used as an independent variable.

Finally, we performed the correlation analysis of the remaining variables in order to determine whether the multi-collinearity problem remains – Table 6.4. We first built the correlation matrix using Spearman rank correlations: as can be seen from Table 6.4a, there are weak-to-moderate correlations in some of the variables. However, according to Stevens [152, pp74], the presence of such correlations does not necessarily affect the regression results. Therefore, we also calculated the variance inflation factors of each variable (Table 6.4b), which is a widely used measure for assessing the degree of multi-collinearity of independent variables. According to the rule of thumb proposed by Myers [108, pp369], these values are acceptable and indicate that the selected explanatory variables do not have significant cross influences.

Table 6.5: Variables used for analysis

The table provides a short description of each selected explanatory variable, as well the rationale for including it into the models. The last column shows the expected impact that a variable may have on the effort (positive or negative).

Factor	**Description**	**Rationale**	**Exp.** β
locs (x_{ij})	Number of lines of code in various programming languages (excluding the source code comments).	The more there are lines of code, the more there will be new vulnerabilities.	+
locsEvolution (y_{ij})	The fraction of the total number of added and deleted lines of code by the total number of lines of code.	We could not use the added/deleted lines as factors, as they have a strong correlation with each other, and with the total lines of code. As **locsEvolution** does not have a strong correlation with the total lines, it can be used as an independent variable that reflects the total level of changes in a project.	+
userCount (y_{ij})	The number of active users (measured by Open Hub).	The more popular the project is, the more it is likely that new vulnerabilities will be discovered by the users.	+
debianInst (y_{ij})	The number of package downloads from the Debian repository.	This variable provides an additional measure for popularity, however, these two factors are not exactly correlated, as some software is usually downloaded from the Web (e.g., Wordpress), so it may be unlikely that someone would install it from the Debian repository, even if a corresponding package exists. On the other hand, some software may be distributed only as a Debian package.	+
years (y_{ij})	The age of a project (in years).	More vulnerabilities can be discovered over time.	+
contribs (y_{ij})	The number of unique contributors for the whole history of the source code repository of a project.	Too few contributors might induce vulnerabilities, as there may be not enough workforce or expertise to catch security defects before releases, so that other people report them (resulting in CVE entries).	−

Table 6.5 lists the set of finally selected explanatory variables, and shows their expected impact on the security maintenance effort with respect to the model in Section 6.3. Table 6.6 shows the descriptive statistics of the response and explanatory variables.

Table 6.6: Descriptive statistics of the variables used for the analysis

	Statistic					
Variable	Min	1st Quartile	Median	Mean	3rd Quartile	Max
log(effort_centralized)	0.51	1.29	1.49	1.50	1.75	2.32
log(effort_distributed)	0.69	3.64	4.44	4.81	5.76	10.13
log(effort_hybrid)	0.69	3.24	3.78	4.12	4.94	8.42
years	1.00	7.00	10.00	10.27	13.80	28.00
userCount	0.00	9.00	52.00	258.00	178.00	9390.00
log(debianInst+1)	0.00	3.76	7.25	6.56	9.46	12.08
contribs	1.00	15.00	32.00	115.20	101.20	1433.00
log(locs)	7.88	10.70	12.02	11.89	13.09	16.96
locsEvolution	0.53	1.58	1.96	2.09	2.53	6.46

6.5 Analysis

To assess the potential impact of individual factors that characterize FOSS components on the security maintenance effort, we employ the least-square regression (OLS). This method assumes that the regression function is linear in the input, allowing for an easy interpretation of the impact dependencies between explanatory and response variables, as well as predictions of potential future values of the response. Most of other regression methods can be perceived as modifications of the linear regression method that is relatively simple and transparent as opposed to its successors [80, Chapter 3].

Our reported R^2 values (0.21, 0.34, 0.42) and F-statistic values (7.17, 13.46, 19.28) are acceptable, as our purpose is to see which variables have the impact. We have not considered the variables in Table 6.3, as they cannot be automatically collected by the vendor. Thus, we can only explain part of the variance.

The results of estimates for each security effort model are given in Table 6.7. These results show that there is a positive relation between the size of the code base **locs** and the effort variable (statistically significant only in `distributed` and `hybrid` models). Zhang [172] and Koru et al. [87] show a positive relation between the size of a code base and the number of defects (which is a component of the effort in our model).

The **locsEvolution** and **contribs** variables do not seem to have an impact. We expected the opposite result, as many works (e.g., [63,109,145]) suggest a positive relation between the number and the frequency of changes and defects. However, these works assessed the changes with respect to distinct releases or components, while we are using the cumulative number of changes for all versions in a project; we may not capture the impact because of this.

