

Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study

Amiangshu Bosu
Dept. of Computer Science
University of Alabama
Tuscaloosa, AL USA
asbosu@ua.edu

Jeffrey C. Carver
Dept. of Computer Science
University of Alabama
Tuscaloosa, AL USA
carver@cs.ua.edu

Munawar Hafiz
Dept. of Computer Science
Auburn University
Auburn, AL USA
munawar@auburn.edu

Patrick Hilley
Dept. of Math and Computer
Science
Providence College
Providence, RI USA
philley@friars.providence.edu

Derek Janni
Dept. of Mathematical
Sciences
Lewis & Clark College
Portland, OR USA
derekjanni@lclark.edu

ABSTRACT

To focus the efforts of security experts, the goals of this empirical study are to analyze which security vulnerabilities can be discovered by code review, identify characteristics of vulnerable code changes, and identify characteristics of developers likely to introduce vulnerabilities. Using a three-stage manual and automated process, we analyzed 267,046 code review requests from 10 open source projects and identified 413 Vulnerable Code Changes (VCC). Some key results include: (1) code review can identify common types of vulnerabilities; (2) while more experienced contributors authored the majority of the VCCs, the less experienced contributors' changes were 1.8 to 24 times more likely to be vulnerable; (3) the likelihood of a vulnerability increases with the number of lines changed, and (4) modified files are more likely to contain vulnerabilities than new files. Knowing which code changes are more prone to contain vulnerabilities may allow a security expert to concentrate on a smaller subset of submitted code changes. Moreover, we recommend that projects should: (a) create or adapt secure coding guidelines, (b) create a dedicated security review team, (c) ensure detailed comments during review to help knowledge dissemination, and (d) encourage developers to make small, incremental changes rather than large changes.

Categories and Subject Descriptors

K.6.3 [Management of Computing and Information Systems]: Software Management—*Software development, Software process*

General Terms

Measurement, Security, Human Factors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FSE'14, November 16–22, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...\$15.00.

Keywords

code review, security defects, open source, vulnerability, inspection

1. INTRODUCTION

A project manager's life is full of analyzing trade-offs in the course of making important decisions. One important decision is to allocate appropriate resources to identify security vulnerabilities and fix the root causes in order to deliver a secure and error-free product. It takes an expert to effectively perform security-critical tasks; but experts are a scarce resource [29]. A project manager should ensure that a product gets optimum attention from security experts, that too very early in the development cycle.

Modern software development is incremental and evolutionary—i.e., make a small change, submit the change, validate the suggested change via peer code reviews, and merge the change into the main codebase. A project manager interested in security will find it appealing to be able to filter changes that have potential vulnerabilities and identify the root causes of those vulnerabilities before the changed code becomes production code. The objective of this study is *to use data from Open Source Software (OSS) projects to identify characteristics that indicate which code changes are likely to contain vulnerabilities so that managers can more effectively focus the security experts' effort on those changes*. Specifically, we explore: (1) the characteristics of the vulnerabilities identified during code review, (2) which types of code changes are more likely to contain vulnerabilities, and (3) which types of developers are more likely to introduce code changes that contain vulnerabilities.

To facilitate this exploration, we define *Vulnerable Code Changes (VCC)* as “code changes that contain exploitable vulnerabilities.” There have been numerous efforts to characterize vulnerable code or modules, as described in Section 2.2. Our approach differs from these previous efforts in two key ways: (1) our study focuses on data gathered during the peer code review process, and (2) in addition to identifying characteristics of vulnerable code, we also identify characteristics of the authors of vulnerable code.

First, we focus on data gathered during the peer code review process for a number of important reasons, as follows. Peer code review data is rich because it documents the discussion between the developers and reviewers regarding potential defects. When the defect contains a security vulnerability, these discussions have the potential to provide additional insight about the source of that

vulnerability. Datasets created from defect repositories, the approach used in most prior studies, may contain vulnerabilities that were identified before release as well as vulnerabilities that were identified after release. Conversely, the dataset created from the code review repository contains only vulnerabilities detected before the code was merged into the main branch (i.e., before release). VCCs identified during code review contain vulnerabilities that could be eliminated prior to release, by using careful coding and review techniques. The longer it takes to detect and fix a security vulnerability, the more that vulnerability will cost [30]. Therefore, eliminating vulnerabilities early via careful coding and reviewing should help to reduce the cost of developing secure software.

Second, our focus on identifying characteristics of the authors who contribute vulnerable code is motivated by the importance of human factors in all aspects of software development. We hypothesize that differences among developers' knowledge, experience and motivation will affect their likelihood of authoring VCCs. There have not been many studies investigating this issue. This study characterizes the authors who create VCCs based on their project experience and whether they are volunteers or full-time paid participants in the OSS projects.

Knowing which code changes are more prone to contain vulnerabilities will allow a project manager to focus the security experts' efforts on a smaller subset of submitted code changes. For example, code changes that exhibit the characteristics identified in our study should be integrated to the project only after a careful review from a security expert. Conversely, changes that are less prone to containing a vulnerability would only go to a security expert if one of the other reviewers raised a potential security concern. This approach would prevent the security experts from wasting their valuable time performing tasks that could be performed by other reviewers who lack the security expertise.

To characterize the VCCs, we developed a method for analyzing 267,046 peer code review requests mined from 10 popular Open Source Software (OSS) projects. Each of these projects uses the Gerrit code review tool, which captures the data necessary for our analysis. Based on an empirically built and validated set of keywords, we used a three-stage systematic analysis process to identify 413 VCCs described in those 267,046 peer review requests.

The main contributions of this work are as follows.

- We provide empirical insights into the characteristics common to VCCs, their locations, and their authors in the context of code reviews,
- We perform the first study to analyze security vulnerabilities using peer code review dataset, and
- We describe an empirically-sound methodology to systematically analyze large dataset via a manual and automated process.

The rest of the paper is organized as follows. Section 2 provides background about security defects and peer code review. Section 3 introduces the research questions. Section 4 describes the research methodology. Section 5 presents the results. Section 6 discusses the implications of the results. Section 7 addresses the threats to validity. Finally, Section 8 concludes the paper.

2. BACKGROUND

To familiarize the reader with the context and background of this study, this section describes why we focused on a specific set of security vulnerabilities, introduces previous research on characterizing vulnerable code, and illustrates the benefits of peer code review in identifying security vulnerabilities.

2.1 Security Vulnerabilities

Hundreds of types of security vulnerabilities can occur in code, design, or system architecture. The security community uses Common Weakness Enumerations (CWE) [33] to distinguish security vulnerabilities. Each security vulnerability is described by a CWE entry which contains a unique CWE ID and a short specification (description, language, severity, and common causes). Variants of vulnerabilities (e.g., different ways to overflow a buffer) are each represented by their own CWE ID.

In this paper, we concentrate on the ten most common vulnerability types, as enumerated in Table 2 in Section 4.2. We chose those ten categories based on previous studies [8, 49] and the Open Web Application Security Project (OWASP) list [39].

2.2 Characterizing Vulnerable Code

Previous researchers have sought to predict which components are likely to contain security vulnerabilities. First, regarding *code attributes*, Neuhaus et al.'s vulnerability prediction model for the Mozilla projects predicted 82% of the known vulnerabilities in the top 30% of the vulnerable components with 70% precision [38]. Gegick et al.'s regression tree model built using lines of code, alert density, and code churn information identified 100% of the vulnerable components with 8% false positive rate [24]. In addition, studies that attempted to correlate code complexity with the presence of vulnerabilities found a weak but significant correlation in the Mozilla Javascript engine [47] and varying degrees of correlations in 14 PHP applications [50]. Code complexity metrics were also used to predict vulnerable files in Mozilla Firefox and Linux kernel [46]. However, a follow-up study found the presence of non-security defects as a better predictor of vulnerable components than code complexity metrics [23].

