On the Effort for Security Maintenance of Free and Open Source Components [*]

STANISLAV DASHEVSKYI, University of Luxembourg, Luxembourg
ACHIM D. BRUCKER, University of Sheffield, United Kingdom
FABIO MASSACCI, University of Trento, Italy

The work presented in this paper is motivated by the need to estimate the security effort of maintaining Free and Open Source Software (FOSS) components within the software supply chain of a large international software vendor. We investigated publicly available factors (from number of active users to commits, from code size to usage of popular programming languages, etc.) to identify which ones impact three potential effort models: Centralized (the company checks each component and propagates changes to the product groups), Distributed (each product group is in charge of evaluating and fixing its consumed FOSS components), and Hybrid (seldom used components are checked individually by each development team, the rest is centralized). We use Grounded Theory to extract the factors from a six months study at the vendor. We report the results on a sample of 152 FOSS components used by the vendor.

Additional Key Words and Phrases: Free and Open Source software, software vulnerabilities, security maintenance

1 INTRODUCTION

According to a recent Black Duck study [18], more than 65% of proprietary applications leverage Free and Open Source Software (FOSS). This choice speeds up application development and flexibility [43] as a FOSS component is often used "as-is" [52]. The price to pay is that vulnerabilities discovered in a FOSS component may affect the application that consumes it.

As the security of a software offering depends on all components, FOSS should be subject to the same security scrutiny of one's own code[1] and the relative security of selecting a FOSS component over a proprietary one should be investigated [39, 40, 79] in order to have a fully secure supply chain [60].

Yet, from the perspective of a software vendor who is consuming FOSS components, the question on selecting the most secure (FOSS, outsourced, or self-developed) components, albeit an important one, may not not always be the most pressing one. First, FOSS components may be the de-facto standard (e. g., Hadoop for big data) and the end customers of the vendor expect them to be used. Second, FOSS may offer functionalities that are very expensive to re-implement, so it is the most economical choice [52]. Third, there might only be one FOSS component with the desired functionality to chose from. Finally, some FOSS component might just have a license that is

[*]Paper accepted for presentation in the *Workshop on the Economics of Information Security (WEIS'18).* A much shorter, prelimnary version was also presented at ESSOS'16 [28]. This work has been partially financed by the European Union under the FP7-PEOPLE-2012-ITN grant 317387 (SECENTIS), and the EIT Digital UNBIAS project Task ID 18213.
[1]For example, SAP, a large software manufacturer, runs static code analysis tools to verify the combined code base of its application and the FOSS component, and audits the results [19].

Authors' addresses: **Stanislav Dashevskyi**, University of Luxembourg, Luxembourg, stanislav.dashevskyi@uni.lu; **Achim D. Brucker**, University of Sheffield, United Kingdom, a.brucker@sheffield.ac.uk; **Fabio Massacci**, University of Trento, Italy, fabio.massacci@unitn.it.

incompatible with the chosen business model (e.g., it might require companies to release their source code).[2]

Another and often more interesting question is to understand which factors are likely to impact the "security maintenance effort" of the selected FOSS component. Indeed, when a new security issue in a FOSS component becomes publicly known, a software vendor has to verify whether that issue affects customers who consume software solutions where that particular FOSS component was shipped by the vendor. In ERP systems and industrial control systems this event may occur years after deployment of the selected FOSS product (see Figure 2 in Section 3).

It is therefore important to understand which characteristics of FOSS components (number of contributors, popularity, lines of code, or choice of a programming language, etc.) are likely inducing security issues in these components. The impact of these characteristics may be significant for the consumers of FOSS components, as these components may be used by hundreds of products.

Motivated by the need to estimate the efforts and risks of consuming FOSS in proprietary software products of a large international software vendor, we formulate our research question as follows:

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

In this paper we propose and empirically test the factors that can impact the vulnerability resolution process 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 the maintenance effort scales 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 team

To validate this question, we first start with a grounded theory building process from a case study at a multinational corporation, that identifies the key problem drivers. Then, by looking at previous research, we consolidate the theory and its qualitative findings in order to identify and sharpen the general theoretical issues that are broadly applicable to the field, as opposed to those that may be idiosyncratic to the specific firm.

2 FOSS AND THIRD-PARTY COMPONENTS

To a consistent terminology, we provide an overview of the different kinds of software components integrated by proprietary software vendors, as well as describe the place of FOSS among them. The most important components types are as follows:

- *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 [88], the consuming party can specify the required compliance, security guidelines as well as the support and maintenance model. Depending on the contract, such components can be shipped in binary or in 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.

[2]A notable example is the German Court decision on GPL infringement by Sitecom Europe in 2004 [42].

- *FOSS components:* grant free access to the source code, as well as the freedom to distribute modified versions, provided that certain restrictions with respect to their licenses are respected [75]. There are many open source licenses that describe different legal aspects in different ways, including the detailed conditions under which FOSS can be distributed.

FOSS components share aspects with outsourced development, subcontracting as well as with proprietary standard software components. For example, FOSS components can be modified, adapted, and maintained by the customer (they have this in common with outsourcing and subcontracting). As proprietary standard software components, they usually provide a fixed set of interfaces and functionality that consuming products need to be adapted to (instead of having a custom made component that "just fits").

2.1 What Makes FOSS Special?

Stol and Babar described the challenges of integrating FOSS components into proprietary software, according to the past literature [84]. They identify maintenance among the most important challenges, which suggests 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. Thus, one could expect that there is no need to handle them differently from other types of third-party components. In practice, there are at least five aspects that are often considered to be special[3]:

(1) The absence of initial costs for FOSS components might misguide developers to use them without properly assessing their licenses [75]. In large companies, the legal check of the software license and the warranty is part of the purchasing process. As FOSS is often just downloaded from the Internet, it is more difficult for organizations to enforce these checks.
(2) In any case, FOSS components have to be integrated into the target application [89], which increases overall costs due to additional development and maintenance activities [2].
(3) Most FOSS licenses contain rather strong "no warranty" clauses [75]. This often comes 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 [30], or by investing into in-house maintenance of the component. The lack of documentation in FOSS projects [11, 84] prevent in-house developers from learning, thus increasing potential maintenance efforts and costs.
(4) Proprietary software vendors are often unable to influence development processes and patch release policies of consumed FOSS components [84]. Given that vulnerability patching is prioritized among other maintenance tasks [9], it is nearly impossible to establish the equilibrium between patch releases in FOSS projects and updates of versions used by proprietary vendors (see [22] for a discussion).
(5) The security response processes for proprietary software often aim to release detailed information about a vulnerability only after a patch for that vulnerability was released. The goal is to provide customers a safety period during which they can patch their systems before a vulnerability gets publicly known, as well as *not* to publish fixes that can be transformed into zero day vulnerabilities for the previous versions of the product. This might conflict, one the 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, as

[3]Some of these aspects are based on empirical studies that are already five to 10 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 the risk that not all of the findings are still true. For example, most larger FOSS projects nowadays support the confidential reporting of security issues as well as a responsible patch and disclosure process. Thus, for these projects the aspect (4) is less of an issue.

publishing a security patch in the source code form prior a public disclosure of a vulnerability can be considered as a public disclosure [73].

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

(1) Besides the FOSS vendors, there are other companies that can offer commercial support and bug fixes for a component. Thus, one has the choice between different maintenance offerings for the same component.
(2) As the source code is available and modifications, under certain conditions, are allowed, one can fix issues independently from the FOSS vendors.

Unfortunately, these opportunities also lead to certain risks (as indicated in points (3) and (4) above). From now on we call companies that integrate FOSS components into their products as *FOSS adopters* (or simply, *adopters*).

3 THEORY BUILDING FROM A CASE STUDY AT A LARGE INTERNATIONAL SOFTWARE VENDOR

To identify the right factors to consider when evaluating the impact of FOSS selection choices on the security maintenance effort we conducted an exploratory case study at a large international software vendor. In our study we adopted several elements of the *Grounded Theory* approach initially proposed by Glaser and Strauss [33]. The goal of this approach is to construct a theory based on a phenomenon that can be explained with data [36]. The approach follows the principle of emergence [35]: 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 [85].

We followed Yin [94] as a guidance on conducting case studies for performing our study. Our main goal was to explore 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 of 15 months was spent by one of the authors at the premises of our industrial partner, including 12 months at the Research Lab, and 3 months with the central *Product Security Team* at headquarters. During the visit at headquarters, the authors worked closely with a senior member of the central Product Security Team, who had been already working in that position for 8 years.

Various techniques exist for knowledge elicitation [41], and structured and semi-structured interviews are considered to be among the most important sources of information [94]. We used purposive sampling [37] while performing informal discussions with developers, members of the Security Testing Team, security champions of the Maintenance Teams for two different product areas, and experts in the Product Security Team being responsible for the definition and implementation of the Security Development Process (i. e., the owner of the Secure Software Development Process), and both the in-bound (adopting FOSS) and out-bound (releasing software as FOSS) Open-Source processes (i. e., the owner of these processes). We had two dedicated meetings with the owner of the Open-Source processes and one meeting with the owner of the Secure Software Development Process. Moreover, we organized five meetings with two different development and maintenance experts. Additionally, we organized both an internal development meeting, at which we did more than 50 short interviews of three to five minutes. However, we could not hold formally recorded interviews, as they would require an extremely heavy and lengthy authorization process through the legal department.

Finally, we presented our findings during the weekly team meetings of the Security Testing Team. The senior member of the Product Security Team, being the "key informant" in Yin's terminology [94], 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, security experts, 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.

3.1 FOSS Adoption as Part of the Secure Software Development Lifecycle

Our industrial partner followed a security development lifecycle (*SDL*) as a part of its secure software development process. The *SDL* of our industrial partner is very similar to the well-known *SDL* of Microsoft [44].

