2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation

A security perspective on code review: The case of Chromium

Marco di Biase
Software Improvement Group
Amsterdam, The Netherlands
m.dibiase@sig.eu

Magiel Bruntink
Software Improvement Group
Amsterdam, The Netherlands
m.bruntink@sig.eu

Alberto Bacchelli
Delft University of Technology
Delft, The Netherlands
A.Bacchelli@tudelft.nl

***Abstract*—Modern Code Review (MCR) is an established software development process that aims to improve software quality. Although evidence showed that higher levels of review coverage relates to less post-release bugs, it remains unknown the effectiveness of MCR at specifically finding security issues.**

We present a work we conduct aiming to fill that gap by exploring the MCR process in the Chromium open source project. We manually analyzed large sets of registered (114 cases) and missed (71 cases) security issues by backtracking in the project's issue, review, and code histories. This enabled us to qualify MCR in Chromium from the security perspective from several angles: Are security issues being discussed frequently? What categories of security issues are often missed or found? What characteristics of code reviews appear relevant to the discovery rate?

Within the cases we analyzed, MCR in Chromium addresses security issues at a rate of 1% of reviewers' comments. Chromium code reviews mostly tend to miss language-specific issues (*e.g.*, C++ issues and buffer overflows) and domain-specific ones (*e.g.*, such as Cross-Site Scripting); when code reviews address issues, mostly they address those that pertain to the latter type. Initial evidence points to reviews conducted by more than 2 reviewers being more successful at finding security issues.

I. Introduction

Code review is a practice of manual source code analysis with the goal of increasing software quality (*e.g.*, reliability and maintainability). Code review comes in different flavors, such as formal code inspections [1], security audits by external experts [2], and more lightweight, asynchronous assessments of source code changes by other developers [3]. The latter form is also known as Modern Code Review (MCR) [4].

Nowadays, MCR is being adopted by many organizations [5] and its popularity is growing with the advent of the pull-based development model [6] and the availability of many tools to support its logistics [7]. Overall, MCR is a process that is (1) informal (in contrast to inspections), (2) tool-based, (3) asynchronous, and that (4) occurs regularly in practice.

Previous research provided evidence that MCR is useful in improving overall software quality level [8], [9], particularly by addressing issues related to software evolvability [10], [11]. Little is known, however, of the value of MCR in relation to software security issues. Past research, in fact, mostly focused on the effectiveness of security audits conducted by external experts on the entire codebase of software systems [2], [12], without considering the MCR practices. With the aim of starting to fill this knowledge gap, we conduct an exploratory case study on MCR from a security perspective.

In particular, we focus on answering questions that can provide initial useful information for both researchers and practitioners. For example: Does MCR find security related issues? If so, are there issues that are more frequently found or missed? Which factors could hinder or support finding security issues in MCR? From these answers, software engineers and managers can start taking informed decisions on whether and how to use MCR for security concerns, and researchers can focus their attention on developing and improving source code analysis tools that address the most problematic aspects of MCR when used for security.

As subject of our study, we consider the case of Chromium [13], an open-source software (OSS) web browser that forms the base of the most popular web browser, *i.e.*, Chrome. Since Chromium uses MCR for each proposed code change and has the highest number of security bugs reported in the CVE database [14] for an OSS product, it gives us the opportunity to investigate the relationship between MCR and security in a significant, real-world context.

We conduct our investigation by exploring the history of MCR and security issues in Chromium. In particular, we (1) manually inspected a total of 1,155 code review comments to determine the proportion of those related to security; (2) semi-automatically extracted review comments raising concerns on security issues and manually analyzed and classified them into known vulnerabilities; (3) and manually analyzed and classified security issues not found during review.

Based on the results of our exploration, we discuss unexpected findings, also providing initial indications to practitioners as well as outline promising future investigation directions for researchers.

This paper is structured as follows: Related work is discussed in Section II. Section III details the research questions, as well as the research method used, and enumerates some threats to construct validity. In Section IV we present our findings, following which in Section V we discuss the same. Finally, in Section VI we summarize the work.

II. Related Work

Previous work on code review started with the investigations by Fagan on formal code inspections [15]. According to his studies, defects are primarily found during the actual inspection meeting [15]. Subsequently, Votta recommended

978-1-5090-3848-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SCAM.2016.30

to involve only the author and one reviewer because of scheduling difficulties [16]. Porter *et al.* found that the number of reviewers and authors were the most relevant factors of software inspection performance. [17]. Kollanus and Koskinen did a software inspection survey, addressing the need of more empirical research to validate the effects of the different processes in practice. Their work led to better understanding of the actual impact of inspections on different organizations [3].

Although useful, formal inspections have been gradually replaced by a more lightweight process by practitioners, for instance to suit agile-oriented development methods [18], [19]. The effectiveness of the more lightweight process has been a topic of research, comparing it to other quality-improving processes such as testing and pair programming [3], [20], [21]. Furthermore, some analysis has been done in understanding the usefulness of MCR. Thongtanunam *et al.* found that developers are often most concerned about documentation and structure to enhance evolvability and fix functional issues [22]. Beller *et al.* revealed that most changes of Open-Source systems in MCR are indeed related to the evolvability and functionality aspect, with a ratio of 75/25 [11]. The study by Bacchelli and Bird [5] showed similar MCR outcomes for industrial projects at Microsoft; Lassenius *et al.* [10] reported similar outcomes for other industrial and academic projects.

Even if MCR is now widely adopted in both open source and industrial projects, the impact of MCR on security is still unclear. The most relevant work in this field has been done by Edmundson *et al.* [12]. Their main focus was to assess the effectiveness of manual code review in improving software security. Specifically, they hired 30 developers and tested their review efficacy on a web application. Their findings suggest that developers are not able to address every security issue in the analyzed system. Furthermore, there was no relation between experience and effectiveness of a developer in finding security-related problems during code review.