Second, regarding *code commit characteristics*, researchers have focused on predicting vulnerabilities based on large number of commits, commit size, and code churn. They have found that Linux kernel files that underwent a large number of revisions were more likely to contain vulnerabilities [32]. Also, commits that contain vulnerabilities had twice as much code churn (lines added, modified or deleted to a file from one version to another) as non-vulnerable code commits [31], which also suggests that larger commit size may harbor more vulnerabilities. Moreover, three metrics on code churn were able to predict vulnerable files in the Linux kernel and Mozilla Firefox [46].

Third, regarding *human factors*, there are relatively few studies that use developer characteristics to predict vulnerabilities (i.e., the type of developers that are most likely to create vulnerabilities). Studies found Linux kernel files modified by more than nine developers [32], and Apache files modified by a new effective author (i.e., modifying a file for the first time) were more likely to be vulnerable. Finally, Shin et. al. found 9 metrics about a developer's social network that could predict vulnerable files [46].

Most of the prior studies focused on code and file metrics, with a few studies that consider human aspects. Our current study differs from these previous works by focusing on three factors (i.e., human aspects, the characteristics of the VCCs, and vulnerability locations) together using one dataset.

2.3 Peer Code Review

Peer code review is the process of analyzing code written by a teammate (i.e., a peer) to judge whether it is of sufficient quality to be integrated into the main project codebase. This informal type of software inspection has been established as an effective quality improvement approach [17, 51]. For many software teams, the relatively high cost of the traditional inspection process often

outweighs its benefits, thereby reducing the prevalence of formal inspections in practice [25]. On the other hand, OSS communities have championed the use of a more informal code review process via electronic communication (e.g., mailing lists). Recently many mature, successful OSS projects have mandated peer code reviews to ensure the quality and integrity of code integrated into the main project branch. Compared with the traditional Fagan-style code inspection [21], contemporary peer code reviews are: (1) more informal, (2) tool-based, and (3) occur regularly in practice [9].

In addition to their positive effect on software quality in general, Code reviews can be an important practice for detecting and fixing security bugs early [28]. For example, an expert reviewer can identify potentially vulnerable code and help the author eliminate the vulnerability or decide to abandon code. Peer code review can also identify attempts to insert malicious code.

OSS and commercial organizations have both been increasingly adopting tools to manage the peer code review process [45]. For this study, we chose ten OSS projects that used the Gerrit code review tool to manage their code review process. Gerrit [4] is one of the most popular open-source code review tools. This web-based tool is under active development and is used by many large and popular OSS projects. In addition, Gerrit integrates with the Git version control system. Developers make local changes and then submit the *patchset* (all files added or modified in a single revision) to a Gerrit repository for review. The Gerrit interface facilitates the review process by showing the changes side-by-side and allowing reviewers to add inline comments about the code (an example is shown in Figure 2 in Section 4). The author can then respond to the comments. Gerrit documents this discussion between the author and the reviewer. If reviewers suggest changes, the author can make those changes and upload a new patchset. This process repeats until the reviewers approve the patchset, which can then be merged into the main project branch. For more details, a typical workflow of a Gerrit code review can be found in the Android Developer Guide (Life of a Patch [7]).

3. RESEARCH QUESTIONS

The goal of this study is to empirically analyze the types of vulnerabilities detected in code reviews, the characteristics of VCCs, and the characteristics of developers who are more likely to authors these VCCs. The following subsections introduce six research questions, two about each of the following topics: (1) characteristics of vulnerabilities, (2) locations of vulnerabilities, and (3) characteristics of developers writing vulnerabilities.

3.1 Characteristics of Vulnerabilities

Types of vulnerabilities detected: The most foundational question is which types of vulnerabilities can be identified via peer code review. This information will help project managers understand what effects the addition of peer code review can have on the overall security of their software projects. To gather this information, the first research question is:

RQ1 *Which types of vulnerabilities do reviewers detect during peer code review?*

VCCs likely to be abandoned: Not only is it important to identify which types of vulnerabilities can be detected through peer code review, it is also important to understand whether developers fix those vulnerabilities or abandon them. When a reviewer identifies a vulnerability in a patchset, the author can choose either to fix the vulnerability or to abandon the change. The patchset cannot be merged into the main code branch unless the author fixes the

vulnerability. Because a code author's goal is to have her changes merged into the main code branch, she will attempt to address all comments by a reviewer and only abandon those changes that are too difficult to fix. Therefore, the abandonment rate of VCCs during peer code review is a good indication of the relative difficulty of fixing various types of vulnerabilities. The next question addresses this issue.

RQ2 *VCCs containing which types of vulnerabilities are more likely to be abandoned?*

3.2 Locations of Vulnerabilities

Size of the files/patchsets containing vulnerabilities: Prior studies have shown that code churn is a predictor of defect density [37]. To determine whether this relationship holds specifically for security vulnerabilities, the next research question is:

RQ3 *Is the amount of code churn in patchsets and files that contain vulnerabilities significantly larger than it is in patchsets and files that do not contain vulnerability?*

Type of the files/patchsets containing vulnerabilities: There is no clear evidence about whether new code or changes to existing code is more likely to contain vulnerabilities. We can make the following arguments on both sides of the issue. Regarding *new code*, during early stages of code development, it is reasonable to assume that a developer may be more focused on the functionality of that module than on security. Thus, a developer may introduce more security vulnerabilities in early versions of the code. Regarding *changes to existing code*, because that code has already been examined by numerous developers, many of the security vulnerabilities will likely have been identified and removed. When developers modify existing code, they can see how previous authors have coded securely and copy the coding style. These practices would suggest that existing code has less vulnerabilities than new code. Conversely, developers may modify files without understanding the context and full implications of the changes. This lack of knowledge could result in the introduction of vulnerabilities. The next research question seeks to gather data to identify whether either of these relationships hold:

RQ4 *Which type of files (new or modified) are more likely to have vulnerabilities?*

3.3 Characteristics of Developers

Effect of author experience on VCCs: OSS projects have code authors with varying experience and backgrounds. One could argue that because novice programmers are often unaware of secure code practices, they are more likely to introduce vulnerabilities. Conversely, in more mature OSS projects, experienced developers are more likely to develop or modify the security-critical components. Because they are working in the portion of the code that is more likely to be vulnerable, experienced developers may author more VCCs. Finally, if most of the OSS developers lack sufficient knowledge about security, then developer experience might have no effect on the likelihood to introduce vulnerabilities. This uncertainty leads to the next research question:

RQ5 *Does an author's experience affect the likelihood of writing VCCs?*

Effect of author's employment: Many popular OSS projects are sponsored by commercial organizations (e.g., Google/Android and RedHat/oVirt). Although the majority of the contributions in a

sponsored OSS project come from the employees of the sponsoring organization (*sponsored-employee*), there may be many other developers (volunteers or employees from other organization that have an interest in the project). Sponsored-employees and other contributors may have different motivations for participating in an OSS project. A volunteer’s motivation may be learning through participation, being a part of an intellectually stimulating project, or trying to integrate his innovative ideas into the project [26]. The motivation of an employee from another organization may be to prioritize features that are important for his or her employer. The primary motivation for a sponsored-employee may be to align the project goals with the organizational focus and enhance the reputation of the sponsoring organization through reliable software. These different motivations suggest that security may have a different priority for each type of contributor. The motivation for a sponsored-employee suggests that he or she may be less likely to create vulnerabilities than the other types of contributors. Our last research question investigates this question.