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 analysis) 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 [19] 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 [12] for example), which motivated the work carried out in this paper. 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.

To understand the role of FOSS components in the development process of our industrial partner, we collected data for 152 most popular FOSS projects that were requested by developers of internal projects as components during the last five years[4]. 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[5] or Cloud Foundry[6].

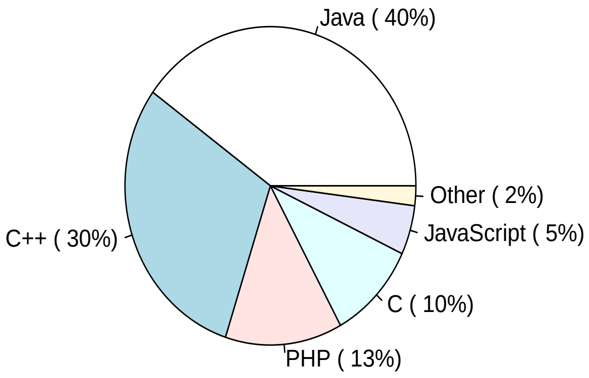
As we can see from Figure 1, FOSS components are integrated (or are requested for integration) into a large number of projects of our industrial partner. Figure 2a illustrates the cumulative size of the code bases of the components in the sample broken down by different programming languages in which they were implemented: the distribution suggests that the largest code base corresponds to Java. Figure 2b shows the distributions of the number of internal projects into which a FOSS component from the sample was integrated (or is requested for integration), 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 applied non-parametric

[4]This information is publicly available and can be reconstructed from the bill of materials of individual projects found on the web community of the large international vendor (although, it was significantly easier to collect this information using the internal sources).
[5]https://www.openstack.org/
[6]https://www.cloudfoundry.org/

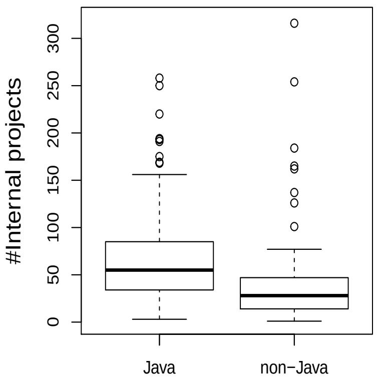
Wilcoxon test, since the data that we collected is not normally distributed (Shapiro-Wilk test returned $p < 0.05$), and it contains unpaired samples. The results of Wilcoxon test confirmed that Java-based components are indeed preferred more by the developers of our industrial partner in comparison to the others: the two distributions have small-to-medium and statistically significant difference ($p < 0.05$, Cohen's $d = 0.44$).

Fig. 1. Descriptive statistics of FOSS components used by internal projects

The two figures characterize the sample of the most popular 152 FOSS projects integrated (or requested for integration) into 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 integrations/requests of Java FOSS components.



(a) The distribution of the source code in used components by programming languages



(b) The number of internal projects using a component

Table 1. Popular Java projects used by our industrial partner

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

3.2 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 [34, 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 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 various license and 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.

While, with respect to the *inbound* FOSS approval process, security checks are an important ingredient, they are not the main decision point. First, certain FOSS components may be the de-facto standard (e. g., Apache Hadoop[7] 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 [52], 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 FOSS Maintenance And Response

After we identified how the choice of FOSS components is carried out, our next 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 enterprise resource planing (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 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 the software was released: most customers are using systems that are between eleven and nine years old. To meet the demands of the customers, the manufacturer offers support and

[7]https://hadoop.apache.org/

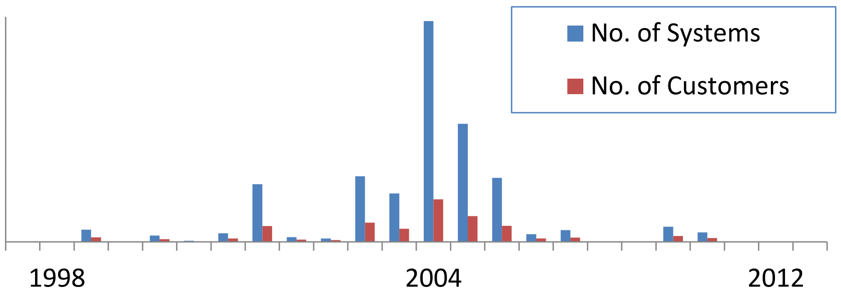


Fig. 2. Customer distribution of an enterprise software system (as of 2014)
The distribution (as of 2014) shows #systems (blue) and #customers (red) using different versions of an ERP application: values on the x-axis show the year when a particular version was released, and values on the y-axis are the numbers of releases and customers of a particular version (the actual 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.

maintenance, for selected products, mainstream support for a number of years and, additional, customer-specific extended maintenance for several additional years. Thus, all third-party products need to be supported at least with security fixed 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[8] or POODLE[9]), 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 is the opposite: cloud offerings usually have rapid release-cycle and, thus, do not require a long maintenance phase. Here, it is more important that the 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. [29] 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 Black Duck Code Center.[10] 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 3.1.

The number of FOSS components per product depends on the actual product. For example, while traditional ERP systems written in proprietary languages (e. g. ABAP or PeopleCode) usually do

[8]http://heartbleed.com/
[9]https://www.oracle.com/technetwork/topics/security/poodlecve-2014-3566-2339408.html
[10]http://www.blackducksoftware.com

Table 2. Historical vulnerabilities of 152 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%
Cross-Site Scripting	5.9%	SQL injection	0.1%

not contain a lot of FOSS components, the situation is quite the opposite for recent cloud offerings based on OpenStack or Cloud Foundry (moreover, both OpenStack and Cloud Foundry themselves consume already several hundreds of FLOSS components).

Table 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 vulnerabilities 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 [24]. Given the numbers of internal projects that use and request FOSS components, the problem of their security maintenance becomes of great importance.

In summary, to minimize the effort associated with integrating FOSS components as well as to maximize the usability of the developed product, the development teams consider different factors. Below we list our findings on the factors that they consider from the security maintenance effort perspective:

F1 *The community around a component is active.* Components from active and well-known FOSS communities (e. g., Apache Software Foundation[11]) should attract more volunteer developers [1]. This in turn should enable the development teams of proprietary vendors that integrate FOSS components to leverage external expertise, as well as externally provided security fixes.

F2 *A component is widely used by many products.* The components that have been already used should require lower effort as licensing checks are already done, and the internal technical expertise can be tapped as well, so that the effect of the economies of scale may be achieved [13]. Thus, the effort for fixing issues or integrating new versions of a component can be shared across multiple development teams.

F3 *Technical aspects of a component are familiar to developers.* If various technicalities, such as programming languages and build systems used in certain FOSS components are already familiar to the development teams, a lower effort for integration and support for those components can be expected [13, 14, 23].

F4 *A component is "covered" by planned maintenance from its own developers or third-parties.* If the security maintenance 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 [21].

F5 *Proxies for estimating the maintenance effort.* Unfortunately, it is very difficult to measure the working hours of developers that represent the software maintenance effort directly [45, 95].

[11]https://www.apache.org/

> Thus, software managers, stakeholders, and developers may employ various proxies which have also been studied in the literature [6, 74, 87, 95] instead of the missing measures of direct maintenance effort. Further, a team of software developers is normally assigned to several tasks, security maintenance being only one of them, so it is hard to get analytical accounting for security maintenance to the level of individual vulnerabilities [67]. Further, 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.

Additional elements, such as the compatibility of the license, or requests from customers that need integration with their code base also play a role. We do not consider them in this paper, as our focus is security maintenance as opposed to software component selection or production. For further discussion see [88] on alliances for software production, or [75] on legal issues and licensing models.

4 THEORY CONSOLIDATION FROM RESEARCH

We now consolidate the theory by a critical overview of the relevant relevant research about selection, evaluation, and consumption of third-party FOSS components by FOSS adopters.

4.1 Selection and Evaluation of FOSS

The first consistent observation is that FOSS components are often selected based on the previous experience of developers, even without considering other alternatives, as emerged from interviews within several FOSS adopters [11]. Whilst some aspects of this experience are clearly idiosyncratic to both firms and developers and may be hard to measure, the previous knowledge of the programming language used for the FOSS component is expected to be a driver as it simplify the developers' own integration effort [13, 14]. Hence the popularity of the programming language might be an indicator of the likelihood that the developer might have actually experienced it beforehand [76, 96].

While past experiences of developers may influence the selection of FOSS certain components, the final decision about the integration of a specific component is typically done after the component is evaluated against the firm-specific criteria [59]. This process may differ from case to case even within the same software company [84].

However, a common ground has been identified in the supplied functionality which played the major role among a set of factors that include various licensing aspects, the speed of evolution, the community characteristics (including the availability of support), as well as the quality of documentation [3, 7, 10]. Such identification relied also on a survey based on software developers that reuse open source software components [11].

Several open source software selection models [77, 93] considered software security aspects, however, only to the limited extent (for example, the model by Samoladas et al [77] considered only the presence of "null dereferences" and "undefined values" in the source code).

Unfortunately, there is no common agreement on which characteristics of FOSS projects should be considered by FOSS adopters when selecting a component (especially with respect to software security). This is possibly because the information about these factors may be very heterogeneous, limited, or even absent [11]: ben Othmane et al. [67] concluded that FOSS adopters' developers may be aware of various factors that may impact the security maintenance effort, however they often fail to identify them. Also, the majority of FOSS projects may provide insufficient documentation [59, 84], furthermore, there may be a lack of support from the FOSS developers [84]. This complicates the industrial FOSS adoption by interfering with the FOSS adopters' ability to learn how to use and fix a component. As developers often lack time for performing a thorough evaluation of all third-party components [84], this may cause additional security problems in the future.

