Security Defect Detection via Code Review: A Study of the OpenStack and Qt Communities

Jiaxin Yu[1,2], Liming Fu[1,2], Peng Liang[1,2*], Amjed Tahir[3], Mojtaba Shahin[4]
[1] School of Computer Science, Wuhan University, Wuhan, China
[2] Hubei Luojia Laboratory, Wuhan, China
[3] School of Mathematical and Computational Sciences, Massey University, Palmerston North, New Zealand
[4] School of Computing Technologies, RMIT University, Melbourne, Australia
{jiaxinyu, limingfu, liangp}@whu.edu.cn, a.tahir@massey.ac.nz, mojtaba.shahin@rmit.edu.au

***Abstract*—*Background*: Despite the widespread use of automated security defect detection tools, software projects still contain many security defects that could result in serious damage. Such tools are largely context-insensitive and may not cover all possible scenarios in testing potential issues, which makes them susceptible to missing complex security defects. Hence, thorough detection entails a synergistic cooperation between these tools and human-intensive detection techniques, including code review. Code review is widely recognized as a crucial and effective practice for identifying security defects. *Aim*: This work aims to empirically investigate security defect detection through code review. *Method*: To this end, we conducted an empirical study by analyzing code review comments derived from four projects in the OpenStack and Qt communities. Through manually checking 20,995 review comments obtained by keyword-based search, we identified 614 comments as security-related. *Results*: Our results show that (1) security defects are not prevalently discussed in code review, (2) more than half of the reviewers provided explicit fixing strategies/solutions to help developers fix security defects, (3) developers tend to follow reviewers' suggestions and action the changes, (4) *Not worth fixing the defect now* and *Disagreement between the developer and the reviewer* are the main causes for not resolving security defects. *Conclusions*: Our research results demonstrate that (1) software security practices should combine manual code review with automated detection tools, achieving a more comprehensive coverage to identifying and addressing security defects, and (2) promoting appropriate standardization of practitioners' behaviors during code review remains necessary for enhancing software security.**

***Index Terms*—Code Review, Security Defect, OpenStack, Qt, Empirical Study**

I. Introduction

Security defects can have serious consequences, such as data breaches, intellectual property theft and disruption of services [1], [2]. Numerous studies have emphasized the significance of keeping software under control to reduce the risk of exploitation [3], [4], [5]. Nevertheless, the practice of leaving a large number of security defects unaddressed in the production environment for extended periods of time and only patching them after they have been released publicly [6], has a negative impact on software quality and leads to increased maintenance costs. Therefore, effectively minimizing the financial and reputational costs of security incidents by detecting security defects as early as possible remains the major focus for the stakeholders involved in software production.

Many organizations are shifting security practices to earlier stages of software development, hoping to address security concerns before they become more difficult and expensive to fix [7]. Under this circumstance, code review is proven to be an effective method to identify and locate security defects early [8], [9]. Code review is a valuable practice of systematically and internally examining revisions before code is released to production to detect defects and ensure quality. Code review is one of the most important practices of modern software development [10]. Compared with security defect detection tools, code review participants are mostly project members who can take full account of the code context [11]; thus, they are in a position to identify security defects effectively.

Several studies have focused on security defects detection in code review (e.g., [12], [13], [14], [15], [8]). Bosu *et al.* investigated the distribution and characteristics of security defects identified by code reviewers [8], while Paul *et al.* focused on the security defects that were missed during code review [14]. However, most of the research mainly concentrated on the identification of security defects, rather than delving into their resolution procedures. Specifically, little is known about the actions taken by practitioners and the challenges they face when resolving identified security defects in code review. Exploring these aspects could help increase the fixing rate of identified security defects during code review.

To this end, this work **aims** to explore the resolution of security defects through the means of code review, thus contributing to develop a more comprehensive body of knowledge on security defect detection via code review. We first collected 432,585 review comments from four active projects of two widely known communities: OpenStack (Nova and Neutron) and Qt (Qt Base and Qt Creator). After a keyword-based search on these review comments, we manually analyzed 20,995 potential security-related comments, resulting in 614 comments that actually identified

This work is funded by the NSFC with Grant No. 62172311 and the Special Fund of Hubei Luojia Laboratory. Amjed Tahir is supported by a MU SREF grant.

978-1-6654-5223-6/23/$31.00 ©2023 IEEE

security defects. We then studied the types of security defects identified, how the practitioners treat the identified defects, and why some of them are finally unresolved in code review.

Our **findings** show that: (1) security defects are not widely identified in code review; (2) when faced with security defects, most reviewers express their opinions on fixing them and provide specific solutions, which are generally agreed and adopted by developers; (3) *Disagreement between the developer and reviewer* and *Not worth fixing the defect now* are the most frequent causes of not resolving security defects.

The **contributions** of this work are: (1) We highlight the importance of manual and context-sensitive security review of code, which may reveal security defects undetected by automated tools. (2) We complement the datasets of previous works on the types of security defects identified during code review. (3) We provide the best practices for practitioners' behaviour in modern code review for security defects detection.

II. Related Work

A. Security Defect Detection

A body of research has focused on the current status of security defect detection across software ecosystems. Alfadel *et al.* discussed vulnerabilities propagation, discovery, and fixes in Python ecosystem [6]. It was found that most exposed security defects were not being fixed in a timely manner. A similar study of `npm` packages demonstrated that delays in fixing security defects were often caused by the fact that the fix was bundled with other features and did not receive the necessary prioritization [16]. Lin *et al.* investigated the security defect management in Debian and Fedora ecosystems [17], and found that over 50% of security defects fixes in Linux distributions can be integrated within one week. Our work differs from the aforementioned studies in that the security defects discussed in these works are publicly disclosed, while in our work we focused on security defects that practitioners may notice during their daily coding activities (but may not have been already disclosed).

Security defects can be detected through automated approaches or manually. Tudela *et al.* utilized hybrid analysis to detect the OWASP Top Ten security vulnerabilities and discussed the performance of different tool combinations [18]. Singh *et al.* compared the difference in automated (belong to DAST) and manual approaches for penetration testing, indicating that humans can locate security defects missed by automated scanners [19]. Osterweil *et al.* formulated a framework using IAST to improve human-intensive approaches in security defect detection and proved its effectiveness [20]. Inspired by the above-mentioned studies, we were motivated to explore an effective human-intensive practice for detecting security defects, i.e., code review, and pave the way for further integrating automated tools into the code review process.

B. Security Defect Detection in Code Review

Several studies have studied security defect detection in code review. For example, di Biase *et al.* explored the value of modern code review for system security and investigated the factors that can affect security testing based on the Chromium project [14]. Thompson *et al.* conducted a large-scale analysis of the dataset obtained from GitHub [9] and reaffirmed the crucial relationship between code review coverage and software security.