RQ6 *In cases where an OSS project is sponsored by an organization, are authors employed by that organization less likely to write VCCs than other authors?*

4. RESEARCH METHOD

In our previous work on peer code review in OSS projects, we reported that members of OSS communities often provide detailed review comments and suggestions about potential defects [13]. In another exploratory work, we validated a set of keywords describing vulnerabilities and showed that using those keywords to text-mine code review comments could identify code reviews that detected security vulnerabilities [12]. For example, if a reviewer suspects the presence of a potential buffer overflow, his/her review comments will likely contain either `buffer`, `overflow`, or both. Building on those results, we developed the 3-stage methodology shown in Figure 1 and described as follows.

4.1 Stage 1 - Mining Code Review Repositories

Following a similar approach as Mukadam et al. [36], we developed the *Gerrit-Miner* tool that can mine the publicly accessible code review data posted in a Gerrit code review repository. We identified 26 OSS projects that had publicly available Gerrit repositories with regular code review activity. In May 2013, we used *Gerrit-Miner* to mine the Gerrit repositories of those 26 OSS projects. To ensure that we had sufficient data for our analysis, we chose the ten projects that each contained more than 4,000 code review requests each. Each of these projects requires all code changes to be posted to Gerrit for review prior to inclusion in the project. We conducted a more detailed analysis on those ten projects (listed in Table 1).

4.2 Stage 2 - Building Keyword Set

We searched the review comments with a set of keywords to identify potential vulnerabilities detected by reviewers. Because the effectiveness of a keyword-based mining approach is highly dependent upon having the proper set of keywords, we used an eight-step empirical approach (described below) to identify the sets of keywords shown in the third column of Table 2. We based steps 1-6 of the approach on prior text-mining studies [27] with modifications to fit our current context. We used the text mining (tm) package [22] of R to support the calculations in steps 4-7.

1. We created and refined an initial set of keywords.

Table 2: Keywords Associated with Vulnerabilities

Vulnerability Type	CWE ID	Keywords
Race Condition	362 - 368	race, racy
Buffer Overflow	120 - 127	buffer, overflow, stack
Integer Overflow	190, 191, 680	integer, overflow, signedness, widthness, underflow
Improper Access	22, 264, 269, 276, 281 -290	improper, unauthenticated, gain access, permission
Cross Site Scripting (XSS)	79 - 87	cross site, CSS, XSS, htmlspecialchars (PHP only)
Denial of Service (DoS) / Crash	248, 400 - 406, 754, 755	denial service, DOS, crash
Deadlock	833	deadlock
SQL Injection	89	SQL, SQLI, injection
Format String	134	format, string, printf, scanf
Cross Site Request Forgery	352	cross site, request forgery, CSRF, XSRF, forged
Common keywords	-	security, vulnerability, vulnerable, hole, exploit, attack, bypass, backdoor
Common (added later)	-	threat, expose, breach, violate, fatal, blacklist, overrun, insecure

2. We searched the Gerrit databases using the initial set of keywords to build a corpus of *documents* where each *document* was a code review comment containing at least one keyword.
3. To handle the code snippets in the corpus documents, we applied identifier splitting rules (i.e. `isBufferFull` became “is Buffer Full” or `read_string` became “read string”).
4. We then cleaned the corpus by removing white-space, punctuation, and numbers. We also converted all words to lowercase. Then we created a list of tokens for each document.
5. We applied the Porter stemming algorithm [40] to obtain the stem of each token (e.g., `buffer`, `buffered`, `buffering` all became `buffer`).
6. We created a Document-Term matrix [48] from the corpus.
7. We determined whether any additional words co-occurred frequently with each of our initial keywords (co-occurrence probability of 0.05 in the same document).
8. We manually analyzed this list of frequently co-occurring additional words to determine whether to include any in the final keywords list. The last row of Table 2 lists the keywords added through this method.

The result of this stage is an empirically built set of keywords that feed into Stage 3.

4.3 Stage 3 - Building a Dataset of Potential VCCs

We divided this stage into three steps based on the study selection procedure in systematic literature reviews [19]. In the first step, *database search*, we queried the code-review database (created in Stage 1) for each project to identify code review requests with comments containing at least one of the keywords identified in Stage 2. This search resulted in a total of 11,043 review comments.

In the second step, *comments audit*, two of the authors independently audited the comments of each code review identified in the *database search* to eliminate any reviews that clearly did not contain

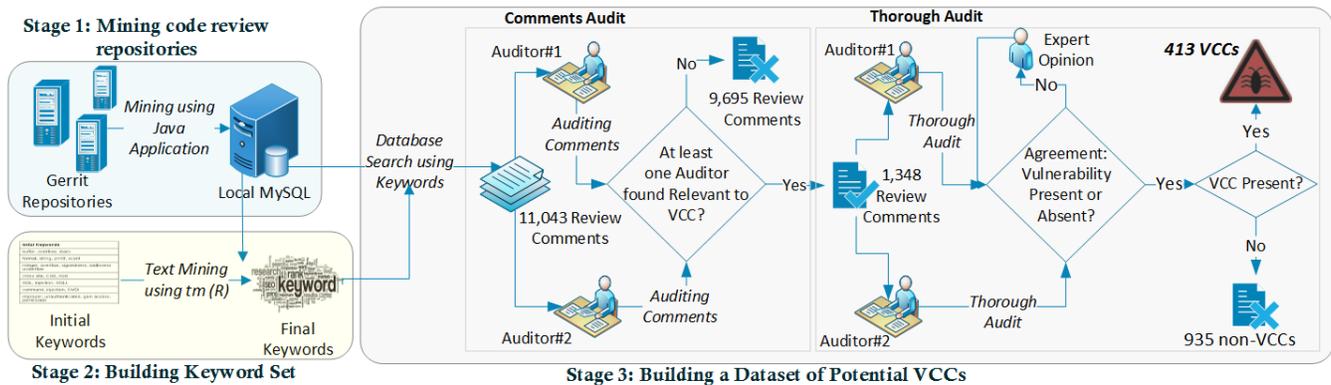


Figure 1: Three-stage Research Method

Table 1: Overview of Projects

Project	Domain	Technology	LOC*	Using since	Gerrit	Requests mined †	Comments audited	Thoroughly audited
Android	Mobile OS	C, C++, Java	13M	October, 2008		18,710	741	205
Chromium OS	Operating system	C, C++	12M	February, 2011		47,392	4,148	442
Gerrit	Code review management	Java, GWT, Servlet	169K	August, 2009		4,207	190	26
ITK/VTK	Visualization toolkit	C++	3.1M	August, 2010		10,549	344	34
MediaWiki	Wiki software	PHP, SQL, JavaScript	985K	September, 2011		59,738	930	89
OmapZoom	Mobile dev. platform	C, C++	10M	February, 2009		31,458	1,506	180
OpenAFS	Distributed file system	C	1M	July, 2009		9,369	267	79
oVirt	Virtualization management	Java	843K	October, 2011		13,647	467	118
Qt	Application framework	C++	4.5M	May, 2011		53,303	1,985	131
Typo3	Content management	PHP, SQL, JavaScript	3.1M	August, 2010		18,673	465	44
Total						267,046	11,043	1,348

* Based on Ohloh (<http://www.ohloh.net/>)

† Mined during May, 2013

vulnerabilities. We excluded a review request only if both auditors independently determined that it should be irrelevant to vulnerability. Review requests that passed this step had the potential to have a security vulnerability. This step reduced the number of candidates for VCCs from 11,043 to 1,348.