4.2 The Economic Impact of Security Maintenance

The maintenance of software components from the security economics perspective is relatively unexplored. In contrast, the cost of general software maintenance is well investigated in the literature [14, 15, 21, 23].

Conceptually, there is an important distinction between parties that are using software components (FOSS adopters) and parties that provide software components and support them (providers). For instance, the security patch management model by Cavusoglu et al. [22] specifies the costs of providers due to developing and shipping patches, as well as vulnerabilities that are exploited before patches are released (reputation losses). The costs of FOSS adopters [22, 86] may emerge due to potential security damage (unavailability of a patch or inability to apply it) and updates induced by components (testing and installing patches). This implies different types of costs for different parties, however, for our scenario, a FOSS adopter would have to bear all these costs, as the adopter is providing to its customers the software that includes FOSS components.

More specifically, the costs of FOSS adopters are affected by the security maintenance effort that they must invest into FOSS components. There are many reasons why security patches provided by FOSS communities cannot be applied effortlessly by industrial FOSS adopters: for instance, due to large numbers of vulnerabilities being disclosed periodically[12], or the fact that third-party security patches should be additionally verified, or due to the reason that patches must be applied for all supported versions of a proprietary application that relies on a potentially unsupported version of a vulnerable FOSS component. Therefore, the security maintenance effort increases significantly when FOSS adopters must identify (or develop), test, distribute, and deploy security patches for FOSS components to their end customers [13, 15, 22].

Several studies [13, 14, 23, 31] observed that a significant part of maintenance costs is generated when the FOSS adopter developers must understand how the software from various third-party providers should be modified or patched. Therefore, an additional reason for high security maintenance costs and efforts may be the fact that proprietary FOSS adopters may integrate hundreds of FOSS components into their products.

Banker and Slaughter [15] investigated how software maintenance in organizations can be improved in order to achieve economical benefits – they find support for a hypothesis that software maintenance can be characterized by economies of scale. Therefore, to the larger is the number of FOSS components used by proprietary applications, and the larger is the number of vulnerabilities that "interfere" with regular maintenance activities, the more the effort due to security maintenance activities may affect the operational costs of FOSS adopters.

Therefore, to account for a possible effect of the economies of scale, we consider three different models that represent different ways of managing security maintenance of FOSS components (see Section 5.2). Specifically, we intend to consider situations in which FOSS adopters could have a centralized team of FOSS security experts that mitigates the negative effect of the lack of knowledge about the entire portfolio of FOSS components integrated into the vendors' products. In these two models, the "bulk" security issue resolution may be beneficial when the number of usages of a component is high.

4.3 Factors of FOSS projects and Security Maintenance Effort

In the rest of this Section we overview various characteristics of FOSS projects that have been discussed in the past literature as being directly or indirectly relevant to the security maintenance of software.

[12]The study by Rescorla [73] suggests that vulnerability discovery/fix rates for software projects do not decrease through their lifetimes.

4.3.1 FOSS Community. Several studies considered the popularity of FOSS projects as being relevant to their quality and maintenance [72, 76, 96]. It is a folk knowledge that "Given enough eyeballs, all bugs are shallow" [72], meaning that FOSS projects have the unique opportunity to be tested and scrutinized not only by their developers, but also by their user community. However, the user community of any FOSS component can be divided into active users (adopters) that learn, integrate, contribute to, or modify a component for their own needs; and passive users that only download a component and install it.

The number of adopters should positively impact the number of proprietary products into which a component will be integrated (as suggested by developer experience from [11]). Also, the number of adopters should positively impact the number of vulnerabilities as active users are more likely to modify and inspect the source code, finding and reporting them. The number of downloads (passive users) of a component can serve as another measure for popularity [27] as identified by the less active subset of users that only download and install a component and are less likely to inspect the source code. Therefore, while the impact of this factor on vulnerability reporting (and on the security maintenance) should still be positive, it should be smaller than the number of adopters.

Apart from external popularity measures, there are internal ones, such as the number of developers of a FOSS component. This measure may serve as an independent factor that impacts the number of bugs or vulnerabilities in a component [17]. However, for our scenario, the impact may be either negative (more experienced developers – less bugs and vulnerabilities), or negative (more inexperienced developers or occasional contributors – more bugs and vulnerabilities).

Understanding how software works is a necessary prerequisite for successful software maintenance and development [14, 15, 31] – this is also supported by our finding F3. Ostrand et al. [66] 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 [66] did not suggest that certain languages may be more prone to bugs or vulnerabilities, they stress the importance of considering various programming languages in connection to their number. Also, the increasing familiarity of users with particular programming languages or frameworks may increase the likelihood that malicious users will able to "break" a piece of software that relies on that particular programming language or framework [5].

Based on the discussion of the above literature, as well as our findings F1, F2, and F3, we formulate the following hypothesis:

H_1 *The factors that characterize various popularity aspects of a FOSS component have positive impact on the security maintenance effort of that component.*

4.3.2 Factors that Impact Vulnerabilities. As we have discussed above, various approaches and recommendations for selection of FOSS components (that may be mostly performed by the management) may not consider many of the factors that are directly relevant to software security (that are important for security experts). As software developers and engineers are focusing on the quality and security of individual software components, the major research efforts had been focused so far on predicting bugs and vulnerabilities in software (see [55]). Although our focus is specifically on security vulnerabilities that may stand aside from generic software bugs, Ozment [68] showed that methods for estimating trends in generic bugs used in software engineering literature can be also applied for security vulnerabilities.

Specifically, an extensive body of research explored the applicability of various metrics that can be used for estimating the number of bugs and security vulnerabilities in *future* releases of a software component. The simplest such metric is time since initial release (i.e., the age of a component), and the corresponding model is a Vulnerability Discovery Model. Massacci and Nguyen [55] provide a comprehensive survey and independent empirical validation of several vulnerability discovery models.

The age of a project, its size and the number of code changes are traditionally used in various studies that investigate defects and vulnerabilities in software [32, 50], software evolution [16, 20] and maintenance [96].

For example, the study by Koru et al. [50] demonstrated a positive relationship between the size of a code base of a project and its defect- proneness. Zhang [97] evaluated the LoC metric for the software defect prediction and concluded that larger modules tend to have more defects. Many other works (see [32, 49, 63, 81, 96, 99]) suggest a positive relation between the number and the frequency of changes in the source code (e.g., repository commits, added/deleted lines of code), and the number of software defects and vulnerabilities.

The works by Ostrand et al. [66] and Bell et al. [17] aimed on predicting files in new 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. [66] considered bug modification histories of files in previous releases, while the follow-up study by Bell et al. [17] 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 [17] find evidence that prediction capabilities of the previous model by Ostrand et al. [66] improves when adding the number of unique developers of a system as an additional factor.

Shin et al. [81] analyzed several developer activity metrics showing that poor developer collaboration can potentially lead to vulnerabilities, and that code complexity metrics alone are not sufficient for vulnerability prediction. Similarly to [63], the authors of [81] suggest that code churn metrics are better indicators for approximate locations of vulnerabilities than complexity.

Several other metrics have been used: static analysis defect densities [90], frequencies of occurrence of specific programming constructs [78, 91], etc. We illustrate some representative cases with Table 3.

While most of the above approaches and models can be efficiently used for reasoning about several projects, it is unclear whether they can be applied in a scenario where there are hundreds of FOSS components integrated into various products of a FOSS adopter. Therefore, we formulate the following hypothesis:

> H_2 *The factors that can be used as predictors for bugs and vulnerabilities in individual components can be also used for assessing the potential security maintenance impact within a large third-party component portfolio.*

4.3.3 Secure Development, Testing, Maintenance And Contribution. Wheeler [92] suggested that successful FOSS projects should use static analysis security testing (SAST) tools, which should at least reduce the amount of trivial vulnerabilities [25] (see [26] for the examples of such vulnerabilities). Penetration testing and dynamic analysis security testing (DAST) tools facilitate the early discovery of security vulnerabilities [8], while maintaining the security regression tests for past vulnerabilities, and tests for the security-critical functionality ensures that the same (or similar) security issues are not re- introduced, thus lowering the security maintenance effort.

Secure design specifications of software components help the adopters to build more secure products – the availability of documentation relevant to security aspects and considerations helps to eliminate the security defects at the early stage of product development [57]. Additionally, the practice of internal reviews when the source code commits are checked before the code is pushed into production improves the overall quality and the security of the product [58]. Finally, the presence of the secure coding standards as a taxonomy of common programming errors [47, 80] reduces the amount of future vulnerabilities and the efforts for security maintenance.

Table 3. Vulnerability and bug prediction approaches

We provide a brief overview of various approaches for bug and vulnerability prediction in the existing literature (we refer to [38] and [71] for a more complete discussion).

Paper	Predictors	Bug data	Predicted vars
[66]	Bug and change histories of files	Internal data on previous releases of a commercial system	Files with largest bug concentration
[63]	Relative Code churn	Internal defect dataset (Windows Server 2003)	Bug density
[82]	Complexity metrics	MFSA, NVD, Bugzilla	Vulnerable functions
[65]	Member and Component dependency graphs, Complexity metrics	MFSA, NVD	Vulnerable functions
[81]	Complexity metrics, Code churn, Developer activity	MFSA, Red Hat Linux package manager	Vulnerable files
[90]	Static analysis vulnerability density	NVD	Num. of Vulnerabilities
[17]	Developer activity metrics	Internal data on previous releases of a commercial system	Files with largest bug concentration
[55]	Known vulnerabilities	MFSA, NVD, Bugzilla, Microsoft Security Bulletin, Apple Knowledge Base, Chrome Issue Tracker	Num. of Vulnerabilities
[78]	Frequencies of prog. constructs	SAST warnings (Fortify SCA)	Vulnerable files
[91]	Complexity metrics, Frequencies of prog. constructs	NVD, Security notes from a project	Vulnerable files

There are several security related-factors that may not impact the security maintenance effort significantly, but have a direct effect on the reputation a FOSS component, making it more or less appealing for selection by adopters. For instance, a project that provides means for downloading its source and binary packages securely (e.g., via https, cryptographically signed or hashed) protects its users from the malware that malicious third parties could inject into downloads. Additionally, if the private vulnerability reporting process is used by FOSS developers, it allows FOSS adopters to resolve security issues in components in a timely manner and notify their customers before the vulnerability becomes publicly known.