III. Methodology

We present the research questions, as well as a description of the research context and the research methodology.

A. Research Questions

Our examination of the literature revealed that our scientific knowledge of code review and security issues does not cover the case of MCR. Our study aims to gather insight on the MCR process with respect to security issues, considering the Chromium project as a case study.

We know that most comments in MCR regard code improvements or clarification questions [5] and most changes triggered by review pertain to maintainability and evolvability issues [11]. We currently have less knowledge on what proportion of reviewers' comments in real-world MCR regards security concerns. This motivates our first research question:

RQ1. What proportion of comments in code reviews address potential security issues?

Vulnerabilities and security flaws come in diverse flavors and pose different challenges when they are to be manually or automatically detected [23]. We explore which types of security threats are more frequently discovered or overlooked during MCR in Chromium, to get an initial indication of the suitability and points for improvements of MCR for this task. This motivates our second and third research questions:

RQ2. What categories of security issues are typically discovered during code reviews?

RQ3. What categories of security issues are typically missed during code reviews?

Finally, knowing which factors in the MCR process may lead to detecting/missing security flaws is important to guide practice and future research on the topic. To this aim, we look for initial evidence that can be tested in further empirical studies. This motivates our last research question:

RQ4. What factors might lead to finding or missing security issues at review time?

B. Research Setting

Our study is focused on the OSS web-browser Chromium, the project on which the popular Google Chrome is based.

Subject system. Chromium consists of over 14 million Source Lines of Code, mainly written in C and C++. The project uses a public issue repository,[1] employs a public MCR process[2] that is strictly enforced ("all code should be reviewed prior to checkin" [24]), and is the OSS product with the highest number of security bugs in CVE [14] [25]. These features make it a valuable case for our exploratory investigation, as they allow us to consider a real-world, extensive project with rich data available for analysis.[3]

Common Vulnerabilities and Exposures (CVE). CVE is a list of publicly available information about security vulnerabilities for software products. It aims to ease the sharing of data on different vulnerability capabilities with a *common enumeration*. With this enumeration, one can access information about the problem on multiple data sources using the same CVE Identifiers. The CVE List entries provide information for each CVE Identifier, such as data on fix information and severity scores. CVE offers a non-exclusive, royalty-free license for research and development.

Code review process in Chromium. Chromium uses Rietveld [26] as code review tool, which allows issuing reviews to the system directly from the code repository. The work flow to create a new review request starts from the change that a developer makes in his workspace: After committing

[1]https://bugs.chromium.org/p/chromium/

[2]https://codereview.chromium.org/

[3]Chromium has dependencies that once were part of Chromium itself, such as V8, Skia, blink, and are developed with the same process and use the same tools. Hence, we include them in our analysis too.

the code into a branch, the author must create a *change list* to describe the patch content; then, to start the review process, the author publishes the change list and selects (at least) one reviewer. It is an author's responsibility to choose a relevant reviewer for the specific patch; guidelines in the developers' contribution page for Chromium suggest to base the choice of the reviewer on who did the latest changes on the modified code. The review must also contain an *owner*, who has the responsibility of ensuring the highest quality for the subsystem being touched by the change.

Figure 1 shows the user interface of Rietveld. After the review process has started, reviewers can **(1)** browse the textual differences (*aka* diff) between the original version of the files and the proposed patch, as well as **(2)** insert inline comments to start a discussion thread. If the reviewer(s) suggests some changes, developers can upload a newer version of the patch, thus initiating a feedback cycle. For a patch to be merged, the owner must give it a 'LGTM' (Looks Good to Me) **(3)**.

Issues and vulnerabilities in Chromium. The Chromium project uses Monorail [27] as issue tracking system. Monorail offers a public way for users and developers to file issues as well as publicly open historical issue data. The project provides detailed information on the life cycle of bugs and reporting guidelines [28].

The special case of security and vulnerability issues is managed as in the following. Labels are heavily used in this context,[4] and specific use for some of those are strictly controlled (*e.g.*, the severity level[5]). Chromium aims to deploy a patch for a critical vulnerability within 30 days. For high-severity ones, the aimed time span is 60 days. Access to security bug data is normally activated within 14 weeks.[6] Once the bug is externally reported, it gets its CVE label assigned.

C. Research Method

Our research method is based on the manual and semi-automatic analysis of historical data on the development and review process of Chromium. Our sources of historical data are: the code review data stored by Rietveld, the issue data saved through Monorail, and vulnerabilities and exposures data stored in CVE. In the following, we detail how we use them.

RQ1. Proportions of review comments on security. To better understand the role of security concerns in code reviews for Chromium, we start by estimating how frequently these concerns are raised by reviewers. To do so, we collect sets of sample review comments from the entire population of comments for Chromium and manually inspect which proportion pertains to security. We focus on code review taking place in the year 2014 to make sure the data on the corresponding vulnerabilities is fully accessible. In fact, choosing 2015 would have led to data not accessible due to the non-disclosure period that security issue have before they can be fully accessible.

[4]https://www.chromium.org/Home/chromium-security/security-labels
[5]https://www.chromium.org/developers/severity-guidelines
[6]https://www.chromium.org/Home/chromium-security

We wrote a Python script to automatically retrieve comments via the Rietveld API.[7] The resulting population consists of 132,000 comments belonging to over 155,000 code reviews for 2014. The lower number of comments relative to reviews is due to the 14-week non-disclosure policy for security issues enforced by Chromium. We found several issues that were not accessible due to other restrictions (*e.g.*, vulnerabilities that affects third-party products).