There is a growing interest in improving the effectiveness of security code review. Paul *et al.* analyzed 18 attributes of a code review to explore factors that influence the identification of security defects, in order to pinpoint areas of concern and provide targeted measures [21]. Braz *et al.* analyzed the impact of two external assistance measures on the identification of security defects [22] and found that explicitly requiring practitioners to concentrate on security can greatly increase the probability of finding security defects, while the further provision of security checklists did not show better results.

Some studies qualitatively analyzed the implementation of security defect detection in code review. Alfadel *et al.* investigated security-related reviews in `npm` packages [15] to analyze the proportion, types, and solutions of identified security defects in these reviews. In comparison, we targeted different data sources and provided a more in-depth analysis, which includes the causes for not resolving security defects and the actions of developers and reviewers when facing security defects in code review; therefore providing a holistic understanding of the current status of security code review. Motivated by these related works, we aim to bridge the knowledge gap with a view to inspire new research directions and enhance the effectiveness of detecting security defects.

III. Methodology

A. Research Questions

The goal of this study is to examine the implementation of security defect detection in code review. Specially, we analyzed review comments to investigate how security defects are identified, discussed, and resolved by reviewers and developers. To achieve this goal, we formulated the following Research Questions (RQs):

RQ1: *What types of security defects are identified in code reviews*?

Previous studies have explored the distribution of security defects found in code reviews [8], [14], [15], [23]. However, those studies have largely focused on specific systems and the types of security defects may vary in different systems, warranting additional research encompassing diverse projects to establish more general findings [14]. Driven by this, RQ1 investigates the frequency of each security defect type within the OpenStack and Qt communities, aiming to complement the findings from existing studies.

RQ2: *How do developers and reviewers treat security defects identified in code reviews?*

Given that strict reviewing criteria were mostly abandoned

in modern code review [24], it is necessary to establish a good understanding of the current practices employed by practitioners and how they influence the quality of security code review, so as to capture the undesirable behaviors and formulate corresponding suggestions for best practices. This RQ aims to explore concrete actions of developers and reviewers after security defects were identified. Answering this RQ helps to better understand the resolution process and the extent to which manual security defect detection is implemented in code review. In addition, the common solutions of each security defect type extracted from the changed source code can be used to support developers in addressing security defects in the future. This RQ is further decomposed into four sub-RQs:

RQ2.1: *What actions do reviewers suggest to resolve security defects?*

RQ2.2: *What actions do developers take to resolve security defects?*

RQ2.3: *What is the relationship between the actions suggested by reviewers and those taken by developers?*

RQ2.4: *What are the common solutions to each security defect type identified in code reviews?*

RQ3: *What are the causes for developers not resolving the identified security defects?*

In some cases, security defects are identified by reviewers but not ultimately resolved by developers. However, little research has been conducted to understand the reasons behind these cases, which could shed light on potential obstacles developers encounter and help in facilitating the resolution of identified security defects. As a result, RQ3 explores potential causes of why some defects are not fixed, with the objective of filling this gap and providing valuable insights.

B. Data Collection

The data collection, labelling, extraction, and analysis process is described below (an overview is shown in Fig. 1).

1) Projects Selection: This study analyzes security defects in code reviews collected from four projects of two communities: Nova[1] and Neutron[2] from OpenStack[3], and Qt Base[4] and Qt Creator[5] from Qt[6]. These two communities are selected based on the following two criteria [25]: 1) *Reviewing Policy* - the community has established a strong review process, and 2) *Traceability* - the review process of the community should be traceable.

OpenStack is a platform that builds and manages public or private cloud, with a set of projects responsible for processing different core cloud computing services. Qt is a cross-platform application for creating GUI applications. We deemed these two communities to be appropriate for our study as they have a large number of code reviews, which are performed using a traceable code review tool - Gerrit[7]. Gerrit offers on-demand tracking of the review process [26]. The projects from the two communities have been widely used in previous code review studies (e.g., [27], [28], [29], [30]). Similar to Hirao *et al.* [31], we selected two active projects from OpenStack (i.e., Nova and Neutron) and Qt (i.e., Qt Base and Qt Creator), which have the highest number of patches.

2) Review Comments Collection: Using the RESTful API provided by Gerrit, we obtained a total of 432,585 review comments from the four projects (166,237 review comments from OpenStack and 266,348 from Qt) spanning from January 2017 to June 2022, the time when we started this work. Considering that our study aims to analyze the practices of developers and reviewers when dealing with security defects, any comments made by bots should be excluded. Hence, we filtered out the review comments of which the author is a bot account (i.e., "*Zuul*" in OpenStack and "*Qt Sanity Bot*" in Qt). We also removed review comments in files that do not correspond to any programming language or are clearly outside the scope of code review, by checking the filename extension (e.g., "`.orig`" and "`.svg`").

3) Potential Security-related Comments Collection: We employed a keyword-based search approach to identify security-related review comments We adopted the keyword set proposed in Paul *et al.*'s work [21], as it is considered the most comprehensive keyword set in previous research, with the largest number of types and keywords. The set includes 103 keywords, which were classified into 11 security defect types and an extra *Common Keywords* type, with each security defect type containing Common Weakness Enumerations (CWEs) [32] to clarify its definition. After thoroughly analyzing the keyword set proposed by Paul *et al.* [21], we made the following adjustments to the set:

First, we adapted parts of the types of security defect and corresponding keywords. For example, we split *Denial of Service (DoS)* from the *Denial of Service (DoS) / Crash* type defined in Paul *et al.*'s work [21], since we considered *DoS* as one clear security defect type based on its definition in CWEs. The keywords relevant to *DoS* were also separated and reclassified into the new *DoS* type.

Second, we collected differentiated keywords and security defect types from previous studies [14], [23] and extended the keyword set obtained from the last step. One additional security defect type was added (i.e., the *Command Injection* type [23]). Moreover, another one additional security defect type was created since part of keywords from [14] could not be mapped into the existing keyword set (i.e., *Use After Free* was created to include "*use-after-free*" and "*dynamic*" based on the definition of CWEs). 19 differentiated keywords collected from previous studies were assigned to specific types (including *Common Keywords*) according to their meanings, (e.g., adding "*crypto*" to the *Encrypt* type).