Our finding F4 suggests that the presence of the planned maintenance and support lifecycles in FOSS projects can be also an important factor that influences the adopters' choice of components: this factor enables FOSS adopters to plan ahead when they should make a decision on whether to replace a component when its maintenance becomes too complicated, or to fork a component and provide support internally when there is no adequate replacement available. The ease of contribution to a FOSS project (such as clear guidelines for new developers and transparent contribution processes) may help to reduce the maintenance effort when a project needs to be forked by adopters. The same is also true for FOSS projects which provide a clear development and distribution models: if these models do not match the models and processes typically used by FOSS adopters, this may disturb their software maintenance processes, significantly increasing the associated efforts and costs.

Based on the above discussion, as well as our finding F4, we formulate the following hypothesis:

H_3 *The factors that characterize various aspects of security and development lifecycle within FOSS components have negative impact on the security maintenance effort of that component.*

5 A MODEL OF BUSINESS AND TECHNICAL DRIVERS FOR FOSS SECURITY MAINTENANCE

The case study described earlier (Section 3), as well as the analysis of existing literature (Section 4), helped us to delineate the key features that a theoretical model for capturing the effort of software maintenance should have to extend to security.

5.1 A Conceptual Model of Business And Technical Drivers

We identified four main areas of factors that may have direct or indirect impact on the security maintenance effort:

(1) *FOSS community:* includes both quantitative and qualitative factors that reflect characteristics of a FOSS project community, being a function of project's general popularity and appeal to contributors. This area also affects 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., 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.

(2) *Secure development and testing:* factors that characterize how well secure development and testing activities are built into the lifecycle of a FOSS project, influencing all potential challenges of consumers with respect to software security, including vulnerabilities.

(3) *Maintenance and contribution models:* 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.

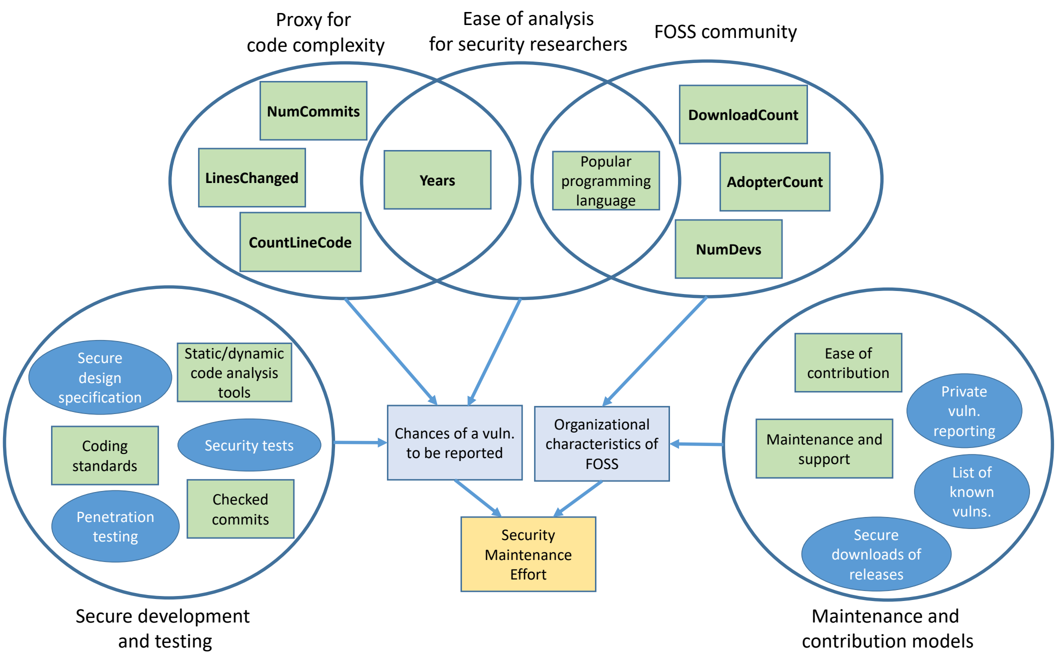
(4) *Proxy for code complexity:* this area of factors comprises of various quantitative characteristics of FOSS projects that represent their overall complexity. This complexity may have an impact on the number of disclosed vulnerabilities, thus affecting the maintenance effort of resolving them.

Tables 4, 6 and 5 summarize each factor for every area, as well as specify the data collection method that can be used to extract the information on these factors (see Section 6 for the description of data collection process). Figure 3 shows the relationships between these factors and the security maintenance effort.

Table 4. FOSS community drivers

Factor	Source	Collection method	Description	References
Popular programming language	Project website, Open Hub, code repository	Automatic	Project is mostly written in Java, C, C++, PHP, JavaScript, SQL, etc.	[56, 67]
NumDevs	Project website, Open Hub, code repository	Automatic	The number of developers.	[63, 81, 96]
AdopterCount	Project website, Open Hub	Automatic	The number of adopters (active users) of a project taken from Open Hub.	[1, 70, 72, 76, 93, 96]
DownloadCount	Project website, CII Census	Semi-automatic	The number of downloads of project releases or packages.	[27]

We left out a key metric that has been used in several papers [81, 99] – the number of different developers that make a number of changes to different areas of the source code of a single project. While this variable is a good predictor for vulnerabilities in individual units of which a single


Proxy for code complexity
Ease of analysis for security researchers
FOSS community
NumCommits
LinesChanged
CountLineCode
Years
Popular programming language
DownloadCount
AdopterCount
NumDevs
Secure design specification
Static/dynamic code analysis tools
Coding standards
Security tests
Penetration testing
Checked commits
Chances of a vuln. to be reported
Organizational characteristics of FOSS
Security Maintenance Effort
Ease of contribution
Private vuln. reporting
Maintenance and support
List of known vulns.
Secure downloads of releases
Secure development and testing
Maintenance and contribution models

Both security-related features (blue ovals), and business-related/functional features (green rectangles) can potentially impact the security maintenance effort, as it may be the case that the main driver for a choice of a FOSS component is not a security feature.

Fig. 3. Theoretical model for the impact of various factors on security maintenance

project or component is built (e.g., files or library modules), it cannot be used in our study, as we consider a FOSS component to be a single "unit". Therefore, we consider the number of unique developers that modify different FOSS components. For the same reason we only use the cumulative number of changes to a FOSS component instead of the number of changes within its specific units.

Table 5. Proxy for code complexity drivers

Factor	Source	Collection method	Description	References
CountLineCode	Open Hub, code repository	Automatic	Total size of the code base (LoC)	[13, 16, 20, 32, 50, 67, 97]
LinesChanged	Open Hub, code repository	Automatic	The development activity of a project (added/deleted lines of code)	[32, 49, 63, 81, 95, 96, 99]
NumCommits	Open Hub, code repository	Automatic	Number of total commits	[63, 81, 96]
Years	Open Hub, code repository	Automatic	Age of a project in years.	[7, 49, 69, 96]

Table 7 shows the final set of independent variables that we collected, and used for the analysis: along with the description of each variable, we provide a rationale for including it into the models, as well as connect it to a corresponding hypothesis that we have formulated in Section 4.

Table 6. 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) or security functionality tests.	[1, 13, 26, 92, 93]
Private vuln. reporting	Project website	Manual	There is a possibility to report security issues privately.	[61]
SAST/DAST tools	Project website, code repository, Coverity website	Manual	A project is using code analysis tools during development.	[1, 25, 67, 92, 93]
Secure design specs	Project website, documentation	Manual	The secure design specification of the project is documented.	[57, 67]
Penetration testing	Project website, documentation	Manual	The penetration testing is performed regularly by the project developers.	[8]
Coding standards	Project website, documentation, code repository	Manual	Secure coding standards are documented.	[13, 47, 67, 80]
List of known vulns	Project website, vulnerability databases	Manual	Past security vulnerabilities of the project are documented and are publicly available.	[93]
Maintenance and support	Project website, documentation	Manual	The patch and release cycles are documented. The maintenance roadmap and support cycles for different versions of a project are documented.	[92, 98]
Ease of contribution	Project website, documentation, code repository	Manual	Clear guidelines for new developers or potential contributors are present.	[1]
Checked commits	Project website, documentation, core repository	Manual	There exists a review process for new contributions, including security code reviews.	[1, 58, 72, 93]

5.2 The Effort Variable For FOSS Maintenance

At first, we observe that a systematic review of software development cost estimation studies [48] have highlighted that "*main cost driver in software development projects is typically the effort and we, in line with the majority of other researchers in this field, use the terms cost and effort interchangeably*". This is further confirmed by recent empirical studies which used the dataset of International Software Benchmark Standards Group [45]. Empirical studies on software maintenance have confirmed the same relation between the cost and the effort [13].

Many studies (see for example [13] and [45]) employed the number of Function Points [4] as a measure of the design and analysis activities of software developers, which is one of the main factors that impacts the software maintenance effort [13]. Several empirical studies [51, 64] on historical vulnerabilities report that typical security fixes are small and local in terms of modified source lines of code (SLOC) and the number of affected functions (Function Points). Since Albrecht and Gaffney [4] observed that Function Points have a high degree of correlation with SLOC, it is fair to assume that resolving a single Function Point during regular maintenance corresponds to resolving a single security vulnerability during security maintenance.