From the initial sample, we selected comments written only by reviewers, as we are interested in reviewers' behavior during the process. It is indeed their responsibility to ensure that code under review has the highest quality. We selected 60,655 comments from the initial dataset. We filter out comments written by non-reviewers by discarding them if their author was not in the list of reviewers.[8] Messages written by bots are marked as `"auto_generated":true` and are automatically discarded as well.

Determining the proportion of comments about security concerns is hard to do automatically, thus we manually read the review comments and flagged them as security-related or not. Being a manual effort, we could not inspect the entire initial dataset, rather we proceeded selecting statistically significant sample sets. As we had no prior details about the distribution of security related comments, we picked comments using random sampling without replacement (as opposed to other techniques, *e.g.*, stratified random sampling) to extract reliable sample sets. We establish the size (n) of such sets with the following formula [29]:

$$n = \frac{N \cdot \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}{(N-1)\,E^2 + \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}$$

Since the proportion ($\hat{p}$) of the comments referring to security concerns is not known *a priori*, we consider the worst case scenario (*i.e.*, $\hat{p} \cdot \hat{q} = 0.25$). We have a population that, from a statistical point of view, is relatively small, so we included the finite population correction factor in the formula: It allows us to take the population size (N) into account (*i.e.*, 60,655 review comments). We keep the standard confidence level of 95% and error (E) of 5%, *i.e.*, if security issues are raised in $f\%$ of the sample set comments, we are 95% confident they will be cited in $f\% \pm 5\%$ of the population comments. To strengthen this sample set selection, we repeat this process three times, creating three non-overlapping sets with size 385 comments each.

Finally, the first author of this paper manually performed the following steps on each comment: (1) He read the comments' content, to establish whether it is security-relevant or not; (2) if relevant, he read the whole code review discussion to obtain a deeper understanding; (3) if the security concern was confirmed by further comments on the raised issue (*i.e.*, by the developer, the reviewer itself or other involved people), he marked it as a security concern.

[7]https://github.com/rietveld-codereview/rietveld/wiki/APIs
[8]This list also contains reviewers added after the code review is started.

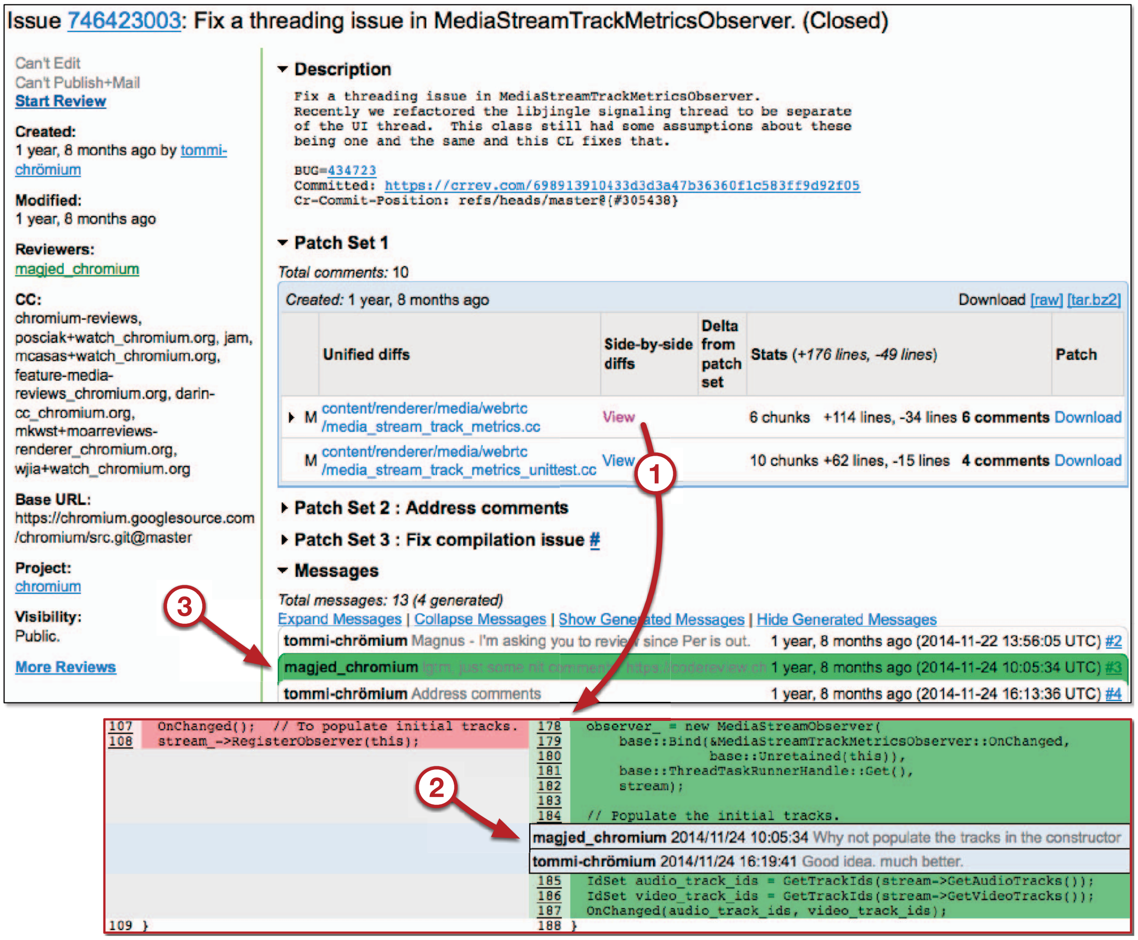

Figure 1. Rietveld Code Review tool in Chromium

RQ2. Types of security flaws discovered by reviews. As a taxonomy to classify the types of security concerns raised in reviews, we rely on the one described in: "24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them" by Howard, LeBlanc, and Viega [23]. This taxonomy has the advantage of focusing on the most common design and coding errors, with examples that can be easily mapped to the historical data we analyze; moreover, it is an exhaustive, yet reasonably sized compilation of problems, as opposed to the Common Weakness Enumeration (CWE) [30], which totals over one thousand very detailed, but loosely organized entries.