Security bugs grow over time [91], which can be explained by the interest of attack-

Table 6.7: Regression Results

locs *has positive and statistically significant impact in* ***`distributed`*** *and* ***`hybrid`*** *models.* ***years*** *has positive and statistically significant impact in all three models.* ***userCount*** *and* ***debianInst*** *have statistically significant, but small impact in all three models.*

	Centralized model		Distributed model		Hybrid model	
Intercept	$8.90 \cdot 10^{-1}$	(3.89)***	1.83	(2.15)*	$7.92 \cdot 10^{-1}$	(1.21)
years	$1.97 \cdot 10^{-2}$	(2.64)**	$8.14 \cdot 10^{-2}$	(2.93)**	$7.12 \cdot 10^{-2}$	(3.34)**
userCount	$7.86 \cdot 10^{-5}$	(2.12)*	$5.02 \cdot 10^{-4}$	(3.63)***	$4.09 \cdot 10^{-4}$	(3.86)***
debianInst	$1.75 \cdot 10^{-6}$	(2.54)*	$9.31 \cdot 10^{-6}$	(3.63)***	$8.51 \cdot 10^{-6}$	(4.33)***
contribs	$-1.17 \cdot 10^{-4}$	(-0.85)	$-5.33 \cdot 10^{-4}$	(-1.04)	$-2.32 \cdot 10^{-4}$	(-0.59)
log(locs)	$3.02 \cdot 10^{-2}$	(1.48)	$1.56 \cdot 10^{-1}$	(2.06)*	$1.95 \cdot 10^{-1}$	(3.35)**
locsEvolution	$2.43 \cdot 10^{-4}$	(0.43)	$1.04 \cdot 10^{-3}$	(0.49)	$1.21 \cdot 10^{-3}$	(0.75)
N	166		166		166	
Multiple R^2	0.21		0.34		0.42	
Adjusted R^2	0.18		0.31		0.40	
F-statistic	7.17 ($p < 0.01$)		13.46 ($p < 0.01$)		19.28 ($p < 0.01$)	

Note, t-statistics are in parentheses. Signif.codes: * 1%, ** 0.01%, *** 0.001%

ers [6], and the vulnerability discovery rate being highest during the active development phase of a project [95]. Our results show that **years** - the age of a project, has a significant impact in all three models, thus supporting these observations.

We found that in our models the number of external users (**userCount** and **debianInst**) of a FOSS component has a small but statistically significant impact. This could be explained by the intuition that only a major increase of the popularity of a FOSS project could result in more development teams that select the project for consumption, and that not every user would have enough knowledge in software security for finding and reporting new vulnerabilities.

6.6 Threats to validity

The construct validity might be affected by errors in the data collection process, as well as the accuracy of data in the data sources that we used. To combat the first threat we carefully checked the collected data, removing duplicates and performing manual spot checks. The latter threat should be minimal, as we used the same data sources that the developers of our industrial partner are typically using.

The internal validity might suffer from wrong interpretation of the results and the choice of the dependent variable. We could not measure the direct security maintenance effort (e.g., working hours of developers) as it is not separable from the regular maintenance, and it could not be separated by distinct FOSS components. Therefore, we had to choose a proxy variable for the security maintenance effort that consists of the number of publicly known vulnerabilities in a FOSS component and the number of usages of these components in the internal software applications. While this approximation may lead to a potential threat to validity, we selected this dependent variable as being relevant to the security maintenance effort based on our discussions with the developers and researchers of our industrial partner (being also limited on the data that is available to the developers of our industrial partner).

To minimize the threats due to potential lack of generalizability and potential overfitting of the results, we had to limit the number of independent variables that we considered for regression analysis. Therefore, our conclusions are based on the analysis of a subset of factors that we initially identified. Moreover, as we were deliberately using only the high-level information that is available to the developers of our industrial partner, there could be a lack of causation between the factors that we assessed and our measure of the effort. Still, our findings are supported by the existing literature on software defect and vulnerability prediction, which lets us to assume that this threat is minimized.

The external validity might suffer from the lack of generalizability. The sample of FOSS projects that we considered is relevant for our industrial partner, which may not be the case for other software vendors. Still, the majority of FOSS components correspond to the Java ecosystem, and Java is one of the most popular programming languages (according to TIOBE index[5]). This suggests that the study is likely relevant for other software vendors as well.

6.7 Conclusions

In this chapter, we have investigated publicly available factors that can impact the effort required for performing security maintenance within large software vendors that have extensive consumption of FOSS components. We have defined three security effort models – `centralized`, `distributed`, and `hybrid`, and selected variables that may impact these models. We automatically collected data on these variables from 166 FOSS components currently consumed by the products of our industrial partner, and analyzed the statistical significance of these variables.