However, as we identified in the previous section, software developers and managers of our industrial partner are considering different proxies for the maintenance effort (`F5`), as it is very

Table 7. Variables used for analysis

Factor	Description	Rationale	Hypothesis
AdopterCount	The number of active users (from Open Hub).	The more popular the project is, the more likely it will be included by as a component.	H_1
DebInst	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 is very 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.	H_1
NumDevs	The number of unique contributors to the source code repository of a project.	Too many contributors might induce vulnerabilities as they might not have exhaustive knowledge on the project and can incidentally break some features they are unaware of.	H_1
Years	The age of a project (in years).	More vulnerabilities could be discovered over time. Alternatively, the age may be the sign of the maturity of a project.	H_2
CountLineCode	The 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.	H_2
LinesChangedRatio	The fraction of the total number of added and deleted lines of by the total number of lines of code.	More historical changes might indicate that FOSS developers are fixing vulnerabilities, so that the adopters do not have to fix them anymore. This variable should have negative impact on the effort.	H_2
PrivateVulnReporting	Indicates whether a projects enforces private vulnerability reporting.	Projects that call for responsible security vulnerability disclosure may be more attractive for the adopters.	H_3
ExternalSecTestng	A project is tested for security by third-party organizations.	External security testing indicates sufficient attention to the project, signifying its importance. It may also provide benefits of additional testing.	H_1, H_3
SecTools	Indicates whether security testing tools are used by FOSS developers.	The usage of security testing tools should decrease the number of post-release vulnerabilities, impacting the security maintenance effort for the adopters.	H_3
SecDesignSpec	A project describes how the security is built in, and/or provides a reference to the relevant security standards, and/or provides secure coding guidelines.	These means may help to improve the security of a project in general, impacting security maintenance for both in-house FOSS developers as well as for the adopters.	H_3
VulnList	A project publishes and maintains a list of its known vulnerabilities.	This variable indicates that security is treated responsibly within a project, making it more attractive for the adopters, and potentially impacting the security maintenance for them.	H_1, H_3
MaintSupport	A project provides details about supported versions, release and patching processes.	This information allows the adopters to plan their maintenance actions in advance, thus impacting the security maintenance as well.	H_2
EaseContrib	A project provides clear guidelines for potential new developers.	This factor may attract more new developers, and indirectly impact the security maintenance effort via the **NumDevs** variable.	H_1
CodeStandards	A project specifies coding guidelines for its contributors.	Clear coding guidelines for project contributors may improve the overall quality of a project, and improve the comprehensibility of the source code for the adopters, impacting their security maintenance efforts.	H_1, H_3
CheckedCommits	There exists a code review process for the contributors.	Code reviews may improve the overall quality of a project, and facilitate security maintenance efforts for the adopters. The presence of code reviews may also give the adopters the perception of better quality.	H_1, H_3

difficult to accurately measure working hours of developers as in Banker et al. [13]. Moreover, it was neither impossible to measure the working hours of developers on security maintenance alone (as this data is not separable from "functional" maintenance), nor we could measure them on each consumed FOSS component (as this data is simply not available). Hence, we need to identify suitable proxies.

During our exploratory case study described in Section 3, we also identified that managers and developers perceive the number of products that are using a vulnerable FOSS component as an important factor for the security maintenance (F2). Our further discussions with developers and researchers of our industrial partner confirmed that instead of working hours spent by developers on specific maintenance tasks, 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. At first, 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[13]. This effort is amplified when several development teams are asking to use the FOSS component as a vulnerability which eschewed detection may impact several hundred products and may lead to several security patches for different products.

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) The `Distributed model` covers the case when security fixes are not centralized within a company. For instance, most of vulnerabilities apply to one internal product, so that each development team takes care of security issues in FOSS components that they use [67]. However, as the number of vulnerabilities and products grow over time, vulnerabilities may affect more and more products at once, therefore the effort for security maintenance increases linearly with the number of products using a FOSS component:

$$e_i \propto |vulns_i| \cdot |products_i| \tag{1}$$

(2) In the `Centralized model` a security fix for all instances of a FOSS component is issued once by the 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. Additionally, as pointed by ben Othmane et al. [67], vulnerabilities affecting several products may be addressed by a single generic solution that applies to all of them. Finally, Banker et al. [13] have already shown that maintenance models do not scale linearly, and that software maintenance may be affected by economies of scale: the effort spent by a single maintenance team that distributes fixes for all consumers within a company may be the most economical choice. To reflect this case, we use two versions of this model: `Centralized constant` and `Centralized network` models. In these models the effort for security maintenance scales either logarithmically (essentially distribution effort of pushing fixes being minimal) or quadratically with the number of products using a FOSS component

[13]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 dataset: rho = 0.71, $p < 0.01$) 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.

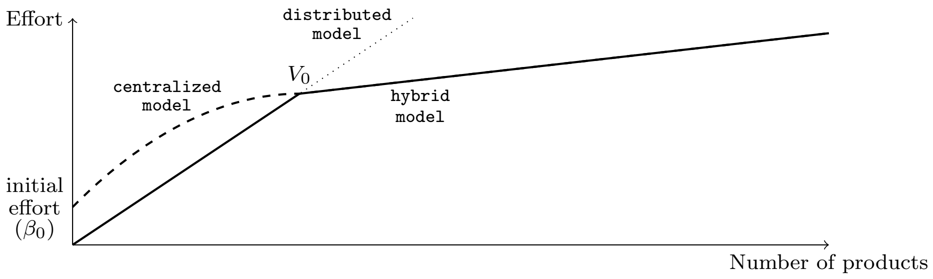


Fig. 4. Illustration of the three effort models

(network effect for the cost of pushing fixes). Their equations are as follows:

$$e_i \propto c_{log}(vulns_i, products_i) = |vulns_i| \cdot \log\big(|products_i|\big) \tag{2}$$

$$e_i \propto c_{net}((vulns_i, products_i)) = |vulns_i| \cdot \sqrt{\big(|products_i|\big)} \tag{3}$$

(3) The `Hybrid model` combines the two previous models: security issues in the least consumed FOSS components (e. g., the number of products that consume them is less than the mean of all consumptions) 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. For this model, we devise two versions as `Hybrid constant` and `Hybrid network` models. Their equations are as follows:

$$e_i \propto \begin{cases} |vulns_i| \cdot |products_i| & \text{if } |products_i| \leq p_0 \\ p_0 \cdot |vulns_i| + c_{hyb\,x}(vulns_i, products_i) & \text{otherwise for} x \in \{log, net\} \end{cases} \tag{4}$$

Where $c_{hybx}(vulns_i, products_0) = 0$ to guarantee continuity at the indifference point and, as the number of products grows $products_i > products_0$ the cost function $c_{hyb\,x}$ asymptotically scales as c_x, i.e. $c_{hybx}(vulns_i, products_i) = O(c_x(vulns_i, products_0))$.

As shown in Figure 4, the `Hybrid` model is a combination of the `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 the centralized and distributed effort models. The hybrid model captures the possibility of a company to switch models after (or before) the indifference point. The fixed effort of the centralized model is obviously higher than the one of a distributed model (e. g., setting up a centralized vulnerability team, 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 or exponential 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/exponential in the overall number of products after $|products_i| > p_0$. This would create a sharp drop in the effort for the security analysis of FOSS components used by several products after p_0 is reached. This phenomenon is neither justified on the field, nor by economic theory. In the sequel, we have used for p_0 the mean of the distribution of the proprietary products that are using a FOSS component.

6 EMPIRICAL DATA ANALYSIS

6.1 Data Collection

We considered the following public data sources to obtain the data on factors of FOSS components that we outlined in Section 5:

1. **National Vulnerability Database (NVD)** – the US government public vulnerability database, we use it as the main source of public vulnerabilities (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 we obtain from the NVD (http://osvdb.org).
3. **Black Duck Code Center** – a commercial platform for the open source governance 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 vendor's installation to identify its most popular FOSS components.
4. **Open Hub (formerly Ohloh)** – a free offering from the Black Duck that is supported by the online community. The Open Hub retrieves data from source code repositories of FOSS projects and maintains statistics that represent various properties of the code base of a project (https://www.openhub.net/).
5. **Coverity Scan Service** – in 2006 Coverity started the initiative of providing free static 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 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 to obtain information about Debian installations (https://www.coreinfrastructure.org/programs/census-project).
7. **HackerOne** – an online platform for white-hat hackers, where various companies publish security bug bounties, including vulnerabilities in FOSS projects. We used this information to identify whether various companies seek to perform external security testing of the FOSS projects in our sample (https://www.hackerone.com/).

We have collected the dataset of 152 FOSS projects (projects that are consumed by at least 5 products as indicated in the Black Duck Code Center repository of our industrial partner). We have showed some of the descriptive statistics of these projects earlier in Table 2 and Figure 1.

For each of these projects we have collected the dependent variables (discussed in Section 5.2), as well as interval and dummy independent variables described in Tables 4, 5, and 6. During the manual collection procedure, we have examined the data sources (4), (5), and (7) described above. The interval independent and the dependent variables have been collected automatically, while the dummy variables have been collected manually (see Table 7 for the final list of variables used for the analysis). In spite of their intuitive appeal, we excluded some of the dummy variables related to the programming languages[14] 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 the variable related to the presence of security tests, as we were unable to find this information with the above data sources. Our experience with data collection suggests that this information could be obtained from source code repositories, however it would still require significant manual effort, and it is unclear whether this effort will eventually pay off.

[14]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 [93].