The security-related comments found in RQ1 form a sample that is not large enough to have even an initial answer to our RQ2. Extending this set of comments using the same method (*i.e.*, the analysis of randomly sampled review comments) would require to extend the initial sample set by an order of magnitude to have enough initial data for our exploration (approximately 80 security concerns, due to the number of possible classes of security issues). This would have been a time-consuming and, more importantly, error-prone approach. For this reason, we built our dataset using a different strategy.

We defined a set of security-related keywords that could have been used by reviewers when commenting security related concerns. To define this list of keywords, we use generic terms, such as 'security', and terms specific to the security flaw described in our taxonomy, such as 'cookie', also considering synonyms. We then used these keywords to retrieve security related review comments. Our keyword list is made by the following terms: *buffer*, *cast*, *command*, *cookie*, *crypto*, *emismatch*, *exception*, *exec*, *form*, *field*, *heap*, *injection*, *integer*, *ondelete*, *out of memory*, *overflow*, *password*, *printf*, *privilege*, *race*, *random*, *sanitize*, *security*, *sensitive*, *sql*, *URL*, *use-after-free*, *vulnerability*, *xhttp*, *xml*. We also used regular expressions and *stemming* based on the same matching intent for different words. For instance, to retrieve *Input Sanitization* issues belonging to the category *Format String problems*, we reduced both *sanitization* and *sanitize* to the common prefix *sanitiz*. With this approach, we retrieved 9,765 comments for our dataset. To ensure that the retrieved comments were pertinent, we proceeded according to the following steps, per keyword: (1) we selected the comments including the specific keyword, (2) we manually inspected every comment to determine whether it was indeed raising a security concern, (3) if so, we analyzed the whole review discussion reading all the comments handling the same issue and analyzed the code changes to assign the comment to the right category in the taxonomy, (4) instead, if the review comment did not raise a security concern, we discarded it. To determine whether a comment raised an actual security problem, we verified that the issue did exist and was fixed in the subsequent proposed patches. We could do it by reading the content of the review discussion. After filtering from the dataset consisting of 9,765 items, the resulting dataset of security-related review comments included 71 elements.

RQ3. Types of security flaws overlooked by reviews. To retrieve a sample of security issues that reviewers failed to notice, we took advantage of the information on CVE as a starting point. Our research method consisted in the following steps: (1) we select a CVE security issue for Chromium and go to the matching entry in the Monorail issue repository (either by taking advantage of the link available in CVE or by manually searching the CVE ID in the issue tracker), (2) as the issue is closed (otherwise it would not be public on CVE), we retrieve the files interested in fixing the issue and we identify the lines that introduced the security issue in the first place, (3) after evaluating the patch diff, we position the issue in the taxonomy, (4) we find the *last relevant change*[9] to these lines (using git blame), and (5) retrieve the data on the code review(s) that allow them to be introduced.

To avoid any latent overlap in the data about found vs. overlooked security problems, we collected data on issues entered in CVE in 2015. Analyzing all the 187 resulting CVE entries, we could link the patches for their corresponding 114 bugs, which we traced back to 139 original code reviews. The difference between the complete set of bugs and the ones we analyzed was due to the aforementioned non-disclosure period (in some cases we also found inaccessible data due to restrictions that applied even after the non-disclosure period) and that we did not always successfully retrieve the original code review (the most frequent reason is that Rietveld was not used by Chromium when the last relevant change was made).

RQ4. Factors (possibly) influencing security reviews. The manual analysis we conducted to answer RQ2 required us to carefully inspect many cases in which security issues were discovered by the participants of the MCR process. In addition to quantitatively comparing the categories of these issues, we attempted to obtain initial qualitative insights on factors that appeared to have an influence on the effectiveness of MCR for security. To formalize this approach, we annotated our manual classification for later analysis to discover emergent patterns or factors.

In detail, this analysis process began with reading comments in a review such that we could understand what happened in each of them. Initially, we annotated each review with possible patterns; then, we repeated the process on our dataset iteratively as we found more or repeated patterns. Finally, we aggregated our findings around different clusters that emerged with our analysis. We found the number and role of reviewers to be the most interesting and recurrent factors that showed an impact during the activity. In particular, multiple reviews did not comply with the Chromium policies on the suggested number of reviewers (Section III-B) and the role of a reviewer was a factor influencing the process.

D. Threats to the validity of the results

The goal of this work is to explore the usefulness of code reviews for security-related issues. We recognize that our research method presents some limitations. In particular, our analysis is limited only to Chromium code reviews. The cultural and workplace habits have an influence on what happens during the process that we aim to analyze, *i.e.*, having guidelines that are in place for the specific development process.

The method to answer RQ1 is based on the manual analysis done by only one researcher. This may pose potential threats, which we tried to mitigate by trying to gain a deep understanding of an issue and by performing an iterative analysis. The classification lacks of further validation via known strategies.

The method to answer RQ2 also poses potential threats to the validity of the results. Since we adopted a keyword-based approach, we could have missed some relevant comments in code reviews. An alternative approach could have been to build our list on results given by RQ1; however, the data generated to answer RQ1 was insufficient to build a larger set of issues. The choice of our keywords is strictly related to the taxonomy that we choose for our study, thus may be biased towards it. In spite of this, our results did not expose any obvious bias and we found categories not defined by the taxonomy. As an alternative, one could use the pre-existing classification available in CVE. However, this classification regards the *outcome* of a vulnerability issue rather than the *cause*. For this reason, we decided to go deeper into the source code and classify the cause. The classification patterns that we used in our study, finally, lacks some of the specific sets of patterns that we found in our analysis; we address this problem by complementing it with the taxonomy offered by CWE.

