Why Security Defects Go Unnoticed during Code Reviews? A Case-Control Study of the Chromium OS Project

Rajshakhar Paul, Asif Kamal Turzo, Amiangshu Bosu
Department of Computer Science
Wayne State University
Detroit, Michigan, USA
{r.paul, asifkamal, amiangshu.bosu}@wayne.edu

Abstract*—Peer code review has been found to be effective in identifying security vulnerabilities. However, despite practicing mandatory code reviews, many Open Source Software (OSS) projects still encounter a large number of post-release security vulnerabilities, as some security defects escape those. Therefore, a project manager may wonder if there was any weakness or inconsistency during a code review that missed a security vulnerability. Answers to this question may help a manager pinpointing areas of concern and taking measures to improve the effectiveness of his/her project's code reviews in identifying security defects. Therefore, this study aims *to identify the factors that differentiate code reviews that successfully identified security defects from those that missed such defects.

With this goal, we conduct a case-control study of Chromium OS project. Using multi-stage semi-automated approaches, we build a dataset of 516 code reviews that successfully identified security defects and 374 code reviews where security defects escaped. The results of our empirical study suggest that the are significant differences between the categories of security defects that are identified and that are missed during code reviews. A logistic regression model fitted on our dataset achieved an AUC score of 0.91 and has identified nine code review attributes that influence identifications of security defects. While time to complete a review, the number of mutual reviews between two developers, and if the review is for a bug fix have positive impacts on vulnerability identification, opposite effects are observed from the number of directories under review, the number of total reviews by a developer, and the total number of prior commits for the file under review.

***Index Terms*—security, code review, vulnerability**

I. Introduction

Peer code review (a.k.a. code review) is a software quality assurance practice of getting a code change inspected by peers before its integration to the main codebase. In addition to improving maintainability of a project and identification of bugs [3], [11], code reviews have been found useful in preventing security vulnerabilities [12], [34]. Therefore, many popular Open Source Software (OSS) projects such as, Chromium, Android, Qt, oVirt, and Mozilla as well as commercial organizations such as, Google, Microsoft, and Facebook have integrated code reviews in their software development pipeline [11], [50]. With mandatory code reviews, many OSS projects (e.g., Android, and Chromium OS) require each and every change to be reviewed and approved by multiple peers [3], [50]. Although mandatory code reviews are preventing a significant number of security defects [12], [44], these projects still report a large number of post-release security defects in the Common Vulnerabilities and Exposure (a.k.a. CVE) database[1]. Therefore, a project manager from such a project may wonder if there was any weakness or inconsistency during a code review that missed a security vulnerability. For example, she/he may want to investigate: i) if reviewers had adequate expertise relevant to a particular change, ii) if reviewers spent adequate time on the reviews, or iii) if the code change was too difficult to understand. Answers to these questions may help a manager pinpointing areas of concern and taking measures to improve the effectiveness of his/her project's code reviews in identifying security defects.

To investigate these questions, this study aims *to identify the factors that differentiate code reviews that successfully identified security defects from those that missed such defects.* Since code reviews can identify vulnerabilities very early in the software development pipeline, security defects identified during code reviews incur significantly less cost, as the longer it takes to detect and fix a security vulnerability, the more that vulnerability will cost [35]. Therefore, improving the effectiveness of code reviews in identifying security defects may reduce the cost of developing a secure software.

With this goal, we conducted a case-control study of the Chromium OS project. Case-control studies, which are common in the medical field, compare two existing groups differing on an outcome [52]. We identified the cases and the controls based on our outcome of interest, namely whether a security defect was identified or escaped during the code review of a vulnerability contributing commit (VCC). Using a keyword-based mining approach followed by manual validations on a dataset of 404,878 Chromium OS code reviews, we identified 516 code reviews that successfully identified security defects. In addition, from the Chromium OS bug repository, we identified 239 security defects that escaped code reviews. Using a modified version of the SZZ algorithm [9]

[1] https://cve.mitre.org/cve/

followed by manual validations, we identified 374 VCCs and corresponding code reviews that approved those changes. Using these two datasets, we conduct an empirical study and answer the following two research questions:

(RQ1): Which categories of security defects are more likely to be missed during code reviews?

Motivation: Since a reviewer primarily relies on his/her knowledge and understanding of the project, some categories of security defects may be more challenging to identify during code reviews than others. The results of this investigation can help a project manager in two ways. First, it will allow a manager to leverage other testing /quality assurance methods that are more effective in identifying categories of vulnerabilities that are more likely to be missed during code reviews. Second, a manager can arrange training materials to educate developers and adopt more effective code review strategies for those categories of security vulnerabilities.

Findings: The results suggest that some categories of vulnerabilities are indeed more difficult to identify during code reviews than others. The identification of a vulnerability that requires an understanding of a few lines of the code context (e.g., unsafe method, calculation of buffer size, and resource release) are more likely to be identified during code reviews. On the other hand, vulnerabilities that require either code execution (e.g., input validation) or understanding of larger code contexts (e.g., resource lifetime, and authentication management) are more likely to remain unidentified.

(RQ2): Which factors influence the identification of security defects during a code review?

Motivation:

Insights obtained from this investigation can help a project manager pinpoint areas of concern and take targeted measures to improve the effectiveness of his/her project's code reviews in identifying security defects.

Findings: We developed a Logistic Regression model based on 18 code review attributes. The model, which achieved an AUC of 0.91, found nine code review attributes that distinguish code reviews that missed a vulnerability from the ones that did not. According to the model, the likelihood of a security defect being identified during code review declines with the increase in the number of directories/files involved in that change. Surprisingly, the likelihood of missing a vulnerability during code reviews increased with a developer's reviewing experience. Vulnerabilities introduced in a bug fixing commit were more likely to be identified than those introduced in a non-bug fix commit.

The primary contributions of this paper are:

- An empirically built and validated dataset of code reviews that either identified or missed security vulnerabilities.
- An empirical investigation of security defects that escaped vs. the ones that are identified during code reviews.
- A logistic regression model to identify relative importance of various factors influencing identification of security defects during code reviews.
- An illustration of conducting a case-control study in the software engineering context.
- We make our script and the dataset publicly available at: https://zenodo.org/record/4539891.

Paper organization: The remainder of this paper is organized as follows. Section II provides a brief background on code reviews and case-control study. Section III details our research methodology. Section IV describes the results of our case-control study. Section V discusses the implications based on the results of this study. Section VI discusses the threats to validity of our findings. Section VII describes related works. Finally, Section VIII provides the future direction of our work and concludes this paper.

II. Background

This section provides a brief background on security vulnerabilities, code reviews, and case-control studies.

A. *Security Vulnerabilities*

A vulnerability is a weakness in a software component that can be exploited by a threat actor, such as an attacker, to perform unauthorized actions within a computer system. Vulnerabilities result mainly from bugs in code which arise due to violations of secure coding practices, lack of web security expertise, bad system design, or poor implementation quality. Hundreds of types of security vulnerabilities can occur in code, design, or system architecture. The security community uses Common Weakness Enumerations (CWE) [41] to provide an extensive catalog of those vulnerabilities.