After that, the initial keyword set of our study was formulated and presented in Table I. We ultimately obtained

[1] https://github.com/openstack/nova
[2] https://github.com/openstack/neutron
[3] https://www.openstack.org/
[4] https://github.com/qt/qtbase
[5] https://github.com/qt-creator/qt-creator
[6] https://www.qt.io/
[7] https://www.gerritcodereview.com/

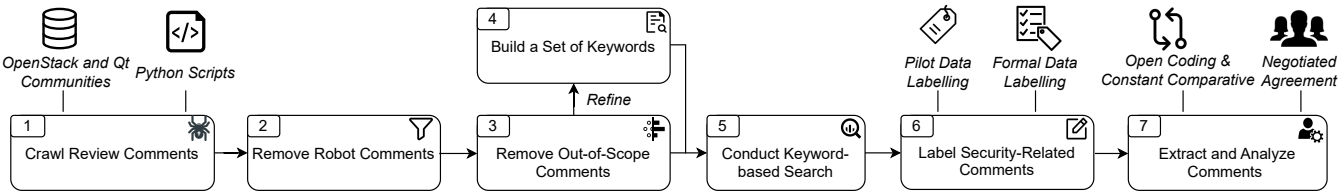

Fig. 1. An overview of our data acquisition, processing and analysis process

122 keywords, which were categorized into 15 security defect types and the *Common Keywords* type. To explicitly illustrate our adjustments, the sources of each type are presented, and newly added keywords compared to the keywords from Paul et al.'s work [21] are emphasized in italics.

Given that the effectiveness of the keyword-based approach heavily depends on the set of keywords used, we followed the approach proposed by Bosu *et al.* [8] to refine the initial set of keywords, which includes the following steps:

1) build a corpus by searching for review comments that contain at least one keyword of our initial set of keywords (e.g., "*racy*", "*overflow*") in the review comments collected in Section III-B2.
2) perform tokenization to each document on the corpus. Considering code snippets contained in review comments, we also applied the identifier splitting rules in this progress (e.g., "*FlavorImageConflict*" becomes "*Flavor Image Conflict*", *security_group* becomes "*security group*").
3) remove stopwords, punctuations, and numbers from the corpus and convert all tokens into lowercase.
4) use `SnowballStemmer` from the NLTK toolkit [33] to obtain the stem of each token (e.g., "*merged*", "*merging*", and "*merges*" have the same token "*merg*").
5) create a Document-Term matrix [34] from the corpus and identify the additional words that frequently co-occur with each of our initial keywords (co-occurrence probability of 0.05 in the same document, as also utilized in [8]).
6) manually analyze the additional words to determine whether to include them into the initial keyword set.

No additional words were found that co-occurred with any one of the initial keywords. Therefore, we were of the opinion that the present keyword set is adequate for supporting keywords-based search and filtering. After that, a script was developed to search for code review comments that contain at least one of the keywords identified in Table I. All these steps led to 20,995 review comments from the four projects, which is called **potential security-related review comments**.

C. Manual Labelling

The 20,995 potential security-related review comments obtained from the previous step may contain many false positives. Hence, we manually inspected the content of these comments, their corresponding discussions, and related source code to determine and label whether they are actually security-related. We defined the labelling criteria, i.e., the review comment should be clearly related to security and meet the definition of one of the CWEs [32] presented in Table I. Aimed at ensuring consistency and improving inter-rater reliability, a pilot labelling was independently conducted by the first and second authors on 200 potential security-related comments randomly selected from the Nova project. The labelling results were compared and the level of agreement between the two authors was measured using Cohen's Kappa coefficient test [35]. For review comments in which the judgements of two raters differ, they were reviewed, evaluated, and discussed with the third author until a consensus was reached. The calculated Cohen's Kappa coefficient is 0.87, thus indicating that the two authors reached a high level of agreement. The first author proceeded to label all the remaining potential security-related comments, and the review comments that the first author was unsure were discussed with the second author to reach a consensus. This process led to the identification of a total of 614 **security-related review comments** for further analysis and the distribution of data points across the four projects is presented in Table II:

D. Data Extraction and Analysis

A set of data items (see Table III) was formulated and extracted from the contextual information of each of the 614 security-related comments, including their corresponding discussion thread and source code, to answer our RQs.

1) RQ1: We classified 614 security-related review comments into 15 security defect types predefined in Table I. Based on this table, for each review comment, we identified the CWE corresponding to the issue described in the comment, and categorized the comment under the security defect type to which that CWE belongs. As shown in the example below, the reviewer pointed out that the calculation of `pos+n` may overflow and lead to undefined behavior, which is consistent with the description of CWE-109, that is "*The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value*", hence is labelled as *Integer Overflow*.

Link: http://alturl.com/td7ej
Project: Qt Base
Type: Integer overflow
Reviewer: This is UB when pos + n overflows...
Developer: Done

TABLE I
KEYWORDS TO MINE CODE REVIEWS THAT IDENTIFY SECURITY DEFECTS

Security Defect Type	Source	CWE ID	Keywords*
Race Condition	[21]	362-364, 366-368	race, racy
Crash	Adapted from [21]	248, 754, 755	crash, *exception*
Resource Leak	Adapted from [21]	401, 404	leak
Integer Overflow	[21]	190, 191, 680	integer, overflow, signedness, widthness, underflow
Improper Access	[21]	22, 264, 269, 276, 281-290	improper, unauthenticated, gain access, permission, hijack, authenticate, privilege, forensic, hacker, root, *URL, form, field, sensitive*
Buffer Overflow	[21]	120-127	buffer, overflow, stack, strcpy, strcat, strtok, gets, makepath, splitpath, heap, strlen, *out of memory*
Denial of Service (DoS)	Adapted from [21]	400, 402, 403, 405-406	denial service, dos, ddos
Deadlock	[21]	833	deadlock
Encryption	[21]	310, 311, 320-327	encrypt, decrypt, password, cipher, trust, checksum, nonce, salt, *crypto, mismatch*
Cross Site Scripting (XSS)	[21]	79-87	cross site, CSS, XSS, malform, *htmlspecialchar*
Use After Free	Created in this work	416	*use-after-free, dynamic*
Command Injection	[23]	77-78, 88	*command, exec*
Cross Site Request Forgery	[21]	352	cross site, request forgery, CSRF, XSRF, forged, *cookie, xhttp*
Format String	[21]	134	format, string, printf, scanf, *sanitize*
SQL Injection	[21]	89	SQL, SQLI, injection, *ondelete*
Common Keywords	[21]	-	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, steal, worm, phishing, cve, cwe, collusion, covert, mitm, sniffer, quarantine, scam, spam, spoof, tamper, zombie, *cast, xml*

* Most of the keywords in this list are adopted from the prior study of Paul *et al.* [21]. The keywords in italic are our additions to this list.