To illustrate this process, consider three example review requests returned when we searched the keyword `buffer` in the Android project. In request 17286, a reviewer commented "is this needed? How do you notify the other cpu that the buffer is available?" Although this comment has the keyword `buffer`, the auditor determined that it is not related to a vulnerability and can therefore be eliminated. Figure 2 shows two additional review requests (18718 and 13244) that contain the `buffer` keyword in the comments. For these two requests, because the comments suggested that there may be a vulnerability present, we did not eliminate them.

In the third step, *thorough audit*, we thoroughly audited the code review discussions and associated patches for the 1,348 review requests. Two authors independently audited the review request in detail to determine whether the reviewer identified a vulnerability. If we found a vulnerability, we classified the vulnerability based on CWE specification. We considered a code change vulnerable only if: (a) a reviewer raised concern about potential vulnerabilities, (b) our manual analysis of the associated code segment indicated the presence of a vulnerability, and (c) the code author either explicitly acknowledged the presence of the vulnerability in the review comments or implicitly acknowledged the presence of the vulnerability by making the recommended changes in the subsequent patches.

Consider the two code reviews shown in Figure 2. In request 13244 (top of Figure 2), the reviewer described a scenario in which the buffer overflow vulnerability may occur. The author then ac-

knowledged that possibility explicitly in the comments. This discussion confirmed that this change is a VCC and should be retained for further analysis. In contrast, in request 18718 (bottom of Figure 2), the reviewer asked the author to add a description in the method signature to explain the behavior in case the caller method supplied an overflowed buffer (e.g., the 'name' parameter is not null-terminated). This more detailed information made it clear that this change is not a VCC and should be excluded.

In this step, we considered a code review request to be vulnerable only if both auditors agreed. In case of any doubt, author Hafiz, an expert in security vulnerabilities, provided his suggestion to help the auditors reach a conclusion. We calculated inter-rater reliability using Cohen's Kappa [16]. Kappa value for this step was found 0.64 (substantial)¹. Table 1 indicates the number of code review requests inspected during the three stages for each of the 10 projects.

5. RESULTS

Using the approach in Section 4, we identified 413 VCCs across the 10 projects (see the Total column in Table 3). For each VCC, we extracted the following metrics about the vulnerability: project, review request ID, author, reviewer, date, vulnerability type, patch set number, and number of lines changed. In addition, we computed metrics about the characteristics of the VCC author at the time the VCC was posted (details in Section 5.5). The following subsections examine each of the research questions in detail.

¹ Cohen's Kappa values are interpreted as following: 0 - 0.20 as slight, 0.21 - 0.40 as fair, 0.41 - 0.60 as moderate, 0.61 - 0.80 as substantial, and 0.81 - 1 as almost perfect agreement

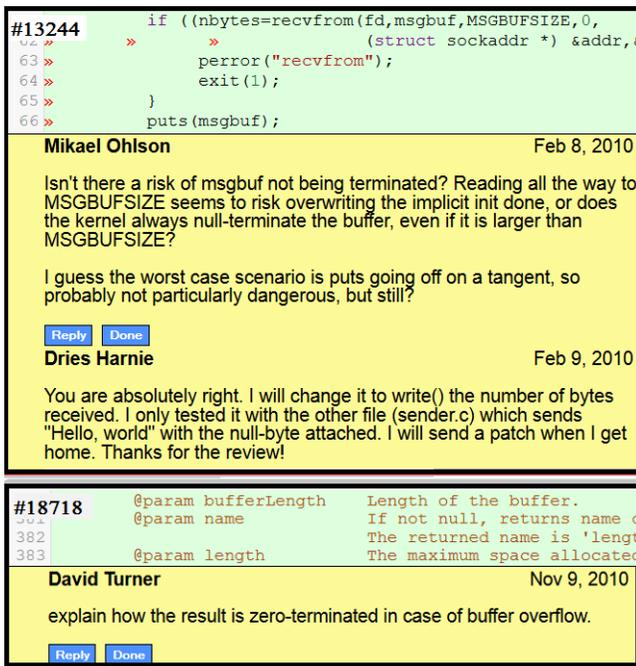


Figure 2: Gerrit Example - Keyword *buffer overflow*: 1) True buffer overflow (top # 13244), 2) Not a security defect (bottom # 18718)

5.1 RQ1: Types of Vulnerabilities Detected

Table 3 shows the VCC distribution across vulnerability types. Among the 413 VCCs we identified, race conditions vulnerabilities were the most common ($\approx 32\%$), specifically the time of check to time of use (TOCTOU: CWE-367). This result may be because many of the projects in the study have to manage shared resources across threads, which may result in race conditions.

Buffer overflow vulnerabilities were present in about 20% of the VCCs. This result is expected because the buffer overflow vulnerability is prominently featured among all vulnerabilities reported in vulnerability repositories such as the National Vulnerability Database (NVD) [15]. Also, six of the ten projects that we analyzed are written in C/C++. Typically, the buffer overflows we identified originated from either incorrect calculations of buffer size or from use of unsafe string functions (e.g., `sprintf`, `strlen`, and `strcmp`), which commonly occur in C/C++ code.

Integer overflow vulnerabilities, which are similar to buffer overflows, were the third most common type of VCCs ($\approx 10\%$). The two most common sources of these VCCs were (1) mistakes in integer arithmetic and (2) failure to account for the differences in integers between 32-bit and 64-bit systems. Interestingly, integer vulnerabilities were not limited to the C/C++ projects. Eight of the ten projects included VCCs that reported integer overflow issues.

We identified 39 VCCs ($\approx 9\%$) reporting improper access vulnerabilities. These review requests often generated a lot of attention and resulted in the involvement of project security experts during the review process (See also RQ2 in Section 5.2).

The ‘Other’ category ranks fifth. It includes : cross-site request forgery (CWE-352), memory leak (CWE-401), information leakage (CWE-200, 203, 210, 532), security misconfiguration (CWE-815), and command injection (CWE-77, 78, 88).

Cross-site scripting (XSS) and denial of service (DoS) vulnerabilities were less common. Web-based applications are more sus-

ceptible to XSS vulnerabilities. Our dataset contains only three web applications that could harbor XSS vulnerabilities. The low frequency of DoS vulnerabilities may result from the fact that these vulnerabilities are often classified as other vulnerabilities. For example, a buffer overflow can lead to denial of service. In our study, we classified a VCC as reporting a DoS vulnerability only if the review comments explicitly discussed a DoS scenario.

The results show that peer code review identified various types of vulnerabilities. Peer code review identified all of the top 10 security vulnerabilities as ranked by NVD [15]. However, the ranking of the vulnerabilities found in our study differ from the ranking suggested by NVD. XSS, buffer overflow, and SQL injection are the most commonly reported vulnerabilities in NVD [15]. In our study, we identified fewer XSS and SQL injection vulnerabilities because there were fewer projects (web and database applications) that could have the vulnerabilities. However, considering web applications only, XSS vulnerabilities were very prominent (MediaWiki - 38% and Typo3 - 53%). SQL injections were also present in MediaWiki and Typo3 (4 each). Nevertheless, the main conclusion from these results is not the ranking, but the fact that all kinds of vulnerabilities are reported in VCCs. This results indicate that *peer code review leads to the identification of different vulnerability types. Therefore, projects concerned about security should adopt peer code review to identify vulnerabilities early.*

5.2 RQ2: VCCs Likely to be Abandoned

Overall, approximately 77% of the VCCs identified during peer review were merged into the code repository (i.e., they were not abandoned). Within that set of changes, most authors fixed the vulnerability prior to merging the code into the trunk. In a very few cases, rather than fixing the vulnerability, the author opened a new bug for the vulnerability and asked the reviewers to approve the change as-is (i.e., with the vulnerability present).