B. *Code reviews*

Compared with the traditional heavy-weight inspection process, peer code review is more light-weight, informal, tool-based, and used regularly in practice [3]. In addition to their positive effects on software quality in general, Code review can be an important practice for detecting and fixing security bugs early in a software development lifecycle [34]. For example, an expert reviewer can identify potentially vulnerable code and help the author to fix the vulnerability or to abandon the code. Peer code review can also identify attempts to insert malicious code. Software development organizations, both OSS and commercial, have been increasingly adopting tools to manage the peer code review process [50]. A tool-based code review starts, when an author creates a patch-set (i.e. all files added or modified in a single revision), along with a description of the changes, and submits that information to a code review tool. After reviewers are assigned, the code review tool then notifies selected reviewers regarding the incoming request. During a review, the tools may highlight the changes between revisions in a side-by-side display. The review tool also facilitates communication between the reviewers and the author in the form of review comments, which may focus on a particular code segment or the entire patchset. By uploading a new patchset to address the review comments, the author can initiate a new review iteration. This review cycle repeats until either the reviewers approve the change or the author abandons. If the reviewers approve the changes, then the author commits the patchset or asks a project committer to integrate the patchset to the project repository.

C. Case-Control Study

Case-control studies, which are widely used in the medical field, is a type of observational study, where subjects are selected based on an outcome of interest, to identify factors that may contribute to a medical condition by comparing those with the disease or outcome (cases) against a very similar group of subjects who do not have that disease or outcome (controls) [31]. A case-control study is always retrospective, since it starts with an outcome and then traces back to investigate exposures. However, it is essential that case inclusion criteria are clearly defined to ensure that all cases included in the study are based on the same diagnostic criteria. To measure the strength of the association between a given exposure and an outcome of interest, researches who conduct case-control studies usually use Odds Ratio (OR), which represents the odds that an outcome will occur given an exposure, compared to the odds of the outcome occurring in the absence of that exposure [52].

Although case control studies are predominantly used in the medical domain, other domains have also used this research method. In the SE domain, Allodi and Massaci conducted case-control studies to investigate vulnerability severities and their exploits [1]. In a retrospective study, where two groups naturally emerge based on an outcome, the case-control study framework can provide researchers guidelines in selecting variables, analyzing data, and reporting results. We believe that a case-control design is appropriate for this study, since we are conducting a retrospective study, where two groups differ based on an outcome. In our design, each of the selected cases is a vulnerability contributing commit forming two groups: 1) cases–vulnerabilities identified during code reviews and 2) controls –vulnerabilities escaped code reviews.

III. Research Methodology

Our research methodology focused on identifying vulnerability contributing commits that went through code reviews and had its' security defects either getting identified or escaping. In the following subsections, we detail our research methodology.

A. Project Selection

For this study, we select the Chromium OS project for the following five reasons– (i) it is one of the most popular OSS projects, (ii) it is a large-scale matured project containing more than 41.7 million Source Lines of Code (SLOC) [25]. (iii) it has been conducting tool-based code reviews for almost a decade, (iv) it maintains security advisories[2] to provide regular updates on identified security vulnerabilities, and (v) it has been subject to prior studies on security vulnerabilities [14], [30], [38], [39], [43].

B. Data Mining

The code review repositories of the Chromium OS project is managed by Gerrit[3] and is publicly available at: https://chromium-review.googlesource.com/. We wrote a Java application to access Gerrit's REST API to mine all the publicly available code reviews for the project and store the data in a MySQL database. Overall, we mined 404,878 code review requests spanning March 2011 to March 2019. Using an approach similar to Bosu et al. [10], we filtered the bot accounts, using a set of keywords (e.g., 'bot', 'CI', 'Jenkins', 'build', 'auto', and 'travis') followed by manual validations, to exclude the comments not written by humans. To identify whether multiple accounts belong to a single person, we follow a similar approach as Bird *et al.* [8], where we use the Levenshtein distance between two names to identify similar names. If our manual reviews of the associated accounts suggest that those belong to the same person, we merge those to a single account.

C. Building a dataset of cases (i.e. vulnerabilities identified during code reviews)

We adopted a keyword-based semi-automated mining approach, which is similar to the strategy used by Bosu *et* al. [12], to build a dataset of vulnerabilities identified during code reviews. Our keyword-based mining was based on the following three steps:

(Step I) Database search: We queried our MySQL database of Chromium OS code reviews to select review comments that contain at least one of the 105 security-related keywords (Table I). Bosu *et* al. [12] empirically developed and validated a list of 52 keywords to mine code review comments associated with the 10 common types of security vulnerabilities. Using Bosu *et* al's keyword list as our starting point, we added additional 53 keywords to this list based on the NIST glossary of security terms [27]. Our database search identified 7,572 code review comments that included at least one of these 105 keywords (Table I).

(Step II) Preliminary filtering: Two of the authors independently audited each code review comment identified during the database search to eliminate any reviews that clearly did not raise a security concern. We excluded a review comment in this step only if both auditors independently determined that the comment does not refer to a security issue. To illustrate the process let's examine two code review comments with the same keyword 'overflow'. The first comment– "*no check for overflow here?*" potentially raises a concern regarding an unchecked integer overflow and therefore was included for a detailed inspection. While the second comment –"*I'm not sure but can specifying overflow: hidden; to a container hide scroll bars?*" seems to be related to UI rendering and was discarded during this step. This step discarded 6,235 comments

[2] https://www.chromium.org/chromium-os/security-advisories

[3] https://www.gerritcodereview.com/

TABLE I
KEYWORDS TO MINE CODE REVIEWS THAT IDENTIFY SECURITY DEFECT

Vulnerability Type	**CWE ID**	**Keywords***
Race Condition	362 - 368	race, racy
Buffer Overflow	120 - 127	buffer, overflow, stack, *strcpy*, *strcat*, *strtok*, *gets*, *makepath*, *splitpath*, *heap*, *strlen*
Integer Overflow	190, 191, 680	integer, overflow, signedness, widthness, underflow
Improper Access	22, 264, 269, 276, 281 -290	improper, unauthenticated, gain access, permission, *hijack*, *authenticate*, *privilege*, *forensic*, *hacker*, *root*
Cross Site Scripting (XSS)	79 - 87	cross site, CSS, XSS, *malform*,
Denial of Service (DoS) / Crash	248, 400 - 406, 754, 755	denial service, DOS, DDOS, 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
Encryption	310, 311, 320-327	*encrypt*, *decrypt*, *password*, *cipher*, *trust*, *checksum*, *nonce*, *salt*
Common keywords	-	security, vulnerability, vulnerable, hole, exploit, attack, bypass, backdoor, threat, expose, breach, violate, fatal, blacklist, overrun, insecure, *scare*, *scary*, *conflict*, *trojan*, *firewall*, *spyware*, *adware*, *virus*, *ransom*, *malware*, *malicious*, *risk*, *dangling*, *unsafe*, *leak*, *steal* , *worm*, *phishing*, *cve*, *cwe*, *collusion*, *covert*, *mitm*, *sniffer*, *quarantine*, *scam*, *spam*, *spoof*, *tamper*, *zombie*

*Approximately half of the keywords in this list are adopted from the prior study of Bosu et al. [12]. Keywords in *italic* are our additions to this list.

and retained the remaining 1,337 comments for a detailed inspection.

