A Double-Edged Sword? Software Reuse and Potential Security Vulnerabilities

Gkortzis, Antonios; Feitosa, Daniel; Spinellis, Diomidis

Published in:
Proceedings of the 18th International Conference on Software and Systems Reuse (ICSR '19)

DOI:
10.1007/978-3-030-22888-0_13

Publication date:
2019

A Double-Edged Sword? Software Reuse and Potential Security Vulnerabilities

Antonios Gkortzis[1(✉)], Daniel Feitosa[2], and Diomidis Spinellis[1]

[1] Department of Management Science and Technology,
Athens University of Economics and Business, Athens, Greece
{antoniosgkortzis,dds}@aueb.gr
[2] Data Research Centre, University of Groningen, Groningen, The Netherlands
d.feitosa@rug.nl

Abstract. Reuse is a common and often-advocated software development practice. Significant efforts have been invested into facilitating it, leading to advancements such as software forges, package managers, and the widespread integration of open source components into proprietary software systems. Reused software can make a system more secure through its maturity and extended vetting, or increase its vulnerabilities through a larger attack surface or insecure coding practices. To shed more light on this issue, we investigate the relationship between software reuse and potential security vulnerabilities, as assessed through static analysis. We empirically investigated 301 open source projects in a holistic multiple-case methods study. In particular, we examined the distribution of potential vulnerabilities between the native code created by a project's development team and external code reused through dependencies, as well as the correlation between the ratio of reuse and the density of vulnerabilities. The results suggest that the amount of potential vulnerabilities in both native and reused code increases with larger project sizes. We also found a weak-to-moderate correlation between a higher reuse ratio and a lower density of vulnerabilities. Based on these findings it appears that code reuse is neither a frightening werewolf introducing an excessive number of vulnerabilities nor a silver bullet for avoiding them.

Keywords: Software reuse · Security vulnerabilities · Case study

1 Introduction

Code reuse is a widely advocated and adopted practice in software development. A Linux distribution is a great example of software reuse, bundling together several packages to provide the functionality of a modern operating system. In a similar manner, the dominant mobile operating system, Android,[1] is based on a customized Linux kernel and bundles additional open source packages. To

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

X. Peng et al. (Eds.): ICSR 2019, LNCS 11602, pp. 187–203, 2019.
https://doi.org/10.1007/978-3-030-22888-0_13

develop user applications, the Android platform provides a set of more than 3 million Java libraries from the Maven repository.[2]

Nevertheless, similarly to any other design decision, code reuse has limitations. A prominent side-effect of code reuse is the existence of serious potential security risks. Kula et al. [12] analyzed 4 659 open source software systems and showed that more than 80% of them used outdated external libraries and dependencies, while 69% of the developers they interviewed were unaware of any security risks in their reused code.

As a concrete example, Heartbleed[3] was a severe security vulnerability in the OpenSSL cryptographic software library that allowed any user on the Internet to read arbitrary memory contents. Through this vulnerable version of the library, a malicious user could retrieve secret keys that protected communications, usernames and passwords, personal emails, documents and messages. The bug was detected two years after its introduction in the code. It affected the web servers that were powering 66% of the active web sites at that time [1]. Another, more recent, example is the Equifax incident [2], in which hackers exploited a known vulnerability in a third-party Java library that Equifax knowingly used, 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[4] service aiming to increase users' awareness and mitigate the potential security risks. Similarly, any Linux or BSD system by default notifies users for available security updates in vulnerable versions of installed packages and system libraries.

Despite the existence of well-known security mishaps due to software reuse, to the best of our knowledge there is a lack of large-scale studies that investigate how security vulnerabilities are associated with code reuse in software systems. This paper aims to contribute towards this direction by analyzing a large set of open source software systems and comparing the levels of vulnerabilities between the native application source code written by the software development team and external source code introduced through dependencies on third-party libraries. To achieve this, we collected a set of 301 Java projects and compared the native and reused parts of the code with regards to potential security vulnerabilities, which were detected based on static analysis.

