The Journal of Systems & Software 172 (2021) 110653

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities

Antonios Gkortzis [a,*], Daniel Feitosa [b], Diomidis Spinellis [a]

[a] Department of Management Science and Technology, Athens University of Economics and Business, Patision 76, Athens, 10434, Greece
[b] Campus Fryslân, University of Groningen, Wirdumerdijk 34, 8911 CE Leeuwarden, The Netherlands

ARTICLE INFO

Article history:
Received 3 November 2019
Received in revised form 13 April 2020
Accepted 18 May 2020
Available online 20 May 2020

Keywords:
Software reuse
Security vulnerabilities
Case study
Open-source software

ABSTRACT

Software reuse is a widely adopted practice among both researchers and practitioners. The relation between security and reuse can go both ways: a system can become more secure by relying on mature dependencies, or more insecure by exposing a larger attack surface via exploitable dependencies. To follow up on a previous study and shed more light on this subject, we further examine the association between software reuse and security threats. In particular, we empirically investigate 1244 open-source projects in a multiple-case study to explore and discuss the distribution of security vulnerabilities between the code created by a development team and the code reused through dependencies. For that, we consider both potential vulnerabilities, as assessed through static analysis, and disclosed vulnerabilities, reported in public databases. The results suggest that larger projects in size are associated with an increase on the amount of potential vulnerabilities in both native and reused code. Moreover, we found a strong correlation between a higher number of dependencies and vulnerabilities. Based on our empirical investigation, it appears that source code reuse is neither a silver bullet to combat vulnerabilities nor a frightening werewolf that entail an excessive number of them.

1. Introduction

Software reuse is a part of the state-of-practice in software development, being supported by practitioners and researchers alike. The dominant mobile operating system, Android,[1] is a modern, large-scale example of software reuse. The operating system is highly modular, allowing smartphone providers to deploy flavors of it, reusing and customizing most of the functionality. For that, the platform provides a set of more than 3 million Java libraries from the Maven repository.[2] Moreover, Android's core is another great example, since it reuses the Linux kernel, which is among the earliest examples of reuse. UNIX-based systems emerged and evolved thanks to systematic reuse, from which some are still maintained until the present time.

However, software reuse is not a silver bullet. Some of its limitations are not characterized as "concerning" but as "dangerous", in the sense that an important side-effect is the security risks that it may entail. In a study with 4659 open-source software systems, Kula et al. (2018) showed that, although more than 80% of the systems' depended on outdated external libraries, 69% of the interviewed developers were unaware of any security risks that were introduced into the system due to incorporating the reused code. Moreover, in the State of Open-Source Security report,[3] Snyk shares the worrisome findings that between 2017 and 2019, they observed an increase of 88% in the number of disclosed vulnerabilities in open-source libraries.

As a concrete example, Heartbleed[4] was a severe security vulnerability that resided in OpenSSL cryptographic software library, which is a popular open-source component. The vulnerability enabled malicious users to read arbitrary memory contents. By exploiting this vulnerability any user could get access to keys that protected communications, usernames and passwords, personal emails, documents and messages. The bug was detected two years later, after it affected the web servers that were powering 66% of the active web sites at that time.[5] Another, more recent, example is the Equifax incident,[6] in which hackers exploited a known vulnerability in a third-party Java library that Equifax reused,

* Corresponding author.
E-mail addresses: antoniosgkortzis@aueb.gr (A. Gkortzis), d.feitosa@rug.nl (D. Feitosa), dds@aueb.gr (D. Spinellis).

1 https://www.android.com/.

2 https://mvnrepository.com/repos/central.

3 https://snyk.io/blog/88-increase-in-application-library-vulnerabilities-over-two-years/.

4 https://nvd.nist.gov/vuln/detail/CVE-2014-0160.

5 Netcraft, Web Server Survey, April 2014 - https://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html.

6 https://www.equifaxsecurity2017.com/.

https://doi.org/10.1016/j.jss.2020.110653
0164-1212/

and stole personal private information of more than 147 million American citizens.

Various initiatives try to battle this problem. GitHub introduced the Security Alert for Vulnerable Dependencies[7] service aiming to increase users' awareness and mitigate the potential security risks. Similarly, any Linux or BSD system notifies its users for available security updates in vulnerable versions of installed packages and system libraries. Additionally, several popular security assessment tools (e.g., SpotBugs, Snyk, OWASP Dependency Check]) have plugins available for integrating them in any build automation tool and Continuous Integration (CI) service.

Development teams can reuse software through third-party libraries to add functionality to their system without the need to implement an available feature from scratch. Software reuse can be performed in two ways: (a) black-box, in which the reused code is in binary form and (b) white-box, in which the third-party source code is inserted into the application. In black-box reuse, developers interact with the library through APIs provided by the third-party developers without editing and maintaining its code. On the other hand, in white-box reuse, developers are able to adjust the reused code and also select only a subset of it to reuse. In our study we focus on black-box reuse, considering that developers do not have direct visibility of the library's implementation and as a consequence, no awareness of the security risk, they might inherit. For the rest of the paper with "reused code" we refer to black-box type of reuse, unless stated otherwise.

Despite the existence of security mishaps and the initiatives to counteract them, to the best of our knowledge, there is a lack of large-scale studies that attempt to obtain an overview of how security vulnerabilities are associated with code reuse, so as to understand the phenomenon. To start filling this gap, we carried out a first exploratory study (Gkortzis et al., 2019) to investigate how potential vulnerabilities are distributed in open-source software-intensive systems, with regards to *native code*, i.e., written in-house by the software development team, and reused code, introduced through dependencies. We scope our research to answer concerns of software practitioners and researchers related to the potential security risks when they select to reuse software. Specifically, we aim at answering the following questions: (1) Will the third-party library that I want to reuse suffer from security vulnerabilities? (2) How are security vulnerabilities in open-source projects distributed between native and reused code? (3) Are third-party libraries from well-known open-source communities less vulnerable than those of less known ones? (4) Is the reuse frequency and the number of developers using a third-party library associated with the amount of vulnerabilities in a specific library?

The findings of our previous study suggest that software reuse has a positive effect on reducing security risks. However, this study had the following main limitations. First, the observed relation was not strong, which indicated that a larger sample size could enlighten the discussion, and further factors that might explain the relation could be explored. Second, the investigation was limited to potential vulnerabilities. Although potential vulnerabilities can be used as a proxy of lack of quality and risk due to unmet security levels, they may not reflect the existing exploitable threats reported on repositories of disclosed vulnerabilities.

This paper aims at further alleviating the aforementioned limitations by analyzing a considerably larger set of open-source software systems and, not only compare the levels of security between the native and reused code, but also triangulate the results by investigating an additional source of information, namely disclosed vulnerabilities. To achieve this goal, we considered a new set of 1244 Java projects and collected both disclosed vulnerabilities (reported in public datasets),[8] and potential vulnerabilities (detected based through static analysis). Adding to the initial characteristics we investigated (Gkortzis et al., 2019), we collected information regarding four characteristics of the projects and dependencies of our dataset, namely, (1) supported by well-known communities, (2) belonging to an enterprise organization, (3) the number of their contributors, and (4) the frequency of usage in projects. In addition to the statistical analysis presented in our previous work we extended our analysis to incorporate the aforementioned dimensions.

The analysis of the produced dataset suggests that the native code seems to be more vulnerable than reused code, although the reused code is dominant in the majority of the projects. Additionally, 65% of the analyzed projects suffer from at least one security vulnerability introduced through a dependency. Moreover, the numbers of both disclosed and potential vulnerabilities are strongly correlated to the number of dependencies.

In summary, the contributions of our work are: (a) an enhanced toolkit and associated processes to build a dataset that fosters the investigation of security vulnerabilities with regard to software reuse in open-source Java projects, (b) the aforementioned updated dataset per se, (c) an additional dataset on the dependencies and its characteristics, and (d) an extended statistical analysis of the dataset. We note that the toolkit and guidelines to reproduce the process are available on GitHub[9] and the dataset on Zenodo.[10]

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 describes our theoretical model and the approach for designing our study. Also, it presents the steps necessary to construct and analyze the dataset. Section 4 presents our findings, which we further discuss in Section 5. Section 6 presents the limitations of our study and Section 7 our conclusions.

2. Related work

As we could not find studies that are similar to ours, we broadened the scope of this section to describe efforts dealing with software defects and vulnerabilities in reused code.

Pashchenko et al. (2018) studied the SAP software ecosystem with regards to the vulnerable open-source dependencies that they use. Their dataset comprised the 200 most commonly used open-source Maven dependencies in their systems. Regarding vulnerabilities, they included those that are disclosed in public databases, such as, the CVE database, and thus, their study does not suffer from false positives vulnerability reports. However, the nonexistence of known vulnerabilities does not guarantee the absence of any other undetected vulnerabilities. Their finding showed that 13% of the direct and transitive used dependencies were reported with at least one disclosed vulnerability. In their analysis they excluded non-deployed dependencies (e.g., test dependencies). In the same direction, Neuhaus and Zimmermann (0000) studied the Red Hat Linux (RHEL) distribution and provided empirical evidence that certain packages (can be used as dependencies) increase the risk of vulnerabilities in the system, while other packages decrease it. Their goal is to support developers in decision making regarding which package they should use in their native code. In a more recent study, Zimmermann et al.