(Step III) Detailed Inspection: In this step, two of the authors independently inspected the 1,337 review comments identified from the previous step, any subsequent discussion included in each review, and associated code contexts to determine whether a security defect was identified in each review. If any vulnerability is confirmed, the inspectors also classified it according to the CWE specification [41]. Similar to Bosu et al. [12], we considered a code change vulnerable only if: (a) a reviewer indicated potential vulnerabilities, (b) our manual analysis of the associated code context found the code to be potentially vulnerable, and (c) the code author either explicitly acknowledged the presence of the vulnerability through a response (e.g., '*Good catch*', '*Oops!*') or implicitly acknowledged it by making the recommended changes in a subsequent patch. Agreement between the two inspectors was computed using Cohen's Kappa (κ) [15], which was measured as 0.94 (almost perfect[4]). Conflicting labels were resolved during a discussion session. At the end of this step, we identified total 516 code reviews that successfully identified security vulnerabilities.

D. Building a dataset of controls (i.e. vulnerabilities escaped during code reviews)

We searched the Monorail-based bug tracking system hosted at: https://bugs.chromium.org/, to identify a list of security defects for the Chromium OS project. We used the bug tracker instead of the CVE database, since the bug tracker includes more detailed information for each security defect (e.g., link to fixing commit and link to Gerrit where the fix was code reviewed). Moreover, some of the security defects may not be reported in the CVE, if it was identified during testing prior to its public release. We used the following five-step approach to build this dataset. We also illustrate this process using an example security defect: *#935175*.

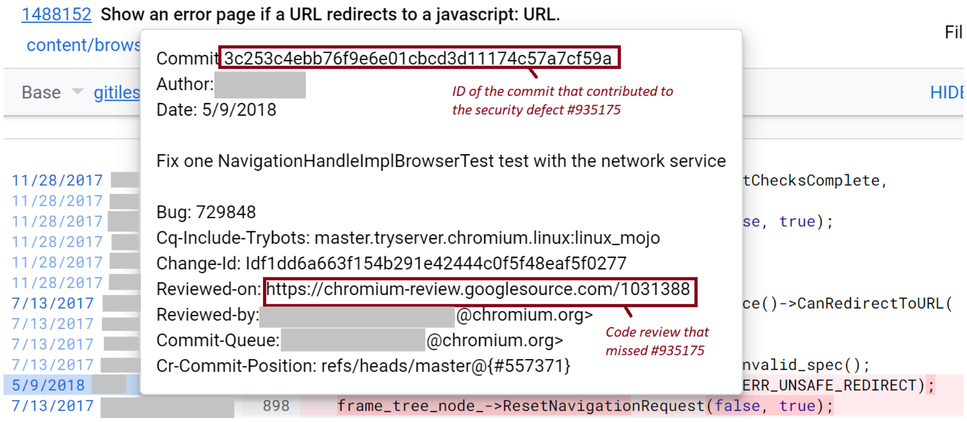
(Step I) Custom search: We use a custom search (i.e., (`Type=Bug-Security status:Fixed OS=Chrome`), to filter security defects for the Chromium OS projects with the status as 'Fixed'. Our search result identified total 591 security defects. We exported the list of defects as a comma-separated values(i.e., csv) file, where each issue is associated with a unique ID.

(Step II) Identifying vulnerability fixing commit: The Monorail page for each 'Fixed' issue includes detailed information (e.g., commit_id, owner, review URL, list of modified files, and reviewer) regarding its fix. For example, http://crbug.com/935175 details the information for the security defect *#935175* including the ID of the vulnerability fixing commit (i.e. ' `56b512399a5c2221ba4812f5170f3f8dc352cd74`'). We wrote a Python script to automate the extraction of the review URLs and commit_ids for each security defect identified in Step I. Finally, we excluded the security fixes that were not reviewed on Chromium OS's Gerrit repository (e.g., third-party libraries). At the end of this step, we were left with 239 security defects and its' corresponding fixes.

(Step III) Identifying vulnerability contributing commit(s): We adopted the modified version of the SZZ algorithm [9] to identify the vulnerability introducing commits from the vulnerability fixing commits identified in Step II. Our modified SZZ algorithm uses the `git blame` and `git bisect` subcommands and is adopted based on the approaches followed in two prior studies [38], [49] on VCCs. For each line in a given file, the `git blame` subcommand names the commit that last commit_id that modified it. The heuristics behind our custom SZZ are as following:

1) Ignore changes in documentations such as release notes or change logs.
2) For each deleted / modified, blame the line that was deleted / modified, since if a fix needed to change a line, that often means that it was part of the vulnerability.
3) For every continuous block of code inserted in the bug fixing commit, blame the lines before and after the block, since security fixes are often done by adding extra checks, often right before an access or after a function call.
4) If multiple commits are marked based on the above steps, mark commits as VCCs based on higher amount of lines

[4] 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



1488152 Show an error page if a URL redirects to a javascript: URL.

Commit 3c253c4ebb76f9e6e01cbcd3d11174c57a7cf59a
Author: [redacted]
Date: 5/9/2018

Fix one NavigationHandleImplBrowserTest test with the network service

Bug: 729848
Cq-Include-Trybots: master.tryserver.chromium.linux:linux_mojo
Change-Id: Idf1dd6a663f154b291e42444c0f5f48eaf5f0277
Reviewed-on: https://chromium-review.googlesource.com/1031388
Reviewed-by: [redacted]@chromium.org>
Commit-Queue: [redacted]@chromium.org>
Cr-Commit-Position: refs/heads/master@{#557371}

ID of the commit that contributed to the security defect #935175

Code review that missed #935175

```
11/28/2017 [redacted]                tChecksComplete,
11/28/2017 [redacted]
11/28/2017 [redacted]                se, true);
11/28/2017 [redacted]
11/28/2017 [redacted]
11/28/2017 [redacted]
7/13/2017  [redacted]                ce()->CanRedirectToURL(
7/13/2017  [redacted]
7/13/2017  [redacted]
7/13/2017  [redacted]                nvalid_spec();
5/9/2018   [redacted]                ERR_UNSAFE_REDIRECT);
7/13/2017  [redacted]   898   frame_tree_node_->ResetNavigationRequest(false, true);
```

Fig. 1. A vulnerability contributing commit (VCC) for the security defect *#935175* and the code review that missed it