The analysis of the produced data revealed a weak-to-moderate inverse correlation between the code reuse ratio and the vulnerability density in open source software systems. This means that software systems with higher reuse ratio tend to have fewer potential vulnerabilities compared to projects where native code is dominant. The main contribution of our work is that, although we observed that the amount of potential vulnerabilities in both native and reused code increases with larger project sizes, a higher reuse ratio is associated with a lower vulnerability density. Additionally, we contribute: (a) the construction process of

[2] https://mvnrepository.com/repos/central.
[3] https://nvd.nist.gov/vuln/detail/CVE-2014-0160.
[4] https://help.github.com/articles/about-security-alerts-for-vulnerable-dependencies/.

a dataset that correlates the software reuse ratio of open source Java projects with their potential security vulnerabilities, (b) the aforementioned dataset per se, and (c) a statistical analysis of this dataset. The source code to reproduce the process is available on GitHub[5] and the dataset on Zenodo.[6]

The remainder of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the approach of our study regarding the dataset construction and the analysis tools. Section 4 presents our findings, which we further discuss in Sect. 5. Section 6 presents the limitations of our study and Sect. 7 our conclusions.

2 Related Work

In this section, we present related work. We note that since we could not identify studies that are directly related to ours, we broadened the scope of this section to describe efforts dealing with software defects and vulnerabilities in reused code.

Pashchenko et al. [19] conducted a study on the `SAP` software ecosystem, investigating how much of the reused code in `SAP` is affected by known vulnerabilities. The authors, similarly to our study, analyzed the top 200 open source Maven systems that `SAP` is reusing. Thus, their study is not affected by false positives. However, the nonexistence of known vulnerabilities does not guarantee the absence of any other undetected vulnerabilities. The authors reported that 13% of the direct and transitive libraries that were reused were affected by at least one known vulnerability. In their analysis they excluded none-deployed dependencies (e.g., test dependencies). Regarding vulnerable dependencies Neuhaus et al. [18] investigated the Red Hat Linux (RHEL) distribution and provided empirical evidence that certain packages are correlated to system vulnerabilities.

Shin et al. [23] studied that three software metrics, i.e., complexity, code churn and developer activity, can be used in order to create a prediction model for potentially vulnerable code chunks in RHEL kernel and in Mozilla Firefox. Similarly, Meneely et al. [14] investigated the RHEL kernel and provided empirical evidence that show files modified by more than nine developers or files maintained by independent developer groups are more likely to have vulnerable code compared to files developed by the main core or smaller groups.

Mohagheghi et al. [16] studied historical data of software defects for 12 consequent releases of a large-scale telecom system developed by Ericsson. Their goal was to investigate the impact of reuse on the defect density (defined as defects per lines of code) and the stability of the system (defined as the degree of modification). Their findings showed that reused code components had a lower defect density compared to non-reused ones. Moreover, reused components had a higher stability compared to the non-reused ones.

Additionally, Mitropoulos et al. [15] used FindBugs to statically examine the Maven ecosystem and presented a dataset of the bugs (including security bugs) of more than 17 000 libraries (155 000 considering all their versions). Their

[5] https://github.com/AntonisGkortzis/Vulnerabilities-in-Reused-Software.
[6] http://doi.org/10.5281/zenodo.2566055.

dataset can be used to analyze the risk of using outdated libraries that exist in the Maven Central repository. Although, this 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.

Concerning the detection of vulnerable reused code, Pham et al. [20] introduced SecureSync, an automatic approach that analyzes existing vulnerabilities, in open source systems and creates models in order to detect suspicious patterns in similar 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 this area. Ponta et al. [21] presented their approach to identify exploitable vulnerabilities based on function call graphs. Recently they made their tool[7] 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 association between code reuse and vulnerabilities, as obtained by means of static analysis, in multiple open source systems.

Table 1. Comparison against related work

Study	Context	Focus on security	Number of projects	Source of vulnerabilities	Relate security to reuse
[19]	Open source	Yes	200	Manual analysis	Yes
[16]	Proprietary	No	1	Defect reporting system	Yes
[15]	Open source	Yes	17 505	Static analysis	No
[20]	Open source	Yes	119	Static analysis and clone detection	Yes
[21]	Open source	Yes	500	Static and dynamic analysis	No
[14]	Open Source	Yes	1	Vulnerability reporting system	No
[23]	Open Source	Yes	2	Vulnerability reporting system	Partially
[18]	Open Source	Yes	1	Vulnerability reporting system	Yes
Ours	Open source	Yes	301	Static analysis	Yes

3 Study Design

In this section, we present the protocol of our case study, which was designed according to the guidelines of Runeson et al. [22], and reported based on the Linear Analytic Structure [22].

3.1 Objective and Research Questions

The goal of the study was formulated according to the Goal-Question-Metric (GQM) approach [24], and is described as follows: "***analyze*** *native and reused*

[7] https://sap.github.io/vulnerabilityassessmenttool/.

code, ***for the purpose of*** *evaluation*, ***with respect to*** *the differences in the estimated levels of security*, ***from the point of view of*** *software developers*, ***in the context of*** *open-source software.*" *To fulfill this objective, we have set two research questions (RQs), as follows*:

RQ$_1$: What factors can group projects with regards to security vulnerabilities?