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 several projects, but also as a development framework in few other projects (indeed, it can be considered to be both a framework and a set of libraries). Splitting the *Apache Struts 2* data point into another two instances marked as "library" and "framework" would introduce dependency relations between these data points. On the other hand, 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, as well as the time spent for the interviews.

6.2 Demographics

Table 8 shows the descriptive statistics on the interval response variables (the security maintenance effort that corresponds to the three models discussed in Section 5.2), as well as statistics on the explanatory variables that we collected from the data sources discussed above. According to previous studies [81], several of these explanatory variables are already known for having the discriminative power with respect to software vulnerabilities.

As we are interested in assessing the factor of scale of the impact of our explanatory variables on the security maintenance, we apply the *log* and *square root* transformations to these variables (their distributions after this transformation are normal according to the Shapiro-Wilk test). We added 1 to the number of adopters and downloads variables (**DebInst** and **AdopterCount**) before transforming them with *log*, since some of the data points initially contained zeros for their values. We justify this by the fact that the value of these variables should be at least 1, as we know that our industrial partner integrates all components from the sample.

Some FOSS components from our sample did not have historical vulnerabilities at the moment of this writing, however, it does not mean that there may be no vulnerabilities in these components in the future. Therefore, we added 1 to the total number of vulnerabilities (**Vulns**) of all FOSS components from the sample to reflect the case in which a vulnerability had just appeared in a component: if there are no historical vulnerabilities in the component, the effort for security maintenance will be proportional to the resolution of only that vulnerability. However, when there are historical vulnerabilities present for that component, the resolution of the new vulnerability will be likely complicated: the Security Response and Maintenance teams will have to ensure that none of the historical vulnerabilities is re-introduced due to the applied fix (let alone the compatibility issues between older versions of a FOSS component and the fix which is typically provided for a recent version only).

Some of the variables shown in Table 8 have strong correlations with other variables: for instance, **NumCommits** correlates strongly with **CountLineCode** and **NumDevs**. Also, to assess the potential impact of of the popularity of a programming language, we tried to divide the size of the code base into two different variables: **CountLinePopular** – the size of the code base written in popular languages (e. g., Java, C/C++, PHP, JavaScript), and **CountLineOther** – 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 **LinesDeleted** and **LinesNew** correlate strongly with **NumCommits** and **NumDevs**, as well as with **CountLineCode**. Therefore, we had to further limit the number of variables for the regression.

For the latter, we had to come up with another metric that would capture the changes to the source code – **LinesChangedRatio**. Figure 5 illustrates why we could not use **LinesNew**

Table 8. Descriptive statistics of the collected variables

	Statistic					
Variable	Min	1st Quartile	Median	Mean	3rd Quartile	Max
distributed_effort	5.0	41.8	85.0	880.6	384.3	34752.0
centralized_log_effort	1.7	3.9	6.1	60.2	28.9	1025.2
centralized_netw_effort	2.2	7.1	12.2	110.9	54.4	1025.5
hybrid_log_effort	10.0	65.9	96.0	602.0	335.2	12116.4
hybrid_netw_effort	10.0	66.2	96.0	619.5	348.0	13218.9
AdopterCount	1.0	13.5	59.5	279.1	201.0	9391.0
DebInst	1.0	70.0	1450.0	22034.0	12846.0	175852.0
NumDevs	1.0	16.0	36.0	123.1	105.5	1433.0
NumCommits	18.0	1440.0	4462.0	10410.0	9656.0	174803.0
Years	1.0	7.0	10.0	10.5	14.0	28.0
CountLineCode	3353.0	43406.0	161081.0	564292.0	482120.0	23220236.0
LinesNew	10066.0	287832.0	875808.0	2671514.0	210747.3	80217058.0
LinesDeleted	1205.0	147049.0	562421.0	1741998.0	1449512.0	45837572.0
LinesChangedRatio	1.7	5.28	7.7	16.0	12.83	638.1

and **LinesDeleted** as factors, as they correlate with each other and with **CountLineCode**. As **LinesChangedRatio** does not have a strong correlation with **CountLineCode**, it can be used as an independent variable.

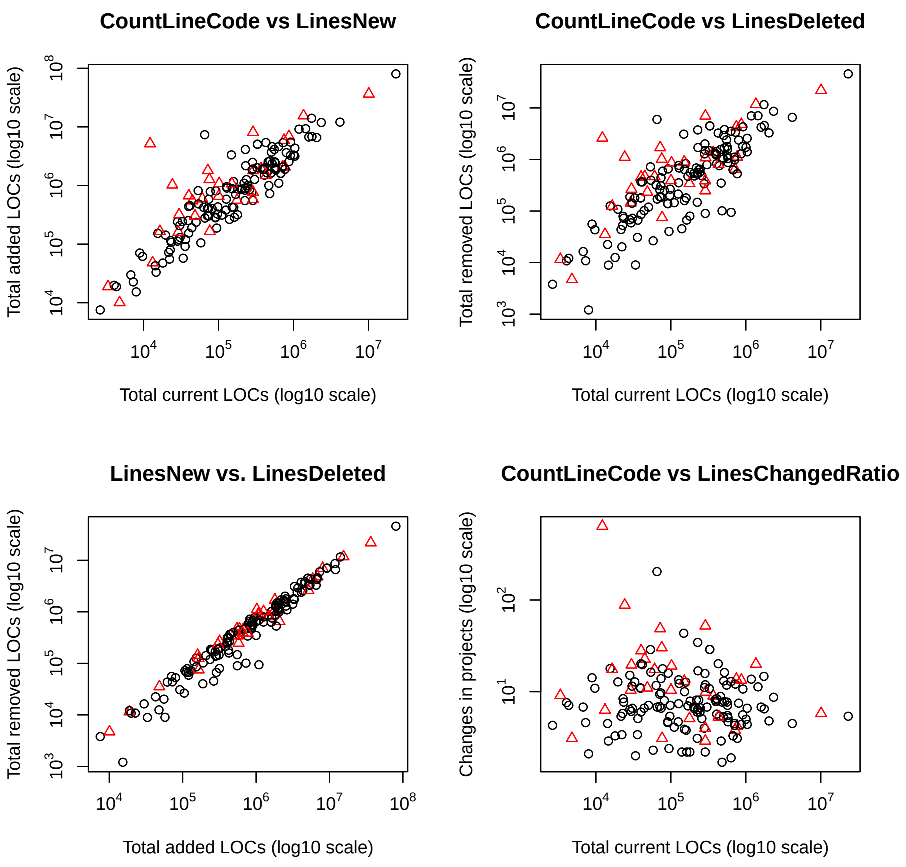
Table 9. Variance inflation factors of the explanatory variables

Variable	VIF	Variable	VIF
Years	1.71	**SecDesignSpec**	1.53
AdopterCount	1.88	**VulnList**	1.73
DebInst	1.82	**ExternalSecTesing**	1.31
NumDevs	1.37	**MaintSupport**	1.85
CountLineCode	1.65	**EaseContrib**	1.63
LinesChangedRatio	1.33	**CodeStandards**	1.87
PrivateVulnReporting	1.63	**CheckedCommits**	1.80
SecTools	1.53		

Finally, we performed the correlation analysis of the remaining variables in order to determine whether the multi-collinearity problem remains. We first built the correlation matrix using Spearman rank correlations and observed that there were still weak-to-moderate correlations in some of the variables. However, according to Stevens [83, 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 9), 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 [62, pp369], these values are acceptable and indicate that the selected explanatory variables do not have significant cross influences.

6.3 Analysis

To assess the potential individual impact of the above factors that characterize FOSS components on the security maintenance effort, we employ the OLS method of linear regression [46, Chapter 3]. The results of estimates for each security effort model are given in Table 10.



*The figure shows that the numbers of added and deleted lines of code (**LinesNew** and **LinesDeleted**) could not be used as independent variables, since they have a strong correlation with each other (as well as with the number of total lines of code **CountLineCode**). On the other hand, the number of added and deleted lines of code divided by the total size of a component (**LinesChangedRatio**) can be used as an independent predictor. The red triangles additionally show the fraction of the lines of code written in scripting languages, and the black circles indicate the non-scripting languages.*

Fig. 5. The rationale for using the **LinesChangedRatio** metric

We found that in our models the number of active users **AdopterCount** has much larger positive impact on the security maintenance effort than the number of downloads by regular users **DebInst**. As we anticipated, active users may be more likely to find and report vulnerabilities, while user that simply download and install software packages may be less likely to do it. The impact of **AdopterCount** is statistically significant only in `Centralized` models. The impact of **DebInst** is negative, albeit it is rather small and is not significant in any of the models. The latter could be explained by the intuition that only a major increase of the popularity of a FOSS project could result in more regular users finding and reporting vulnerabilities, as not every regular user is likely to have enough knowledge and motivation to discover and report security issues in software that they are using.