TABLE II
ATTRIBUTES OF A CODE REVIEW THAT MAY INFLUENCE IDENTIFICATION OF A VULNERABILITY

Type	Name	Definition	Rationale
Location	Number of files under review	Number of files under review in a review request.	Changes that involve greater number of files are more likely to be defect-prone [28], yet more time-consuming to review.
	Number of directory under review	Number of directory where files have been modified in a review request.	If the developers group multiple separate changes into a single commit, the review of those comprehensive changes could be harder.
	Code churn	Number of lines added / modified / deleted in a code review.	Larger changes are more likely to have vulnerability [12], [45], [46] and require more time to comprehend.
	Lines of code	Numbers of lines of code in the file before fix.	Larger components are more difficult to understand.
	Complexity	McCabe's Cyclomatic Complexity [33].	Difficulty to comprehend a file increases with its cyclomatic complexity.
	is_bug_fix	Code review request that is submitted to fix a bug	A bug fix review request may draw additional attention from the reviewers, as bugs often foreshadow vulnerabilities [14]
Participant	Author's coding experience	Number of code commits the author has submitted (i.e., both accepted and rejected) prior to this commit.	Experienced authors' code changes may be subject to less scrutiny and therefore may miss vulnerabilities during reviews.
	Reviewer's reviewing experience	Number of code reviews that a developer has participated as a reviewer (i.e., code not committed by him/her) prior to this commit.	Experienced reviewers may be more likely to spot security concerns.
	Reviewer's coding experience	Number of code commits that a reviewer has submitted (i.e., both accepted and rejected) prior to this commit.	Experienced developers provide more useful feedback during code review [13] and may have more security knowledge.
Review process	Review time	The time from the beginning to the end of the review process. We define the review process to be complete when the patchset is 'Merged' to the main project branch or is 'Abandoned'.	A cursory code review is more likely to miss security defects that require thorough reviews.
	Number of reviewers involved (NR_f)	Number of reviewers involved in reviewing file f	As Linus's law suggest, the more eyeballs, the less likelihood of a defect remaining unnoticed.
Historical	Review ratio ($RR_{a,f}$)	The ratio between the number of prior reviews from developer a to a file f and the total number of prior reviews to that file. If the developer a participated in i of the r prior reviews in file f then: $RR_{a,f}$ = $\frac{i}{r}$	A developer who has reviewed a particular file more may have better understanding of its design.
	Commit ratio ($CR_{a,f}$)	The ratio between the number of commits to a file f by author a and the total number of commits to that file. If author a makes i of the c prior commits then $CR_{a,f}$ = $\frac{i}{c}$	A developer who makes frequent changes in a file may have better understanding of its design
	Weighted recent commits ($RC_{a,f}$)	If a file f has total n prior commits and author a makes three of three of the prior n commits (e.g., i, j, k), where n denotes the latest commit, then: $RC_{a,f}$ = $\frac{(i+j+k)}{(1+2+3+...+n)}$ = $\frac{2(i+j+k)}{n(n+1)}$	A developer who makes recent commits may have better understanding about the current design.
	Total commit	Total number of commits made on the current file	Files that have too many prior commits might require extra attention from the reviewers.
	Mutual reviews	Number of reviews performed by the current reviewer and author	Better understanding about the author's coding style might help the reviewer to investigate defects.
	Number of review comments	Total number of review comments in the current file	Higher number of review comments indicate the file has gone through a more detailed review.
	File ownership ($FO_{a,f}$)	The ratio between the number of lines modified by a developer and total number of lines in that file. If developer a writes i of total n lines in file f, then $FO_{a,f}$ = $\frac{i}{n}$	The owner of a file may be better suited to review that file.

until at least 80% lines are accounted for.

We manually inspect each of the VCCs identified by our modified SZZ algorithm as well as corresponding vulnerability fixing commits to exclude unrelated commits or include additional relevant commits. At the end of this step, we identified total 374 VCCs. Figure 1 shows a VCC for the security defect *#935175* identified through this process.

(Step IV) Identifying code reviews that approved VCCs: A git repository mirror for the Chromium OS project is hosted at https://chromium.googlesource.com/ with a gitiles[5] based frontend. We used the REST API of gitiles to query this repository to download commit logs for each VCC identified in the previous step. Using a REGEX parser, we extract the URLs of the code review requests that approved VCCs identified in Step III. For example, Figure 1 also includes the URL of the code review that missed the security defect *#935175*. At the end of this step, we identified total 374 code reviews that approved our list of VCCs.

(Step V) CWE classification of the VCCs: 124 out of the 374 VCCs in our dataset had a CVE reported in the NIST NVD database[6] For example `CVE-2019-5794` corresponds to the security defect *#935175*. For such VCCs, we obtained the CWE classification from the NVD database. For example, NVD classifies *#935175* as a 'CWE-20: Improper Input Validation'. For the remaining 250 VCCs, two of the authors independently inspected each VCC as well as its fixing commits to understand the coding mistake and classify it according to the CWE specification. Conflicting labels were resolved through discussions.

E. Attribute Collection

To answer the research questions motivating this study, we computed 18 attributes for each of the 890 code reviews (i.e. 516 cases + 374 controls). Majority of the attributes selected in this study have been also used in prior studies investigating the relationship between software quality and code review attributes [28], [29], [36], [55]. Table II presents the list of our attributes with a brief description and rationale behind the inclusion of each attribute to investigate our research objectives. Those attributes are grouped into four categories: 1) vulnerability location, 2) participant characteristics, 3) review process, 4) historical measures. We use several Python scripts and SQL queries to calculate those attributes from our curated dataset and our MySQL database of Chromium OS code reviews.

IV. Results

Following subsections detail the results of the two research question introduced in the Section I based on our analyses of the collected dataset.

A. RQ1: Which categories of security defects are more likely to be missed during code reviews?

For both identified and escaped security defects cases, we either obtained a CWE classification from the NVD database or manually assign one for those without any NVD reference. The 890 VCCs (i.e., both identified and escaped cases) in our dataset represented 86 categories of CWEs. However, for the simplicity of our analysis, we decreased the number of distinct CWE categories by combining similar categories of CWEs into a higher level category. The CWE specification already provides a hierarchical categorization scheme[7] to represent the relationship between different categories of weaknesses. For example, both CWE-190 (Integer Overflow or Wraparound) and CWE-468 (Incorrect Pointer Scaling) belong to the higher level category: CWE-682 (Incorrect Calculation). Using the higher level categories from the CWE specification [41], we reduce the number of distinct CWE types in our dataset to 15. During this higher level classification, we also ensured no common descendants among these final 15 categories. Table III shows the fifteen CWE categories represented in our dataset, their definitions, and both the number and ratios of identified /escaped cases, in a descending order based on their total number of appearances.

The results of a Chi-Square (χ^2) test suggest that some categories of CWEs are significantly (χ^2=491.69, $p-value < 0.001$) more likely to remain undetected during code reviews than the others. Chromium OS reviewers were the most efficient in identifying security defects due to '*CWE-676: Use of potentially dangerous function*'. For example, following C functions are `strcpy()`, `strcat()`, `strlen()`, `strcmp()`, `sprintf()` unsafe as they do not check for buffer length and may overwrite memory zone adjacent to the intended destination. As the identification of a CWE-676 is relatively simple and does not require much understanding of the associated context, no occurrences of dangerous functions escaped code reviews. Reviewers were also highly efficient in identifying security defects due to '*CWE-404: Improper resource shut down or release*' that can lead to resource leakage. '*CWE 682: Incorrect calculation*', which includes calculation of buffer size and unsecured mathematical operation (i.e., large addition/multiplication or divide by zero), were also more likely to be identified during code reviews (≈80%). The other categories of CWEs that were more likely to be identified during code reviews include: improper exception handling (CWE-703) and synchronization mistakes (i.e., CWE- 662, and CWE-362).

On the other hand, Chromium OS reviewers were the least effective in identifying security defects due to insufficient verification of data authenticity (CWE-345), as all such occurrences remained undetected. Insufficient verification of data can lead to an application accepting invalid data. Although Improper input validations (CWE-20) were frequent occurrences (i.e., 72), those remained undetected during ≈88% code reviews. Improper input validation can lead to many

[5] https://gerrit.googlesource.com/gitiles/

[6] https://nvd.nist.gov/

[7] https://cwe.mitre.org/data/graphs/1000.html

critical problems such as uncontrolled memory allocation and SQL injection. Approximately 88% security defects caused by improper access control (CWE-284) also remained undetected as reviewers were less effective in identifying security issues due to improper authorization and authentication, and improper user management. The other categories of CWEs that were more likely to remain unidentified during code reviews include: operation on a resource after expiration or release (CWE-672) and exposure of resources to wrong spheres (CWE-668).