Our results are limited to the number of issues and code reviews analyzed, representing a threat to the generalizability of our conclusions. Furthermore, we retrieved the starting dataset from a known database of security issues. This, although represents a reliable source, still has breadth limitations. Nonetheless, our work focuses on facts affecting the cases we analyzed. Threats to the results' validity are hence to be restricted to the field of inappropriate conceptualization about the process under analysis.

Our study can make statements only for the aforementioned process, because it is limited to it. A larger dataset would allow for both a quantitative and qualitative analysis that could further improve our study, but this would have gone beyond the scope of this exploratory study.

[9]We define as relevant change those that modify functional aspects. For example, we discarded every refactoring that changed an identifier or the way a variable was accessed (value, reference or pointer), as well as a library or API refactoring.

IV. Results

We present the results of our exploratory investigation, following the structure of the aforementioned research questions.

RQ1. What proportion of comments in code reviews address potential security issues?

Through the manual analysis of our three samples (each consisting of 385 randomly selected review comments), we found they contained respectively 4, 2, and 5 security-related issues. These results give us statistical evidence that approximately 1% of the review comments in Chromium relate to security.

> **Result 1:** *Approximately 1% of the review comments in Chromium are about potential security flaws.*

We manually classified the 11 security-related comments using the taxonomy of Howard *et al.*: Two comments address *Race conditions*, two *Failing to protect network traffic*, one *Catching all exceptions*, one *Format String Problems*, and one *C++ Catastrophes*. The remaining four belonged to other categories, and address issues in *Credential Management Issues*, *Improper Control of a Resource Through its Lifetime*, *Incorrect Type Conversion or Cast* and *Improper Privilege Management*. A likely cause of issues not fitting into Howard's taxonomy is the type of software system that Chromium represents, *i.e.*, a web browser. The taxonomy is geared towards more generic application-type software systems.

As random sampling led to a very low number of cases, we deviate from our current approach and use the one presented in Section III-C to answer our next research question.

RQ2. What categories of security issues are typically discovered during code reviews?

The second column of Table I presents the absolute number of review comments and the proportion of those (enclosed by parentheses) that belong to the different categories in the security flaw taxonomy of Howard *et al.* [23], as we found them by manually inspecting our sample comments. The categories are sorted by decreasing occurrence in code review comments and the color intensity reflect the relative amount of items for the specific category.

The most popular category of potential security flaws discovered during MCR in Chrome is client side *Cross-Site Scripting (XSS)*, followed by *C++ Catastrophes* and *Buffer Overruns*. The first flaw is domain-specific: It enables attackers to execute client-side scripts in web pages viewed by other users; it is among the most popular security vulnerabilities found in web applications [31]. The second and third flaw are language-specific: The category *C++ Catastrophes* includes bugs such as use-after-free and use of uninitialized variables, and other bugs caused by non-cautious use of pointers, such as a destructor for a pointer that does not set its content to null; the category *Buffer Overruns* comprises of problems ranging from the copy without checking size of its input

Table I

The security flaws found and missed in the Chromium project by modern code review, manually classified according to the taxonomy of Howard *et al.* [23]

Security flaw	Found by code review	Missed by code review
XSS (client side)	15 (21%)	12 (10%)
C++ catastrophes	8 (11%)	33 (29%)
Buffer overruns	6 (8%)	18 (16%)
Too much privilege	5 (7%)	5 (4%)
Information leakage	5 (7%)	1 (1%)
Race conditions	4 (6%)	5 (4%)
Format String problems	4 (6%)	3 (3%)
Catching all exceptions	3 (4%)	0 (0%)
Failing to protect network traffic	2 (3%)	4 (3%)
Integer overflows	2 (3%)	3 (3%)
Use of Magic URLs, predictable cookies	1 (1%)	1 (1%)
Use of weak password-based systems	1 (1%)	1 (1%)
Command injection	0 (0%)	7 (6%)
XSS (server side)	0 (0%)	1 (1%)
Other	15 (21%)	21 (18%)

to heap-based and stack-based buffer overflows. Interestingly, *Command injection*, which has been the most popular security vulnerability for several years [31] appears neither in this sample of comments found through keywords (despite also using keywords specific to this flaw) nor in the sample of comments we found in RQ1.

The *Other* category in Table I contains all the cases that did not fall into the taxonomy by Howard *et al.* [23]. For these cases we use an alternative taxonomy based on CWE; Table II reports the results, represented by absolute values and proportions (in parenthesis) of the total. Language-specific issues are prominent (CWE-704 is common in C and C++ [32]).[10]

> **Result 2:** *The majority of potential security flaws detected during MCR relate to domain and language-specific issues. Injections, despite being the most diffused security flaw, are not detected in our sample of Chromium reviews.*

[10] For the detailed description of the categories, we refer the reader to the definitions provided by the taxonomies we use (*i.e.*, [23], [32]).

Table II
THE SECURITY FLAWS, FOUND AND MISSED IN THE CHROMIUM PROJECT BY MCR, THAT COULD NOT BE CLASSIFIED WITH THE TAXONOMY OF HOWARD *et al.* [23] AND WERE CLASSIFIED ACCORDING TO CWE [30]

Other security flaw	CWE id	Found by code review	Missed by code review
Incorrect Type Conversion or Cast	**CWE-704**	6 (40%)	2 (10%)
Improper Control of a Resource in its Lifetime	**CWE-664**	1 (7%)	2 (10%)
Use of Potentially Dangerous Function	**CWE-676**	0 (0%)	2 (10%)
Weaknesses that Affect System Processes (IPC)	**CWE-634**	1 (7%)	1 (5%)
Privilege / Sandbox Issues	**CWE-265**	2 (13%)	0 (0%)

RQ3. What categories of security issues are typically missed during code reviews?