[7] https://help.github.com/articles/about-security-alerts-for-vulnerable-dependencies/.

[8] For example, the Common Vulnerabilities and Exposure repository, available at https://cve.mitre.org/.

[9] https://github.com/AntonisGkortzis/Vulnerabilities-in-Reused-Software.

[10] http://doi.org/10.5281/zenodo.2566054.

(2019) investigated security risks attached to JavaScript packages distributed via the NPM package manager. Upon analyzing their dependencies and maintainers, the authors found that, due to transitive vulnerabilities and lack of maintenance, individual packages pose a considerable threat. They also showed that the number of vulnerabilities tends to increase with the number of transitive dependencies. The JavaScript's NPM dependency network was investigated also by Decan et al. (2018). In their study, the authors analyzed 400 vulnerability reports covering a 6-year period and observed that the number of security vulnerabilities and the packages affected by them is growing over time. Additionally, they reported that 54% of the packages in the network have at least one version that is affected by a vulnerable transitive dependency.

Regarding the effect that the size of the developers team have on the security defects in the code, Meneely and Williams (0000) performed a study on the RHEL kernel. The authors provided empirical evidence that large developers teams (with more than nine members) and independent developer groups tend to introduce more security defects in the code compared to smaller development teams or files developed by the core developers.

Mohagheghi et al. (2004) performed an analysis on software defects data for 12 consequent releases of a large-scale telecom system developed by Ericsson. Their goal was to examine how reuse, affects two factors of the system: (1) the defect density (defined as defects per lines of code); and (2) the stability (defined as the degree of modification). The authors provided evidence that both defect density and stability showed better results in reused components compared to those in the non-reused components.

Additionally, Mitropoulos et al. (2014) used FindBugs to perform a large scale analysis on the Maven ecosystem. The outcome of their work is a dataset of the bugs (including security bugs) of more than 17 000 Maven dependencies (155 000 considering all their versions). Their dataset can be used to analyze the risk of using outdated libraries that exist in the Maven Central repository. Although, their work does not examine reuse we find it relevant to mention, since among the results, the authors reported a weak correlation between potential security vulnerabilities and the project size. In a similar direction, Shin et al. (0000) investigated the RHEL kernel and the Mozilla Firefox web-browser to create a prediction model for detecting potentially vulnerable code based on the following three code metrics: (1) complexity; (2) code churn; and (3) developer activity.

Concerning the effects of reusing code snippets from publicly available web sources on the quality of the software, Fischer et al. (2017) reported that 15.4% of the 1.3 million Android applications that they analyzed, contained code snippets related to security, published on StackOverflow.[11] Interestingly, 97.9% of those applications contained one or more vulnerable code snippets. Similarly, Abdalkareem et al. (2017), analyzed 22 Android applications on the extent, and the conditions under which, developers use code snippets copied from StackOverflow. Their findings showed that there was a statistically significant medium increase of bug-fixing commits after reusing code from StackOverflow.

On the subject of detecting vulnerable code, Pham et al. (2010) contributed towards the automated detection of suspicious code. Authors introduced SecureSync, a tool that analyzes existing disclosed vulnerabilities, in open-source systems and creates models in order to detect similar suspicious patterns in other systems. The authors evaluated their approach by analyzing 176 releases of 119 open-source projects and identified suspicious code in 51% of them. Practitioners have also made significant contributions in the area of classifying existing vulnerabilities as exploitable.

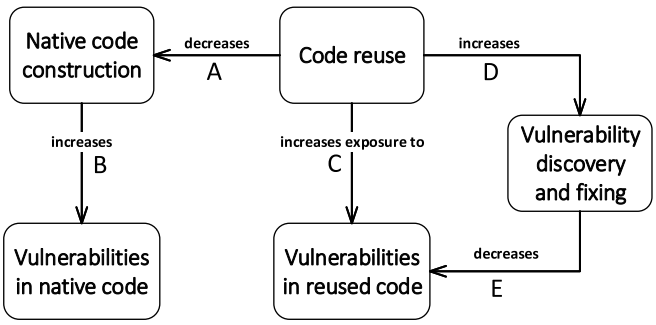

Fig. 1. Theoretical model.

Specifically, Ponta et al. (2018) presented their approach to identify exploitable vulnerabilities based on function call graphs. Vulnerabilities in places of the reused code which are not accessible by the native code can be considered of a lower risk for the system. Recently, they made their tool[12] and the vulnerability dataset available for detecting known vulnerabilities in Java and Python software systems.

In Table 1, we highlight the main differences of our study compared to related work. In particular, to the best of our knowledge, the study reported in this paper is the first to investigate the correlation between code reuse and vulnerabilities, as obtained by means of static analysis, specially in conjunction with disclosed vulnerabilities, in multiple open-source systems.

3. Theoretical and empirical design

In this section, we present the theoretical model and the protocol of our case study, which was designed according to the guidelines of Runeson et al. (2012), and reported based on the Linear Analytic Structure (Runeson et al., 2012).

3.1. Theoretical model

Based on the insights we obtained from the state of the art on the analysis of vulnerabilities and reuse in software-intensive systems (Section 2), we drew the assumptions and designed the theoretical model for our study. The aspects of our analysis and their relationships are visualized in Fig. 1. The relationships presented in Fig. 1 establish the main research questions that are investigated in the following Sections.

Initially, with relationship A we theorize that developers using existing available reusable libraries need to write fewer lines of code to satisfy the requirements of the software system they are implementing. Among other reasons, such an action may also be taken to avoid the accumulation of vulnerabilities (relationship B), as a larger source code base (SLOC) introduces more security risks (Chowdhury and Zulkernine, 2010). However, a side-effect of increasing the number of dependencies is that source code size of the reused code also increases (relationship C), which may bring in more vulnerabilities (Zimmermann et al., 2019; Chowdhury and Zulkernine, 2010).

Despite a potential increase of software size due to reuse, we theorize that code reused through open-source dependencies is more probable to have faster detection and patching of security defects (relationship D) through the application of the so-called Linus' law: "given enough eyeballs, all bugs are shallow" (Raymond, 1999, p. 30), (Wang and Carroll, 2011). Consequently, if projects track their dependencies and update them

11 https://stackoverflow.com/.

12 https://sap.github.io/vulnerabilityassessmenttool/.

Table 1
Comparison against related work.

Study	Context	Focus on security	Number of projects	Language	Source of vulnerabilities	Relate security to reuse
Pashchenko et al. (2018)	Open-source	Yes	200	Java	Manual analysis	Yes
Mohagheghi et al. (2004)	Proprietary	No	1	Java, C & Erlang	Defect reports	Yes
Mitropoulos et al. (2014)	Open-source	Yes	17 505	Java	Static analysis	No
Pham et al. (2010)	Open-source	Yes	119	C & C++	Static analysis and clone detection	Yes
Ponta et al. (2018)	Open-source	Yes	500	Java	Static and dynamic analysis	No
Meneely and Williams (0000)	Open-source	Yes	1	C & C++	Vulnerability reports	No
Shin et al. (0000)	Open-source	Yes	2	C & C++	Vulnerability reports	Partially
Neuhaus and Zimmermann (0000)	Open-source	Yes	1	C & C++	Vulnerability reports	Yes
Zimmermann et al. (2019)	Open-source	Yes	5 386 239	JavaScript	Vulnerability reports	Partially
Decan et al. (2018)	Open-source	Yes	610 000	JavaScript	Vulnerability report	Yes
Fischer et al. (2017)	Open-source	Yes	1 600 000	Java	Static analysis	Yes
Abdalkareem et al. (2017)	Open-source	No	22	Java	Commit changes	No
Ours	Open-source	Yes	1 244	Java	Static analysis and vulnerability reports	Yes

when necessary, there would be fewer security vulnerabilities overall (relationship E) (Pashchenko et al., 2018).

In summary, our goal is to evaluate these relationships based on the findings of our analyses on the produced datasets.

3.2. Objective and research questions

The goal of the study was formulated according to the Goal-Question-Metric (GQM) approach (van Solingen et al., 2002), and is described as follows: "***analyze*** *native and reused code,* ***for the purpose of*** *evaluating,* ***with respect to*** *the differences in the estimated and actual levels of security vulnerabilities,* ***from the point of view of*** *software developers,* ***in the context of*** *open-source software*". To fulfill this objective, we have set three research questions (RQs), as follows:

RQ_1: What size and reuse factors are related with potential security vulnerabilities of a project?

RQ_1 aims at acquiring an overview of how two size factors and two reuse factors are related to the potential security vulnerabilities of a project. In RQ_1 we investigate the relationships B and C presented in Fig. 1. Additionally, we investigate how dependencies from well-known and less known communities affect the number of potential vulnerabilities in a project.

RQ_2: How are potential security vulnerabilities distributed between native and reused code?

RQ_2 aims at investigating an important question correlated with software reuse, namely, the extent to which reuse influences the security of a project. This correlation is depicted with relationships A, B, and C. For that, we exploit static analysis to identify potential vulnerabilities and investigate how native code developed by the project's team and reused code stemming from dependencies on third-party components contribute to the overall estimated security level.

RQ_3: To what extent do open-source projects suffer from vulnerabilities introduced through dependencies?

The purpose of RQ_3 is to collect evidence of disclosed vulnerabilities that affect dependencies used in the projects as Fig. 1 depicts with relationship C. To achieve that, we analyze all dependencies with the OWASP Dependency-Check tool and report the findings.