Our manual examinations of the characteristics of these CWE categories suggest that security defects that can be identified based on a few lines of the code context (e.g., unsafe method, calculation of buffer size, and resource release) are more likely to be identified during code reviews. On the other hand, Chromium OS reviewers were more likely to miss CWEs requiring either code execution (e.g., input validation) or understanding of larger code contexts (e.g., resource lifetime, and authentication management).

Finding 1: *The likelihood of a security defect's identification during code reviews depends on its CWE category.*

Observation 1(A): *Security defects related to the synchronization of multi-threaded application, calculation of variable and buffer size, exception handling, resource release, and usage of prohibited functions were more likely to be identified during code reviews.*

Observation 1(B): *Security defects related to the user input neutralization, access control, authorization and authentication management, resource lifetime, information exposure, and datatype conversion were more likely to remain undetected during code reviews.*

B. RQ2: Which factors influence the identification of security defects during a code review?

To investigate this research question, we developed a logistic regression model. Logistic regression is very efficient in predicting a binary response variable based one or more explanatory variables [7]. In this study, we use the factors described in the Table II as our explanatory variables, while our response variable is a boolean that is set to *TRUE*, if a code review's security defect was identified by reviewer(s) and *FALSE* otherwise.

Being motivated by recent Software Engineering studies [37], [55], we adopt the model construction and analysis approach of Harrell Jr. [22], which allows us to model nonlinear relationships between the dependent and explanatory variables more accurately. The following subsections describe our model construction and evaluation steps.

1) Correlation and Redundancy Analysis: If the explanatory variables to construct a model are highly correlated with each other, they can generate a overfitted model. Following, Sarle's VARCLUS (Variable Clustering) procedure [51], we use the Spearman's rank-order correlation test (ρ) [54] to determine highly correlated explanatory variables and construct a hierarchical representation of the variable clusters (Figure 2). We retain only one variable from each cluster of highly correlated explanatory variables. We use $|\rho \geq 0.7|$ as our threshold, since it has been recommended as the threshold for high correlation [24] and has been used as the threshold in prior SE studies [37], [55].

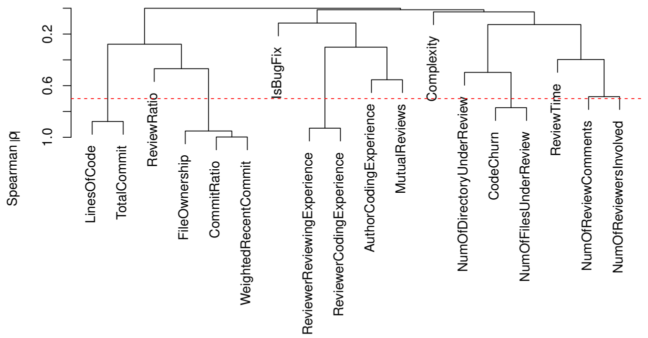


Fig. 2. Hierarchical clustering of explanatory variables according to Spearman's $|\rho|$ and Sarle's VARCLUS. The dashed line indicates the high correlation coefficient threshold ($|\rho| = 0.7$)

We found four clusters of explanatory variables that have $|\rho| > 0.7$ – (1) total lines of code (*totalLOC*) and number of commits (*totalCommit*), (2) commit ratio, weighted recent commit, and file ownership, (3) reviewer's coding experience and reviewer's reviewing experience, (4) amount of code churn, directory under review, and number of files under review. From the first cluster, we select total number of commit in the file. From the second cluster, we select file ownership. From the third and fourth cluster, we select reviewer's reviewing experience and number of directory under review respectively. Despite not being highly correlated, some explanatory variables can still be redundant. Since redundant variables can affect the modelled relationship between explanatory and response variables, we use the `redun` function of the `rms R` package with the threshold $R^2 \geq 0.9$ [19] to identify potential redundant factors among the remaining 12 variables and found none.

2) Degrees of Freedom Allocation: A model may be overfitted, if we allocate degrees of freedom more than a dataset can support (i.e., number of explanatory variables that the dataset can support). To minimize this risk, we estimate the budget for degrees of freedom allocation before fitting our model. As suggested by Harrell Jr. [22], we consider the budget for degrees of freedom to be $\frac{min(T,F)}{15}$, where T represents the number of rows in the dataset where the response variable is set to TRUE and F represents the number of rows in the dataset where the response variable is set to FALSE. Using this formula, we compute our budget for degrees of freedom = 24, since our dataset has 516 TRUE instances and 374 FALSE instances.

For maximum effectiveness, we allocate this budget among

TABLE III
Distribution of Chromium OS CWEs identified /escaped code reviews

CWE ID	CWE Definition	#Identified	%Identified	#Escaped	%Escaped
662	Improper Synchronization	109	89.34	13	10.66
362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	73	87.95	10	12.05
682	Incorrect Calculation	65	79.27	17	20.73
20	Improper Input Validation	9	11.11	72	88.89
703	Improper Check or Handling of Exceptional Conditions	58	72.50	22	27.50
404	Improper Resource Shutdown or Release	71	94.67	4	5.33
284	Improper Access Control	8	12.12	58	87.88
672	Operation on a Resource after Expiration or Release	3	4.62	62	95.38
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	42	72.41	16	27.59
676	Use of Potentially Dangerous Function	53	100.0	0	0.0
668	Exposure of Resource to Wrong Sphere	15	30.0	35	70.0
704	Incorrect Type Conversion or Cast	1	5.88	16	94.12
345	Insufficient Verification of Data Authenticity	0	0.0	13	100.0
665	Improper Initialization	5	50.0	5	50.0
19	Data Processing Errors	0	0.0	10	100.0

all the survived explanatory variables in such a way that the variables that have more explanatory powers (i.e., explanatory variables that have more potential for sharing nonlinear relationship with the response variable) to be allocated with higher degrees of freedom than the explanatory variables that have less explanatory powers. To measure this potential, we compute Spearman rank correlations (ρ^2) between the dependent variable and each of the 12 surviving explanatory variables (Figure 3). Based on the results of this analysis, we split the explanatory variables into two groups– (1) we allocate three degrees of freedom to three variables, i.e., *number of directory under review*, *review ratio*, and *reviewer's reviewing experience*, and (2) we allocate one degree of freedom for the remaining nine variables. Although *isBugFix* has higher potential than *reviewer's reviewing experience*, we cannot assign more than one degree of freedom for *isBugFix* as it is dichotomous. As suggested by Harrell Jr. [22], we limit the maximum allocated degree of freedom for an explanatory variable below five to minimize the risk of overfitting.

3) Logistic Regression Model Construction: After eliminating highly correlated explanatory variables and allocating appropriate degrees of freedom to the surviving explanatory variables, we fit a logistic regression model using our dataset. We use the `rcs` function of the `rms R` package [19] to fit the allocated degrees of freedom to the explanatory variables.

4) Model Analysis: After model construction, we analyze the fitted model to identify the relationship between the response variable and each of the explanatory variables. We describe each step of our model analysis in the following.