Using the same taxonomies employed for RQ2, we manually categorized the 114 security issues that were missed by code review and that we identified based on our research method (Section III-C). The third column in Table I reports the absolute and relative (in parenthesis) results. Since the security flaws are ordered by found ones, we again used the color to quantify the frequency with which each flaw was missed during review (the redder, the higher the frequency).

We found that the most commonly overlooked flaw is *C++ Catastrophes*, which represents one third of all missed flaws. With *Buffer Overruns* and *Cross-Site Scripting (XSS)* (second and third, respectively), it covers the majority of the missed flaws. We did not find any bug in the categories *Catching All Exceptions*, *Information Leakage*, *Magic URLs or predictable cookies* and *use of weak password-based systems.* Similar to RQ2, the category *Others* is split using CWE as can be seen in Table II, column 'Second sample'.

Contrasting found and missed security flaws, the set including the top three is the same in both cases. This is to be expected as the frequencies are probably partly caused by differences in base rates of distribution of those flaws. For example, XSS issues are very prominent in web applications [31] and accordingly appear in the top-3. Nevertheless, we notice that *C++ Catastrophes* is the most common missed flaw (29% of occurrences), but it is far less frequently found during review (11% of occurrences), as opposed to *XSS*.

To have a quantitative overview of how the ranking of security flaws is related between the two sets, we compute their Spearman correlation [33], which is non-parametric, and found it to be 0.56 (excluding the *other* category). This value is considered a moderate relationship, thus providing evidence that the missed and found flaws indeed have a common base, but not negligible differences exist, too.

> **Result 3:** *The majority of overlooked security flaws by MCR in Chromium relate to language and domain-specific issues. The ranking in the occurrences of missed vs. found security flaws presents non-negligible differences.*

RQ4. What factors might lead to finding or missing security issues at review time?

With this research question we seek to make a preliminary step to further the understanding the nature of security-issue-finding reviews. We investigate the set of code reviews that we built for RQ2 (*i.e.*, reviews *finding* security flaws) to let emerge interesting insights and patterns. We found that the reviews discovering security flaws could be structured meaningfully according to who raised the security concern and the number of people involved:

A main reviewer finds the flaw (2 reviewers). Per Chromium policies (Section III-B), each patch must be reviewed by the *owner* of the changed subsystem and another reviewer, we define them in our analysis as *main reviewers*. In this scenario only these two reviewers participate in the review and one of them raises the security concern.

A main reviewer finds the flaw (>2 reviewers). Even if not required by the policy, some reviews see the participation of more than 2 reviewers. In this scenario, one of the main reviewers raises the concern, but the number of participating people is larger than normal (we found a maximum of eight people involved in a single review).

An optional reviewer finds the flaw (>2 reviewers). In this scenario, one of the optional reviewers, which is either added to the review or joins the process spontaneously, raises the security-related concern.

A security expert finds the flaw (>2 reviewers). In this pattern, a security expert is explicitly invoked by one of the reviewers to join the review and detect security related problems. It is indeed the expert who finds the security flaw.

The author of the patch. In this case, the developer submitting the patch raises a security issue about the patch while discussing the code changes with the reviewers.

Off-line discussion. In this scenario, the issue is detected during a face-to-face discussion with an unspecified person relevant to the submitted change and this is reported in the review.

Other. The cases not belonging to the previous categories.

In Table III, we split the 71 reviews raising security-concerns into the aforementioned scenarios. Column *ID* has been introduced to give each scenario a unique identifier, later used in Table IV. The fourth column has absolute and relative (in parenthesis) values. We see that in majority (55%) of the cases, the security concern was raised when more than three people were involved in the review; moreover, main reviewers are more likely to raise security concerns when more than three people are involved (41% vs. 31%).

We further analyze the code reviews finding faults, by relating these scenarios with the categories of found issues. Results are reported in Table IV. Data is too sparse to generate any statistical evidence, but there is no noticeable relation between the scenario and the type of issue found.

> **Result 4:** *Initial evidence suggests that reviews in which more than 2 reviewers are involved tend to raise more security concerns.*

Table III
THE REVIEWS IN WHICH SECURITY CONCERNS HAVE BEEN RAISED, SPLIT BY THE PERSON WHO RAISED THE CONCERN AND THE NUMBER OF PEOPLE INVOLVED IN THE REVIEW. THREE PEOPLE (*i.e.*, AUTHOR, OWNER, AND SECOND REVIEWER) IS REQUIRED BY CHROMIUM POLICIES.

Person raising the security concern	Reviewers involved	ID	Number of reviews
Main reviewer	**= 2**	**M=2**	22 (31%)
Main reviewer	**> 2**	**M>2**	29 (41%)
Optional reviewer	**> 2**	**O>2**	7 (10%)
Security expert	**> 2**	**S>2**	3 (4%)
Patch author		**Aut**	5 (7%)
Off-line discussion		**Off**	2 (3%)
Other			3 (4%)

Table IV
THE SCENARIO IN WHICH THE SECURITY FLAW WAS RAISED IN RELATION TO ITS CATEGORY. SCENARIO'S ID CORRESPOND TO THOSE INTRODUCED FIRST IN TABLE III.

Security flaw	Scenario					
	M=2	**M>2**	**O>2**	**S>2**	**Aut**	**Off**
XSS (client side)	4	6	3	0	0	1
C++ catastrophes	1	6	1	0	0	0
Buffer overruns	4	1	1	0	0	0
Too much privilege	1	2	0	1	1	0
Information leakage	2	3	0	0	0	0
Race conditions	1	2	0	1	0	0
Format String problems	0	2	0	0	2	0
Catching all exceptions	1	1	0	1	0	0
Failing to protect network traffic	1	1	0	0	0	0
Integer overflows	0	0	2	0	0	0
Use of Magic URLs, predictable cookies	0	0	0	0	0	1
Use of weak password-based systems	1	0	0	0	0	0
Other	6	5	0	2	0	0

V. DISCUSSION

This section presents a discussion based on our results, including some indications to practitioners.