RQ_4: How are the characteristics of a dependency related to its potential and actual vulnerabilities?

$RQ_{4.1}$: How is the reuse frequency of a dependency related to potential and actual vulnerabilities?

$RQ_{4.2}$: How is the community type of a dependency related to potential and actual vulnerabilities?

RQ_4 aims at investigating the validity of Linus' law, by looking at whether the many eyeballs brought-in through increased reuse actually find and fix potential and disclosed vulnerabilities, as presented in relationships D and E in Fig. 1. Additionally, RQ_4 aims at investigating if the type of the dependency, i.e., from a well-known community or an enterprise organization, is associated with the dependency's number of potential and actual vulnerabilities.

3.3. Cases and unit of analysis

To answer the aforementioned research questions, we designed a multiple-case study, i.e., one in which the multiple cases are also the units of analysis (Runeson et al., 2012). For this study, we chose open-source projects as cases and units of analysis. We selected this particular type of study because the case granularity (i.e., project-level) is sufficient, and multiple cases will provide statistical power to the analysis. Moreover, the selected unit of analysis allows answering the set research questions and pinpoint cases that researchers or practitioners may want to investigate in more detail.

The cases were collected from GitHub Activity Data dataset[13] which is publicly available on the Google Cloud Public Datasets.[14] The GitHub Activity Data 3TB+ dataset contains a full snapshot of the content of more than 2.8 million open-source GitHub repositories including more than 145 million unique commits. Additionally, it contains over 2 billion different file paths, and the contents of the latest revision for 163 million files, all of which are searchable with regular expressions. Users can execute queries on the dataset through the Google BigQuery API.

3.4. Variables and data collection

To address the research questions, we built a containing two groups of variables for each unit of analysis: (a) project information; and (b) vulnerability information. We built the dataset by following a five-step procedure, which is described in the

[13] https://console.cloud.google.com/marketplace/details/github/github-repos?filter=solution-type:dataset&id=46ee22ab-2ca4-4750-81a7-3ee0f0150dcb.
[14] https://cloud.google.com/public-datasets/.

Table 2
List of recorded variables for the projects dataset.

Variable	Description
Project	Full project name
D	Number of dependencies
CVE	Number of disclosed vulnerabilities introduced through dependencies
C	Number of classes in project
C_n	Number of native classes
C_r	Number of reused classes
L	Number of source lines of project
L_n	Number of source lines of code in native classes
L_r	Number of source lines of code in reused classes
L_re	Number of source lines of code in reused classes from an enterprise organization
L_rne	Number of source lines of code in reused classes from a non enterprise organization
L_rw	Number of source lines of code in reused classes from well-known communities
L_rnw	Number of source lines of code in reused classes from less-known communities
V	Number of potential vulnerabilities in project
V_n	Number of potential vulnerabilities in native code
V_r	Number of potential vulnerabilities in reused code
V_re	Number of potential vulnerabilities in reused classes from an enterprise organization
V_rne	Number of potential vulnerabilities in reused classes from a volunteer based contribution
V_rw	Number of potential vulnerabilities in reused classes from well-known communities
V_rnw	Number of potential vulnerabilities in reused classes from less-known communities
VC_n	Number of potentially vulnerable native classes
VC_r	Number of potentially vulnerable reused classes

Table 3
List of recorded variables for the dependencies dataset.

Variable	Description
Dependency	Full dependency name
W	Provided by an open-source well-known community
E	Provided by an enterprise Github organization
CVE_d	Number of disclosed vulnerabilities
V	Number of potential vulnerabilities in dependency
P	Number of projects this dependency is used in
C_b	Number of contributors in projects that use this dependency

following paragraphs together with the associated variables. Fig. 2 illustrates the data collection. A summary of the recorded variables is presented in Table 2. Additionally to the aforementioned dataset, we built a dataset that comprises (a) the dependency information, such as, the community type of its author (Enterpise and well-known); and (b) each dependencies' potential and publicly disclosed vulnerabilities. A summary of the recorded variables for the second dataset is presented in Table 3.

We note that the complete procedure is automated in a set of scripts available on GitHub.[15]

Step 1: Filter projects. First, we queried the GitHub Activity Data database[16] and selected the projects that met the following criteria: (1) contain Java code, and (2) contain at least one Apache Maven[17] build automation configuration file, (i.e., *pom.xml*). We selected Java as a programming language so as to take advantage of automated build support provided by Maven, and the security violation identification capabilities of the SpotBugs[18] tool and the OWASP Dependency Check tool.[19] Maven is well-established tool, and it allowed us to automate the build process of multiple projects and retrieve their dependencies. Both operations were necessary for collecting the potential vulnerabilities. Finally, we queried the GitHub API[20] and retrieved the stars for each project of our aforementioned list. We used the *stars* as an indicator of popularity and we sorted the projects based on that criterion.

Step 2: Download repositories and detect build paths. In this step, we selected the 3500 most popular GitHub projects of the list that we generated in Step 1. We selected a large amount of projects to improve the representativeness of the study sample towards the population and strengthen the statistical analyses. Next, using the Git tool, we cloned locally the projects. Several projects consist of many modules and components written in various programming languages and managed by different build automation tools. To identify the projects that are in the scope of our analysis, we created a tool that automatically detects the root Maven configuration file. We manually resolved cases with multiple root build paths.

Step 3: Build projects and retrieve dependencies. Working on the local copies of the repositories, we built each project. To accelerate this step we skip (1) testing tasks, (2) Java documentation generation tasks, and (3) any static analysis code review tool execution (such as checkstyle[21] and PMD).[22] When the building process is complete, the generated compiled package (i.e., a `.jar` or `.war` file) is stored in the local Maven repository (the `.m2` directory by default). The dependencies (along with any transitive dependencies) that a project defines in its configuration file are also downloaded and stored in the local Maven repository. From the initial 3500, we discarded 1181 projects that failed to build. The main failure reasons were: (a) Java versions incompatibilities, (b) non-accessible Maven dependencies, and (c) compilation errors. For the remaining 2319 successful builds, we created and stored their transitive dependency trees, i.e., the paths to the packages of the project and its dependencies. The dependency trees were retrieved with the use of the `mvn dependency:tree` Maven command.

Step 4: Collect project information. In this step, we analyzed each successfully build project's dependencies' tree that was produced in the previous step. With this process we collected the first groups of variables: *project*, D, C_n, C_r, L_n and L_r (see their definitions in Table 2). For that, we collected the class files from each jar file and also used them to retrieve the source lines of

15 https://github.com/AntonisGkortzis/Vulnerabilities-in-Reused-Software.
16 https://console.cloud.google.com/marketplace/details/github/github-repos?filter=solution-type:dataset&id=46ee22ab-2ca4-4750-81a7-3ee0f0150dcb.
17 https://maven.apache.org/.
18 https://bugs.github.io/.
19 https://www.owasp.org/index.php/OWASP_Dependency_Check.
20 https://developer.github.com/v3/.
21 https://github.com/checkstyle/checkstyle.
22 https://pmd.github.io/.

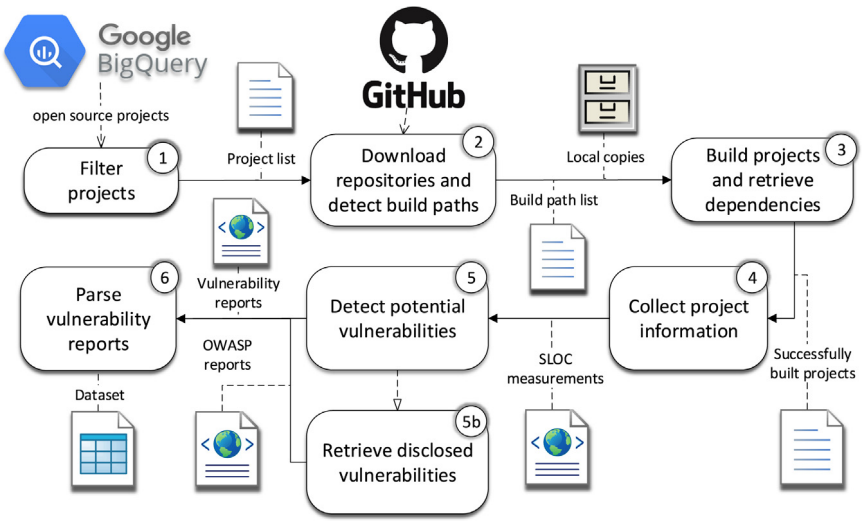

Fig. 2. The dataset construction procedure.

code (SLOC), which is estimated based on the number of the statements. When analyzing the dependencies we count only those that are deployed with the application or used at runtime. These are characterized as `compile`, `runtime`, `provided` in the corresponding Maven configuration file. All other dependencies are ignored since they do not cause an exploitable threat in the deployed application.