TABLE II
KEYWORDS TO MINE CODE REVIEWS THAT IDENTIFY SECURITY DEFECTS

Project	Comments	Potential Security-related	Security-related
Neutron	56,846	3,289	88
Nova	109,391	7,677	192
Qt Base	170,820	7,677	241
Qt Creator	95,528	2,352	93
Total	432,585	20,995	614

2) RQ2: We categorized the actions suggested by reviewers into three categories with reference to what was formulated by Tahir *et al.* in [36], [37].

1) **Fix**: recommend fixing the security defect.
2) **Capture**: detect the security defect, but do not provide any further guidance.
3) **Ignore**: recommend ignoring the security defect.

Confronted with review comments posted by reviewers, there are three possible behaviors for developers:

1) **Resolve**: The developer resolved the security defect identified by the reviewer.
2) **Not resolve**: The developer ignored the security defect identified by the reviewer.
3) **Unknown**: We are unable to determine the behavior of the developer.

We defined *Unknown* to describe the case that the developer responds to the reviewer with a promise to fix the security defect in the future, but we could not obtain specific resolution evidence from the source code due to the overwhelming amount of manual inspection of unlimited commits. An example of such a case is shown below:

```
Link: http://alturl.com/8g2p9
Project: Nova
Type: Buffer overflow
Developer: ...In a future patch that adds the
    ability to configure the executor type, we will
    need to deal with the issue you raise here.
```

We inspected the discussion and the follow-up submitted code to determine whether a security defect has been resolved. A security defect was considered resolved only when the situation meets the following three possible categories:

1) Code is modified in the subsequent patchsets by the developer to resolve the security defect before the code change is merged.
2) Developer mentioned clearly in the reply to comments that the security defect has been fixed in another code change.
3) The code change with the security defect was abandoned. As insecure code is not merged, it would not pose a harmful threat to the source code base.

As shown in Fig. 2, the developer added an assert statement to check the buffer size in Line 70 of `tst\_qlocalsocket.cpp` in patchset 5 to fix the buffer overflow, thus we can confirm that the security defect identified in this review comment was resolved.

Employing the open coding and constant comparative method [38], we used MAXQDA[8] as the coding tool and extracted the solutions developers adopted from the specific code modification for fixing security defects in resolved instances, so as to investigate the common solutions of each security defect types.

3) RQ3: To further understand why unresolved security defects were ultimately ignored by practitioners, we also utilized the open coding and constant comparative method [38] to examine the discussions between developers and reviewers.

For the purpose of minimizing bias, this data extraction was performed by the first author and verified by two other co-authors. Any conflicts were discussed and addressed by the three authors, using a negotiated agreement approach [39]. The complete extraction results in this step is available online [40].

[8] https://www.maxqda.com/

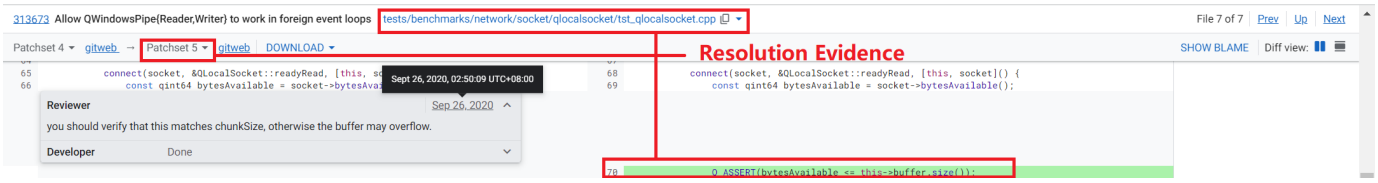

Fig. 2. An example of adding an assert statement to fix buffer overflow.

TABLE III
DATA ITEMS TO BE EXTRACTED FROM REVIEW COMMENTS

Data Item	Description	RQ
Security-Related	*Whether the review comment is security-related.*	RQ1
Security Defect	*The type of the identified security defect.*	RQ1
Reviewer's Action	*The action suggested by the reviewer to cope with the security defect.*	RQ2.1
Developer's Action	*The action adopted by the developer to cope with the security defect.*	RQ2.2, RQ2.3
Solution	*The final coding solution adopted to fix the security defect.*	RQ2.2
Relationship	*Relationship between the solution adopted by the developer and the solution suggested by the reviewer.*	RQ2.3
Resolution Evidence	*The location of code modification for resolving the security defect.*	RQ2.4
Cause	*The cause of not resolving the security defect.*	RQ3

IV. RESULTS

A. RQ1: Category of Security Defects Identified in Code Reviews

As explained in Section III-C, 614 review comments were identified as security-related comments, which account for less than 1% of all comments in code reviews. As detailed in Table IV, the majority of security defects (539 out of 614, 87.8%) were identified by reviewers, which is considerably more than those raised by developers. Therefore this study is based on 539 security-related review comments that meet the former case. As described in Section III, we have predefined 15 types of security defects with their distribution (see Table V). On the whole, we found that *Race Condition* is the most frequently identified type and was discussed in as many as 39.0% of instances. The second and third most frequently identified types are *Crash* and *Resource Leak*, accounting for 22.8% and 10.9%, respectively. There are 41 (7.6%) review comments identified *Integer Overflow*, followed by *Improper Access* with 31 (5.8%) instances. As can be seen in Table V, there are also nine types that were identified on rare occasions with proportions lower than 5%. Although *SQL Injection* is a common network attack and listed as the top 10 web application security risks by the Open Web Application Security Project (OWASP) in the past 15 years [41], no instance of this type was found in this study.

RQ1 summary: Security defects are not prevalently discussed in code review, with the proportion less than 1%. Of those security-related review comments, a considerable amount of review comments detected the security defects *race condition* (39.0%), *crash* (22.8%), and *resource leak* (10.9%).