RQ$_1$ aims at acquiring an overview of open-source projects with regards to the security vulnerabilities identified through static analysis. This overview allows the provision of demographics for the dataset and the identification of groups of projects with similar features. This information is also useful to support decision-making in software development activities related to reuse, and to drive future research efforts.

RQ$_2$: How is software reuse associated with security vulnerabilities?
- **RQ$_{2.1}$:** How does native code contribute to the overall amount of vulnerabilities?
- **RQ$_{2.2}$:** How does reused code contribute to the overall amount of vulnerabilities?

RQ$_2$ aims at investigating an important question associated with software reuse, namely the extent to which reuse influences the security of a project. 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.

3.2 Cases and Unit of Analysis

To answer the aforementioned research questions, we designed a holistic multiple-case study, i.e., one in which the multiple cases are also the units of analysis [22]. 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 Reaper [17], a subset of the GHTorrent data set [8]. GHTorrent is a large openly-available database of GitHub metadata. Reaper is a curated dataset comprising more than 2 million unique projects. It retrieves information from GHTorrent and filters it on the following criteria: (1) Select only projects that are of the Java, Python, PHP, Ruby, C++, C, or C# programming languages. (2) The project's repositories contain evidence of an engineered software project such as, documentation, testing, and project management. (3) This dataset contains only projects that are publicly accessible, excluding forked and deleted repositories.

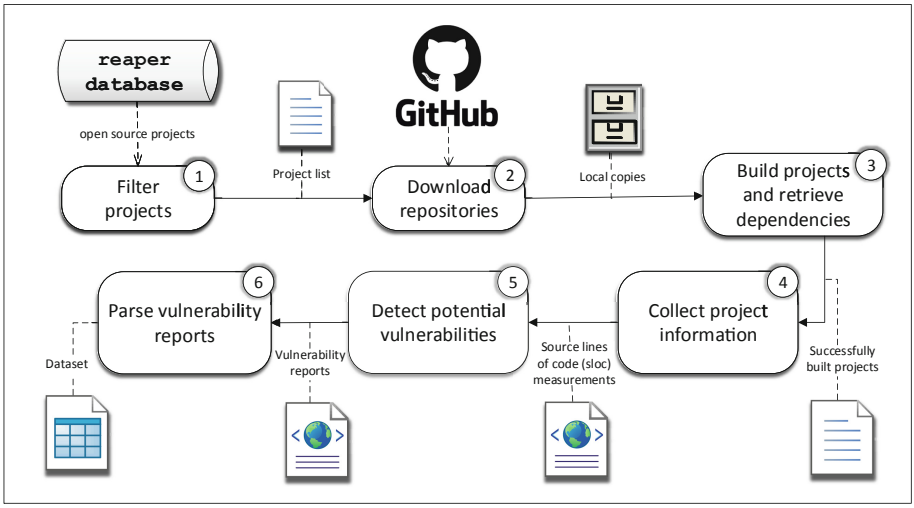

Fig. 1. The dataset construction procedure.

3.3 Variables and Data Collection

To address the research questions, we built a dataset 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 following paragraphs together with the associated variables. Figure 1 illustrates the data collection. A summary of the recorded variables is presented in Table 2. We note that the complete procedure is automated in a set of scripts available on GitHub.[8]

Step 1: Filter projects. First, we queried the Reaper database [17] and selected the GitHub projects written in Java. We selected Java as a programming language so as to take advantage of automated build support provided by Maven, and the security violations identification capabilities of the SpotBugs[9] tool. Thus, we filtered the projects by selecting only those that were using the Apache Maven automation tool.[10] We applied this filter because this tool is well-known, 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 sorted the projects based on their popularity, by retrieving their *stars* using the GitHub API.[11]

Step 2: Download repositories. Next, using the Git tool, we cloned the top 1000 projects. We selected this amount to improve the representativeness of the sample towards the population and strengthen the statistical analyses.

[8] https://github.com/AntonisGkortzis/Vulnerabilities-in-Reused-Software.
[9] https://spotbugs.github.io/.
[10] https://maven.apache.org/.
[11] https://developer.github.com/v3/.