As a proxy for security maintenance effort of consumed FOSS components we used the combination of the number of products using these components, and the number of known vulnerabilities in them. As the summary of our findings, the main factors that influence the security maintenance effort whose are its age, size, and popularity. In fact, the external popularity of a FOSS component has statistically significant but small impact on the effort, meaning that only large changes in popularity will have a visible effect.

[5] http://www.tiobe.com/tiobe-index/

We had to limit the number of variables that we consider in our effort models to avoid potential over-fitting and the lack of generalizability of the results. This is a direct consequence of the fact that the ground truth data is difficult to obtain, and the amount of data that is available is not uniform across all population of FOSS projects.

Chapter 7

Conclusions and Future Work

The aim of this dissertation is to aid large software vendors (in particular, our industrial partner), that integrate many FOSS components into their software products, in facilitating security maintenance of these components: we provide a solution for testing and adaptation of existing exploits against web applications to identify whether vulnerabilities can be reproduced in specific environments in which these applications are deployed (Chapter 4); we propose a vulnerability screening method for identification of versions likely affected by a newly disclosed vulnerability (Chapter 5); finally, we assess the of impact of various FOSS component characteristics on their security maintenance effort (Chapter 6).

In Chapter 4 we discussed TestREx– a testbed for repeatable exploits. The work behind TestREx is currently protected with a US patent [136], and, besides expanding our corpus, we intend to apply TestREx for several research activities, such as large-scale testing of static analysis tools and semi-automatic generation of test cases for web applications. We also used it successfully for teaching a Master course on security vulnerabilities in web applications.

To move towards the generation of test cases, we plan to refine our implementation of exploit scripts into a hierarchy of exploit classes that would help to write exploits faster. This could be achieved by factoring common attributes of exploit types and altering the exploit attributes in case if a given exploit did not work. Another possible future direction for TestREx can be in helping testers to find various sets of software environments that serve as the necessary pre-condition for a certain vulnerability exploit to successfully execute.

In Chapter 5 we discussed the vulnerability screening method for estimating the likelihood of an older version of a FOSS component to be affected by a newly disclosed vulnerability, using the vulnerability fix. The software prototype that implements the method can be used as a standalone tool, or be integrated into other tools that already exist and are used by the developers during the SDL phase or post-release maintenance activities. Currently, we are actively refining the research prototype for the possibility of further integration with the existing toolchain of our industrial partner. To improve the vulnerability screening we see several lines of future work that would allow to better understand the "quality-versus-speed" trade-offs as well as to extend the scope of our approach:

- Improve the quality of the vulnerability screening by associating changes across multiple files, as well as investigating the impact of more precise slicing algorithms. However, we do not expect a significant improvement in the latter direction as, in our experience, vulnerabilities were mostly fixed locally by modifying few lines.
- Improve the quality of estimation of the update effort by including changes in build dependencies (direct and transitive), which might influence the estimate. For example, consider an application that only runs on Java 1.4 and a FOSS upgrade that would require Java 1.8: to resolve this dependency conflict, either the fixes for the FOSS components need to be back-ported to Java 1.4, or the entire application needs to be ported to Java 1.8.
- Extend the approach to more programming languages and test the approach on different types of projects.

In Chapter 6 we have investigated the publicly available factors that can impact the effort required for performing security maintenance of FOSS components in software products of our industrial partner. We have defined three security effort models – `centralized`, `distributed`, and `hybrid`, and selected variables that may impact these models. We automatically collected data on these variables from 166 FOSS components currently consumed by various products of our industrial partner, and analyzed these models. As a proxy for security maintenance effort of consumed FOSS components we used the combination of the number of products using these components, and the number of known vulnerabilities in them. As the summary of our findings, the main factors that influence the security maintenance effort are the amount of lines of code of a FOSS component, the age of the component, and its popularity. We have also observed that the external popularity of a FOSS component has statistically significant but small impact on the effort, meaning that only large changes in popularity will have a visible effect.

For further development of security maintenance effort models, we plan to collect a wider dataset on FOSS projects and their factors (as well as their vulnerabilities), and assess the potential effect of these other factors. Using this data for the effort prediction is also a promising direction for the future work.

Bibliography

[1] Mark Aberdour. Achieving quality in open-source software. *IEEE Software*, 24(1):58–64, 2007.

[2] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, 2011.

[3] Bram Adams, Ryan Kavanagh, Ahmed E. Hassan, and Daniel M. German. An empirical study of integration activities in distributions of open source software. *Empirical Software Engineering*, 21(3):960–1001, 2016.