TABLE IV
FREQUENCY OF EACH ROLE THAT IDENTIFIED THE SECURITY DEFECTS

Who Identified the Security Defect	Number	Percentage
Reviewer	539	87.8%
Developer	75	12.2%
Total	614	100.0%

TABLE V
FREQUENCY OF EACH SECURITY DEFECT TYPES

Security Defect Type	Number	Percentage
Race Condition	210	39.0%
Crash	123	22.8%
Resource Leak	59	10.9%
Integer Overflow	41	7.6%
Improper Access	31	5.8%
Buffer Overflow	24	4.5%
Denial of Service (DoS)	12	2.2%
Deadlock	11	2.0%
Encryption	9	1.7%
Cross Site Script (XSS)	8	1.5%
Use After Free	8	1.5%
Command Injection	1	0.2%
Cross Site Request Forgery	1	0.2%
Format String	1	0.2%
SQL Injection	0	0.0%
Total	539	100.0%

B. RQ2: Treatment of Security Defects by Developers and Reviewers

RQ2.1: Table VI shows that over half of the reviewers (290 out of 539, 53.8%) expected developers to *fix* the identified security defects. A large portion (251 out of 290, 86.6%) of these cases include specific solutions to assist developers in resolution, which may provide suggestions or even detailed code snippets for resolving the defects. Below is an example where the reviewer recommended that the developer should add verification logic to avoid the *Buffer Overflow* defect.

```
Link: http://alturl.com/tk8t9
Project: Qt Base
Type: Buffer Overflow
Reviewer: you should verify that this matches
        chunkSize,otherwise the buffer may overflow.
```

Only 13.4% (39 out of 290) of those fixes asserted that the defect needed to be fixed, without any guiding solutions. For example:

```
Link: http://alturl.com/783po
Project: Qt Base
Type: Integer Overflow
Reviewer: This could overflow too...You'll need to
       fix it otherwise the commit won't integrate.
```

TABLE VI
ACTIONS SUGGESTED BY REVIEWERS TO COPE WITH SECURITY DEFECTS

Reviewers' Actions	Number	Percentage
Fix with a Specific Solution	251	46.6%
Fix without a Specific Solution	39	7.2%
Capture	210	39.0%
Ignore	39	7.2%
Total	539	100.0%

TABLE VII
ACTIONS TAKEN BY DEVELOPERS TO COPE WITH SECURITY DEFECTS

Developers' Actions	Number	Percentage
Resolve	355	65.9%
Not Resolve	161	29.9%
Unknown	23	4.2%
Total	539	100.0%

There are also 210 (39.0%) cases where reviewers only identified security defects without indicating the next step that the developer should follow, which fall under the definition of *Capture* type. Besides that, there were a few reviewers (39, 7.2%) who explicitly suggested *ignoring* the identified security defects for various reasons, such as the issues not worth fixing.

RQ2.2: We inspected the discussion and subsequent patchsets to determine whether a security defect was fixed finally. As shown in Table VII, developers chose to fix the identified security defect more often, accounting for 65.9%. The actions developers took to each security issue are presented in Table VIII. The overall result is that almost every type of security defect has a fix rate upwards of 50.0%, except for *Deadlock*, as low as 36.4%. As analyzed in RQ1, defects of type *Race Condition*, *Crash*, and *Resource Leak* are the top three frequently identified security defects. As demonstrated in Table VIII, these types of security defect are also frequently addressed by developers in code reviews with fix rates of 64.3%, 71.5%, and 79.7%, respectively. In addition to the top three types of security defects, there are another 11 security defect types from "Integer Overflow" to "Format String", totalling 147, and the fixing rate of these 11 types is comparatively low, at 57.8% (85 out of 147).

RQ2.3: The relationship between the action developers took and the action reviewers suggested is illustrated in Fig. 3.

TABLE VIII
DEVELOPERS' RESOLUTION RATE FOR EACH SECURITY DEFECT TYPES

Security Defect Type	Reviews	Resolve	Percentage
Race Condition	210	135	64.3%
Crash	123	88	71.5%
Resource Leak	59	47	79.7%
Integer Overflow	41	26	63.4%
Improper Access	31	17	54.8%
Buffer Overflow	24	14	58.3%
Denial of Service (DoS)	12	6	50.0%
Deadlock	11	4	36.4%
Encryption	9	5	55.6%
Cross Site Script (XSS)	8	4	50.0%
Use After Free	8	6	75.0%
Command Injection	1	1	100.0%
Cross Site Request Forgery	1	1	100.0%
Format String	1	1	100.0%
Total	539	355	65.9%

When reviewers provide a clear idea for fixing the identified security defects with specific solutions (*Fix with a specific solution*), the fix rate by developers reaches 81.3% (204 out of 251). When reviewers only point out that fixing is needed but do not offer any guidance (*Fix without a specific solution*), developers choose to address these defects in 61.5% (24 out of 39) of the cases. Furthermore, when reviewers indicate the existence of security defects without further instructions (*Capture*), only 59.5% (125 out of 210) of these issues are fixed. Based on our findings, it can be speculated that reviewers' suggestions that include guidance in code review, such as whether and how to resolve defects, are crucial to improving the overall fix rate of identified security defects. As shown in Fig. 3, for the instances in which the actions suggested by reviewers are the *Fix* type, 78.6% (228 out of 290) of developers fixed the identified security defects. For the instances in which reviewers suggested ignoring the defects, nearly all (36 out of 39, 92.3%) developers ignored the defects. Overall, it can be concluded that the majority of developers tend to agree with reviewers' opinions when the reviewers express clear perspectives on defect handling. Hence, the participation and enthusiasm of reviewers are crucial for detecting security defects during code review.

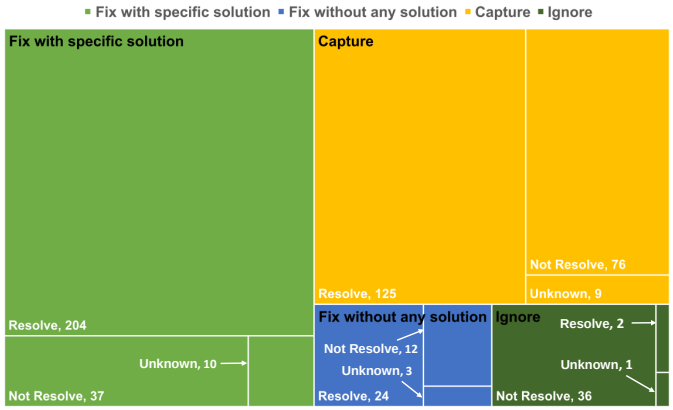

Fig. 3. The treemap of relationship between the action developers taken and the action reviewers suggested

RQ2.4: The coding solutions adopted by developers to resolve different security defects are further investigated and presented in Table IX. In order to make the sample size large enough to ensure the credibility of the conclusions, we only selected the top three security defects based on their prevalence for analysis, i.e., *Race Conditions*, *Crash*, and *Resource Leak*.

In terms of *Race Condition*, the most common approach adopted by developers is to **take thread-safety measures**. These measures include using thread-safe functions, such as atomic operations, the `invokeMethod` function, or synchronization functions that utilize signals and slots (e.g., `QFutureWatcher` in Qt). They also employed custom logic when working with resources, including measures such as adding locks, usage limitations, and updating before usage to ensure consistency. **Code refactoring** is also an important solution for *Race Condition*, with 33 instances. A few

TABLE IX
THE SOLUTIONS OF THE TOP THREE MOST FREQUENT SECURITY DEFECT TYPES

Type	Solution	Number
Race Condition	Take thread-safety measures	55
	Code refactoring	33
	Concurrency management	9
	Handle side effects	7
Crash	Code refactoring	30
	Add condition check to avoid crash	26
	Capture exceptions by try/catch block	13
	Safely terminate execution in advance	6
	Use safe functions	4
Resource Leak	Add release function	20
	Use resource-management techniques	9
	Reduce usage of resource	8
	Code refactoring	4
	Handle side effect	2

cases adopted **concurrency management**, which includes passing messages between threads and adding wait functions. Additionally, 7 developers solved the issue by **handling side effects**, which means dealing with the consequences of *Race Conditions* indirectly, such as capturing exceptions.