Additionally, we analyzed every project and dependency individually and detected those that are maintained by an enterprise organization. For this process we used the dataset provided by Spinellis et al. (2020) which contains a list of 17 252 identified Github enterprise repositories. The dataset defines as an enterprise project "one that is likely to be mainly developed by financially compensated employees, working full time under an organization's management". Furthermore, during this step, we compiled a list of well-known open-source communities that are popular in Github, e.g., Apache, Google, Facebook, Microsoft, MySql and Eclipse. Well-known communities are software development groups that provide high-quality open-source software systems that are widely used by other developers and teams (e.g., the Apache web-server, the Facebook React web-framework, the MySql community database, the Microsoft dotnet framework and the vscode editor. To detect which dependencies are maintained by well-known communities, we mapped the Maven unique identifier of each dependency (i.e., `groupId`) to the `groupIds` of projects belonging to the Github organizations in the list of well-known communities list.

Finally, we performed the next three filtering steps: (1) identified and discarded projects that had no dependencies, (2) discarded projects that had fewer than 1000 lines of native code, and (3) projects that were used in their entirety only as dependencies in other projects. For example, `aws/aws-lambda-java-libs` and `spring-cloud/spring-cloud-(bus|stream|netfix)` appear as dependencies in the dependency-trees of other projects of our dataset. Applying the three filtering rules led us to a final dataset of 1244 projects.

Step 5: Detect potential vulnerabilities. To detect potential vulnerabilities we performed a static analysis of the each project's code base. This type of analysis gives us the ability to assess a large set of projects without the need of test cases and execution scenarios. The latter techniques can prove to be time consuming and prone to missing cases in code coverage. On the other hand, static analyzers look for patterns in the code base of a system while covering all possible execution paths. Kulenovic and Donko (2014) compared different static analysis methods for detecting security vulnerabilities in code bases. They found that there is a constant improvement of the algorithms used for static analysis. Consequently, static analyzers have better performance in terms of accuracy and precision when detecting security vulnerabilities.

For selecting our analyzer we consulted the Open Web Application Security Project's (OWASP) list of static analysis tools,[23] considering only those that: (1) can analyze Java code, (2) can operate offline, (3) are actively maintained by the open-source community, and (4) have rules for detecting patterns of potential security violations. Based on the aforementioned criteria, we selected the static analyzer SpotBugs[24] (v3.1.11) (Hovemeyer and Pugh, 2004; Zheng et al., 2006; Tomassi, 2018). This tool identifies violations of good coding practices (Hovemeyer and Pugh, 2004) by creating rules based on bug patterns. There are nine categories of rules and two of them related to security: *Security* and *Malicious Code*. Moreover, based on the completeness of a rule matching on a bug detection, SpotBugs classifies this detection into one of three levels of confidence (low, medium, high). The tool has already been evaluated in independent studies (Hovemeyer and Pugh, 2004; Feitosa et al., 2015) and (Ayewah et al., 2007), which reported an average precision of 66%. Additionally, using only medium or high level of confidence in the detection rules the precision showed to be significantly increased. Nevertheless, SpotBugs like any static analysis tool, is still prone to introducing noise (false positives) to the data collection. However, other studies showed that SpotBugs findings can be valuable pointers to parts of the system that need to be maintained (Ayewah and Pugh, 2010; Feitosa et al., 2018; Hovemeyer and Pugh, 2004; Khalid et al., 2016; Tripathi and Gupta, 2014; Zheng et al., 2006).

To further enhance the security related detection capabilities of SpotBugs, we included its plugin FindSecBugs.[25] This plugin adds several new bug patterns related to the Open Web Application Security Project (OWASP) top-10 vulnerabilities[26] and several other listed in the Common Weaknesses Enumerations (CWE) list.[27] CWE is a community-list of common software security

[23] https://owasp.org/www-community/Source_Code_Analysis_Tools.
[24] This is the well-known *FindBugs* tool further developed under a new name.
[25] https://find-sec-bugs.github.io/.
[26] https://www.owasp.org/index.php/Top_10-2017_Top_10.
[27] https://cwe.mitre.org/.

weaknesses types, and serves as a common language for classifying security vulnerabilities in software systems. The combination of SpotbBugs core functionality[28] and FindSecBugs' specialized bug patterns[29] offers a capability to detect 163 potential security vulnerabilities. To perform an analysis, SpotBugs requires the path to the compiled Java project and its dependencies. We acquired this information from the lists that we created in Step 3. Next, SpotBugs generates an XML file that reports all the potential vulnerabilities in the given Java classes for both native code base and dependencies. Due to failures in the SpotBugs' analysis, we excluded 260 projects during this step. The two most commons of errors were: (a) executable files missing compiled code, and (b) Java version incompatibilities.

Finally, we analyzed with SpotBugs all dependencies detected in Steps 3 and 4, and collected their potential vulnerabilities. We analyzed the dependencies as standalone `jars`, to avoid including vulnerabilities from other dependencies related to the one that we analyzed. We applied the same filtering that we presented earlier in this Step on the SpotBugs findings for the dependencies.

Step 5b: Retrieve disclosed vulnerabilities. We performed this step in parallel with Step 5. The purpose of this step is use the OWASP Dependency-Check tool in order to analyze all dependencies and to retrieve the information for its disclosed vulnerabilities. The OWASP Dependency-Check tool reports the unique identifier (CVE) for the dependency of our interest and the complete tree of transitive dependencies. We exclude disclosed vulnerabilities that refer to non-Java transitive dependencies. This step populated the variable *CVE* in Table 2.

Step 6: Collect vulnerability information. In this final step, we collected the second groups of variables for each project: V_n, V_r, VC_n, VC_r, VL_n, and VL_r. For that, we parse each XML report that we generated by SpotBugs in Step 5. From these reports we select only the potential security vulnerabilities and we discard all other data. Then, we aggregate the results separately for the native source code and the reused source code. Next, we parse the JSON reports that OWASP Dependency Check tool generated for each dependency in Step 5b and assign a list of unique disclosed vulnerabilities (CVEs) to each project. In this list we include only vulnerabilities related to Java code and dismiss all others.

3.5. Analysis procedure

To investigate the collected data, we performed various statistical analyses. First, to answer RQ_1, we calculated the descriptive statistics on all collected variables, and used linear regression analysis for four selected variables associated with project size and reuse. Additionally, we investigate how the overall amount of vulnerabilities are associated with dependencies maintained by different communities: (a) well-known vs. less-known, and (b) enterprise vs. volunteer-based. Next, to answer RQ_2, we first calculated the ratio of reuse Rr and vulnerabilities density Dv as described in (1a) and (2b) below.

$$Rr = \frac{L_r}{L_n + L_r}\ (1a), \quad \text{and} \quad Dv = \frac{V_n + V_r}{L_n + L_r}\ (1b) \tag{1}$$

Then, similarly with RQ_1, we performed a linear regression analysis to evaluate the correlation between reuse and security vulnerabilities.

Regarding RQ_3, we collected the disclosed vulnerabilities of each projects' dependencies and performed a linear regression analysis between the number of dependencies and the number of disclosed vulnerabilities. Finally, to answer RQ_4, we collected data related to the use frequency of each dependency in our dataset and we performed a linear regression analysis on the use frequency and its number of vulnerabilities.

We note that this complete procedure is automated and available online together with all other scripts used in this study.[30]

4. Results

In this section, we present details about the obtained dataset and answers to the study's results research questions. Based on our unit of analysis, i.e., each project, and with regards to the variables we presented in Section 3 we obtained the descriptive statistics shown in Table 4.

4.1. RQ_1 - Relationship between vulnerabilities and size and reuse

To investigate how the factors of (1) source code size (SLOC), (2) number of classes, (3) number of dependencies, (4) reuse ratio are related to the number of potential vulnerabilities, we performed a multivariate ordinary linear regression with standardized beta coefficients on the aforementioned variables. The summary of this analysis is presented in Table 5.

The results show that the source code size (SLOC) is strongly correlated to the number of potential vulnerabilities. Interestingly, the number of classes appears to have no effect on the potential vulnerabilities. Although this may seem to contradict previous findings, we note that the majority of our dataset comprise smaller projects, which may encapsulate more functionality in single classes. Regarding the reuse factors, there is no statistical evidence that they are correlated with the number of potential vulnerabilities as both the number of dependencies and reuse ratio, have below-weak correlation.

In order to investigate how dependencies from well-known communities contribute to the total amount of potential vulnerabilities, we calculated the well-known ratio, which is the SLOC of well-known communities divided by the total SLOC of the reused code. We then analyzed the correlation between the number of potential vulnerabilities and the well-known ratio, performing a non-parametric test. The results (Kendall's $\tau \approx 0.05$, p-value ≈ 0.02) show that the number of potential vulnerabilities is not correlated to the well-known ratio.

> ***RQ***$_1$: The multivariate linear regression provides empirical evidence for the common belief that the number of potential vulnerabilities increases along with the source code size (SLOC). However, it shows no evidence that the number of potential vulnerabilities is correlated with reuse factors. Similarly, the number of potential vulnerabilities shows no correlation with the type (well-know and less well-known) of the reused code.

4.2. RQ_2 - Distribution of vulnerabilities in native and reused code

Fig. 3 depicts three boxplots, which illustrate the distribution of the vulnerability density (per 1000 lines of code) in the native, reused, and total code respectively. Comparing the vulnerability density in the native code (left boxplot) and the vulnerability density in the reused code (middle boxplot), we observe that the vulnerability density median is higher in native code. Also, there are more projects with higher vulnerability density in native code than in reused code.

[28] https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#malicious-code-vulnerability-malicious-code.

[29] https://find-sec-bugs.github.io/bugs.htm.

[30] https://github.com/AntonisGkortzis/Vulnerabilities-in-Reused-Software.