Figure 3 shows that VCCs containing some types of vulnerabilities were significantly more likely to be abandoned than VCCs containing other types ($\chi^2_{9,413} = 28.61, p = .001$). VCCs containing the *improper accesses* vulnerabilities (the fourth most common type of vulnerability), which threaten application integrity and data privacy, were the most likely to be abandoned. Unlike *buffer overflows* and *integer overflows*, that are easier to fix, *Improper access* issues are more complicated architectural issues and have critical consequences that require involvement of managers or developers with design knowledge. If the reviewer can hypothesize a scenario where an access violation may occur, she will reject the change and instruct the author to redesign the solution. This requirement for additional work likely led to the high abandonment rate for VCCs with *improper access* vulnerabilities.

VCCs related to *deadlock* and *other* vulnerabilities, which were found at relatively low rates, were also abandoned at a high rate. VCCs related to the two most common vulnerabilities, *buffer overflow* and *race condition*, were abandoned at a moderate rate (between 20% to 25%). *While most vulnerability types were frequently fixed, some of the most critical vulnerabilities were fixed less often. In general, vulnerabilities identified during code review are easy to fix. Although, special attention needs to be given to the most critical vulnerabilities due to their relatively high abandonment rate.*

5.3 RQ3: Size of the Files/Patchsets Containing Vulnerabilities

We define *NLC* (*Number of Lines Churned*) as “the total number of lines changed, added, or removed in a file” between revisions.

Because the 10 projects included in this study all required code changes to be posted to Gerrit, it can be said that Gerrit is the

Table 3: Number of Vulnerabilities Identified

Project	Race Condition	Buffer Overflow	Integer Overflow	Improper Access	XSS	DoS/ Crash	Deadlock	SQL Injection	Format String	Other	Total
Android	16	16	3	5	0	5	2	0	6	7	60
Chromium OS	38	38	21	15	0	6	5	2	2	12	139
Gerrit	6	0	1	0	0	4	0	1	0	2	14
ITK/VTK	1	4	7	0	0	1	0	0	0	1	14
MediaWiki	7	2	2	3	14	0	0	4	1	4	37
OmapZoom	12	11	4	7	0	0	5	0	0	6	45
OpenAFS	12	7	2	0	0	1	0	0	0	0	22
OVirt	24	0	0	5	1	1	1	0	0	3	35
Qt	17	3	3	1	0	1	3	0	1	1	30
Typo3	0	0	0	3	9	0	0	4	0	1	17
Total	133	81	43	39	24	19	16	11	10	37	413
% of Total	32%	20%	10%	9%	6%	5%	4%	3%	2%	9%	100%

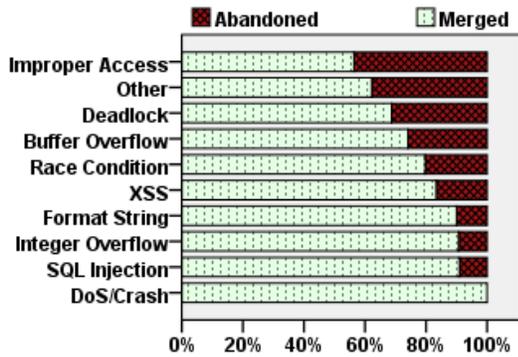


Figure 3: Status of Vulnerable Code Changes

gatekeeper between the developer’s local branch and the main code base [36]. This requirement means that some very large patchsets were submitted to Gerrit, to follow the policy, but were likely not reviewed. Consistent with the result of a prior study [41], our manual inspection of some of these large patchsets indicated that they were performing tasks like merging branches, importing new versions of third-party libraries, importing externally developed code, or changing copyright information. To ensure that our analysis only included changes that actually underwent peer code review, we needed to eliminate this type of change. Because it was not possible to manually review all the large changes, we used an alternate approach to identify those changes that should be excluded.

The most objective way to determine whether a change was reviewed was the presence of inline comments (Section 2.3) in Gerrit. Using this concept, for each project we identified the largest patchset (in terms of NLC) that contained at least one inline comment (i.e., it was reviewed). We then excluded any patchsets that contained more NLCs, under the assumption that they were likely not peer reviewed. This process excluded a total of 1820 review requests (~0.7% of the total). We used this smaller set of review requests (i.e. 265,226 out of total 267,046) for RQ3, and RQ4.

Table 4 lists the average NLC - (1) for all patchsets, (2) for all files, (3) for the patchsets containing a vulnerability, and (4) for the files containing a vulnerability. For each project, the average NLC for the vulnerable files was larger than the average NLC for all files. Analyzing the average NLC observations from the 10 projects together (i.e. two from each project), this difference is significant (Wilcoxon Signed Rank test, $Z=-2.805$, $p=.005$). Additionally, in 9/10 projects, the average NLC for the vulnerable patchsets was

Table 4: Average NLC for Patchsets and Files

Project	All		Vulnerable		Ratio (All/Vulnerable)	
	Patch-sets	Files	Patch-sets	Files	Patch-sets	Files
Android	230	47	846	202	3.67	4.29
Chromium OS	151	38	364	153	2.41	4.02
Gerrit	108	25	226	103	2.09	4.12
ITK/VTK	273	43	576	96	2.10	2.23
MediaWiki	88	27	619	127	7.03	4.70
OmapZoom	160	46	143	77	0.80	1.51
OpenAFS	111	28	424	266	3.81	9.50
oVirt	117	25	121	73	1.03	2.92
Qt	179	32	260	96	1.45	3.00
Typo3	183	34	713	82	3.89	2.41

higher than the average NLC for all patchsets. Again, analyzing all projects together, this difference is significant (Wilcoxon Signed Rank test, $Z=-2.599$, $p=.009$).

To further understand these results, we calculated the ranking of each vulnerable patchset and each vulnerable file in terms of NLCs relative to all patchsets and files. Approximately 50% of the vulnerable patchsets were in the top 20% of all patchsets and 30% of the vulnerable patchsets were in the top 10%. The same ratios held for the vulnerable files. This result supports earlier findings that vulnerability contributing commits were significantly larger than non-vulnerability contributing commits [31], and increased likelihood of defects in large changes [42]. Therefore, *the probability of a file or patchset containing a vulnerability increases with the number of lines churned. Because larger changes are more difficult to review and more likely to have vulnerabilities, developers should strive to make small, incremental changes where possible.*

5.4 RQ4: Type of the Files/Patchsets Containing Vulnerabilities

Table 5 shows the average number of vulnerabilities per 10K files and the average NLC for new files and modified files. Overall, the number of vulnerabilities is significantly higher in the modified files than in the new files (Wilcoxon Signed Rank test, $Z=-2.599$, $p=0.009$) This result indicates that modified files are more likely to contain vulnerabilities than new files. Based on the result from RQ3, it is possible that this observation was caused by the NLC in the files rather than whether they were new or modified. An analysis of NLC showed that the NLC was significantly higher in the new files than in the modified files. This result is the exact opposite of what we would

Table 5: Modified Files Vs. New Files

Project	Modified Files		New Files	
	Vulnerabilities /10K Files	NLC	Vulnerabilities /10K Files	NLC
Android	8.06	30	5.89	66
Chromium OS	8.82	27	6.15	49
Gerrit	8.52	18	6.88	37
ITK/VTK	2.35	26	2.16	72
MediaWiki	2.28	26	1.51	26
OmapZoom	6.06	38	1.25	56
OpenAFS	7.99	25	1.86	34
Ovirt	6.78	22	3.31	30
Qt	1.36	23	0.38	48
Typo3	1.81	23	2.01	47

have expected if the new vs. modified distinction was not relevant (i.e. new files should have had more vulnerabilities because they had a higher NLC on average). Therefore we can conclude that whether a file is new or modified does affect the likelihood of vulnerabilities. Modified files are more likely to contain vulnerabilities than the new files.