In the instances of *Crash*, there are five possible solutions. **Code refactoring** and **adding condition check** are the two main solutions adopted by developers to fix the *Crash* defects. In 13 review comments, developers **captured exceptions by try/catch block** to avoid *Crash*. Furthermore, 6 developers **safely terminated execution in advance** to prevent damage caused by an abrupt *Crash*, and a specific example of this case is to add an assert statement to immediately trigger an exception and terminate the execution of the program, if a certain condition or constraint is not met. There are also 4 cases where developers **used safe functions** that can eliminate potential exceptions, thus improve the overall stability of the program and minimize the likelihood of crashing.

Approximately half of *Resource Leak* defects are fixed by **adding resource release functions**, where developers may explicitly close resources or prevent skipping of the deletion function through modification in code logic. 9 developers also **used resource-management techniques**, such as smart pointers, Resource Acquisition Is Initialization (RAII), or bridge technologies during fixing. Additionally, 8 developers **reduced resource allocation** to avoid leaks through converting to passing by reference, transferring resource ownership and so on. Only 4 cases involve **code refactoring** as a solution, while just 2 cases addressed the security defects through **handling side effects**, as previously mentioned.

RQ2 summary: 53.8% of the reviewers indicated a need to fix the identified security defects after their detection, and most of them were willing to provide specific solutions for developers to fix the defects. From the developers' perspective, majority of developers tend to agree with reviewers' suggestions, and over half of the identified security defects were resolved by developers.

C. RQ3: Causes of Not Resolving Security Defects

According to the aforementioned result of RQ2.2, there are 161 instances where the identified security defects were not resolved. By manually inspecting the discussion for each review comments, we excluded 64 (39.8%) review comments neither developers nor reviewers involved in these instances clearly indicate the causes for ignoring the identified security defects, leaving us with 97 instances (60.2%) for further analysis. The statistical results of the remaining instances can be found in Table X, and six causes were then identified.

Nearly half of (44, 45.4%) unresolved security defects are because either developers or reviewers think it is *Not worth fixing the defect now*, which is the most common cause of not resolving the identified security defects. From the perspective of security defects, the identified security defects in these cases may be harmless and acceptable for developers, or the occurrence scenarios of security defects are so tricky that they will not become system hazards under normal utilization. It may also be because that there are other security defects in the code that will have a greater impact, and those currently found are negligible comparatively. On the developer side, fixes might cost too much effort and require tons of changes. If existing solutions had other adverse effects on the system and were irreconcilable, developers would also choose to ignore the identified defects in light of the benefit of current code changes. In addition, some developers noted that the resolutions for identified security defects were not an immediate concern and could be considered in the future. Two examples corresponding to the above two situations are presented below, the cruxes has been emphasized in bold:

Link: http://alturl.com/hm4o7
Project: Nova
Developer: after discussing it on IRC [1], we went on a consensus that it's **acceptable** to remove the VIF from the metadata since the NIC on the VM already detached, even if the Neutron action could potentially fail.

Link: http://alturl.com/xcb6n
Project: Qt Base
Developer: I'll leave it as it is - otherwise I'll have to **change the example too much and write tons of code** obscuring the real example.

We attribute *Disagreement between the developer and the reviewer* to be the reason why developers do not resolve the security defects in 33 review comments (34.0%). In these cases, some developers could not comprehensively understand reviewers' opinions, while others indicated that the identified security defects did not exist. Furthermore, some developers believed that fixing was unnecessary or the solution was unreasonable. In the following example, the developer objected to the reviewer's suggestion to control traffic by adding a security group, asserting that no modification was required.

Link: http://alturl.com/9uwgv
Project: Neutron
Reviewer: i suspect this requires security groups.
Developer: Why we need this? Could you explain because I don't think we need anything here.

TABLE X
THE DISTRIBUTION OF CAUSES FOR IGNORING SECURITY DEFECTS IDENTIFIED IN CODE REVIEWS

Cause	Number	%
Not worth fixing the defect now	44	45.4%
Disagreement between the developer and the reviewer	33	34.0%
No clear solution to solve the problem	11	11.3%
Out of the scope of this commit	6	6.2%
Think users should make correct choices	2	2.1%
Developers had no time to rework	1	1.0%
Total	97	100.0%

Due to the lack of knowledge or limitation by other system logic, 11.3% of identified security defects were ignored for the reason that practitioners *had no effective solution to thoroughly resolve the defects*, and below is an example of this case:

```
Link: http://alturl.com/t5paz
Project: Qt Creator
Reviewer: ...If you are not happy with a crash you
    can add a check against 0. This will avoid the
    crash here but I am pretty sure that it will
    crash sooner or later on a different location...
```

In 6.2% review comments, the reason for not resolving security defects is that the resolution is considered *out of the scope of the commit*. As shown in the example below, the identified security defect was historical and thus orthogonal with the feature of this commit. Accordingly, the developers reckoned that those defects should wait to be resolved in specific logic changes in the future, rather than now.

```
Link: http://alturl.com/joasw
Project: Nova
Developer: ...I think that the multi-attach problem
    is orthogonal and should be investigated in
    another patch.
```

In addition, three occasional instances were found, in two (2.1%) of which the developers believed that it was *users' responsibility to make correct choices* to guarantee the system running appropriately, and no any modification was conducted to the source code. While in the remaining one (1.0%), the developer clearly indicated that he/she *had no time to rework* and left the reviewer to accept identified defects or directly abandon the whole change.

RQ3 summary: Generally speaking, 39.8% related instances did not provide the cause of failure to resolve. *Not worth fixing the defect now* and *Disagreement between the developer and the reviewer* are the main reasons of ignoring security defects.

V. IMPLICATIONS

Here we discuss several implications of the findings reported in this paper.