Table 4
Descriptive statistics.

Variable	Sum	Min	Max	Mean	Median	σ	Dataset
D	11 328	1	182	14 751	8	18 144	Dependencies
CVE	1 728	0	102	7 508	2	12	Projects
C	10 031 775	9	191 859	8 064	3 835	14 237	Projects
C_n	2 616 877	3	184 717	2 103	190	9 001	Projects
C_r	7 414 898	2	120 000	5 960	2 862	9 398	Projects
L	315 948 364	1 231	3 619 934	253 977	124 871	372 688	Projects
L_n	69 350 897	1 000	2 650 782	55 748	5 210	191 022	Projects
L_r	246 597 467	3	3 330 189	117 209	96 265	286 261	Projects
L_re	96 144 679	0	893 172	77 286	39 211	107 552	Projects
L_rne	150 810 685	0	2 531 786	121 230	50 470	208 833	Projects
L_rw	112 555 320	0	2 132 518	90 478	38 364	162 380	Projects
L_rnw	134 319 010	0	2 004 584	107 973	47 687	164 581	Projects
V	828 315	0	10 064	665	301	972	Projects
V_n	212 873	0	8 790	171	23	580	Projects
V_r	615 442	0	7 955	494	177	747	Projects
V_re	188 732	0	2 183	151	55	240	Projects
V_rne	426 710	0	5 772	343	95	591	Projects
V_rw	283 137	0	3 940	227	56	427	Projects
V_rnw	332 305	0	4 433	267	85	444	Projects
VC_n	150 241	0	6 670	120	19	413	Projects
VC_r	451 375	0	5 082	362	145	541	Projects
CVE_d	10 074	0	55	1	0	3	Dependencies
V_d	860 027	0	4 716	92	18	259	Dependencies
P	18 888	1	124	2	1	4	Dependencies
C_b	830 975	1	4 589	89	20	236	Dependencies

Table 5
Multivariate regression analysis for potential vulnerabilities.

Variable	Description	coeff	p-value
L	Number of lines of code	0.7882	0.000
C	Number of classes	0.0125	0.699
D	Number of dependencies	0.1225	0.000
R_r	Reuse ratio	0.0399	0.004

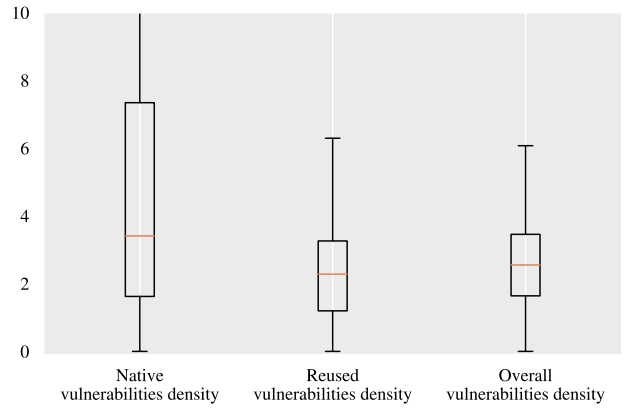

Fig. 3. Boxplots of vulnerability density in native code (left), reused code (center), and overall (right).

Furthermore, we notice that the overall density (right boxplot) is similar to the density in reused code compared to the native code. This is due to the fact that the size of reused code is considerably larger than native code, and the density is calculated after the vulnerabilities and corresponding code sizes re combined.

To investigate RQ_2 with regards to the correlation between the reuse ratio and the vulnerability density, we performed an ordinary linear regression with standardized coefficients. The result (statistic $= -0.0221$, p-value $= 0.436$) shows no evidence of a statistically significant relationship between these two variables. Further interpreting the results, one can suggest that there is no correlation between relationships AC and B as depicted in Fig. 1. The current dataset does not provide strong evidence to either confirm or deny whether projects with higher reuse ratio tend to have lower vulnerability density.

To further investigate the distribution of the potential vulnerabilities between the native and reused code we list the most occurring types of vulnerabilities as reported by the SpotBugs tool. In Table 6, we list the integrated top-10 recurrent types of potential vulnerabilities in native and reused code. For each type of potential vulnerability we calculated its density, as the number of detected potential vulnerabilities per 10 000 lines of code. In their description we include a reference number to the CWE software weaknesses types list.[31]

In Table 6 we observe that potential vulnerabilities that belong to the last two types (11 and 12) were detected in the reused code more often than in the native code with a difference of $> 80\%$. Similarly, for the types 6, 7 and 8 we observe a moderately greater frequency of detection in the reused code. On the contrary, for types 1, 3, 4 and 10, we observe a moderately greater frequency of appearances in the native code. Regarding types 2, 5 and 9 we observe similar frequency of appearance in both native and reused code.

> ***RQ***$_2$: The median vulnerability density is higher in native code. However, the results do not present any statistically significant correlation between the reuse ratio and the vulnerability density.

4.3. RQ_3 - Disclosed vulnerabilities in reused code

The first step to answer RQ_3 was to analyze all dependencies in our dataset with the OWASP Dependency-Check tool. The results showed that, at the time of the analysis, 2821 out of the 11 328 dependencies (24.9%) were reported to have at least

[31] https://cwe.mitre.org/data/definitions/699.html.

Table 6
Densities of most occurring types of vulnerabilities.

#	Vulnerability description	Densities in code		Difference
		Native	Reused	
1	Potential CRLF Injection for logs (CWE-93/117)	0.150	0.086	25.72%
2	Potential Path Traversal (file read) (CWE-22)	0.128	0.139	−7.96%
3	May expose internal representation by returning reference to mutable object	0.093	0.076	22.30%
4	May expose internal representation by incorporating reference to mutable object	0.092	0.071	29.53%
5	Information Exposure Through An Error Message (CWE-209/211)	0.060	0.064	−6.77%
6	Field is not final but should be	0.048	0.076	−36.62%
7	Predictable pseudo-random number generator (CWE-330)	0.039	0.048	−19.23%
8	URLConnection Server-Side Request Forgery (SSRF) and File Disclosure (CWE-73/918)	0.035	0.056	−37.98%
9	Field should be package protected	0.022	0.025	−9.19%
10	Format String Manipulation (CWE-134)	0.020	0.012	30.73%
11	Object de-serialization is used (CWE-502)	0.010	0.064	−84.13%
12	MD2, MD4 and MD5 are weak hash functions (CWE-327)	0.006	0.035	−82.08%

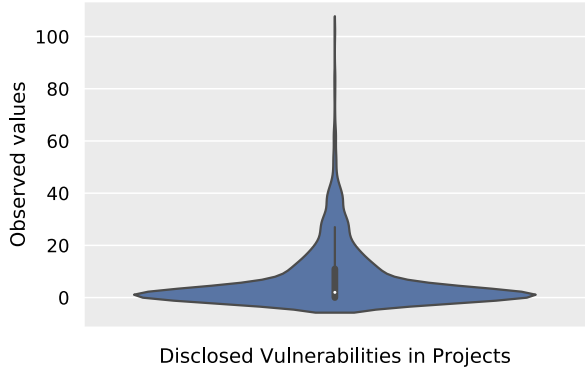

Fig. 4. Violin plots of number of disclosed vulnerabilities in projects.

one disclosed vulnerability. Consequently, mapping those findings to projects, we accounted for 65% of projects being vulnerable through their dependencies.

Fig. 4 presents the distribution of number of disclosed vulnerabilities in our dataset. It is clear that the majority of projects have very few disclosed vulnerabilities.

However, this is a concerning finding, because even one vulnerability can lead to a security breach with severe consequences. This is also interesting, because many open-source projects use outdated third-party dependencies with disclosed vulnerabilities; see references Kula et al. (2018) and Ponta et al. (2018). Disclosing vulnerability details along with the code patch fixing the security defect, motivates users to update to a newer, secure version. On the other hand, the disclosure also gives time and necessary details for malicious users to prepare attacks that target exploiting those specific defects, as happened in the Equifax incident.[32]

To investigate if the number of disclosed vulnerabilities in a project is correlated with the number of dependencies used in this project we performed a linear regression analysis between these two variables. The results, (statistic $=$ 0.6151, p-value $<$ 0.001) show that the number of disclosed vulnerabilities is strongly correlated to the number of dependencies. Similarly, with respect to potential vulnerabilities, we performed the linear regression analysis. The results (statistic $=$ 0.7730, p-value $<$ 0.001) are in line with the previous analysis and show a strong correlation between the number of dependencies and the potential vulnerabilities.

These findings suggest that a larger amount of dependencies used in a project may be correlated with a higher risk of bringing on board disclosed vulnerabilities. This can be described as the "effect of complex configuration", because developers select the direct dependencies in their projects but are unaware of the number of indirect dependencies brought in the project through other dependencies. Kula et al. (2018) interviewed several developers that affirmed being unaware of security risks in the code that they reuse. Additionally, Snyk reported that 78% of disclosed vulnerabilities are found in indirect dependencies.[33]

Table 7
Regression analysis for dependencies' use frequency.

Variable	Description	coeff	p-value
V	Potential vulnerabilities	−0.0326	0.083
CVE	Disclosed vulnerabilities	−0.0355	0.059

> ***RQ***$_3$: The analysis shows that 24.9% of the dependencies have at least one reported disclosed vulnerability. These vulnerable dependencies affect 65% of the projects analyzed. Additionally, the regression analysis showed that the numbers of both disclosed and potential vulnerabilities are strongly correlated to the number of dependencies included in a project.