Assessment of explanatory ability and model stability: To assess the performance of our model, we use Area Under the Receiver Operating Characteristic (AUC) curve [21]. Our model achieves an AUC of 0.914. To estimate how well the model fits our dataset, we calculate Nagelkerke's Pseudo R^2 [47][8]. Our model achieves a R^2 value of 0.6375, which is considered to be a good fit [47].

Power of explanatory variables estimation: We use the Wald statistics (Wald χ^2) to estimate the impact of each explanatory variable on the performance our model. We use the `anova` function of the `rms R` package to estimate the relative contribution (Wald χ^2) and statistical significance (p) of each explanatory variable to the model. The larger the Wald χ^2 value is, the more explanatory power the variable wields on our model. The results of our Wald χ^2 tests (Table IV) suggest that *number of directory under review* wields the highest predictive power on the fitted model. *ReviewRatio*, *isBugFix*, *ReviewerReviewingExperience*, and *TotalCommit* are the next four most significant contributors. *Number of review comments*, *number of mutual reviews*, *cyclomatic complexity of the file*, and *review time* also wield significant explanatory powers. However, *experience of code author*, *number of reviewers involved in the review process*, and *proportion of ownership of the file* do not contribute significantly on the fitted model.

We use the `summary` function of the `RMS R` package to analyze our model fit summary. Table IV also shows the contributions of each explanatory variable to fit our model using the '*deviance reduced by*' values. For a generalized linear model, deviance can be used to estimate goodness / badness of fit. A higher value of residual deviance indicates worse fit and a lower value indicates the opposite. A model with a perfect fit would have zero residual deviance. The `NULL` deviance value, which indicates how well the response variable is predicted by a model that includes only one intercept (i.e., the grand mean), is estimated as 1211.05 for our dataset. The deviance of a fitted model decreases once we add explanatory variables. This decrement of residual

[8] For Ordinary Least Square (OLS) regressions, $Adjusted\ R^2$ is used to measure a model's goodness of fit. Since it is difficult to compute $Adjusted\ R^2$ for a logistic regression model, the $Pseudo\ R^2$ is commonly used to measure its goodness of fit. The advantage of using Nagelkerke's $Pseudo\ R^2$ is that it's range is similar to the $Adjusted\ R^2$ range used for OLS regressions [53].

deviance would higher for a variable with higher predictive power than for a variable with lower predictive power. For example, the explanatory variable "*directory_under_review*", which has the highest predictive power, reduces the residual deviance by 195.701 with a loss of three degrees of freedom. We can imply that the variable "*directory_under_review*" adds $\frac{195.701}{1211.05} \times 100\% = 16.16\%$ explanatory power to fit the model. Similarly, "*reviewers_experience*" reduces the residual deviance by 104.104 with a loss of three degrees of freedom. Therefore, "*reviewers_experience*" adds $\frac{104.104}{1211.05} \times 100\% = 8.6\%$ explanatory power to fit the model. Overall, our explanatory variables decrease the deviance by 571.73 with a loss of 17 degrees of freedom. Hence, we can imply that our explanatory variables add $\frac{571.74}{1211.05} \times 100\% = 47.21\%$ explanatory power to fit the model which can be considered as a significant improvement over the null model.

Examination of variables in relation to response: Since Odds Ratio (OR) is recommended to measure the strength of relationship between an explanatory variable and the outcome [52], we compute the OR of each explanatory variable in our fitted model (Table IV) using 95% confidence interval. In this study, the OR of an explanatory variable implies how the probability of getting a true outcome (i.e., a vulnerability getting identified) increases with a unit change of that variable. Therefore, an explanatory variable with $OR > 1$ would increase the probability of a security defect getting identified during code reviews with its increment and vice versa. Since the explanatory variables used in our model have varying ranges (i.e., while 'number of directory under review' varies between from 1 to 10, the 'reviewing experience' varies between 0 to several hundreds), we cannot draw a generic conclusion by comparing the numeric OR value of an explanatory variable against the OR of another variable that has a different range.

Table IV shows that the OR of '*number of directory under review*' is 0.76 (i.e. < 1), indicating that, if the '*number of directory under review*' increases, a code review is more likely to miss a security defect. On the other hand, the odds ratio of the variables *ReviewRatio* and *IsBugFix* are well above 1, which imply that if the review request is marked as a bug fix commit or the review conducts a significant number of prior review to that file, the security defect is more likely to be identified during code review. Since *IsBugFix* is a dichotomous variable, interpretation of its OR value (4.55) is straightforward. It indicates that vulnerabilities in a bug fix commit were 4.55 times more likely to be identified during code reviews than a non-bug fix commit. The results also suggest positive impact of review time on vulnerability identification. Surprisingly, the overall reviewing experience of a developer does not increase his/her ability to identify security defects.

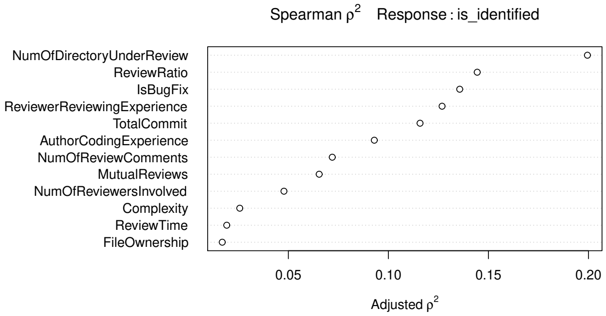


Fig. 3. Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the response. The larger values of ρ^2 indicate higher potential for a nonlinear relationship.

Finding 2: *Our model has identified nine code review factors that significantly differ between code reviews that successfully identified security defects and those failed. Number of directories impacted by a code change has the most predictive power among those nine factors.*

Observation 2(A): *The probability of a vulnerability getting identified during a code review decreases with the increase in number of directories, number of prior reviews by the reviewer, number of prior commits in the file, and number of review comments authored on a file during the current review cycle.*

Observation 2(B): *The probability of a vulnerability getting identified during a code review increases with review time, number of mutual reviews between the code author and a reviewer, cyclomatic complexity of the file under review, if the change belongs to a bug fix, and a reviewer's number of priors review with the file.*

V. Implications

In this section, we describe the implications of our findings.

A. Findings from RQ1

Table III suggests that reviewers detect CWE-662 and CWE-362 in most of the cases. Both of the CWEs are related to the security issues for multi-threaded applications (Improper synchronization and race condition). Hence, we can infer that Chromium OS developers have adequate expertise in securing multi-threaded programs. Developers are also detecting issues related to improper calculation of array buffer or variable size in most of the cases which can overcome the possibility of potential buffer and/or integer overflow/underflow. However, identifying security issues with user input sanitization remains a concern. Most of the issues related to improper input validation have been escaped during code review which can lead to security vulnerabilities such as SQL injection attack, cross-site scripting attach, and IDN holograph attack. Chromium OS project manager may tackle this problem in two possible ways. First, they may leverage additional quality assurance

TABLE IV

EXPLANATORY POWERS OF THE CODE REVIEW ATTRIBUTES TO PREDICT THE LIKELIHOOD OF A VULNERABILITY TO BE IDENTIFIED DURING CODE REVIEWS