The total number of unique developers **NumDevs** does not seem to have a major impact. As we pointed in Section 5, we could not use the number of different developers that make changes to

Table 10. Regression Results

VulnList *is statistically significant in all models, and has positive impact.* ***AdopterCount*** *and* ***ExternalSecTesting*** *have positive impact and are significant on all* `Centralized` *and* `Hybrid` *models respectively.* ***Years*** *has positive impact in all models, but is statistically significant only in* `Hybrid` *models.* ***CodeStandards*** *has negative impact in all models (significant only in* `Hybrid constant` *model).* ***CheckedCommits*** *has positive impact (statistically significant in* `Centralized constant` *model).*

	Distributed model		Centralized log model		Centralized network model	
Intercept	−2024.92	(−0.76)	−159.48	(−1.33)	−291.17	(−1.25)
Years	475.38	(1.01)	32.25	(1.52)	60.86	(1.47)
AdopterCount	206.78	(1.02)	23.03	(2.55)*	36.53	(2.07)*
DebInst	−11.17	(−0.12)	−4.02	(−0.97)	−5.35	(−0.67)
NumDevs	61.35	(0.28)	1.25	(0.13)	4.01	(0.21)
CountLineCode	−12.87	(−0.06)	0.30	(0.31)	0.41	(0.02)
LinesChangedRatio	−192.61	(−0.53)	−7.34	(−0.45)	−16.10	(−0.51)
PrivateVulnReporting	−55.10	(−0.08)	49.89	(1.62)	59.31	(0.99)
ExternalSecTesting	1347.06	(1.55)	53.77	(1.37)	120.18	(1.58)
SecTools	795.82	(1.14)	44.47	(1.42)	89.03	(1.46)
SecDesignSpec	−126.44	(−0.19)	0.39	(0.01)	−5.71	(−0.10)
VulnList	1872.48	(2.32)*	128.95	(3.56)***	238.23	(3.38)***
MaintSupport	652.65	(0.88)	−23.19	(−0.70)	−3.36	(−0.05)
EaseContrib	346.84	(0.52)	9.10	(0.30)	26.55	(0.46)
CodeStandards	−731.18	(−0.91)	−47.26	(−1.31)	−91.27	(−1.29)
CheckedCommits	442.21	(0.49)	70.35	(1.73) .	103.05	(1.30)
N	152		152		152	
Multiple R^2	0.25		0.41		0.38	
Adjusted R^2	0.16		0.34		0.31	

Note, t-statistics are in parentheses. Signif.codes: . 5%, * 1%, ** 0.01%, *** 0.001%

	Hybrid log model			Hybrid network model	
Intercept	−2150.32	(−1.70)	.	−2153.60	(−1.62)
Years	444.67	(1.98)	*	447.88	(1.90) .
AdopterCount	96.89	(1.01)		102.31	(1.02)
DebInst	−6.52	(−0.15)		−6.61	(−0.14)
NumDevs	28.23	(0.28)		29.68	(0.28)
CountLineCode	35.83	(0.36)		33.79	(0.32)
LinesChangedRatio	−52.62	(−0.30)		−59.48	(−0.33)
PrivateVulnReporting	56.71	(0.17)		46.58	(0.14)
ExternalSecTesting	967.64	(2.34)	*	987.06	(2.27)*
SecTools	454.85	(1.37)		474.23	(1.36)
SecDesignSpec	−54.27	(−0.17)		−59.71	(−0.18)
VulnList	1214.21	(3.17)	**	1255.50	(3.12)**
MaintSupport	262.23	(0.75)		287.19	(0.78)
EaseContrib	253.44	(0.80)		259.78	(0.78)
CodeStandards	−646.82	(−1.69)	.	−654.40	(−1.62)
CheckedCommits	87.75	(0.49)		105.72	(0.23)
N	152			152	
Multiple R^2	0.36			0.36	
Adjusted R^2	0.29			0.29	

Note, t-statistics are in parentheses. Signif.codes: . 5%, * 1%, ** 0.01%, *** 0.001%

different parts of a single FOSS component as it was done by previous works [81, 99], therefore, this could be the reason why we could not capture the impact.

Security bugs grow over time [53], which can be explained by the interest of attackers [5], and the vulnerability discovery rate being highest during the active development phase of a project [54]. Our results show that the variable **Years** - the age of a project has significant and relatively large positive impact in `Hybrid` models, thus supporting these observations.

Zhang [97] and Koru et al. [50] who showed a positive relation between the size of a code base and the number of defects (which is a component of the effort variable in our models). In our models, the **CountLineCode** variable has mostly moderate positive impact, albeit it is not statistically significant. While this supports the observations in [50, 97], we cannot fully confirm this fact. On the other hand, the **LinesChangedRatio** has negative impact in all models (albeit, not statistically significant). We expected the opposite result, as many works [32, 63, 81] suggest a positive relation between the number and the frequency of changes and defects. However, these works assessed the changes with respect to *distinct* software releases or distinct components of a single software system, while we are using the cumulative number of changes for all versions of the FOSS components from our sample as this is the parameter of interest for a developer team that must maintain the software for over 10 years (see sample distribution in Figure 2).

For the security-related factors, we observed that while **ExternalSecTesing** may indicate the high interest of adopters in a FOSS project that helps to improve their security status [93], it may have a major impact on the security maintenance effort of individual adopters (statistically significant in `Hybrid` models). Thus, while external security testing practices serve an important role of securing FOSS projects and helps their adopters in the long run, it may have consequences for individual adopters: for example, when all of a sudden a third party publishes a security bug bounty for a project, and other adopters do not take action by monitoring various sources for new vulnerabilities, their security maintenance effort may be significantly increased. This intuition is additionally supported by the fact that the **VulnList** variable has a major and statistically significant impact in all our models.

Finally, we observe that **CodeStandards** has major negative impact on security maintenance, supporting the intuition that coding standards are helpful for future security maintenance [67] (statistically significant in `Hybrid` constant model). Also, according to our results **CheckedCommits**, has moderate positive impact (yet, statistically significant only in `Centralized` constant model). The latter may be due the fact that while code reviews are helpful for supporting the quality of a software product in general, they require a lot of resources from FOSS developers, potentially resulting in errors in specific areas of a project.

Overall, we found the evidence that supports our hypothesis H_1 and H_3, however, the evidence for H_2 is limited. Our reported R^2 values (0.25, 0.41, 0.38, 0.36, 0.36) are acceptable, as our purpose is to see which variables have the impact. We have not considered some of the variables listed in Tables 4, 6, and 5, because of the reasons explained in Section 6.2. Thus, we can only explain part of the variance.

7 DISCUSSION AND IMPLICATIONS

In this study we aimed to bridge the existing knowledge on the economics of software maintenance, as well as the empirical research on software defects and vulnerabilities in order to investigate another the important yet overlooked aspect of software economics – the security maintenance of third-party FOSS components.

We performed an exploratory case study to obtain insights on how the selection and maintenance of FOSS components is handled within the software supply chain of a large international proprietary software vendor, and identified major challenges of FOSS selection and consumption. We have also developed a theory of factors impacting the vulnerability resolution process and investigated their impact on three different security maintenance models, that can be used by software development managers and stakeholders for guiding their decisions on various aspects of the security maintenance.

Our study has implications for developers, managers and stakeholders, as well as for security researchers who are interested in software ecosystems that result from combining proprietary and free software.

7.1 Implications for Practice

We identified that security aspects of a software product are relevant on all stages of its development lifecycle (the SDL process described in Section 3.1), starting from the *Preparation* stage when all the selected third-party components should pass the initial quality gates regulated by the *Inbound* component selection process. The development activities end after the *Transition* phase, where the entire software product has to pass the internal security certification process. However, the security concerns do not actually end at that phase, as the software product finally enters the *Utilization* phase where it remains for support and maintenance for many years.

This long support period implies that the issue of security maintenance for many older versions of FOSS components shipped with older versions of the software product is extremely relevant as well - the importance of security characteristics of FOSS components becomes much more apparent during the *Utilization* phase.

While the SDL process dictates that FOSS components should receive the same treatment with no difference to the in-house coding, this rule is difficult to enforce especially during the *Utilization* phase 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. Unfortunately, traditional security testing strategies cannot be easily applied to a set of integrated FOSS components as soon as the maintenance phase begins: the members of the Security Response and Maintenance teams must react *quickly* to new security issues reported for these components that can potentially affect hundreds of customers.

This suggests that an efficient strategy for securing the software supply chains of proprietary vendors that integrate FOSS components, and decreasing the costs and security maintenance efforts, should start from increasing the awareness among the software development teams about the consequences of their choices of specific FOSS components, as well as the management decisions on the components that have been already integrated. Also, apart from various development processes that regulate the selection of components, vendors should constantly monitor various quantitative and qualitative data about third-party FOSS components from multiple sources, and maintain software repository where this data can be accessed by all stakeholders.

7.2 Implications for Research

A large body of research concentrated on generic software maintenance, however there was little focus on the security maintenance. Therefore, we believe that our study will contribute to closing this gap and encourage other researchers to investigate the security maintenance aspects in more depth, as well as motivate the industrial software vendors to share their datasets with researchers to foster more empirical research in this area.

The biggest challenge in applying the results of empirical research in general, as well as empirical research that aims on studying 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. For instance, Massacci and Nguyen [54] addressed the question of selecting the right source of ground truth for vulnerability analysis. The authors [54] 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 (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.

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 – thus, we confirm several known results in the area of security effort, however, we also show that these metrics are relevant for the variety of software components rather than for different parts of a single software system as in [81] and [99].

7.3 Limitations and 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, removed the duplicates and performed manual spot checks. The threat related to the accuracy of the data sources 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 over-fitting 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[15] index). This suggests that the study is likely relevant for other software vendors as well.

8 CONCLUSIONS

In this paper we have developed a theory of factors impacting the vulnerability resolution process and investigated their impact on several security maintenance models for large software vendors

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

that have extensive consumption of FOSS components. We have instantiated several security effort models – `Centralized`, `Distributed`, and `Hybrid`, and collected variables that represent factors impacting these models. We have collected data on these variables from 152 FOSS components currently consumed by software products of our industrial partner, and analyzed their statistical significance for these models.

As a proxy for security maintenance effort of consumed FOSS components we used the combination of the number of products that rely upon 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 popularity (namely, the number of adopters) of a FOSS component, its age, and several security-related factors: external security testing of a component by other companies or adopters, and the availability of the information about known security vulnerabilities in a project. We also observed that adopters should investigate whether coding guidelines are enforced within FOSS projects, and whether there exist code reviews practices, as these two factors also mat have impact on security maintenance.