A. Missed language-specific security issues

We found that the majority of missed security flaws relate to language-specific issues, as witnessed by the high frequency of *C++ catastrophes* and *Buffer overruns*. We found this to be particularly surprising for especially two reasons: (1) These kinds of flaws are extremely *localized*, thus one can quickly verify their presence by considering only the changed code and little external code that can be easily retrieved with static code analysis (*e.g.*, the code that instantiates or destroys a pointer) [23], as opposed to architectural and design flaws that require more time to be discovered and a more thorough knowledge of the entire codebase [34]; (2) Chromium has suffered from many flaws of this type throughout its history [30], thus we expected developers to be particularly attentive and employ strict policies to avoid/find them, but, even though reviewers do find these errors (Table I), many still slip their attention.

We may hypothesize that the high-incidence of overlooked language-specific issues may be caused by the inherent intricacies of the main languages of Chromium, *i.e.*, C and C++. Moreover, language-specific issues belong to a very broad spectrum and the reviewers' experience may not cover it fully.

Overall, our results provide evidence that the current MCR practice, as implemented in Chromium, is not yet able to fully deal with language-specific security issues.

B. Security issues, found vs. missed by MCR in Chromium

Having used the same taxonomies to classify the results regarding found and missed security flaws in MCR, we have the chance to compare the results.

We hypothesize that the moderate positive rank correlation (0.56) between the frequencies of security flaws that are found vs. missed is due to the underlying distribution of these issues in the code. However, if on the one hand it is reasonable to think that issues that are, by nature, more common in a project are going to be more frequently found/missed in MCR; on the other hand, if one is aware that a project is prone to certain types of issues, (s)he should analyze the code more thoroughly with that in mind to avoid them. Results seem to indicate that reviewers may not take into account historical flaws when conducting reviews, thus the first line of reasoning better explains the moderate positive rank correlation.

Considering the details of the found vs. missed flaws we see that domain-specific issues (*e.g.*, *XSS*) seem to be relatively less problematic to find for reviewers, than language-specific problems. Moreover, *Command injections* are not found entirely by the reviews we analyzed, but they represent the fourth most common security issue that is missed in Chromium reviews in 2014; the set of keywords we used for retrieving the code review finding security issues carefully included a number of terms related to injection, which was also inspired by the review we randomly found answering RQ1, so we would exclude this as due to our data collection mechanism.

We hypothesize that these results are due to reviewers more aware of security issues specific to the particular application they are developing, rather than of generic security concerns that may appear in different types of applications, *e.g.*, due to the programming language used.

Overall, our results provide evidence that domain-specific security flaws in Chromium tend to be found more frequently at review time, compared to other issues; in particular, the case of command injections seems to be particularly problematic, thus calling for further investigation on this topic.

C. Given more eyeballs all security flaws are shallow

The code review practice in Chromium suggests that ideally the reviewer should be one who is familiar with the area of the code submitted [24]; anybody can review code as long as there is at least one owner for each different submodule the contributor is committing the code to [24]. Studies on the topic confirm that code review is an activity that is effective when a reasonably low number of people is involved [16], [35] Focusing on security, we have initial evidence suggesting that a number of reviewers higher than 2 is advisable to find the most security flaws.

Speculating on this result is hard, because of the high variability of the process. Even if Chromium has rules enforcing the modality of how the procedure should be run, the overall idea that we got from manually analyzing the activity in hundreds of code review is that there is a straightforward way of running it, but it is often misinterpreted. The *standard* way of reviewing code in Chromium still finds a large number of security issues in our research findings; despite this, our results suggests that a higher number of *eyes* reading a Change List submitted to Chromium has a higher percentage of addressing a potential future security issue. We did not investigate this finding further, but studies can be designed and carried out to determine if and how the number of reviewers has an effect on the security issues found.

D. Initial indications to practitioners

Practitioners could benefit from initial indications that can help the review process trying to be more prone in addressing security flaws. The first suggestion that this work brings to evidence is that increasing the number of people involved when dealing with code patches could be useful to mitigate future security bugs. This could be because of the larger and different expertise, regarding various part of the codebase, brought into the process by having more people involved. Observing patterns that have been defined, the intervention of an *external* reviewer relative to the ongoing process revealed further bugs that otherwise would have been left unnoticed. Dealing with large projects could be one factor that limits the scope of our indication.

Investing time and effort in writing more thorough and effective test cases is one indication that practitioners could use to prevent issues. Reviewers, in fact, would trust the code patches submitted by developers more. In some cases, when analyzing code reviews that did not address a security flaw, we noticed the complete absence of test cases regarding that particular functionality or component. We also notice that after being patched, test cases were explicitly added. This seems to suggest that both contributors and reviewers may be overlooking the importance of testing.

VI. Conclusion

We conclude by summarizing our study and suggesting viable paths for future work.

A. Summary

Our study focused on the security perspective of MCR in the Chromium project. We presented a categorization of issues that are missed and found in code reviews. From this analysis, we learned that, on the one hand, code reviews in Chromium often miss security-related C++ issues, buffer overflows, and XSS vulnerabilities. On the other hand, if code reviews address security issues, XSS vulnerabilities are most frequent.

Finally, we tentatively identified review characteristics that led to the uncovering of security issues. We found that if more than two reviewers are involved in a review, more security issues were discovered than if a review was done according to the two-reviewer policy set by Chromium. This result contrasts earlier work that found the optimal number of reviewers to be two [35]. Thus, we suggest practitioners trial security reviews with an increased number of reviewers involved.