This result may be due to our initial argument that, when a developer changes someone else’s code, s/he may introduce vulnerabilities due to lack of knowledge of the overall design and implications of the change. Along this line, Meneely et al. found that *New Effective Authors (NEA)*, i.e., authors making changes to a file for the first time, made 41% of the observed vulnerabilities [32]. While we did not calculate NEA in our study, this phenomenon may provide an explanation of our observed result. This argument is also consistent with previous findings that code which changes frequently, computed either by code churn [46] or by number of revisions [32], is more likely to contain vulnerabilities.

Because modified files are more likely than new files to contain vulnerabilities, patchsets that contain large, modified files should be reviewed carefully for vulnerabilities.

5.5 RQ5: Effect of Author Experience on VCCs

The 413 VCCs were written by a total of 292 authors. To answer this research question, we defined the *experience* of an author to be "the number of posted code review requests (CRR) in the current project at the time they posted the VCC." The rationale is that the more code changes an author posts for a given project, the more experienced s/he has with that project. This measure of experience is similar to the ones used in previous software engineering studies [35, 43]. Because of the varying number of contributors to each of the 10 projects, we also calculated percentile ranks of the authors based on experience at the time they posted the VCC.

The average experience of the authors contributing VCCs is significantly higher than the average experience of all authors (Wilcoxon Signed Ranks test $Z=-6.997, p<.001$). This result is bit surprising, because intuition would suggest that inexperienced authors, who are unaware of secured coding practices, should be more likely to write VCCs. In fact, 50% of the VCC authors were in the top 20% of authors based on experience, with 25% in the top 10%.

In terms of number, approximately 25% of the VCC authors had made over 200 code changes prior to authoring VCCs. Zeller stated that because experienced authors usually take the responsibilities of working on the riskiest parts of the system, they may be responsible for the critical bugs [53]. Security critical modules are risky, so this results support Zeller’s argument. In addition, the fact that the most experienced authors introduce the majority of the VCCs supports McGraw’s claim that most developers, even the most experienced,

lack knowledge about secure coding practices [29]. Conversely, Rahman and Devanbu tested Zeller’s hypothesis and they did not find any supporting evidence regarding association between author experience and severity of bugs [43].

Because we calculate experience based on activity level (the more code someone writes, the higher his/her experience metric), it is possible that more experienced authors introduce more VCCs simply because they commit more code. To analyze this result in more detail, we split the top 20% of authors, based on experience, from the remaining 80%. Each group contributed approximately 50% of the VCCs overall. Table 6 shows the percentage of the CRRs and the percentage of the VCCs contributed by members of each group. The percentage of code contributed by the top 20% authors (column 2 of Table 6) is similar to that found in prior OSS studies [18, 34].

In all cases, the percentage of VCCs introduced by top 20% of authors is significantly less than the percentage of code contributed by them ($t_{10,9} = -4.563, p = 0.001$). We computed the ratio between %VCC and % CRR for the two groups (under %VCC/%CRR in Table 6). If the most experienced authors introduced more VCCs simply because they contributed more code, the ratios should be close to 1. However, we found those ratios less than 1 for the top 20% authors and more than 1 for the remaining 80% of authors. The last column of Table 6 shows that less experienced authors (in the remaining 80%) are 1.8 to 24 times more likely to contribute a VCC than more experienced authors (in the top 20%).

While the majority of the VCCs are introduced by the most experienced authors, less experienced authors are 1.8 to 24 times more likely to write VCCs. This result has two key implications: (1) even experienced developers make secure coding mistakes, and (2) code from inexperienced authors needs more careful security reviews.

5.6 RQ6: Effect of Author’s Employment

We were able to reliably determine the sponsoring organizations for six out of the ten projects. Those six projects account for 91% of the VCCs in this study. Using the author’s email address, we determined whether he or she was an employee of the sponsoring organization. Table 7 shows the sponsoring organization, email domains, employee CRR percentages, and employee VCC percentages for the six sponsored projects.

Overall, the percentage of VCCs authored by sponsored-employees was significantly higher than the percentage of CRRs authored by volunteers ($t_{6,5} = -2.873, p = 0.035$) meaning that changes from sponsored-employees were more likely to have vulnerabilities than changes from other contributors. We again checked whether this result was due to the patchset size effect from RQ3. In this case, there is no significant difference between the average NLC for the patchsets submitted by sponsored-employees and

Table 6: Contributions by Bottom 80% of Authors

Project	%CRR		%VCC		%VCC / %CRR		VCC likelihood of Remaining 80% (times)
	Top 20%	Remain-ing 80%	Top 20%	Remain-ing 80%	Top 20%	Remain-ing 80%	
Android	89	11	62	38	0.70	3.5	5.0
Chromium OS	83	17	58	42	0.70	2.5	3.5
Gerrit	93	7	50	50	0.54	7.1	13.1
ITK/VTK	87	13	79	21	0.91	1.6	1.8
MediaWiki	92	8	51	49	0.55	6.0	10.9
OmapZoom	77	23	51	49	0.64	2.1	3.3
OpenAFS	93	7	64	36	0.69	5.1	7.4
Ovirt	75	25	28	72	0.37	2.9	7.8
Qt	86	14	57	43	0.66	3.1	4.7
Typo3	91	9	29	71	0.32	7.9	24.6

Table 7: Code Changes and VCCs Authored by Employees

Project	Sponsoring company	Email domain	% CRR	% VCC
Android	Google	google.com, android.com	53.3%	65.0%
Chromium OS	Google	google.com, chromium.org	89.1%	93.5%
MediaWiki	WikiMedia	wikimedia.org	35.7%	64.9%
OmapZoom	Texas Instruments	ti.com	70.7%	95.6%
OVirt	Redhat Inc.	redhat.com	88.5%	91.4%
Qt	Digia, Nokia (previously)	digia.com, nokia.com	50.7%	56.7%

the average NLC for the patchsets submitted by non-employees ($t_{6,5} = -1.778, p = 0.136$). Therefore, we argue that sponsored-employees are more likely to contribute vulnerable code than other contributors regardless of the patchset size.

This result was contrary to our *a priori* expectation. We pose two hypotheses for this result. First, sponsored-employees may be more likely to work on the security-sensitive components of the projects, therefore writing more VCCs. Second, code changes coming from non-employees may be reviewed more thoroughly prior to submission to Gerrit. For example, we found instances where employees of Samsung, Sony Mobile, and T-mobile submitted code changes to the Android repository. Before submitting those changes the authors may have performed some testing and review of the code on their own. Conversely, code changes from Google employees (the sponsoring company) may not be reviewed prior to submission to Gerrit. We have not tested these hypotheses yet, but we believe that further investigation is needed to better understand this result.

Employees of organization sponsoring the OSS project are more likely to write VCCs. Code from sponsored-employees needs to be reviewed more thoroughly.

6. DISCUSSION

Our results provides empirical evidence about the usefulness of peer code review to identify different types of vulnerabilities. Therefore, in addition to the known benefits of code reviews, i.e., detecting bugs [20, 51], maintaining code integrity [17], improving the relationships between the participants [14], and spreading knowledge, expertise, and development techniques among the review participants [45, 9], peer code review also increases security by facilitating the identification and removal of vulnerable code changes.