Table 2. List of recorded variables

Variable	Description
Project	Full project name
C_n	Number of native classes
C_r	Number of reused classes
L_n	Number of source lines of code in native classes
L_r	Number of source lines of code in reused classes
V_n	Number of vulnerabilities in native code
V_r	Number of vulnerabilities in reused code
VC_n	Number of potentially vulnerable native classes
VC_r	Number of potentially vulnerable reused classes
VL_n	Number of source lines of code in potentially vulnerable native classes
VL_r	Number of source lines of code in potentially vulnerable reused classes

Step 3: Build projects and retrieve dependencies. With the repositories at hand, we built each project. During the building process, the generated compiled package (i.e., a `.jar` or `.war` file) is placed in the local Maven repository (the `.m2` directory by default). The dependencies (third party packages or libraries) of each project are also downloaded and placed in the local repository. From the total 1000, we discarded 490 projects that failed to build. For the remaining 510 successful builds, we stored their tree, i.e., the paths to the packages of the project and its dependencies.

Step 4: Collect project information. In this step, we analyzed each project's dependencies' tree and collected the first groups of variables: *project*, C_n, C_r, L_n and L_r. For that, we collected the class files from each package and also used them to retrieve the source lines of code (SLOC), which is estimated based on the number of the statements. We discarded projects that had less than 1000 lines of native code, which led us to a final dataset of 301 projects.

Step 5: Detect potential vulnerabilities. To perform this step we employed static analysis. The benefit of using static analysis for detecting potential security vulnerabilities is the ability to assess a large set of projects without the need of test cases and execution scenarios. Static analyzers can look for patterns in the code base of a system attempting to cover all possible execution paths. Kulenovic et al. [13] studied several static analysis methods for detecting security vulnerabilities. Their findings show that the algorithms used for detecting security vulnerabilities with static analysis are improving constantly, and consequently are increasing the accuracy and the precision of the static analyzers.

We used the static analyzer SpotBugs[12] (v3.1.11) [10,25,27]. This tool considers bug patterns as rules to identify violations of good coding practices [10]. The rules are organized into nine categories, two of them related to security:

[12] This is the well-known *FindBugs* tool further developed under a new name.

Security and *Malicious Code*. Moreover, SpotBugs classifies the detected violations into three levels of confidence (low, medium, high) related to the likelihood of their veracity. The tool has already been evaluated in independent studies [6,10] and [4], which reported an average precision of 66%. The studies also reported that the precision can be boosted by ignoring vulnerabilities with a low level of confidence. Nevertheless, there is still a possibility that SpotBugs introduces noise (false positives) to the data collection. However, other studies showed that the detected vulnerabilities are valuable pointers to parts of the system that need to be maintained [3,5,10,11,26,27].

Finally, to further improve the findings of SpotBugs, we included its plugin FindSecBugs,[13] which covers the Open Web Application Security Project (OWASP) top-10 vulnerabilities[14] and several other Common Weaknesses Enumerations (CWEs).[15] CWE is a list of common security weaknesses, maintained by the community, and serves as a common language for classifying vulnerabilities. To detect potential vulnerabilities, SpotBugs requires the path to the compiled Java project and its dependencies. For that, we used the project trees obtained in Step 3. The output of this analysis is a XML file that contains information about the potential vulnerabilities among the native and reused classes.

Step 6: Collect vulnerability information. In this final step, we collected the second groups of variables: V_n, V_r, VC_n, VC_r, VL_n, and VL_r. For that, we parse each SpotBugs' XML report that we generated in the previous step. 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.

3.4 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 scatter plots and box plots to aid the interpretation of the collected dataset. To answer RQ_2, we first calculated the ratio of reuse Rr and vulnerabilities density Dv as described in (1a) and (1b) below.

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

Next, we used the Pearson correlation [7] to calculate the association between reuse and security vulnerabilities. To further support this analysis, we created scatter plots between the ratio of reuse and the amounts of both native-code and reused-code vulnerabilities. We note that this complete procedure is automated and available online together with all other scripts used in this study.[16]

[13] https://find-sec-bugs.github.io/.
[14] https://www.owasp.org/index.php/Top_10-2017_Top_10.
[15] https://cwe.mitre.org/.
[16] https://github.com/AntonisGkortzis/Vulnerabilities-in-Reused-Software.

4 Results

Here, we present the results obtained from the execution of the study design presented in the previous section. In particular, we first present the overall statistics of our dataset. Then we address RQ_1 by obtaining an overview of the built dataset. Next, we examine RQ_2 by analyzing the distribution of vulnerabilities between native and reused code.