[4] Alan Agresti and Christine A. Franklin. *Statistics: the art and science of learning from data*. Pearson, 2012.

[5] Norita Ahmad and Phillip A. Laplante. A systematic approach to evaluating open source software. In *Strategic Adoption of Technological Innovations*, pages 50–69. IGI Global, 2013.

[6] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. Security vulnerabilities in software systems: A quantitative perspective. In *Proceedings of the 19th IFIP WG 11.3 Working Conference on Data and Applications Security (DBSEC'05)*, 2005.

[7] Luca Allodi, Vadim Kotov, and Fabio Massacci. Malwarelab: Experimentation with cybercrime attack tools. In *Proceedings of 6th USENIX Workshop on Cyber Security Experimentation and Test (CSET'13)*, 2013.

[8] Luca Allodi and Fabio Massacci. Comparing vulnerability severity and exploits using case-control studies. *ACM Transactions on Information and System Security*, 17(1):1–20, 2014.

[9] Stephanos Androutsellis-Theotokis, Diomidis Spinellis, Maria Kechagia, and Georgios Gousios. Open source software: A survey from 10,000 feet. *Foundations and Trends in Technology, Information and Operations Management*, 4(3-4):187–347, 2011.

[10] Claudio Agostino Ardagna, Ernesto Damiani, and Fulvio Frati. FOCSE: an OWA-based evaluation framework for OS adoption in critical environments. In *Proceedings of IFIP International Conference on Open Source Systems (OSS'07)*, 2007.

[11] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005.

[12] Andre Arnes, Paul Haas, Giovanni Vigna, and Richard A. Kemmerer. Digital forensic reconstruction and the virtual security testbed vise. In *Proceedings of the 3rd International Conference of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06)*, 2006.

[13] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, 2010.

[14] Lerina Aversano and Maria Tortorella. Evaluating the quality of free/open source systems: A case study. In *Proceedings of 12th International Conference on Enterprise Information Systems (ICEIS'10)*, 2010.

[15] Lerina Aversano and Maria Tortorella. Quality evaluation of FLOSS projects: Application to ERP systems. *Information and Software Technology*, 55(7):1260–1276, 2013.

[16] Claudia Ayala, Øyvind Hauge, Reidar Conradi, Xavier Franch, Jingyue Li, and Ketil Sandanger Velle. Challenges of the open source component marketplace in the industry. In *Proceedings of IFIP International Conference on Open Source Systems (OSS'09)*, 2009.

[17] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'07)*, 2007.

[18] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static code analysis to detect software security vulnerabilities-does experience matter? In *Proceedings of the 4th International Conference on Availability, Reliability and Security (ARES'09)*, 2009.

[19] Ruediger Bachmann and Achim D. Brucker. Developing secure software: A holistic approach to security testing. *Datenschutz und Datensicherheit*, 38(4):257–261, 2014.

[20] Rajiv D. Banker, Srikant M. Datar, and Chris F. Kemerer. A model to evaluate variables impacting the productivity of software maintenance projects. *Management Science*, 37(1):1–18, 1991.

[21] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–95, 1993.

[22] Rajiv D. Banker and Sandra A. Slaughter. A field study of scale economies in software maintenance. *Management Science*, 43(12):1709–1725, 1997.

[23] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, 2015.

[24] Karl Beecher, Andrea Capiluppi, and Cornelia Boldyreff. Identifying exogenous drivers and evolutionary stages in FLOSS projects. *Journal of Systems and Software*, 82(5):739–750, 2009.

[25] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. The limited impact of individual developer data on software defect prediction. *Empirical Software Engineering*, 18(3):478–505, 2013.

[26] Lotfi ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, Achim D. Brucker, and Philip Miseldine. Factors impacting the effort required to fix security vulnerabilities. In *Proceedings of 18th International Conference on Information Security (ISC'15)*, 2015.

[27] Terry Benzel. The science of cyber security experimentation: The DETER project. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*, 2011.

[28] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the 7th European Software Engineering Conference (ESEC/FSE'09)*, 2009.

[29] Achim D. Brucker and Uwe Sodan. Deploying static application security testing on a large scale. *Datenschutz und Datensicherheit*, pages 91–101, 2014.

[30] Joan Calvet, Carlton Davis, José M Fernandez, Wadie Guizani, Matthieu Kaczmarek, Jean-Yves Marion, and Pier-Luc St-Onge. Isolated virtualised clusters: testbeds for high-risk security experimentation and training. In *Proceedings of 3rd USENIX Workshop on Cyber Security Experimentation and Test (CSET'10)*, 2010.

[31] Andrea Capiluppi. Models for the evolution of os projects. In *Proceedings of International Conference on Software Maintenance (ICSM'03)*, 2003.