This study produced two types of results. First, some results are consistent with findings from previous studies (large files/patchsets more likely to contain VCCs, and modified files are more likely to contain VCCs). These results are still interesting because they were produced from a different data source than the one used in the previous studies, providing some additional confirmatory power. Second, other results provide some new insights. Specifically, the results about the characteristics of VCC authors are surprising and new (see Section 5.5 and Section 5.6). Even though the projects included in this study are popular and successful OSS projects with a lot of talented developers, the most experienced developers still authored VCCs. During our analysis of the code changes, we observed different developers creating similar types of vulnerabilities. Many of those common mistakes were as simple as using unsafe methods, using non-sanitized inputs, the TOCTTOU race condition, and overflows in integer arithmetic. Because developers can avoid those types of mistakes by learning the practices of secure coding,

we argue that developers lack adequate knowledge of secure coding practices. Moreover, contrary to our initial belief, we found that employees of the organization sponsoring the OSS project were more likely to author VCCs. We believe these findings warrant further research.

Based on the results, and our observations during this study, we make the following recommendations.

- The websites of many OSS projects contain guidelines for coding style [2, 6] and for general development [3]. Unfortunately, the projects included in this study did not provide secure coding guidelines for their developers. Even experienced developers making common insecure coding mistakes (see Section 5.5) suggest their unawareness of secure coding guidelines. To educate their developers, we suggest that projects either reference existing online secure coding guides (e.g., CERT Secure Coding Standards [1]) or publish a few secure coding practices on the project webpage.
- During our *thorough audit* of the review comments (Section 4.3), we found that reviewers in three projects, Android, Chromium OS, and Typo3, would refer a change to the ‘security team’ when they were unsure of the security implications. Other projects may also have security teams that we did not observe in the review comments. We suggest that projects create a security code review team, whose members have the most knowledge about secure coding. As McGraw suggests, these developers can be further trained in software security and can “think like an attacker” in their daily work [29]. This security review team could use our results along with other vulnerability prediction models to focus their efforts on the riskiest modules.
- To enable the information documented during code reviews to be as useful as possible, we suggest that reviewers provide detailed comments that include the rationale for their suggested change. In addition to indicating the security problem, the detailed feedback will also disseminate security knowledge. Two comments about the same problem that we observed during *thorough audit* (Section 4.3) show the contrast between high-detail (Comment-1) and low detail (Comment-2) and illustrate the benefits of providing detailed comments.

Comment-1: *"sprintf() is not a good function to ever use because you can end up with buffer overruns. Please use snprintf, and also ensure that your output is always terminated '\0' (which, I believe snprintf() will do)"*

Comment-2: *"use snprintf"*

While comment-2 does suggest a fix to the problem, comment-1 provides useful information that will not only benefit the patch submitter but also educate others who read the comment. The more detailed comment also provides the rationale behind the suggested change so the developer can better understand how to make the change and why it is necessary. This additional detail also helps disseminate knowledge about a secure coding concept throughout the project.
- We found that large changes are more likely to include vulnerabilities (Section 5.3). Because, it is more difficult for reviewers to review large patchsets in a timely manner, instead of submitting a large patchset, developers should “check in early, check in often”.
- Finally, for researchers, we observed different developers making the same mistakes. There is a need for automatic code review tools to be built and integrated with existing

code review tools like Gerrit, that can detect those common mistakes. The Balachandran [10] extension for ReviewBoard (another code review tool) performs automated code review based on static analysis. While this tool is useful, it does not target security issues specifically. Another alternative is the commercial tool Klocwork [5]. However, this tool will likely not be appealing to most OSS projects due to the cost.

7. THREATS TO VALIDITY

This section is organized around the common validity threat types.

7.1 Internal Validity

The primary threat to internal validity is project selection. That is, it is possible that set of projects analyzed did not provide a good representation of the types of security vulnerabilities that we were interested in studying. While we only included projects that practice modern code review [9] supported by Gerrit, we included most of the publicly accessible Gerrit-based projects that contain a large number of code review requests. As the data in Table 1 shows, these projects cover multiple languages and application domains. Furthermore, it is possible that projects supported by other code review tools could have behaved differently. We think this threat is minimal for three reasons: (1) all code review tools support the same basic purpose—detecting defects and improving the code, (2) the focus of the study was on security vulnerabilities rather than on the specifics of the code review tools themselves, and (3) none of the code review tools focus specifically on security vulnerabilities, so there is no clear reason that one would be favored over another for this study.

7.2 Construct Validity

The first construct validity threat is that if the keyword set was incomplete, the search could have missed some VCCs. Section 4.2 describes our process for building an empirically-validated set of keywords. To further validate the completeness of the keyword set, we randomly chose 400 code review requests that did not contain any of the keywords (40 from each of the 10 projects). The sample size of 400 was designed to provide a 95% confidence interval [52]. A manual review of those 400 code review requests found only one security vulnerability (a memory leak). This analysis increases our confidence in the validity of the keyword set.

The second construct validity threat is the appropriateness of the *comments audit* step of Stage 3 (Section 4.3). This step excluded about 88% of the review requests that contained at least one keyword. The only way to ensure that a VCC was not excluded would have been to review the details of each request. At 5-10 minutes per request, it was not feasible to give a detailed review to all 11043 requests that contained at least one keyword. To reduce the chances of eliminating a VCC, two researchers independently reviewed each comment and discarded it as a VCC only if both of them agreed. We also performed a detailed review of 400 randomly chosen code review requests (40 from each project) that were excluded and found only eight that contained VCCs. While we were not able to identify all of the VCCs, we did identify the majority of them. It is possible that the missed VCCs could have changed the overall results, but we have no reason to believe any specific type of VCC was systematically eliminated in this phase.

To reduce the threat to experimenter bias, two authors independently inspected and classified each of the 1,348 review request that passed the *comments audit* step. The authors discussed any disagreements and consulted with security expert Hafiz for additional input to reach final agreement. Author Hafiz then independently inspected 42 of the identified VCCs (~10%) and found only one that was not

a VCC and one that was misclassified. This small number of errors should not significantly alter the results.

Another threat is the validity of our experience measure (i.e. “number of prior code changes or reviews”). Because experience is multi-factorial, measuring it is complex. For example, how does one account for contributions to other projects? While a different experience measure may have produced different results, we believe that our experience measure is a reasonable representation of a developer’s experience and familiarity with the current project.

Finally, for RQ3 (Section 5.3) and RQ4 (Section 5.4), we excluded a relatively small number of very large code changes under the assumption they were not reviewed. It is possible that we excluded some code review requests that were actually reviewed. We were conservative in setting the minimum NLC to exclude, with the smallest being 9,088. Because most OSS code changes are small [45] and OSS proponents encouraged small patches [44], it is reasonable to assume that very large patchsets with no inline comments were not reviewed.

7.3 External Validity

The projects chosen for our study include OSS that vary across domains, languages, age, and governance. For each project we mined a large number of code review requests for our analysis. Therefore, we believe these results can be generalized to many other OSS projects. However, because OSS projects vary on characteristics like product, participant type, community structure, and governance, we cannot draw general conclusions about all OSS projects from this single study. To build reliable empirical knowledge, we need family of experiments [11] that include OSS projects of all types. To encourage replication, we published our scripts².