Table 3. Dataset size

Variable	Value	Variable	Value
Projects	301	V_n	16 700
Reused dependencies	5 662	V_r	51 744
C_n	288 955	VC_n	7 820
C_r	1 082 995	VC_r	29 140
L_n	8 078 996	VL_n	987 421
L_r	35 279 947	VL_r	3 598 352

4.1 RQ_1 Projects' Overview

In Table 3, we present the overall size of the dataset regarding the variables we presented in Sect. 3. Figure 2 illustrates the distribution of the six variables we presented in Table 2. The Figure comprises six boxplots in a 2×3 matrix. Each column depicts a type of variable (e.g., number of vulnerabilities) and each row the type of code that the variable regards The number of outliers varied for each variable from 6% to 14% (with an average of 11%) of the total amount of projects. For visualization purposes, we omit these outliers in the boxplots.

Table 4. Descriptive statistics

Variable	Minimum	Maximum	Mean	Median	Std. deviation
C_n	3	36 587	960	132	3 641
C_r	4	118 110	3 598	1 715	7 836
L_n	1 002	798 308	26 841	3 710	88 054
L_r	92	2 525 867	117 209	59 679	192 377
V_n	0	2 230	55	5	222
V_r	0	4 175	172	48	351
VC_n	0	801	26	4	88
VC_r	0	2 660	97	28	211

In Fig. 2, we observe that most projects lie in the lowest range of values, a trend that is also visible among all variables. This observation is in line with the descriptive statistics we presented in Table 4, since the mean values are closer to the minimum than to the maximum. Based on these findings, we hypothesize that the number of vulnerabilities in source code increases with the size of the project (measured in SLOC).

We tested this hypothesis by performing independent T-tests. In our first set of tests, we ordered the dataset based on size of native code (L_n) and compared the means between the lower and upper half for: (a) the number of vulnerabilities in native code (V_n) (statistic $= -3.87$, p-value < 0.01) and (b) the number of vulnerabilities in reused code (V_r) (statistic $= -2.26$, p-value $= 0.02$) The results of the tests show a statistical significant difference between the two halves, and that a smaller design size (smaller SLOC) also presents fewer vulnerabilities. Similarly, for the second set of tests, we ordered the dataset based on the size of the reused code (L_r) and compared the means between lower and upper half of the dataset. The results are similar to the first test for both variables.

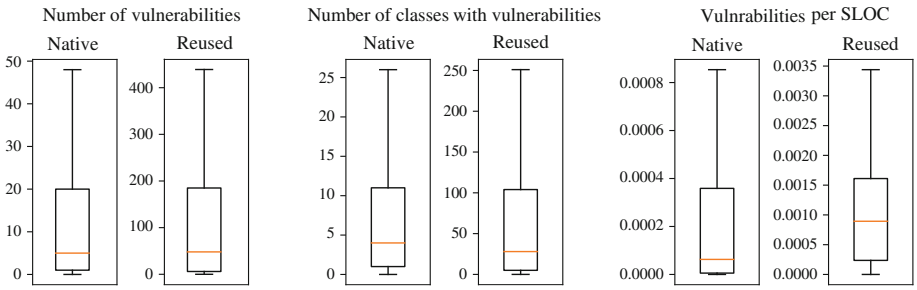

Fig. 2. Boxplot of variables related to vulnerabilities

RQ$_1$: The independent t-tests provide empirical evidence for the common belief that the number of potential vulnerabilities increases along with the design size (SLOC).

4.2 RQ$_2$ - Association between Reuse and Vulnerabilities

Figure 3 depicts three boxplots that illustrate the distribution of the vulnerability density 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 similar on both cases. However, there are more projects with higher vulnerability density in native code than in reused code. We also note 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

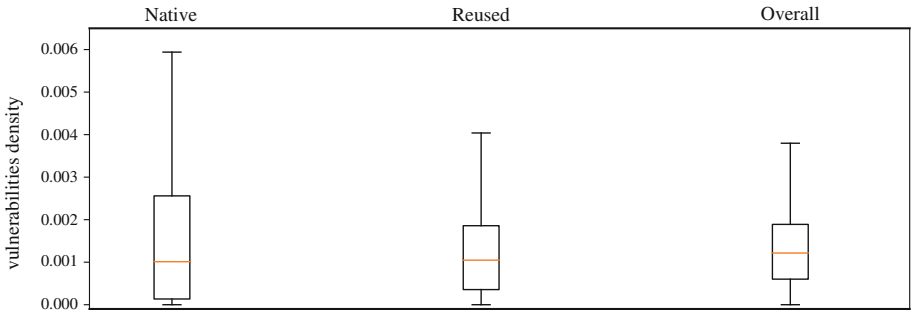

Fig. 3. Boxplots of vulnerability density in native code (left), reused code (center), and overall (right)

is considerably larger than native code, and the normalization procedure is done after the vulnerabilities are combined.