4.4. RQ_4 - Dependencies' use frequency

The dataset analyzed in this RQ regards the 11 328 unique dependencies that appeared in our population of 1244 projects. For these dependencies, we collected the (1) disclosed vulnerabilities as reported and collected by the OWASP Dependency-Check tool, (2) the potential vulnerabilities as reported by the SpotBugs tool, (3) if they are maintained by a well-known community or an enterprise organization, and (4) the total number of contributors of the projects that reuse these dependencies.

To investigate if the use frequency is correlated with the number of disclosed and potential vulnerabilities we performed a linear regression analysis on these variables. The results are presented in Table 7 and show that there is no statistically significant evidence to correlate the number of disclosed and potential vulnerabilities with their use frequency. There is no evidence that more popular dependencies have more disclosed vulnerabilities reports or have a lower number of potential vulnerabilities.

To further investigate Linus' Law, we estimate the total number of contributors in projects reusing a dependency as the number of eyeballs that might detect a vulnerability. We tested how the total amount of contributors is associated with the potential

32 https://www.wired.com/story/equifax-breach-no-excuse/.

33 https://snyk.io/blog/78-of-vulnerabilities-are-found-in-indirect-dependencies-making-remediation-complex/.

Table 8
Regression analysis for dependencies' use frequency.

Dataset	Potential vulnerabilities		Disclosed vulnerabilities	
	τ	p-value	τ	p-value
All dependencies	−0.01	0.28	−0.10	0.00
Well-known communities	−0.02	0.06	−0.13	0.00
Enterprise organizations	−0.03	0.02	−0.14	0.00
Both well-known and enterprise	−0.05	0.00	−0.22	0.00

and the disclosed number of vulnerabilities in the dependencies by calculating Kendall's non-parametric correlation. In Table 8, we report the results (τ and p-value) of the Kendall correlation on (1) the overall dependencies dataset population, (2) dependencies from well-known communities, (3) dependencies from enterprise organizations, and (4) from dependencies that are both well-known and from an enterprise organization.

The τ and p-values of the executed correlation tests show no strong evidence that the type of the community (i.e., well-known, enterprise) is correlated to the number of its potential vulnerabilities. While for the actual vulnerabilities the statistical evidence is poor for each individual type community, we observe a very weak correlation for dependencies that belong both in a well-known community and an enterprise organization.

> ***RQ***$_4$: The statistical analysis showed that there is no evidence that correlates the use frequency of a dependency with its security aspect. Furthermore, the results showed only a very weak correlation between the number of eyeballs associated with a dependency and its disclosed vulnerabilities: only for dependencies that belong in both well-known communities and enterprise organizations.

5. Discussion

In this section, we revisit and explain the findings presented in the previous section, comparing them against related work where applicable. We also elaborate on a point that stems from the discussion, namely, the special case of enterprise open-source projects. Finally, we elaborate on the implications of these observations to both researchers and practitioners.

5.1. Interpretation of the results

We found that the amount of potential vulnerabilities of a project is strongly correlated to its source code size (SLOC). This finding is in line with what Chowdhury and Zulkernine (2010) observed in their study on five consequent versions of the Mozilla Firefox web browser. Similarly, Mitropoulos et al. (2014) found a positive correlation between project size and the amount of vulnerabilities, which also aligns with our findings. Furthermore, our findings agree with those of Yu and Mishra (2013) in that the more a project evolves and adds functionality the more it accumulates defects. Both findings support Lehman's Seventh Law, which states that the quality of a software product decreases with time unless it is restructured (Lehman, 1996; Herraiz et al., 2013). If we assume that reused code stands for code that would otherwise have to be written from scratch, vulnerabilities will ultimately arise either from native or reused code. Depending on the security expertise of the development team, one will have, then, to choose between two strategies. On the one hand, a wiser strategy may be to avoid reuse for developing components with strict security requirements and manage the vulnerability threat internally. On the other hand, it may be desirable to reduce vulnerability risks by reusing as much as possible.

To discuss this subject further, we focus on the results of RQ_2, which suggest the presence of a higher vulnerability density in native code, i.e., higher count of vulnerabilities per SLOC than reused code. However, our results also suggest that the distribution of vulnerabilities between native and reused code is not homogeneous among the studied projects. Perhaps, projects with similar vulnerability density may have features in common. For example, Mohagheghi et al. (2004), who performed a comparable study but in an industrial setting, found a lower defect density (which includes security vulnerabilities) in reused code when compared to native code. In summary, for the time being these findings place a heavier weight for the decision making on the development team, which has to verify the maturity of reused code and balance it with in-house expertise in writing secure code.

Regarding the relatively larger amount of reused code, we note that this is understandable due to the nature of our dataset, i.e., with multiple medium-size projects which is observable in Table 4. On one hand, dependencies (e.g., libraries) have a larger impact on the project size as they may introduce a cascade of included dependencies. On the other hand, the evolution of the project may not depend as much on additional reuse, which decreases the reuse ratio.

Turning to disclosed vulnerabilities, our analysis showed a significantly larger percentage of affected dependencies (24.9%) compared to that reported in related work. In particular, Pashchenko et al. (2018) found that 12.4% of their studied dependencies were vulnerable. This difference can be explained based on the difference between the two datasets. In our dataset, we study dependencies used in open-source projects while Pashchenko et al. (2018) study dependencies in proprietary SAP projects. A possible explanation is that enterprise projects are more selective on the use of third-party code, and they tend to update their versions more often. To shed further light in this matter, in the next section, we explore the differences between the reuse of third-part code on volunteer-based and enterprise open-source communities.

With respect to the use frequency of dependencies, we were not able to clearly identify a correlation with the number of potential or disclosed vulnerabilities. Consequently, we could not establish from our data that Linus' law (Raymond, 1999, p. 30) does in fact hold. It seems that users of third-party code are not necessarily contributors that can catch and fix security vulnerabilities and thus support the aforementioned law. However, absence of evidence is not evidence of absence. A related line that might be worth pursuing, would be to investigate if a project's popularity is associated with how fast security defects are detected and fixed. In that direction, van Liere (2009) studied the Firefox community and found that a large community of bug reporters can be associated with quicker bug fixing, while the addition of new software developers incurs fixing delays. However, with a similar scope, Bissyandé et al. (2013) performed a large scale study on 20 000 GitHub projects and found that the correlation between bug fixing time and the amount of issue reporters is negligible (0.16). These partially contradicting results

Table 9
Descriptive statistics for enterprise and volunteer-based projects.

Variable	Enterprise ($N = 252$)		Volunteer-based ($N = 992$)	
	Mean	σ	Mean	σ
Contributors	42	94	21	42
D	18	24	14	16
CVE	8	12	7	12
L	284 564	442 846	246 181	352 450
L_n	44 501	175 598	58 616	194 783
L_r	240 062	383 684	187 564	254 663
V	725	1 052	650	950
V_n	123	374	183	621
V_r	602	884	467	884

show that more research in this subject is paramount and should also consider other indicators of project popularity (e.g., number of downloads, forks, and positive reviews).

Finally, we investigated to what extent the amount of potential vulnerabilities is correlated with the amount of disclosed vulnerabilities. A linear regression analysis showed that there is a medium correlation between the two factors. Despite the fact that disclosed vulnerabilities cannot be tracked to code level in this dataset, the results show that a high number of potential vulnerabilities is an indicator of higher risk of exploitable vulnerabilities.

5.2. Comparison between enterprise and volunteer-based projects

In the previous section, we noted that different practices between community types could reflect on a more selective process to manage reuse. In particular, one may wonder how enterprise open-source projects compare to volunteer-based ones. As our dataset encompasses both types of projects, it is feasible to perform such comparison. For that, we used the same process to identify dependencies belonging to enterprise organizations (see Section 3.4, Step 4) to also identify enterprise projects. The classification of each project (into 'enterprise' or 'volunteer-based') is also available in the main dataset. We used this extension of the dataset to revisit the research questions in which we perform project-level analyses (i.e., RQ_1–RQ_3). In Table 9, we present a summary of the descriptive statistics to briefly compare the two sub-populations.

Regarding RQ_1, we analyzed the relationship between the number of vulnerabilities and size and reuse for each of the two groups of projects and did not find a significant difference. Regarding RQ_2, we examined the distribution of vulnerabilities between native and reused code and, although we noticed a lower density of vulnerabilities in native code on enterprise projects, we also found it not to be statistically significant. Finally, regarding RQ_3, we first looked into the number of projects affected by disclosed vulnerabilities and found the percentage to be similar to the overall population (enterprise: 64.3%; volunteer-based: 65.3%). However, we estimated the association between disclosed vulnerabilities and the number of dependencies for both groups and noticed that enterprise projects are less likely to suffer from them, compared to volunteer-based projects based on the linear regression analysis ($coeff = 0.5 < coeff = 0.6825$).

In summary, our dataset allows us to further speculate that enterprise projects may indeed be less likely to be suffer from vulnerabilities due to a higher quality of native code and a more careful selection of dependencies. However, we cannot provide strong evidence to support this based on our dataset alone, and more studies are necessary to investigate a larger population and additional factors.

Table 10
Manual inspection of SpotBugs' findings.