A two-step of detection mechanism is suggested to conduct security practices in software development. Our study found that in the process of code review, the majority of reviewers provided useful suggestions to fix identified security defects, and developers usually agreed and adopted solutions suggested by reviewers. This indicates that reviewers' assessment of security defects is trustworthy for developers. Generally speaking, code review is effective in detecting and addressing security defects. Although various tools (e.g., SAST, DAST, IAST) have been used in modern code review to speed up the review process, these tools test only based on known scenarios and have limitations in test coverage, thus resulting in potential false positives [19]. Experienced and knowledgeable code reviewers, due to their deeper understanding of code context, can capture security defects which do not conform to known patterns and cannot be detected by tools. Therefore, automated tools and code review, as two significant approaches of security defect detection, need to complement each other. We recommend a two-step detection mechanism that combines the two approaches: tools to conduct scalable and fast security defect detection as the first check, and then the reviewers to conduct code review referring to the detection results of the tool. During the second step, the reviewers check the results generated by tools to provide developers with instructions for further action, and at the same time review the submitted code to find defects that the tool failed to detect. This mechanism not only improves efficiency, but also enhances the comprehensiveness of security defect detection.

The characteristics of the project can affect the type and quantity of security defects found in it. We found that *XSS (Cross-Site Scripting)* and *SQL Injection* are less discussed during code review, which is consistent with the findings of Paul *et al.* [21], but contrary to the results of di Biase *et al.* [14], which demonstrated that XSS was a frequently identified security defect with a relatively higher number than other types. The projects used in this study (Nova, Neutron, Qt Base, and Qt Creator) and Paul *et al.*'s work (Chromium OS) are the projects that provide infrastructure for higher-level applications to run on, with less direct interaction with users' inputs and outputs, while di Biase *et al.* selected Chromium, a Web browser that has multiple ways of directly interacting with users. One possible reason for this result is that the likelihood of potential input/output-related security defects in core components and projects may be low. This further confirms that project characteristics can influence the types and quantity of security defects that may exist in this project.

Reviewers need to pay more attention to high-risk code with the use of multi-threading or memory allocation. *Race Condition* and *Resource Leak* related security defects are frequently identified in code review. These two defect types are also widely recognized as common defects in software development [42], [43]. Hence, we encourage code reviewers to conduct a rigorous inspection of code involving multi-threading and memory allocation during code review, as they can potentially introduce *Race Condition* and *Resource Leak* defects, making them more susceptible to security risks.

Appropriate standardization of practitioners' behaviors in code review is critical for better detection of security defects. In modern code review, strict reviewing criteria are not mandated [44]. We found that some developers' and reviewers' actions result in ambiguity during the code review

process. For example, some comments that identified security defects were neither responded nor had corresponding code modifications. Hence, code reviews may not foster a sufficient amount of discussion [45], increasing the time and effort of the development process and having a negative impact on software quality. Here are several specific recommendations regarding standardization: (1) For security defects that remain unresolved due to disagreement between developers and reviewers, reviewers could further assess the risk of the security defects. We found that the main reason for not resolving security defects is *Disagreements between the developer and the reviewer* in which the developer did not agree with the reviewer's assessment, and thus decided not to fix the security defects identified. However, due to the different knowledge and experience, it is likely for the developer to merge risky security defects into the source code. Hence, we suggest that when there is a disagreement, reviewers should further assess the risk of identified security defects and communicate with developers if necessary. (2) It is preferable for developers to resolve identified security defects. However, when developers decide not to address a security defect (possibly due to risk assessment or cost-benefit considerations), they should provide clear reasons for this decision in the discussion. It was found that in 40% of the cases, the identified security defects were left unresolved, with no reasons provided. This negatively impacts adequate communication between reviewers and developers, making review details opaque and untraceable. Therefore, we recommend that when a security defect was decided to be left unresolved, sufficient justifications should be provided in the discussion to facilitate further handling of the unresolved security defects. (3) Unresolved security defects should be properly documented, and the developers who decide to fix them in the future should be clearly scheduled for resolution in subsequent stages. According to the results of RQ2.2, 29.9% of security defects were unresolved and merged into source codeDocumenting unresolved security defects in code review helps to effectively track and manage them. Clearly scheduling unresolved security defects that developers decide to fix in the future can ensure they are actually resolved in a timely manner, thus preventing them from causing damage to the system. Therefore, we encourage practitioners to document unresolved defects and schedule needed fixes.

VI. Threats to Validity

Internal Validity: During the data processing phase, there are comments that were either generated by bots or related to non-review target files, which could influence the accuracy of the final results. We filtered these comments to mitigate bias. Furthermore, we employed a keyword-based search approach to obtain potential security-related comments, which can lead to missing security-related comments that do not contain the exact keywords. To reduce this bias, we collected all the keywords utilized in previous studies into the keyword list and refined the list according to the approach proposed by Bosu *et al.* [8], ensuring a comprehensive set of keywords to cover all eligible review comments as much as possible.

External Validity: We selected four projects from the OpenStack and Qt communities (two each) as the primary data source of our study. However, these projects may not fully represent the entire landscape of security defects across all software systems. This limitation poses a potential threat to the generalizability of our results. To address this concern, we compared and discussed with the previous studies that explored similar questions to supplement our own findings and reduce the risk of interpretation bias.

Construct Validity: Since this study predefined the types of security defects and matched practical scenarios with security defect types through manual inspection, there is a potential cognitive bias arising from subjective judgments. To reduce this bias, we based the classification on the security defect types proposed in previous works [21] and clarified these security defects by CWEs, thus ensuring the concepts of each type are accurate, appropriate, and consistent throughout the entire research process. In addition, all the data labeling and extraction processes in this study were carried out manually, which introduces the possibility of subjective and potentially misleading conclusions. Therefore, during the data labelling phase, the first and second authors conducted a pilot data labelling independently and reached a consensus on labelling criteria through discussions. During the data extraction phase, while the first author performed the extraction work, the second and third authors reviewed the results to ensure the accuracy and comprehensiveness of the data extraction results.

Reliability: We drafted a protocol outlining the detailed procedure before conducting our study. The protocol was reviewed and confirmed by all authors to ensure the clarity and repeatability of the method. We also made our full dataset available online for future replications [40].

VII. Conclusions

In this work, we investigated the security defects identified in code review comments. We analyzed the data from four open source projects of two large communities (OpenStack and Qt) that are known for their well-established code review practices. More specifically, we manually inspected 20,995 review comments obtained by keyword-based search and identified 614 security-related comments. We extracted the following data items from each comment: 1) the type of security defect, 2) the action taken by reviewers and developers, 3) reasons for not resolving identified defects from these comments. Our main results are: (1) security defects are not widely discussed in code reviews, and when discussed, *Race Condition* and *Crash* security defects are the most frequently identified types; (2) the majority of the reviewers express explicit fixing suggestions of the detected security defects and provide specific solutions. Most of the developers are willing to agree with reviewers' opinions and adopt their proposed solutions; (3) *Not worth fixing the defect now* and *Disagreement between the developer and the reviewer* are the main reasons for not resolving security defects.