To investigate RQ_2 with regards to the association between the reuse ratio and the vulnerability density, we calculated the Pearson correlation between these variables, which are defined in Sect. 3.4. The result shows a correlation coefficient of -0.18 (p-value < 0.01), indicating a weak inverse correlation between the reuse ratio and the vulnerability density in a project. Figure 4 illustrates the distribution of the vulnerability density in the native code (left scatter plot) and in the reused code (right scatter plot) respectively, with regard to the reuse ratio. Despite the fact that there is more reused code than native, both cases have similar tendency in term of accumulation of vulnerabilities. In particular, there is a clear tendency towards a lower vulnerability density in both native and reused code.

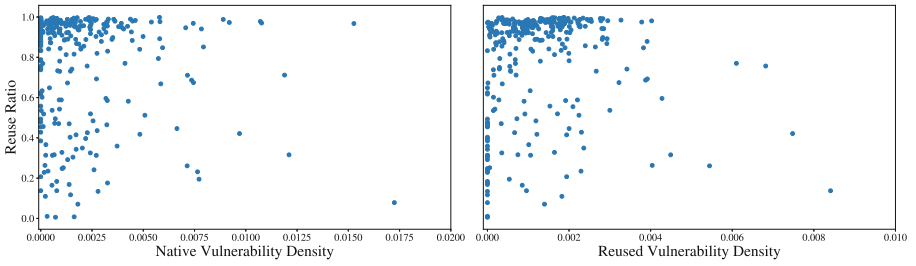

Fig. 4. Scatter plots of vulnerability density in native (left) and reused (right) code

RQ$_2$: The median vulnerability density is similar in both native and reused code. Additionally, the results show a weak inverse correlation between the reuse ratio and the vulnerability density.

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 the implications of these observations to both researchers and practitioners.

5.1 Interpretation of the Results

In summary, we found that the amount of reused code is considerably larger compared to native code. However, the vulnerability density is higher in native code, i.e., it shows a higher count of vulnerabilities per SLOC than reused code. These observations culminate in the fact that the amount of vulnerabilities is mostly associated with the reused code. Viewed simplistically this finding indicates that more reuse leads to more vulnerabilities. However, more reuse is associated with a lower vulnerability density. This result suggests that reused code is mature, and has fewer vulnerabilities. Consequently, if we assume that reused code stands for code that would otherwise have to be written from scratch, the increased reuse of the more mature code may lead to a lower overall density of vulnerabilities. These findings are in line with those of Mohagheghi et al. [16], who performed a comparable study but in a industrial setting and also found a lower defect density (which includes security vulnerabilities) in reused code when compared to native code. Moreover, Mitropoulos et al. [15] found a positive correlation between project size and the amount of vulnerabilities, which also aligns with our findings related to native 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. 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. To assess that, we analyzed the correlation between the reuse ratio and the size of native code (in SLOC), and found a moderate association (coefficient $= -0.43$, p-value < 0.01).

The results reported in this paper are based on abstractions observed on the overall dataset. An interesting observation in the SpotBugs reports is type of the most occurring types of security bugs. In Table 5, we list the top-5 most recurrent types of vulnerabilities. We notice that both native and reused code share the same types of vulnerabilities.

5.2 Implications for Researchers and Practitioners

Security assessment of source code is popular among practitioners and researchers. In many cases, this process is executed before every release. In our study, we provided evidence that code reuse has a positive impact on the security of a software system. Our dataset provides information related to reuse ratio and the existence of potential vulnerabilities in 301 projects. Practitioners can

Table 5. Most occurring types of vulnerabilities

Security bugs description	Reported in code
May expose internal representation by returning/incorporating reference to mutable object	Native & Reused
Field is not final but should be	Native & Reused
Field should be package protected	Native & Reused
Method invoked that should be only be invoked inside a *doPrivileged* block	Native & Reused
Classloaders should only be created inside *doPrivileged* block	Native
Field is a mutable collection which should be package protected	Reused

consult the dataset and gain insight on projects of their interest. Software developers can use this information to prioritize bug fixing and assign resources to improve their native code with regards to security. Moreover, practitioners can employ the provided automation scripts to perform a similar analysis on their own code base.