Project	True positive	False positive	Undecided
`spotify/netty4-zmtp`	1	–	1
`gturri/aXMLRPC`	3	1	1
`twitter/whiskey`	2	2	2

5.3. Inspection of SpotBugs' findings

To acquire more insights over SpotBugs' findings, we selected three projects from our dataset and investigated if the reported potential vulnerabilities are exploitable. This process consists of the following steps:

1. retrieve SpotBugs' XML report for a project,
2. dynamically analyze the project by executing all test cases provided by the developers,
3. manually inspect the source code flagged as vulnerable.

For this process we selected three projects, namely `spotify/netty4-zmtp`, `gturri/aXMLRPC` and `twitter/whiskey` to manually investigate the validity of thirteen potential vulnerabilities reported by SpotBugs. All three projects contained potential vulnerabilities detected by SpotBugs, as well as unit tests that challenge the functionality of the application.

In Table 10, we present our findings from the manual inspection of each potential vulnerability of the three selected projects. We mark as *True positive* the bugs in SpotBugs' report that can lead to actual security vulnerabilities based on the description provided by SpotBugs[34] and its plugin, FindSecBugs.[35] In the *False positive* column, we report SpotBugs' findings that do not constitute a security vulnerability. Finally, as *Undecided*, we report those that partially match the vulnerability description.

We dynamically analyzed the source code of the three projects by executing the provided test cases and applying the Java Code Coverage Library[36] on them. The results showed that the lines containing a potential vulnerability flagged as *True positive* were covered by one ore more test cases. This finding suggests that more extensive testing related to the security requirements is required.

Furthermore, other aspects of dynamic analysis could be used to supplement static analysis in order to test the application's behavior more extensively. For example, *fuzz testing* is a prominent dynamic analysis technique for discovering software bugs and security vulnerabilities. The success of this technique is reflected by the hundreds of bugs detected in popular applications by the fuzzer AFL.[37] Fuzz testing iteratively and randomly generates inputs with which it tests a target program. However, this

[34] https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html.
[35] https://find-sec-bugs.github.io/bugs.htm.
[36] https://www.eclemma.org/jacoco/.
[37] AFL2018.AmericanFuzzingLop(AFL).https://lcamtuf.coredump.cx/afl/.

technique comes with a great computational cost. Klees et al. (2018) performed an evaluation of 32 studies related to fuzz testing and reported that all suffered by one or more violations of the proposed proper methodology for performing this technique. This shows that the effectiveness of fuzz testing depends on the prior-execution configuration of the tester based on the context of each application. This makes fuzz testing difficult to apply on a large-scale analysis such as the one reported in this manuscript.

5.4. Implications for researchers and practitioners

Security assessment and risk analysis are common practices among software developers and researchers. With the prevalence of agile software development and the automations that continuous deployment strategy offers, security assessment can be performed before every version release of a software system. In our study, we provided evidence that source code size has a negative impact on the security of a software system. Additionally, we showed that a higher number of dependencies tend to be associated with more security risks in open-source software systems. To mitigate this risk more strict security assessment methods should be followed. For example, automated build processes could integrate vulnerability detection tools, e.g., SpotBugs, Snyk and OWASP Dependency Check. Such methods can provide valuable information regarding the security status of the native code and the risks introduced through dependencies.

Software developers can consult the dataset and gain insight related to the security vulnerabilities of 1244 open-source projects. Practitioners can use this information to perform risk analysis and prioritize bug-fixing activities related to security defects. Moreover, practitioners can employ the provided automation scripts to perform a similar analyses on their own code base.

The provided dataset can be used by researchers to explore additional research questions based on other characteristics of the projects, e.g., clustering of projects based on one or more of the available variables. Additionally, by taking advantage of SpotBugs plurality of findings, researchers can investigate other software quality attributes (e.g., correctness and performance). To examine this aspect, researchers can modify the provided scripts to enrich SpotBugs' report with information related to these attributes. Our scripts and guidelines are available for researchers to create their own dataset or extend the one analyzed in this study.

6. Threats to validity

In this section, we discuss the three types of validity that are applicable in this study: (1) the construct validity; (2) the reliability; and (3) the external validity. We exclude internal validity since our study does not examine causality. Construct validity examines the relationship between the study's observable object or phenomenon and its research questions. Reliability examines if the study can be replicated and produce the same results. Finally, external validity examines potential threats to generalizing the results of this study to other cases.

Regrading construct validity, we can argue that static analysis can only detect potential security defects and not actually exploitable vulnerabilities. However, as we saw, these reports are correlated with the existence of exploitable vulnerabilities. Furthermore, vulnerabilities reported by static analyzers in the reused code may not be exploitable since some vulnerable elements may never be executed by the native code, and thus be irrelevant. Moreover, our study is limited to identifying only black-box reuse as defined by Heinemann et al. (2011), which requires developers to include a binary version of the dependency. White-box reuse is the integration of the dependency code into the native code. White-box requires clone code-detection and, thus, is out of the scope of this study. Finally, we selected the amount of GitHub stars for measuring the popularity of our projects. There are other criteria, such as watchers and forks, that may render different results.

Concerning reliability, we put our best effort to make this study easy to replicate. The source code, along with the guidelines to execute it, are available on GitHub.[38] To reproduce the same results, researchers should revert the Git repositories of the locally downloaded projects to the date of this study (July 20th 2019). To mitigate reliability risks, two developers were involved in the development of the scripts and all authors reviewed the analysis process.

Finally, concerning external validity, we identified three potential risks. Firstly, the project selection was limited to one programming language (Java), and thus generalization of our findings to other languages requires further investigation. Secondly, the selection of our projects represents only a proportion of the available open-source Java projects on Github and thus, generalization of our findings to open-source Java projects hosted in Github or other web vcs requires further investigation. Finally, despite the fact that Maven provided us a straight-forward way of building the projects and easy access to the dependencies, it also limited our dataset. Almost 34% of the initial project selection (3500) failed to build with Maven or was partially built, and was therefore excluded from the analysis.

7. Conclusion

Software reuse is a widely adopted practice that still raises several concerns when it comes to security risks. There are good arguments to both reuse and not reuse source code, especially with regards to open-source software. In this context, we conducted a multiple-case study to explore and discuss the relationship between software reuse and the amount of security vulnerabilities in open-source projects. For that, we followed up on a previous study (Gkortzis et al., 2019) and further examined the distribution of potential vulnerabilities among the code created by a development team (i.e., native code) and code reused from third-party dependencies. Moreover, we investigated how information about disclosed vulnerabilities from public databases triangulate with previous results especially on studying the association between the ratio of reuse and the density of vulnerabilities.

For that, we looked into the most popular Java projects in the GitHub Activity Data database and constructed a dataset with 1244 projects, containing information regarding the size of both native and reused code, as well as vulnerability information obtained from the static analyzer SpotBugs and the OWASP Dependency-Check tool. Among the results, we observed that larger projects are related with an increased amount of potential vulnerabilities in both native and reused code. Furthermore, native code appears to have a higher vulnerability density. However, our analysis showed no strong evidence that native code contributes to more vulnerabilities than reused code in a project. Additionally, the results suggest that the number of dependencies in a project is correlated to its number of vulnerabilities.

In light of the theoretical and empirical designs, and the observed results, we envisage several opportunities of future work. On the one hand, it is desirable to investigate other programming languages, automated build systems and package managers (e.g., Ant, Gradle, npm and pip). Such data could be used to further enrich the provided dataset, and allow for confirmatory and replication studies. Future studies could explore more in-depth research questions related to, for example, features that

[38] https://github.com/AntonisGkortzis/Vulnerabilities-in-Reused-Software.

could cluster similar projects in terms of size, also including a qualitative analysis to explain each cluster. On the other hand, the toolkit reported in paper could be implemented as a workbench that could benefit practitioners and researchers alike by fostering in-house analyses or future studies.

CRediT authorship contribution statement

Antonios Gkortzis: Conceptualization, Methodology, Software, Data curation, Writing - original draft, Visualization, Investigation, Validation. **Daniel Feitosa:** Conceptualization, Methodology, Software, Writing - original draft, Visualization, Investigation, Writing - review & editing. **Diomidis Spinellis:** Conceptualization, Methodology, Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2020.110653.

References

Abdalkareem, R., Shihab, E., Rilling, J., 2017. On code reuse from StackOverflow: An exploratory study on Android apps. Inf. Softw. Technol. 88, 148–158. http://dx.doi.org/10.1016/j.infsof.2017.04.005.

Ayewah, N., Pugh, W., 2010. The google findbugs fixit. In: Proc. 19th Int. Symp. on Software Testing and Analysis (ISSTA '10). ACM, Trento, Italy, pp. 241–252. http://dx.doi.org/10.1145/1831708.1831738.

Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y., 2007. Evaluating static analysis defect warnings on production software. In: Proc. 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07). ACM Press, San Diego, California, USA, pp. 1–8. http://dx.doi.org/10.1145/1251535.1251536.

Bissyandé, T.F., Lo, D., Jiang, L., Réveillère, L., Klein, J., Traon, Y.L., 2013. Got issues? Who cares about it? a large scale investigation of issue trackers from GitHub. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 188–197. http://dx.doi.org/10.1109/ISSRE.2013.6698918, ISSN: 1071-9458, 2332-6549.