[32] Thomas E. Carroll, David Manz, Thomas Edgar, and Frank L. Greitzer. Realizing scientific methods for cyber security. In *Proceedings of the Workshop on Learning from Authoritative Security Experiment Results (LASER'12)*, 2012.

[33] Brian W. Carver. Share and share alike: Understanding and enforcing open source and free software licenses. *Berkeley Technology Law Journal*, 20(1):443–481, 2005.

[34] Hasan Cavusoglu, Huseyin Cavusoglu, and Jun Zhang. Security patch management: Share the burden or share the damage? *Management Science*, 54(4):657–670, 2008.

[35] Taizan Chan, Siu Leung Chung, and Teck Hua Ho. An economic model to estimate software rewriting and replacement times. *IEEE Transactions on Software Engineering*, 22(8):580–598, 1996.

[36] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of 22nd IEEE Computer Security Foundations Symposium (CSF'09)*, 2009.

[37] Kevin Zhijie Chen, Guofei Gu, Jianwei Zhuge, Jose Nazario, and Xinhui Han. Webpatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, 2011.

[38] Song Chengyu, Paul Royal, and Wenke Lee. Impeding automated malware analysis with environment-sensitive malware. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Security (HotSec'12)*, 2012.

[39] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[40] Steve Christey. Unforgivable vulnerabilities. *Black Hat Briefings*, 2007.

[41] Vittorio Cortellessa, Ivica Crnkovic, Fabrizio Marinelli, and Pasqualina Potena. Experimenting the automated selection of cots components based on cost and system requirements. *Journal of Universal Computer Science*, 14(8):1228–1255, 2008.

[42] Vittorio Cortellessa, Fabrizio Marinelli, and Pasqualina Potena. Automated selection of software components based on cost/reliability tradeoff. In *Proceedings of the Third European Workshop on Software Architecture (EWSA'06)*, 2006.

[43] Kevin Crowston, James Howison, and Hala Annabi. Information systems success in free and open source software development: Theory and measures. *Software Process: Improvement and Practice*, 11(2):123–148, 2006.

[44] Mark Curphey and Rudolph Arawo. Web application security assessment tools. *IEEE Security & Privacy*, 4(4):32–41, 2006.

[45] Linus Dahlander and Mats G. Magnusson. Relationships between open source software companies and communities: Observations from nordic firms. *Research Policy*, 34(4):481–493, 2005.

[46] Stanislav Dashevskyi, Achim D. Brucker, and Fabio Massacci. On the security maintenance cost of open source components. *Submitted to ACM Transactions on Internet Technology.*

[47] Stanislav Dashevskyi, Achim D. Brucker, and Fabio Massacci. A screening test for disclosed vulnerabilities in FOSS components. *To be submitted to ACM Transactions on Software engineering.*

[48] Stanislav Dashevskyi, Achim D. Brucker, and Fabio Massacci. On the security cost of using a free and open source component in a proprietary product. In *Proceedings of the 2016 Engineering Secure Software and Systems Conference (ESSoS'16)*, 2016.

[49] Stanislav Dashevskyi, Daniel Ricardo Dos Santos, Fabio Massacci, and Antonino Sabetta. TestREx: a testbed for repeatable exploits. In *Proceedings of 6th USENIX Workshop on Cyber Security Experimentation and Test (CSET'14)*, 2014.

[50] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. Software bertillonage. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[51] Vieri Del Bianco, Luigi Lavazza, Sandro Morasca, and Davide Taibi. Quality of open source software: The QualiPSo trustworthiness model. In *Proceedings of IFIP International Conference on Open Source Systems (OSS'09)*, 2009.

[52] Massimiliano Di Penta, Luigi Cerulo, and Lerina Aversano. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology*, 51(10):1469–1484, 2009.

[53] Tudor Dumitras, Priya Narasimhan, and Eli Tilevich. To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. *ACM SIGPLAN Notices*, 45(10):865–876, 2010.

[54] Eric Eide. Toward replayable research in networking and systems. *Position paper presented at Archive*, 2010.

[55] Glenn Ellison and Drew Fudenberg. The neo-luddite's lament: Excessive upgrades in the software industry. *The RAND Journal of Economics*, 31(2):253–272, 2000.

[56] Nathan S. Evans, Azzedine Benameur, and Matthew Elder. Large-scale evaluation of a vulnerability analysis framework. In *Proceedings of 6th USENIX Workshop on Cyber Security Experimentation and Test (CSET'14)*, 2014.

[57] Brian Fitzgerald. The transformation of open source software. *MIS Quarterly-Management Information Systems*, 30(3):587–598, 2006.