The findings of this study can also benefit researchers. In particular, the provided dataset can be used to investigate research questions different from the ones discussed in this study, e.g., clustering of projects based on one or more of the available variables. Additionally, our proposed approach can be employed to investigate other software quality attributes (e.g., correctness, performance) since SpotBugs can also provide valuable information related to these quality attributes. To examine this aspect, researchers can modify the provided scripts to include bug reports from SpotBugs related to these attributes. Researchers can also reuse our scripts to extend or create their own datasets.

6 Threats to Validity

In this section, we discuss the construct validity, the reliability, and the external validity of our study. Threats to internal validity, are not applicable in this study since it doesn't 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, these reports are indicators of places that developers should focus when reviewing the code. Furthermore, vulnerabilities reported in the reused code may not all actually affect a project's security, because some vulnerable elements may never be executed by the native code. Moreover, the study can identify only black-box reuse as defined by Heinemann et al. [9]. Black-box reuse requires developers to include a binary version of the dependency, which in our case is a Java package

(`jar` or `war` file). White-box reuse is the incorporation of the third-party source code into the native source code. This approach requires clone code-detection like that performed by Heinemann et al. [9], which is out of the scope of this study. Finally, projects were sorted based on their popularity (GitHub stars). This criterion might not be indicative of the usage of these projects.

Concerning reliability, we put our best effort to make this study easy to replicate. The source code, along with detailed instructions, are available in this link.[17] The dataset variable values may vary based on the date of the study. To retrieve the same values researchers should revert the Git repositories to the date of this study (February 10th 2019). To mitigate any reliability risk, two developers were involved and reviewed the process and the actual scripting.

Finally, concerning external validity, we identified two potential risks. Firstly, the project selection was limited to one programming language (Java), and thus generalization of our findings in other languages requires further investigation. Secondly, 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 45% of the initial project selection (1 000) failed to build with Maven or was partially built, and was therefore excluded from the analysis.

7 Conclusion

In this paper, we reported a holistic multiple-case method study with the goal of investigating the association between security vulnerabilities and software reuse in open source projects. In particular, we looked into the distribution of vulnerabilities among native code created by a project's development team and reused code introduced through third-party dependencies, also identifying characteristics of the studied projects. Moreover, we examined the correlation between the ratio of reuse and the density of vulnerabilities. For that, we constructed a dataset with 301 of the most popular projects in the Reaper repository, and collected information regarding the size of both native and external code, as well as vulnerability information obtained from the static analyzer SpotBugs. Unsurprisingly, the results suggest that larger projects are associated with more vulnerabilities in both native and reused code. However, they also show that higher reuse ratio is correlated with a lower overall vulnerability density.

In light of our study design and findings, we envisage several opportunities of future work. On the one hand, it is desirable to extend the provided dataset and incorporate projects from other programming languages and automated build systems, such as Ant, Gradle, npm and pip. The extended dataset could be used for replication and extension studies. The former could mitigate threats to the validity of our study by providing triangulation of data and results. Extension studies could encompass the current or evolved dataset, and explore more in-depth research questions related to, for example, the features of larger and smaller projects, or with more or less external code. On the other hand, the automation scripts shared through this study could be turned into a tool that

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

could benefit both practitioners and researchers by providing a workbench for in-house analyses or future studies.

Acknowledgments. We express our appreciation to Paris Avgeriou for reviewing the manuscript and providing us with feedback that improved its quality. The research described has been carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

References

1. April 2014 Web Server Survey—Netcraft. https://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html
2. Cybersecurity Incident & Important Consumer Information—Equifax. https://www.equifaxsecurity2017.com/
3. Ayewah, N., Pugh, W.: The Google FindBugs fixit. In: Proceedings of 19th International Symposium on Software Testing and Analysis (ISSTA 2010), pp. 241–252. ACM, Trento (2010). https://doi.org/10.1145/1831708.1831738
4. Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: Proceedings of 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), pp. 1–8. ACM Press, San Diego (2007). https://doi.org/10.1145/1251535.1251536
5. Feitosa, D., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., Nakagawa, E.: What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes? Inf. Softw. Technol. (2018). https://doi.org/10.1016/j.infsof.2018.07.014
6. Feitosa, D., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y.: Investigating quality trade-offs in open source critical embedded systems. In: Proceedings of 11th International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA 2015), pp. 113–122. ACM, Montreal (2015). https://doi.org/10.1145/2737182.2737190
7. Field, A.: Discovering Statistics Using IBM SPSS Statistics, 4th edn. SAGE Publications Ltd., Thousand Oaks (2013)
8. Gousios, G., Spinellis, D.: GHTorrent: GitHub's data from a firehose. In: Proceedings of 9th IEEE Working Conference on Mining Software Repositories (MSR 2012), pp. 12–21. IEEE, June 2012. https://doi.org/10.1109/MSR.2012.6224294
9. Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B., Irlbeck, M.: On the extent and nature of software reuse in open source Java projects. In: Schmid, K. (ed.) ICSR 2011. LNCS, vol. 6727, pp. 207–222. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21347-2_16
10. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM SIGPLAN Not. **39**(12), 92–106 (2004). https://doi.org/10.1145/1052883.1052895
11. Khalid, H., Nagappan, M., Hassan, A.E.: Examining the relationship between FindBugs warnings and app ratings. IEEE Softw. **33**(4), 34–39 (2016). https://doi.org/10.1109/MS.2015.29
12. Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K.: Do developers update their library dependencies? Empirical Softw. Eng. **23**(1), 384–417 (2018). https://doi.org/10.1007/s10664-017-9521-5