	Allocated D.F.	**Deviance**	**Residual Deviance**	**Deviance Reduced By (%)**	**Odds Ratio**	**Pr(>Chi)**
NULL			1211.05			
NumOfDirectoryUnderReview	3	195.70	1015.35	16.16	0.76	<0.001***
ReviewerReviewingExperience	3	104.10	911.25	8.60	0.99	<0.001***
ReviewRatio	3	85.49	825.76	7.06	1.83	<0.001***
IsBugFix	1	67.14	758.62	5.54	4.55	<0.001***
TotalCommit	1	40.03	718.59	3.30	0.97	<0.001***
NumOfReviewComments	1	26.56	692.03	2.19	0.98	<0.001***
ReviewTime	1	23.86	668.17	1.97	1.01	<0.001***
MutualReviews	1	14.95	653.22	1.23	1.01	<0.001***
Complexity	1	12.17	641.05	1.01	1.12	<0.001***
AuthorCodingExperience	1	0.95	640.10	0.08	0.99	0.33
FileOwnership	1	0.60	639.50	0.05	1.52	0.44
NumOfReviewersInvolved	1	0.18	639.32	0.02	1.04	0.67
Total	**18**	**571.73**		**47.21%**		

Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:
* p <0.05; ** p <0.01; *** p <0.001;

practices, such as static analysis, fuzzy testing that are known to be effective in identifying these categories of vulnerabilities. Second, education / training materials may be provided to reviewers to improve their knowledge regard these CWEs.

B. Findings from RQ2

Herzig and Zeller find that when developers commit loosely related code changes that affect all related modules, the likelihood of introducing bug increases [23]. Such code changes are termed as *tangled* code changes. Our study also finds that if the code change affects multiple directories, the security defect is more likely to escape code review. Reviewing tangled code changes can be challenging due to difficulties in comprehension. To tackle this issue we recommend: 1) trying to avoid code changes dispersed across a large number of directories, when possible, 2) spending additional time during such changes as our results also suggest positive impact of review time on vulnerability identification, and 3) integrate a tool, such as the one proposed by Barnett et al. [4] to help reviewers navigate tangled code changes.

Our results also suggest that Chromium OS reviewers, who have participated in higher number of code reviews for were less likely to identify security defects. There may be several possible explanations for this result. First, developers who participates in a large number of reviews may become less cautious (i.e., review fatigue) and miss security defects. Second, developers who review lot of changes may have to spend less time per review, as code reviews are considered as secondary responsibilities in most projects. Therefore, such developers become less effective in identifying security defects due to hasty reviews. Finally, identification of security defects may require special skillsets that do not increase a developer's participation in non-security code reviews. While we do not have a definite explanation, we would recommend project managers to be more aware of 'review fatigue' and avoid overburdening a person with a larger number of reviews.

The likelihood of file's vulnerability escaping increases with the total number of commit it has encountered during its lifetime. A file with higher number of commits indicates more frequent changes in that file than others due to bugs or design changes. Since bugs often foreshadow vulnerabilities [14], developers should be more cautious while reviewing files that frequently go through modifications.

Interestingly, if a code change is marked as a bug fix, developers are more likely to identify security defects (if exists) during code reviews, which suggests extra cautions during such reviews. Therefore, an automated model may be used to predict and assign tags (e.g., 'security critical') to code changes that are more likely to include vulnerabilities to draw reviewers' attentions and seek their cautiousness.

Unsurprisingly, the likelihood of a security defect getting identified increases with review time (i.e., time to conclude a review). Although, taking too much time to complete a review would slow the development process, reviewers should make a trade-off between time and careful inspection, and try to avoid rushing reviews of security critical changes. The number of mutual reviews between a pair of developers also has a positive effect on the likelihood of security defect identification. When two developers review each other's code over a period of time, they become more aware of each other's coding styles, expertise, strengths, and weaknesses. That awareness might help one to pay attention to areas that he/she thinks the other has a weakness or where he/she may make a mistake. Since mutual reviews have positive impact, we recommend promoting such relationships.

VI. THREATS TO VALIDITY

Since case control studies originate from the medical domain,one may question whether we can use this study framework to study SE research questions. We would like to point out that prior SE studies have adopted various research designs, such as systematic literature review, controlled experiment, ethnography, and focus group that have originated in other research domains. Although, the results of this study do not rely on the case-control study framework, we decided to use this design, since: 1) our study satisfies the criteria for using this framework, and 2) following a established methodological framework strengthens an empirical research such as this study.

Our keyword-based mining technique to identify whether a code review identifies security defect or not poses a threat to validity. We may miss a security defect if the review comments do not contain any of the keywords that we used. However, as we are only considering those reviews that belong to security defects and ignoring the rest, we are considering that false-negative labeling of security defect will not make any impact on our study. Nevertheless, as we manually check all the security defects while assigning CWE ID, we find no false-positive labelling a code review as related to security defect.

Another threat to is the categorization of CWE ID. As one of our authors manually checks all the codes to find weakness type and assign the best match CWE ID for each weakness, that author might categorize a weakness with a wrong or less suited CWE ID. To minimize the effect of this threat, another author randomly chooses 200 source code files and manually assign CWE ID following a similar process without knowing the previously labeled CWE ID. We find that 194 out of 200 labels fall in the same group of CWE IDs that were labeled earlier. So, we are considering that this threat will not make any significant change in our results.

Another threat is the measure we take to calculate the developer's experience. We can interpret the term "experience" in many ways. And in many ways, measuring of experience will be complex. For example, we cannot calculate the amount of contribution of a developer to other projects. Although a different experience measure may produce different results, we believe our interpretation of experience is reasonable as that reflects the amount of familiarity with current project.

Finally, results based on a single project or even a handful of projects can be subject to lack of external validity. Given the manual work involved in the data collection process, it is often infeasible to include multiple projects. Moreover, historical evidence provides several examples of individual cases that contributed to discovery in physics, economics, and social science (see "Five misunderstandings about case-study research" by Flyvjerg [20]). Even in the SE domain case studies of the Chromium project [14], [17], Apache case study by Mockus *et al.* [42], and Mozilla case study by Khomh *et al.* [26] have provided important insights. To promote building knowledge through families of experiments, as championed by Basili [5], we have made our dataset and scripts publicly available [48].

VII. Related Work

Code review technologies are widely used in modern software engineering. Almost all the large scale projects have adopted peer code review practices with the goal of improving product quality [50]. Researchers have justified the benefit of code reviews to identify missed defects [6], [32]. Prior studies also find that peer code review can be very effective in identifying security vulnerability [12]. That is why developers use 10-15% of their working hours in reviewing other's code [11]. However, despite the popularity and evidence in support, some researchers explore that peer code reviews are not always performed effectively, which decelerates the software development process [16].

Despite putting lots of efforts in code review to keep product secured, a significant number of security vulnerability is reported every year and the number is ever-increasing. Although some prior studies [6], [18] have questioned about the effectiveness of peer code review in identifying security vulnerabilities, they did not explore the factors that could be responsible for this ineffectiveness. Researchers have introduced several metrics of code reviews over time that can be used to identify security vulnerability [2], [38], [39]. However, they did not investigate the state of those attributes when code review cannot identify security vulnerabilities.