Chowdhury, I., Zulkernine, M., 2010. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: Proceedings of the 2010 ACM Symposium on Applied Computing. In: SAC '10, ACM, New York, NY, USA, pp. 1963–1969. http://dx.doi.org/10.1145/1774088.1774504, event-place: Sierre, Switzerland.

Decan, A., Mens, T., Constantinou, E., 2018. On the impact of security vulnerabilities in the npm package dependency network. In: Proceedings of the 15th International Conference on Mining Software Repositories. In: MSR '18, Association for Computing Machinery, Gothenburg, Sweden, pp. 181–191. http://dx.doi.org/10.1145/3196398.3196401.

Feitosa, D., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., Nakagawa, E., 2018. What can violations of good practices tell about the relationship between gof patterns and run-time quality attributes? Inf. Softw. Technol. http://dx.doi.org/10.1016/j.infsof.2018.07.014.

Feitosa, D., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y., 2015. Investigating quality trade-offs in open source critical embedded systems. In: Proc. 11th Int. ACM SIGSOFT Conf. the Quality of Software Architectures (QoSA '15). ACM, Montreal, QC, Canada, pp. 113–122. http://dx.doi.org/10.1145/2737182.2737190.

Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., Fahl, S., 2017. Stack overflow considered harmful? The impact of copy paste on android application security. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 121–136. http://dx.doi.org/10.1109/SP.2017.31, iSSN: 2375-1207.

Gkortzis, A., Feitosa, D., Spinellis, D., 2019. A double-edged sword? software reuse and potential security vulnerabilities. In: Peng, X., Ampatzoglou, A., Bhowmik, T. (Eds.), Reuse in the Big Data Era. Springer International Publishing, Cham, pp. 187–203.

Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B., Irlbeck, M., 2011. On the extent and nature of software reuse in open source java projects. In: Proc. 12th Int. Conf. Top Productivity Through Software Reuse (ICSR'11). Springer Berlin Heidelberg, Pohang, South Korea, pp. 207–222.

Herraiz, I., Rodriguez, D., Robles, G., Gonzalez-Barahona, J., 2013. The evolution of the laws of software evolution: A discussion based on a systematic literature review. ACM Comput. Surv. 46 (2), http://dx.doi.org/10.1145/2543581.2543595.

Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. ACM SIGPLAN Not. 39 (12), 92–106. http://dx.doi.org/10.1145/1052883.1052895.

Khalid, H., Nagappan, M., Hassan, A.E., 2016. Examining the relationship between FindBugs warnings and app ratings. IEEE Software 33 (4), 34–39. http://dx.doi.org/10.1109/MS.2015.29.

Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M., 2018. Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. In: CCS '18, Association for Computing Machinery, New York, NY, USA, pp. 2123–2138. http://dx.doi.org/10.1145/3243734.3243804.

Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K., 2018. Do developers update their library dependencies? Empir. Softw. Eng. 23 (1), 384–417. http://dx.doi.org/10.1007/s10664-017-9521-5.

Kulenovic, M., Donko, D., 2014. A survey of static code analysis methods for security vulnerabilities detection. In: Proc. 37th Int. Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO '14). pp. 1381–1386. http://dx.doi.org/10.1109/MIPRO.2014.6859783.

Lehman, M.M., 1996. Laws of software evolution revisited. In: Proceedings of the 5th European Workshop on Software Process Technology. In: EWSPT '96, Springer-Verlag, Berlin, Heidelberg, pp. 108–124.

van Liere, D.W., 2009. How shallow is a bug? why open source communities shorten the repair time of software defects. In: ICIS 2009 Proceedings. p. 195.

Meneely, A., Williams, L., Secure open source collaboration: An empirical study of linus' law. In: Proc. 16th ACM Conf. Computer and Communications Security. In: CCS '09, ACM, pp. 453–462. http://dx.doi.org/10.1145/1653662.1653717.

Mitropoulos, D., Karakoidas, V., Louridas, P., Gousios, G., Spinellis, D., 2014. The bug catalog of the Maven ecosystem. In: Proc. 11th Working Conf. Mining Software Repositories (MSR '14). ACM, Hyderabad, India, pp. 372–375. http://dx.doi.org/10.1145/2597073.2597123.

Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H., 2004. An empirical study of software reuse vs. defect-density and stability. In: Proc. 26th Int. Conf. Software Engineering (ICSE '04). IEEE Computer Society, Washington, DC, USA, pp. 282–292, URL http://dl.acm.org/citation.cfm?id=998675.999433.

Neuhaus, S., Zimmermann, T., The beauty and the beast: vulnerabilities in red hat's packages. In: Proc. 2009 USENIX Annual Technical Conf. (USENIX 2009).

Pashchenko, I., Plate, H., Ponta, S.E., Sabetta, A., Massacci, F., 2018. Vulnerable open source dependencies: Counting those that matter. In: Proc. 12th ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM '18). ACM, Oulu, Finland, pp. 42:1–42:10. http://dx.doi.org/10.1145/3239235.3268920.

Pham, N.H., Nguyen, T.T., Nguyen, H.A., Wang, X., Nguyen, A.T., Nguyen, T.N., 2010. Detecting recurring and similar software Vulnerabilities. In: Proc. 32nd ACM/IEEE Int. Conf. Software Engineering (ICSE '10). ACM, Cape Town, South Africa, pp. 227–230. http://dx.doi.org/10.1145/1810295.1810336.

Ponta, S.E., Plate, H., Sabetta, A., 2018. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In: Proc. 34th IEEE Int. Conf.on Software Maintenance and Evolution (ICSME '18). http://dx.doi.org/10.1109/ICSME.2018.00054.

Raymond, E., 1999. The cathedral and the bazaar. Knowl. Technol. Policy 12 (3), 23–49. http://dx.doi.org/10.1007/s12130-999-1026-0.

Runeson, P., Host, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. Wiley Blackwell.

Shin, Y., Meneely, A., Williams, L., Osborne, J.A., Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, 37 (6), 772–787. http://dx.doi.org/10.1109/TSE.2010.81.

van Solingen, R., Basili, V., Caldiera, G., Rombach, H.D., 2002. Goal question Metric (GQM) approach. In: Encyclopedia of Software Engineering. John Wiley & Sons, Inc., Hoboken, NJ, USA, pp. 528–532. http://dx.doi.org/10.1002/0471028959.sof142.

Spinellis, D., Kotti, Z., Kravvaritis, K., Theodorou, G., Louridas, P., 2020. A dataset of enterprise-driven open source software. In: 17th International Conference on Mining Software Repositories. In: MSR '20, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3379597.3387495.

Tomassi, D.A., 2018. Bugs in the wild: Examining the effectiveness of static analyzers at finding real-world bugs. In: Proc. 2018 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE '18). ACM, Lake Buena Vista, FL, USA, pp. 980–982. http://dx.doi.org/10.1145/3236024.3275439.

Tripathi, A.K., Gupta, A., 2014. A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for Java programs. In: Proc. 18th Int. Conf. Evaluation and Assessment in Software Engineering (EASE '14). ACM, London, UK, pp. 23:1–23:4. http://dx.doi.org/10.1145/2601248.2601288.

Wang, J., Carroll, J.M., 2011. Behind Linus's law: A preliminary analysis of open source software peer review practices in Mozilla and Python. In: CTS 2011: International Conference on Collaboration Technologies and Systems. IEEE, pp. 117–124.

Yu, L., Mishra, A., 2013. An empirical study of lehman's law on software quality evolution. Int. J. Softw. Inf. 7, 469–481.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P., Vouk, M.A., 2006. On the value of static analysis for fault detection in software. Softw. Eng. IEEE Trans. 32 (4), 240–253. http://dx.doi.org/10.1109/TSE.2006.38.

Zimmermann, M., Staicu, C., Tenny, C., Pradel, M., 2019. Small world with high risks: A study of security threats in the npm ecosystem. CoRR arXiv:1902.09217. URL http://arxiv.org/abs/1902.09217.

Antonis Gkortzis is a Ph.D. Student at the Athens University of Economics and Business (Greece) in the Software Engineering and Security (SENSE) group. He holds an MSc degree in Software Engineering from University of Groningen (the Netherlands) and a BSc degree in Information Technology from the Technological Institute of Thessaloniki (Greece). His research interests include security, object-oriented design, maintainability, and software quality assessment.

Dr. Daniel Feitosa is an Assistant Professor in the Faculty Campus Fryslân and the Chief Data Scientist at the Data Research Centre of the University of Groningen. He is also an associated researcher in the group of Software Engineering and Architecture of the University of Groningen. He holds a BSc degree (2010) and MSc (2013) in Computer Science from the University of São Paulo, Brazil, and was awarded his Ph.D. degree (2019) in Software Engineering by the University of Groningen. He currently has 20 publications among journal, conference papers and book chapters. His main research interests are in software architecture, software patterns and data analytics.

Diomidis Spinellis is a Professor in the Department of Management Science and Technology at the Athens University of Economics and Business, Greece and director of the University's Business Analytics Laboratory. He is the author of two award-winning books, *Code Reading* and *Code Quality*: *The Open Source Perspective*. His most recent book is *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. He has contributed code that ships with Apple's macos and bsd Unix, and is the developer of *CScout, UMLGraph, dgsh*, and other open-source software packages, libraries, and tools. He served as the Editor in Chief for *IEEE Software* over the period 2015–2018.