13. Kulenovic, M., Donko, D.: A survey of static code analysis methods for security vulnerabilities detection. In: Proceedings of 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2014), pp. 1381–1386, May 2014. https://doi.org/10.1109/MIPRO.2014.6859783
14. Meneely, A., Williams, L.: Secure open source collaboration: an empirical study of Linus' law. In: Proceedings of 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 453–462. ACM (2009). https://doi.org/10.1145/1653662.1653717
15. Mitropoulos, D., Karakoidas, V., Louridas, P., Gousios, G., Spinellis, D.: The bug catalog of the Maven ecosystem. In: Proceedings of 11th Working Conference on Mining Software Repositories (MSR 2014), pp. 372–375. ACM, Hyderabad (2014). https://doi.org/10.1145/2597073.2597123
16. Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H.: An empirical study of software reuse vs. defect-density and stability. In: Proceedings of 26th International Conference on Software Engineering (ICSE 2004), pp. 282–292. IEEE Computer Society, Washington, DC (2004). http://dl.acm.org/citation.cfm?id=998675.999433
17. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating GitHub for engineered software projects. Empirical Softw. Eng. **22**(6), 3219–3253 (2017). https://doi.org/10.1007/s10664-017-9512-6
18. Neuhaus, S., Zimmermann, T.: The beauty and the beast: vulnerabilities in red hat's packages. In: Proceedings of 2009 USENIX Annual Technical Conference (USENIX 2009) (2009)
19. Pashchenko, I., Plate, H., Ponta, S.E., Sabetta, A., Massacci, F.: Vulnerable open source dependencies: counting those that matter. In: Proceedings of 12th ACM/IEEE Internatinal Symposium on Empirical Software Engineering and Measurement (ESEM 2018), pp. 42:1–42:10. ACM, Oulu (2018). https://doi.org/10.1145/3239235.3268920
20. Pham, N.H., Nguyen, T.T., Nguyen, H.A., Wang, X., Nguyen, A.T., Nguyen, T.N.: Detecting recurring and similar software vulnerabilities. In: Proceedings of 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), pp. 227–230. ACM, Cape Town (2010). https://doi.org/10.1145/1810295.1810336
21. Ponta, S.E., Plate, H., Sabetta, A.: Beyond metadata: code-centric and usage-based analysis of known vulnerabilities in open-source software. In: Proceedings of 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018), September 2018. https://doi.org/10.1109/ICSME.2018.00054
22. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. Wiley, Hoboken (2012)
23. 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. https://doi.org/10.1109/TSE.2010.81
24. van Solingen, R., Basili, V., Caldiera, G., Rombach, H.D.: Goal question metric (GQM) approach. In: Encyclopedia of Software Engineering, pp. 528–532. Wiley, Hoboken (2002). https://doi.org/10.1002/0471028959.sof142
25. Tomassi, D.A.: Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs. In: Proceedings of 2018 26th ACM Joint Meeting on European Software Engineering Conference on and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), pp. 980–982. ACM, Lake Buena Vista (2018). https://doi.org/10.1145/3236024.3275439

26. Tripathi, A.K., Gupta, A.: A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for Java programs. In: Proceedings of 18th Interantional Conference on Evaluation and Assessment in Software Engineering (EASE 2014), pp. 23:1–23:4. ACM, London (2014). https://doi.org/10.1145/2601248.2601288
27. Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P., on Vouk, M.A.S.E.I.T.: On the value of static analysis for fault detection in software. IEEE Trans. Softw. Eng. **32**(4), 240–253 (2006). https://doi.org/10.1109/TSE.2006.38