Meneely and Williams find that the engagement of too many developers to write a source code file can make that file more likely to be vulnerable; termed that situation as "too many cooks in kitchen" [40]. But, they do not explore what characteristics of code review was responsible. Munaiah et al. use natural language processing to get the insights from code review that missed a vulnerability [44]. They investigate code review comments of Chromium project and find that code reviews that have discussions containing higher sentiment, lower inquisitiveness, and lower syntactical complexity are more likely to miss a vulnerability. To the best of our knowledge, no prior study has sought to identify the difference in security defects that are identified in code review and security defects that are escaped. Also, no prior studies introduce attributes that can be impactful in distinguishing code reviews where security defects get identified and code reviews where security defects get escaped.

VIII. Conclusion

In this case-control study, we empirically build two datasets– a dataset of 516 code reviews where security defects were successfully identified and a dataset of 374 code reviews where security defects were escaped. The results of our analysis suggest that the likelihood of a security defect's identification during code reviews depends on its CWE category. A logistic regression model fitted on our dataset achieved an AUC score of 0.91 and has identified nine code review attributes that influence identifications of security defects. While time to complete a review, the number of mutual reviews between two developers, and if the review is for a bug fix have positive impacts on vulnerability identification, opposite effects are observed from the number of directories under review, the number of total reviews by a developer, and the total number of prior commits for the file under review. Based on the results of this study, we recommend: 1) adopting additional quality assurance mechanisms to identify security defects that are difficult to identify during code reviews, 2) trying to avoid tangled code changes when possible, 3) assisting the reviewers to comprehend tangled code changes, 4) balancing review loads to avoid review fatigue, and 4) promoting mutual reviewing relationship between developers.

REFERENCES

[1] L. Allodi and F. Massacci, "Comparing vulnerability severity and exploits using case-control studies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 1, pp. 1–20, 2014.

[2] H. Alves, B. Fonseca, and N. Antunes, "Software metrics and security vulnerabilities: dataset and exploratory study," in *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 2016, pp. 37–44.

[3] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 2013, pp. 712–721.

[4] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 134–144.

[5] V. R. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 456–473, 1999.

[6] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 202–211.

[7] V. Bewick, L. Cheek, and J. Ball, "Statistics review 14: Logistic regression," *Critical care*, vol. 9, no. 1, p. 112, 2005.

[8] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 137–143.

[9] M. Borg, O. Svensson, K. Berg, and D. Hansson, "Szz unleashed: an open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project," *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*, 2019. [Online]. Available: http://dx.doi.org/10.1145/3340482.3342742

[10] A. Bosu and J. C. Carver, "Impact of developer reputation on code review outcomes in oss projects: An empirical investigation," in *2014 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14, Torino, Italy, 2014, pp. 33:1–33:10.

[11] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 56–75, 2016.

[12] A. Bosu, J. C. Carver, H. Munawar, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: an empirical study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 257–268.

[13] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at microsoft," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 146–156.

[14] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities?: a study of the chromium project," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 269–279.

[15] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[16] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs. how the current code review best practice slows us down," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 27–28.

[17] M. di Biase, M. Bruntink, and A. Bacchelli, "A security perspective on code review: The case of chromium," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 21–30.

[18] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner, "An empirical study on the effectiveness of security code review," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2013, pp. 197–212.

[19] H. FE, "Regression modeling strategies," https://hbiostat.org/R/rms/, [Online; accessed July 05, 2020].

[20] B. Flyvbjerg, "Five misunderstandings about case-study research," *Qualitative inquiry*, vol. 12, no. 2, pp. 219–245, 2006.

[21] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (roc) curve." *Radiology*, vol. 143, no. 1, pp. 29–36, 1982.

[22] F. E. Harrell Jr, *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer, 2015.

[23] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 121–130.

[24] D. Hinkle, H. Jurs, and W. Wiersma, "Applied statistics for the behavioral sciences," 1998.

[25] B. D. O. Hub, "Summary of chromium os project," https://www.openhub.net/p/chromiumos/analyses/latest/languages_summary, [Online; accessed on July 05, 2020].

[26] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality?: an empirical case study of mozilla firefox," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 2012, pp. 179–188.

[27] R. Kissel, *Glossary of key information security terms*. Diane Publishing, 2011.

[28] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 111–120.

[29] A. Krutauz, T. Dey, P. C. Rigby, and A. Mockus, "Do code review measures explain the incidence of post-release defects?" *Empirical Software Engineering*, vol. 25, no. 5, pp. 3323–3356, 2020.

[30] R. Lagerström, C. Baldwin, A. MacCormack, D. Sturtevant, and L. Doolan, "Exploring the relationship between architecture coupling and software vulnerabilities," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2017, pp. 53–69.

[31] S. Lewallen and P. Courtright, "Epidemiology in practice: case-control studies," *Community Eye Health*, vol. 11, no. 28, p. 57, 1998.

[32] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, 2008.

[33] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[34] G. McGraw, "Software security: building security in," *volume 1. Addison-Wesley Professional*, 2006.

[35] ——, "Automated code review tools for security," *Computer*, vol. 41, no. 12, pp. 108–111, 2008.

[36] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 192–201.

[37] ——, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[38] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, 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*. IEEE, 2013, pp. 65–74.

[39] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, "An empirical investigation of socio-technical code review metrics and security vulnerabilities," in *Proceedings of the 6th International Workshop on Social Software Engineering*, 2014, pp. 37–44.

[40] 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*, 2009, pp. 453–462.

[41] Mitre Corporation, "Common weakness enumeration," http://cwe.mitre.org/, [Online; accessed on January 13, 2020].

[42] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the apache server," in *Proceedings of the 22nd international conference on Software engineering*. Acm, 2000, pp. 263–272.

[43] N. Munaiah and A. Meneely, "Vulnerability severity scoring and bounties: why the disconnect?" in *Proceedings of the 2nd International Workshop on Software Analytics*. ACM, 2016, pp. 8–14.

[44] N. Munaiah, B. S. Meyers, C. O. Alm, A. Meneely, P. K. Murukannaiah, E. Prud'hommeaux, J. Wolff, and Y. Yu, "Natural language insights from code reviews that missed a vulnerability," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2017, pp. 70–86.

[45] 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, pp. 284–292.

[46] ——, “Using software dependencies and churn metrics to predict field failures: An empirical case study,” in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 364–373.

[47] N. J. Nagelkerke *et al.*, “A note on a general definition of the coefficient of determination,” *Biometrika*, vol. 78, no. 3, pp. 691–692, 1991.

[48] R. Paul, A. K. Turzo, and A. Bosu, “A dataset of Vulnerable Code Changes of the Chormium OS project,” Feb. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4539891

[49] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, “Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.

[50] 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*, 2013, pp. 202–212.

[51] W. Sarle, “Sas/stat user’s guide: The varclus procedure. sas institute,” *Inc., Cary, NC, USA*, 1990.

[52] M. S. Setia, “Methodology series module 2: case-control studies,” *Indian journal of dermatology*, vol. 61, no. 2, p. 146, 2016.

[53] T. J. Smith and C. M. McKenna, “A comparison of logistic regression pseudo r2 indices,” *Multiple Linear Regression Viewpoints*, vol. 39, no. 2, pp. 17–26, 2013.

[54] L. Statistics, “Spearman’s rank-order correlation,” *Laerd Statistics*, 2013.

[55] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, “Review participation in modern code review,” *Empirical Software Engineering*, vol. 22, no. 2, pp. 768–817, 2017.