While our study was performed at a single software vendor that adopts FOSS components, we believe that the results of this study can be generalized to other adopters as well. This is due to the fact that most of the theoretical development and measures are not adopter-specific but rather FOSS-specific. As a future work we plan collecting a wider sample of FOSS projects, assessing other explanatory variables and investigating our models further. Using the data for prediction of the effort is also a promising direction for the future work.

REFERENCES

[1] M. Aberdour. Achieving quality in open-source software. *IEEE Software*, 24(1):58–64, 2007. 9, 15, 17

[2] 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. 3

[3] Norita Ahmad and Phillip A Laplante. A systematic approach to evaluating open source software. *Strategic Adoption of Technological Innovations*, page 50, 2013. 10

[4] Allan J. Albrecht and John E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, 1983. 17

[5] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. Security vulnerabilities in software systems: A quantitative perspective. In *Data and Applications Security XIX*, pages 281–294. Springer, 2005. 12, 26

[6] Prasanth Anbalagan and Mladen Vouk. On predicting the time taken to correct bug reports in open source projects. In *Proceedings of International Conference on Software Maintenance (ICSM'09)*, 2009. 10

[7] Claudio Agostino Ardagna, Ernesto Damiani, and Fulvio Frati. Focse: an owa-based evaluation framework for os adoption in critical environments. In *Open Source Development, Adoption and Innovation*, pages 3–16. Springer, 2007. 10, 16

[8] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005. 13, 17

[9] 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. 3

[10] 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. 10

[11] 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. 3, 10, 12

[12] Ruediger Bachmann and Achim D. Brucker. Developing secure software: A holistic approach to security testing. *Datenschutz und Datensicherheit*, 38(4):257–261, 2014. 5

[13] 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. 9, 10, 11, 16, 17, 19

[14] 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. 9, 10, 11, 12

[15] Rajiv D. Banker and Sandra A. Slaughter. A field study of scale economies in software maintenance. *Management Science*, 43(12):1709–1725, 1997. 11, 12

[16] 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. 13, 16

[17] 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. 12, 13, 14

[18] Black Duck Software. The tenth annual future of open source survey. https://www.blackducksoftware.com/2016-future-of-open-source, 2016. Last accessed 04.07.2017. 1

[19] Achim D. Brucker and Uwe Sodan. Deploying static application security testing on a large scale. *Datenschutz und Datensicherheit*, pages 91–101, 2014. 1, 5

[20] Andrea Capiluppi. Models for the evolution of os projects. In *Proceedings of International Conference on Software Maintenance (ICSM'03)*, pages 65–74, Los Alamitos, CA, USA, 2003. IEEE Computer Society. 13, 16

[21] Eugenio Capra, Chiara Francalanci, and Francesco Merlo. The economics of open source software: an empirical analysis of maintenance costs. In *Proceedings of International Conference on Software Maintenance (ICSM'07)*, pages 395–404. IEEE, 2007. 9, 11

[22] Hasan Cavusoglu, Huseyin Cavusoglu, and Jun Zhang. Security patch management: Share the burden or share the damage? *Management Science*, 54(4):657–670, 2008. 3, 11

[23] 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. 9, 11

[24] 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. 9

[25] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004. 13, 17

[26] Steve Christey. Unforgivable vulnerabilities. *Black Hat Briefings*, 2007. 13, 17

[27] 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. 12, 15

[28] 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)*, pages 190–206. Springer, 2016. 1

[29] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. Software bertillonage. *Empirical Software Engineering*, 18(6):1195–1237, 2013. 8

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

[31] 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. 11, 12

[32] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. Prioritizing software security fortification through code-level metrics. In *Proceedings of the 4th Workshop on Quality of Protection (QoP'08)*, 2008. 13, 16, 26

[33] Barney G. Glaser and Anselm L. Strauss. Grounded theory. *Strategien qualitativer Forschung. Bern: Huber*, 1998. 4

[34] Ron Goldman and Richard P. Gabriel. *Innovation happens elsewhere: Open source as business strategy.* Morgan Kaufmann, 2005. 7

[35] 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. 4

[36] Greg Guest, Kathleen M. MacQueen, and Emily E. Namey. *Applied thematic analysis.* Sage, 2011. 4

[37] Mohanad Halaweh. Using grounded theory as a method for system requirements analysis. *Journal of Information Systems and Technology Management*, 9(1):23–38, 2012. 4

[38] 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. 14

[39] 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. 1

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

[41] 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. 4

[42] Julian P. Hoppner. The GPL prevails: An analysis of the first-ever court decision on the validity and effectively of the GPL. *A Journal of Law, Technology & Society (SCRIPTed)*, 1:628, 2004. 2

[43] 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. 1

[44] Michael Howard and Steve Lipner. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software.* Microsoft Press, 2006. 5

[45] Hennie Huijgens, Arie van Deursen, Leandro L. Minku, and Chris Lokan. Effort and cost in software engineering: A comparison of two industrial data sets. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE'17)*, 2017. 9, 17

[46] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 6. Springer, 2013. 23

[47] Russell L. Jones and Abhinav Rastogi. Secure coding: building security into the software development life cycle. *Information Systems Security*, 13(5):29–39, 2004. 13, 17

[48] Magne Jorgensen and Martin Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007. 17

[49] 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. 13, 16

[50] 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. 13, 16, 26

[51] Daoyuan Li, Li Li, Dongsun Kim, Tegawendé F Bissyandé, David Lo, and Yves Le Traon. Watch out for this commit! a study of influential software changes. *arXiv preprint arXiv:1606.03266*, 2016. 17

[52] 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. 1, 7

[53] 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. 26

[54] 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. 26, 27

[55] Fabio Massacci and Viet Hung Nguyen. An empirical methodology to evaluate vulnerability discovery models. *IEEE Transactions on Software Engineering*, 40(12):1147–1162, 2014. 12, 14

[56] 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)*, pages 94–103, 2012. 15

[57] Gary McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006. 13, 17

[58] 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*, pages 1–44, 2015. 13, 17

[59] 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. 10

[60] Mark Merkow. *Risk Analysis and Management for the Software Supply Chain*, 2013. 1

[61] 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. 17

[62] Raymond H. Myers. *Classical and modern regression with applications.* Duxbury Press, Pacific Grove, 2000. 23

[63] 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. 13, 14, 15, 16, 26

[64] 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, 2016. 17

[65] 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. 14

[66] 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. 12, 13, 14

[67] Lotfi Ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, and Achim D. Brucker. Time for addressing software security issues: Prediction models and impacting factors. *Data Science and Engineering*, 2(2):107–124, 2017. 10, 15, 16, 17, 19, 26

[68] Andy Ozment. Software security growth modeling: Examining vulnerabilities with reliability growth models. In *Quality of Protection: Security Measurements and Metrics*, pages 25–36. Springer, 2006. 12

[69] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th USENIX Security Symposium*, 2006. 16

[70] 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. 15

[71] 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. 14

[72] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999. 12, 15, 17

[73] Eric Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005. 4, 11

[74] Gregorio Robles, Jesús M. González-Barahona, Carlos Cervigón, Andrea Capiluppi, and Daniel Izquierdo-Cortázar. Estimating development effort in free/open source software projects by mining software repositories: A case study of openstack. In *Proceedings of the 8th International Working Conference on Mining Software Repositories MSR('11)*, 2014. 10

[75] Michel Ruffin and Christof Ebert. Using open source software in product development: A primer. *IEEE Software*, 21(1):82–86, 2004. 3, 10

[76] 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. 10, 12, 15

[77] 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. 10

[78] 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. 13, 14

[79] Guido Schryen. Is open source security a myth? *Communications of the ACM*, 54(5):130–140, 2011. 1

[80] Robert C. Seacord. Secure coding standards. In *Proceedings of the Static Analysis Summit, NIST Special Publication*, 2006. 13, 17

[81] 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. 13, 14, 15, 16, 22, 25, 26, 28

[82] 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. 14

[83] James P. Stevens. *Applied multivariate statistics for the social sciences.* Routledge, 2012. 23

[84] 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. 3, 10

[85] Anselm Strauss and Juliet Corbin. *Basics of qualitative research*, volume 15. Newbury Park, CA: Sage, 1990. 4

[86] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002. 11

[87] Ferdian Thung. Automatic prediction of bug fixing effort measured by code churn size. In *Proceedings of the 5th International Workshop on Software Mining*, 2016. 10

[88] Amrit Tiwana. Does interfirm modularity complement ignorance? a field study of software outsourcing alliances. *Strategic Management Journal*, 29(11):1241–1252, 2008. 2, 10

[89] 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. 3

[90] James Walden and Maureen Doyle. SAVI: Static-analysis vulnerability indicator. *IEEE Security & Privacy*, 10(3):32–39, 2012. 13, 14

[91] 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. 13, 14

[92] David A Wheeler. How to evaluate open source software/free software (OSS/FS) programs. Whitepaper. Accessed on 04.07.2017, 2011. 13, 17

[93] David A Wheeler and Samir Khakimov. Open source software projects needing security investments. Whitepaper. Accessed on 04.07.2017, 2015. 10, 15, 17, 21, 26

[94] Robert K. Yin. *Case study research: Design and methods.* Sage, 2013. 4, 5

[95] Liguo Yu. Indirectly predicting the maintenance effort of open-source software. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(5):311–332, 2006. 9, 10, 16

[96] 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. 10, 12, 13, 15, 16

[97] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Proceedings of International Conference on Software Maintenance (ICSM'09)*, 2009. 13, 16, 26

[98] Luyin Zhao and Sebastian Elbaum. Quality assurance under the open source development model. *Journal of Systems and Software*, 66(1):65–75, 2003. 17

[99] 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. 13, 15, 16, 25, 28