REFERENCES

[1] R. Telang and S. Wattal, "An empirical analysis of the impact of software vulnerability announcements on firm stock price," *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 544–557, 2007.

[2] H. Cavusoglu, B. Mishra, and S. Raghunathan, "The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers," *International Journal of Electronic Commerce*, vol. 9, no. 1, pp. 70–104, 2004.

[3] E. Iannone, R. Guadagni, F. Ferrucci, A. D. Lucia, and F. Palomba, "The secret life of software vulnerabilities: A large-scale empirical study," *IEEE Transactions on Software Engineering*, vol. 49, no. 01, pp. 44–63, 2022.

[4] G. McGraw, J. H. Allen, N. Mead, R. J. Ellison, and S. Barnum, "Software security engineering: A guide for project managers," tech. rep., CMU/SEI, 2013.

[5] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, vol. 1, 2002.

[6] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.

[7] GitLab, *GitLab: Mapping the DevSecOps Landscape - 2022 Survey*, 2022. https://about.gitlab.com/developer-survey/#operations.

[8] A. Bosu, J. C. Carver, M. Hafiz, 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 (FSE)*, pp. 257–268, ACM, 2014.

[9] C. Thompson and D. Wagner, "A large-scale study of modern code review and security in open source projects," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, pp. 83–92, ACM, 2017.

[10] 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.

[11] S. McConnell, *Code Complete*. Pearson Education, 2004.

[12] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner, "An empirical study on the effectiveness of security code review," in *Proceedings of the 5th International Symposium on Engineering Secure Software and Systems (ESSoS)*, pp. 197–212, Springer, 2013.

[13] R. Paul, "Improving the effectiveness of peer code review in identifying security defects," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1645–1649, ACM, 2021.

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

[15] M. Alfadel, N. Nagy, D. Costa, R. Abdalkareem, and E. Shihab, "Qualitative analysis of security-related code reviews in npm packages: An empirical study," *Available at SSRN 4161317*, 2022.

[16] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Software Engineering*, vol. 26, pp. 1–28, 2021.

[17] J. Lin, H. Zhang, B. Adams, and A. E. Hassan, "Vulnerability management in linux distributions: An empirical study on debian and fedora," *Empirical Software Engineering*, vol. 28, no. 2, p. 47, 2023.

[18] F. M. Tudela, J.-R. B. Higuera, J. B. Higuera, J.-A. S. Montalvo, and M. I. Argyros, "On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications," *Applied Sciences*, vol. 10, no. 24, p. 9119, 2020.

[19] N. Singh, V. Meherhomji, and B. Chandavarkar, "Automated versus manual approach of web application penetration testing," in *Proceedings of the 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pp. 1–6, IEEE, 2020.

[20] L. J. Osterweil, M. Bishop, H. M. Conboy, H. Phan, B. I. Simidchieva, G. S. Avrunin, L. A. Clarke, and S. Peisert, "A comprehensive framework for using iterative analysis to improve human-intensive process security: An election example," *ACM Transactions on Information and System Security*, 2017.

[21] R. Paul, A. K. Turzo, and A. Bosu, "Why security defects go unnoticed during code reviews? a case-control study of the chromium os project," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1373–1385, IEEE, 2021.

[22] L. Braz, C. Aeberhard, G. Çalikli, and A. Bacchelli, "Less is more: supporting developers in vulnerability detection during code review," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pp. 1317–1329, ACM, 2022.

[23] A. Bosu and J. C. Carver, "Peer code review to prevent security vulnerabilities: An empirical evaluation," in *Proceedings of the 7th IEEE International Conference on Software Security and Reliability Companion (SERE-C)*, pp. 229–230, IEEE, 2013.

[24] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice (ICSE)*, pp. 181–190, ACM, 2018.

[25] 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 (MSR)*, pp. 192–201, ACM, 2014.

[26] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Review participation in modern code review: An empirical study of the android, qt, and openstack projects," *Empirical Software Engineering*, vol. 22, pp. 768–817, 2017.

[27] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 677–687, ACM, 2018.

[28] K. Hamasaki, R. G. Kula, N. Yoshida, A. C. Cruz, K. Fujiwara, and H. Iida, "Who does what during a code review? datasets of oss peer review repositories," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pp. 49–52, IEEE, 2013.

[29] X. Han, A. Tahir, P. Liang, S. Counsell, K. Blincoe, B. Li, and Y. Luo, "Code smells detection via modern code review: A study of the openstack and qt communities," *Empirical Software Engineering*, vol. 27, no. 6, p. 127, 2022.

[30] L. Fu, P. Liang, Z. Rasheed, Z. Li, A. Tahir, and X. Han, "Potential technical debt and its resolution in code reviews: An exploratory study of the openstack and qt communities," in *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 216–226, ACM, 2022.

[31] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "Code reviews with divergent review scores: An empirical study of the openstack and qt communities," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 69–81, 2020.

[32] The MITRE Corporation, *Common Weakness Enumeration*, 2022. https://cwe.mitre.org/.

[33] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, 2009.

[34] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Pearson Education India, 2016.

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

[36] A. Tahir, A. Yamashita, S. Licorish, J. Dietrich, and S. Counsell, "Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pp. 68–78, ACM, 2018.

[37] A. Tahir, J. Dietrich, S. Counsell, S. Licorish, and A. Yamashita, "A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites," *Information and Software Technology*, vol. 125, p. 106333, 2020.

[38] B. G. Glaser, "The constant comparative method of qualitative analysis," *Social Problems*, vol. 12, no. 4, pp. 436–445, 1965.

[39] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, 2013.

[40] J. Yu, L. Fu, P. Liang, A. Tahir, and M. Shahin, *Dataset of the Paper "Security Issue Detection in Code Review: An Exploratory Study of OpenStack and Qt communities"*, 2023. https://doi.org/10.5281/zenodo.7886148.

[41] The OWASP® Foundation, *Top 10 Web Application Security Risks in OWASP*, 2022. https://owasp.org/www-project-top-ten/.

[42] T. Wu, J. Liu, X. Deng, J. Yan, and J. Zhang, "Relda2: an effective static analysis tool for resource leak detection in android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 762–767, IEEE, 2016.

[43] L. Zhang and C. Wang, "Rclassify: classifying race conditions in web applications via deterministic replay," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 278–288, IEEE, 2017.

[44] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 13th European Software Engineering Conference (ESEC/FSE)*, pp. 300–310, ACM, 2011.

[45] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, 2016.