[58] Brian Fitzgerald. Open source software: Lessons from and for software engineering. *IEEE Computer*, 44(10):25–30, 2011.

[59] Richard K. Fjeldstad and William T. Hamlen. Application program maintenance study: Report to our respondents. *Tutorial on Software Maintenance, IEEE Computer Society Press*, pages 13–30, 1982.

[60] J. Fonseca, M. Vieira, and H. Madeira. Evaluation of web security mechanisms using vulnerability & attack injection. *IEEE Transactions on Dependable and Secure Computing*, 11(5):440–453, 2014.

[61] Jose Fonseca and Marco Vieira. Mapping software faults with web security vulnerabilities. In *Proceedings of the 3rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, 2008.

[62] Forrester Consulting. Software integrity risk report. the critical link between business risk and development risk. Technical report, 2011.

[63] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. Prioritizing software security fortification throughcode-level metrics. In *Proceedings of the 4th Workshop on Quality of Protection (QoP'08)*, 2008.

[64] Barney G. Glaser and Anselm L. Strauss. Grounded theory. *Strategien qualitativer Forschung. Bern: Huber*, 1998.

[65] Ron Goldman and Richard P. Gabriel. *Innovation happens elsewhere: Open source as business strategy.* Morgan Kaufmann, 2005.

[66] Jurgen Graf. Speeding up context-, object- and field-sensitive SDG generation. In *Proceedings of 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM'10).*

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

[68] David A. Grimes and Kenneth F. Schulz. Uses and abuses of screening tests. *The Lancet*, 359(9309):881 – 884, 2002.

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

[70] Mohanad Halaweh. Using grounded theory as a method for system requirements analysis. *Journal of Information Systems and Technology Management*, 9(1):23–38, 2012.

[71] Tracy Hall, Sarah Beecham, and David Bowes. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[72] Marit Hansen, Kristian Köhntopp, and Andreas Pfitzmann. The open source approach opportunities and limitations with respect to security and privacy. *Computers & Security*, 21(5):461–471, 2002.

[73] Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, 2016.

[74] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012.

[75] Jaap-Henk Hoepman and Bart Jacobs. Increased security through open source. *Communications of the ACM*, 50(1):79–83, 2007.

[76] Robert R. Hoffman, Nigel R. Shadbolt, Mike A. Burton, and Gary Klein. Eliciting knowledge from experts: A methodological analysis. *Organizational behavior and human decision processes*, 62(2):129–158, 1995.

[77] Martin Höst and Alma Oručević-Alagić. A systematic review of research on open source software in commercial software product development. *Information and Software Technology*, 53(6):616–624, 2011.

[78] Michael Howard and Steve Lipner. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.

[79] Christos Ioannidis, David Pym, and Julian Williams. Information security trade-offs and optimal patching policies. *European Journal of Operational Research*, 216(2):434 – 444, 2012.

[80] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 6. Springer, 2013.

[81] Xuxian Jiang, Dongyan Xu, Helen J. Wang, and Eugene H. Spafford. Virtual playgrounds for worm behavior investigation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, 2006.

[82] Russell L. Jones and Abhinav Rastogi. Secure coding: building security into the software development life cycle. *Information Systems Security*, 13(5):29–39, 2004.

[83] Stefan Kals, Engin Kirda, Christopher Krügel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web (WWW'06)*, 2006.

[84] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Aloka Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[85] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, 2011.

[86] Sunghun Kim, Thomas Zimmermann, Kai Pan, and James E. Whitehead Jr. Automatic identification of bug-introducing changes. In *Proceedings of 21st International Conference on Automated Software Engineering (ASE'06)*, 2006.

[87] A. Gunes Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2009.

[88] Chunkit Kwong, Li-Feng Mu, Jiafu Tang, and Xinggang Luo. Optimization of software components selection for component-based software system development. *Computers & Industrial Engineering*, 58(4):618–624, 2010.

[89] Josh Lerner and Jean Tirole. The open source movement: Key research questions. *European Economic Review*, 45(4):819–826, 2001.

[90] Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N. Slyngstad, and Maurizio Morisio. Development with off-the-shelf components: 10 facts. *IEEE Software Journal*, 26(2):80, 2009.

[91] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability (ASID'06)*, 2006.

[92] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[93] Gen Lu. *Analysis of Evasion Techniques in Web-based Malware*. PhD thesis, Tucson, AZ, USA, 2014.

[94] Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172 – 1186, 2006.

[95] Fabio Massacci and Viet Hung Nguyen. Which is the right source for vulnerability studies?: an empirical analysis on Mozilla Firefox. In *Proceedings of the International ACM Workshop on Security Measurement and Metrics (METRISEC'10)*, 2010.