7.4 Conclusion Validity

Our dataset of 413 VCCs was built from the 267,046 review requests mined from the 10 diverse projects, which is large enough to draw conclusion with 95% confidence level [52]. We also tested all data for normality prior to conducting statistical analyses and used appropriate tests based upon the results of the normality test. The threats to conclusion validity are minimal.

8. SUMMARY AND FUTURE WORK

This study created a dataset of 413 VCCs from 267,046 code review requests mined from 10 popular Open Source Software (OSS) projects using an empirically built and validated set of keywords. The results show that peer code review is an effective method for identifying most of the common types of security vulnerabilities and that certain types of VCCs are more difficult to fix than others. While we obtained a number of important results, one result is novel and particularly interesting. The majority of the VCCs are written by the most experienced authors, supporting McGraw’s claim that most developers, regardless of experience, lack knowledge of secure coding practices [29]. Even though the less experienced authors wrote fewer VCCs overall, their code changes were 1.5 to 24 times more likely to be vulnerable.

While this study focused on vulnerabilities identified through peer code review, there is a need to study the characteristics of the vulnerable changes that were missed during peer code review. Therefore, we consider that as our next step for this research work.

9. ACKNOWLEDGMENTS

This research is partially supported by the National Security Agency, and National Science Foundation Grant No. 1156563.

² <http://amiangshu.com/VCC/index.html>

10. REFERENCES

- [1] Cert c secure coding standard. <https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard>. [Online; accessed 6-Mar-2014].
- [2] Chromium coding style. <http://dev.chromium.org/developers/coding-style>. [Online; accessed 6-Mar-2014].
- [3] Chromium developer guide. <http://www.chromium.org/chromium-os/developer-guide>. [Online; accessed 6-Mar-2014].
- [4] Gerrit code review tool. <https://code.google.com/p/gerrit/>. [Online; accessed 6-Mar-2014].
- [5] Klocwork. <http://www.klocwork.com/>.
- [6] Qt coding style. http://qt-project.org/wiki/Qt_Coding_Style. [Online; accessed 6-Mar-2014].
- [7] Android. Android developer guide. <http://source.android.com/source/life-of-a-patch.html>. [Online; accessed 6-Mar-2014].
- [8] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *International Symposium on Empirical Software Engineering and Measurement (ESEM), 2011*, pages 97–106, Banff, Alberta, Canada, 2011. IEEE.
- [9] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721, San Francisco, CA, USA, 2013. IEEE Press.
- [10] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 931–940, San Francisco, CA, USA, 2013. IEEE Press.
- [11] V. R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.
- [12] A. Bosu and J. Carver. Peer code review to prevent security vulnerabilities: An empirical evaluation. In *2013 IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C)*, pages 229–230, Washington, DC, USA, June 2013.
- [13] A. Bosu and J. C. Carver. Peer code review in open source communities using reviewboard. In *Proceedings of the 4th ACM Wksp. on Evaluation and Usability of Programming Language and Tools*, pages 17–24, Tucson, Arizona, USA, 2012. ACM.
- [14] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 133–142, Baltimore, MD, USA, 2013. IEEE.
- [15] S. Christey and R. Martin. Vulnerability type distributions in CVE, version 1.1. <http://cwe.mitre.org/documents/vuln-trends/index.html>, May 2007.
- [16] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.
- [17] J. Cohen, E. Brown, B. DuRette, and S. Teleki. *Best Kept Secrets of Peer Code Review*. Smart Bear, 2006.
- [18] T. Dinh-Trong and J. Bieman. The freebsd project: a replication case study of open source development. *IEEE Transactions on Software Engineering*, 31(6):481–494, 2005.
- [19] T. Dyba, T. Dingsoyr, and G. K. Hanssen. Applying systematic reviews to diverse study types: An experience report. In *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007.*, pages 225–234, Madrid, Spain, 2007. IEEE.
- [20] M. Fagan. Reviews and inspections. *Software Pioneers—Contributions to Software Engineering*, pages 562–573, 2002.
- [21] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15:182–211, 1976.
- [22] I. Feinerer. Introduction to the tm package text mining in r. <http://cran.r-project.org/web/packages/tm/index.html>, 2013.
- [23] M. Gegick, P. Rotella, and L. Williams. Toward non-security failures as a predictor of security faults and failures. In *Engineering Secure Software and Systems*, pages 135–149. Springer, 2009.
- [24] M. Gegick, L. Williams, J. Osborne, and M. Vouk. Prioritizing software security fortification through code-level metrics. In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 31–38. ACM, 2008.
- [25] P. M. Johnson. Reengineering inspection. *Communications of the ACM*, 41(2):49–52, 1998.
- [26] K. R. Lakhani and R. G. Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. *Perspectives on free and open source software*, 1:3–22, 2005.
- [27] S. Lukins, N. Kraft, and L. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *15th Working Conference on Reverse Engineering, 2008. WCRC '08.*, pages 155–164, Antwerp, Belgium, 2008.
- [28] G. McGraw. Software security. *IEEE Security Privacy*, 2(2):80–83, Mar-Apr 2004.
- [29] G. McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.
- [30] G. McGraw. Automated code review tools for security. *Computer*, 41(12):108–111, Dec. 2008.
- [31] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejada, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74, Baltimore, MD, USA, 2013. IEEE.
- [32] A. Meneely and L. Williams. Secure open source collaboration: an empirical study of linus’ law. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 453–462. ACM, 2009.
- [33] Mitre Corporation. Common weakness enumeration. <http://cwe.mitre.org/>. [Online; accessed 6-Mar-2014].
- [34] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering Methodology*, 11(3):309–346, July 2002.
- [35] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.
- [36] M. Mukadam, C. Bird, and P. C. Rigby. Gerrit software code review data from android. In *Proceedings of the Tenth*

- International Workshop on Mining Software Repositories*, pages 45–48, San Francisco, CA, USA, 2013. IEEE Press.
- [37] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, St. Louis, MO, USA, 2005.
- [38] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.
- [39] OWASP. The open web application security project. <https://www.owasp.org/index.php/Category:Vulnerability>, 2013. [Online; accessed 6-Mar-2014].
- [40] M. F. Porter. Snowball: A language for stemming algorithms. <http://www.tartarus.org/~{}martin/PorterStemmer>, 2001.
- [41] L. J. Pratt, A. C. MacLean, C. D. Knutson, and E. K. Ringger. Cliff Walls: An analysis of monolithic commits using Latent Dirichlet Allocation. In *Open Source Systems: Grounding Research*, pages 282–298. Springer, 2011.
- [42] R. Purushothaman and D. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, June 2005.
- [43] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500, Waikiki, Honolulu, HI, USA, 2011. ACM.
- [44] E. S. Raymond. Homesteading the noosphere. *First Monday*, 3(10), 1998.
- [45] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, Saint Petersburg, Russia, 2013. ACM.
- [46] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.
- [47] Y. Shin and L. Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317, Kaiserslautern, Germany, 2008. ACM.
- [48] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*, volume 1. Addison-Wesley Longman Publishing Co., Inc., 2005.
- [49] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security Privacy*, 3(6):81 – 84, Nov.-Dec. 2005.
- [50] J. Walden, M. Doyle, G. A. Welch, and M. Whelan. Security of open source web applications. In *Proceedings of the 2009 3rd international Symposium on Empirical Software Engineering and Measurement*, pages 545–553, Orlando, FL, USA, 2009. IEEE Computer Society.
- [51] K. E. Wiegers. *Peer reviews in Software: A practical guide*. Addison-Wesley Boston, 2002.
- [52] T. Yamane. Elementary sampling theory. 1967.
- [53] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Pub. Inc., San Francisco, CA, USA, 2005.