[96] Fabio Massacci and Viet Hung Nguyen. An empirical methodology to evaluate vulnerability discovery models. *IEEE Transactions on Software Engineering*, 40(12):1147–1162, 2014.

[97] Roy A. Maxion and Kevin S. Killourhy. Should security researchers experiment more and draw more inferences? In *Proceedings of 4th USENIX Workshop on Cyber Security Experimentation and Test (CSET'11)*, 2011.

[98] Philip Mayer and Andreas Schroeder. Cross-language code analysis and refactoring. In *Proceedings of 12th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, 2012.

[99] Gary McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[100] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):1–44, 2015.

[101] Andrew Meneely, Harshavardhan Srinivasan, Afiqah Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *Proceedings of the 7th International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*, 2013.

[102] Janne Merilinna and Mari Matinlassi. State of the art and practice of OpenSource component integration. In *Proceedings of 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'06)*, 2006.

[103] Mark Merkow. Risk analysis and management for the software supply chain. Whitepaper. Accessed on 31.10.2016., 2013.

[104] Vishal Midha and Prashant Palvia. Factors affecting the success of open source software. *Journal of Systems and Software*, 85(4):895–905, 2012.

[105] Charlie Miller. The legitimate vulnerability market: Inside the secretive world of 0-day exploit sales. In *Proceedings of 6th Annual Workshop on the Economics of Information Security (WEIS'07)*, 2007.

[106] Rupert G. Miller Jr. *Survival analysis*, volume 66. John Wiley & Sons, 2011.

[107] Nivedita Mukherji, Balaji Rajagopalan, and Mohan Tanniru. A decision support model for optimal timing of investments in information technology upgrades. *Decision Support Systems Journal*, 42(3):1684–1696, 2006.

[108] Raymond H. Myers. *Classical and modern regression with applications.* Duxbury Press, Pacific Grove, 2000.

[109] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, 2005.

[110] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: a study of the impact of shared code on vulnerability patching. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[111] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

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

[113] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the International ACM Workshop on Security Measurement and Metrics (METRISEC'10)*, 2010.

[114] Gary Nilson, Kent Wills, Jeffrey Stuckman, and James Purtilo. Bugbox: A vulnerability corpus for PHP web applications. In *Proceedings of 6th USENIX Workshop on Cyber Security Experimentation and Test (CSET'13)*, 2013.

[115] Node Security Project. Js-yaml deserialization code execution. `https://nodesecurity.io/advisories/JS-YAML_Deserialization_Code_Execution`, 2013.

[116] Node Security Project. st directory traversal. https://nodesecurity.io/advisories/st_directory_traversal, 2014.

[117] Linus Nyman and Tommi Mikkonen. To fork or not to fork: Fork motivations in sourceforge projects. In *Proceedings of IFIP International Conference on Open Source Systems (OSS'11)*, 2011.

[118] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[119] OWASP. Webgoat. https://www.owasp.org/index.php/Category: OWASP_WebGoat_Project.

[120] Andy Ozment. Software security growth modeling: Examining vulnerabilities with reliability growth models. In *Quality of Protection: Security Measurements and Metrics*, pages 25–36. Springer US, 2006.

[121] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[122] Jannik Pewny and Thorsten Holz. Evilcoder: Automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*, 2016.

[123] Henrik Plate, Serena E. Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, 2015.

[124] Gregor Polančič, Romana Vajde Horvat, and Tomislav Rozman. Comparative assessment of open source software using easy accessible data. In *Proceedings of 26th International Conference on Information Technology Interfaces (ITI'04)*, 2004.

[125] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[126] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Proceedings of 10th International Conference on Information Security (ISC'07)*, 2007.

[127] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *International Journal on Software Tools for Technology Transfer*, 9(5-6):489–504, 2007.

[128] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[129] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of 18th International Conference on Automated Software Engineering (ASE'03)*, 2003.

[130] Eric Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005.

[131] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the 20th ACM SIGSOFT symposium on Foundations of software engineering (FSE'12)*, 2012.

[132] Gregorio Robles and Jesús M. González-Barahona. A comprehensive study of software forks: Dates, reasons and outcomes. In *Proceedings of IFIP International Conference on Open Source Systems (OSS'11)*, 2012.

[133] Maria A. Rossi. Decoding the free/open source software puzzle: A survey of theoretical and empirical contributions. In *The Economics of Open Source Software Development*, pages 15–55. Elsevier, 2006.

[134] Michel Ruffin and Christof Ebert. Using open source software in product development: A primer. *IEEE Software*, 21(1):82–86, 2004.

[135] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, 2008.

[136] Antonino Sabetta, Luca Compagna, Serena Ponta, Stanislav Dashevskyi, Daniel Ricardo Dos Santos, and Fabio Massacci. Multi-context exploit test management, 2015. US Patent App. 14/692,203.

[137] Izzet Sahin and Fatemeh Mariam Zahedi. Policy analysis for warranty, maintenance, and upgrade of software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(6):469–493, 2001.

[138] Hitesh Sajnani, Vaibhav Saini, Joel Ossher, and Cristina Videira Lopes. Is popularity a measure of quality? an analysis of Maven components. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, 2014.

[139] Ioannis Samoladas, Georgios Gousios, Diomidis Spinellis, and Ioannis Stamelos. The SQO-OSS quality model: measurement based open source software evaluation. In

Proceedings of IFIP International Conference on Open Source Systems (OSS'08), 2008.

[140] Mohamed Sarrab and Osama M. Hussain Rehman. Empirical study of open source software selection for adoption, based on software quality characteristics. *Advances in Engineering Software*, 69:1–11, 2014.

[141] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[142] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security (FC'12)*, 2012.

[143] Guido Schryen. Is open source security a myth? *Communications of the ACM*, 54(5):130–140, 2011.

[144] Robert C. Seacord. Secure coding standards. In *Proceedings of the Static Analysis Summit, NIST Special Publication*, 2006.

[145] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

[146] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM'08)*, 2008.

[147] Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 2012.

[148] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[149] Black Duck Software. The tenth annual future of open source survey. `https://www.blackducksoftware.com/2016-future-of-open-source`, 2016. Last accessed 2017-04-11.

[150] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. *ACM SIGPLAN Notices*, 42(6):112–122, 2007.

[151] Stefan Stanciulescu, Sandro Schulze, and Andrzej Wasowski. Forked and integrated variants in an open-source firmware project. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, 2015.

[152] James P. Stevens. *Applied multivariate statistics for the social sciences.* Routledge, 2012.

[153] Klaas-Jan Stol and Muhammad Ali Babar. Challenges in using open source software in product development: a review of the literature. In *Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS'10)*, 2010.

[154] Evan Lawrence Stoner. A foundation for cyber experimentation. Master's thesis, 2015.

[155] Anselm Strauss and Juliet Corbin. *Basics of qualitative research*, volume 15. Newbury Park, CA: Sage, 1990.

[156] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws.* John Wiley & Sons, Inc., New York, NY, USA, 2007.

[157] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[158] Julian Thome, Lwin Khin Shar, and Lionel Briand. Security slicing for auditing xml, xpath, and sql injection vulnerabilities. In *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE'15)*, 2015.

[159] Amrit Tiwana. Does interfirm modularity complement ignorance? a field study of software outsourcing alliances. *Strategic Management Journal*, 29(11):1241–1252, 2008.

[160] Arian Treffer and Matthias Uflacker. Dynamic slicing with Soot. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP'14)*, 2014.

[161] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014.

[162] Kris Ven and Herwig Mannaert. Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology*, 50(9):991–1002, 2008.

[163] James Walden and Maureen Doyle. SAVI: Static-analysis vulnerability indicator. *IEEE Security & Privacy*, 10(3):32–39, 2012.

[164] James Walden, Jeffrey Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE'14)*, 2014.

[165] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*, 1981.

[166] David A Wheeler. How to evaluate open source software/free software (OSS/FS) programs. Whitepaper. Accessed on 28.10.2016, 2011.

[167] David A Wheeler and Samir Khakimov. Open source software projects needing security investments. Whitepaper. Accessed on 28.10.2016, 2015.

[168] Carol Woody, Robert Ellison, and William Nichols. Predicting software assurance using quality and reliability measures. Technical report, CMU/SEI-2014-TN-026, 2014.

[169] Robert K. Yin. *Case study research: Design and methods.* Sage, 2013.

[170] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications.* No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[171] Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E Hassan. How does context affect the distribution of software maintainability metrics? In *Proceedings of International Conference on Software Maintenance (ICSM'13)*, 2013.

[172] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Proceedings of International Conference on Software Maintenance (ICSM'09)*, 2009.

[173] Su Zhang, Xinwen Zhang, Xinming Ou, Liqun Chen, Nigel Edwards, and Jing Jin. Assessing attack surface with component-based package dependency. In *Proceedings of 9th International Conference on Network and System Security (NSS'15)*, 2015.

[174] Luyin Zhao and Sebastian Elbaum. Quality assurance under the open source development model. *Journal of Systems and Software*, 66(1):65–75, 2003.

[175] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'10)*, 2010.

[176] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the 3th International Workshop on Predictor models in Software Engineering (PROMISE'07)*, 2007.