B. Future research directions

In addition to the research directions mentioned in Section V, we foresee two other alternative paths.

First, we were surprised that MCR practices in Chromium do not include the use of static code analysis tools to evaluate the submitted patch. This holds true especially when dealing with a C++ codebase. The only insight that we were able to gather is that Valgrind is used to catch memory and threading issues in test cases [36] [37]. As an example, the lack of use of static analysis tools is particularly evident when analyzing results that concern the Buffer Overflow category. Despite being an historically famous category of issue, with books [38] and tools [39] written to prevent and address them, it is still causing its fair share of trouble. As static tools are available that analyze codebases to address potential vulnerabilities, the question that one could ask in this situation is: Could any of them provide evidence of such issues? Studies can be carried out to understand why this situation happens and whether the most common security issues in a popular software as Chromium could be prevented or exposed enough to reviewers, by using any kind of static source code analysis tools from industry and academia.

Second, our research focused only on Chromium, without investigating other software products from different domains, that also have security issues. Further work can be done in replicating our study and analyzing if, given different software products, other insights could be gathered on the topic.

REFERENCES

[1] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. Software inspections: An effective verification process. *IEEE Software*, 6(3):31–36, 1989.

[2] Matthew Finifter and David Wagner. Exploring the relationship between web application development tools and security. In *USENIX conference on Web application development*, 2011.

[3] Sami Kollanus and Jussi Koskinen. Survey of Software Inspection Research. *The Open Software Engineering Journal*, 3(1):15–34, 2009.

[4] Jason Cohen. Modern code review. In Andy Oram and Greg Wilson, editors, *Making Software*, chapter 18, pages 329–338. O'Reilly, 2010.

[5] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. *Proceedings - International Conference on Software Engineering*, pages 712–721, 2013.

[6] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE 2016, pages 285–296, May 2016.

[7] P. Rigby, B. Cleary, F. Painchaud, M.A. Storey, and D. German. Open source peer review–lessons and recommendations for closed source. *IEEE Software*, 2012.

[8] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, pages 1–44.

[9] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the Qt, vtk, and itk projects. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering*, SANER 2015, pages 171–180. IEEE, 2015.

[10] Mika V. Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[11] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 202–211, 2014.

[12] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An empirical study on the effectiveness of security code review. In *Engineering Secure Software and Systems - 5th International Symposium, ESSoS 2013, Paris, France, February 27 - March 1, 2013. Proceedings*, pages 197–212, 2013.

[13] Google Inc. The Chromium Project. https://www.chromium.org/Home.

[14] CVE - Top 50 Products By Total Number Of "Distinct" Vulnerabilities in 2015. http://www.cvedetails.com/top-50-products.php?year=2015. [Online; accessed 13-06-2016].

[15] ME Fagan. Design and code inspections to reduce errors in program development, ibm systems journal., vol. 15, 1976.

[16] Lawrence G Votta Jr. Does every inspection need a meeting? *ACM SIGSOFT Software Engineering Notes*, 18(5):107–114, 1993.

[17] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, 1998.

[18] Richard A Baker Jr. Code reviews enhance software quality. In *Proceedings of the 19th international conference on Software engineering*, pages 570–571. ACM, 1997.

[19] Bertrand Meyer. Design and code reviews in the age of the internet. *Communications of the ACM*, 51(9):66–71, 2008.

[20] Erik Arisholm, Hans Gallis, Tore Dybå, and Dag IK Sjoberg. Evaluating pair programming with respect to system complexity and programmer expertise. *Software Engineering, IEEE Transactions on*, 33(2):65–86, 2007.

[21] Chris F Kemerer and Mark C Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *Software Engineering, IEEE Transactions on*, 35(4):534–550, 2009.

[22] P Thongtanunam and S McIntosh. Investigating code review practices in defective files: an empirical study of the Qt system. *Mining Software Repositories*, pages 168–179, 2015.

[23] Michael Howard, David LeBlanc, and John Viega. 24 deadly sins of software security: Programming flaws and how to fix them. 2009.

[24] Contributing Code to The Chromium Project. https://www.chromium.org/developers/contributing-code. [Online; accessed 13-06-2016].

[25] Google Chrome - Security Vulnerabilities Published In 2015. http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-15031/year-2015/Google-Chrome.html. [Online; accessed 13-06-2016].

[26] Google Inc. Rietveld Code Review. https://github.com/rietveld-codereview/rietveld.

[27] Google Inc. Monorail. https://bugs.chromium.org/hosting/.

[28] Bug Life Cycle and Reporting Guidelines. https://www.chromium.org/for-testers/bug-reporting-guidelines. [Online; accessed 13-06-2016].

[29] Mario F Triola. *Elementary statistics*. Pearson/Addison-Wesley Reading, MA.

[30] CWE - Common Weakness Enumerator. https://cwe.mitre.org/.

[31] The Open Web Application Security Project. OWASP Top 10 - 2013 – the ten most critical web application security risks. http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf, 2013.

[32] Common Weakness Enumerator. CWE-704: Incorrect type conversion or cast. https://cwe.mitre.org/data/definitions/704.html.

[33] Charles Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904.

[34] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of FSE 2012 (20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering)*, FSE '12, pages 51:1–51:11. ACM, 2012.

[35] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.

[36] Life of a Chromium Developer. https://docs.google.com/presentation/d/1abnqM9j6zFodPHA38JG1061rG2iGj_GABxEDgZsdbJg/present?slide=id.i161. [Online; accessed 13-06-2016].

[37] Using Valgrind - The Chromium Projects. https://www.chromium.org/developers/how-tos/using-valgrind. [Online; accessed 13-06-2016].

[38] Robert C Seacord. *Secure Coding in C and C++*. Pearson Education, 2005.

[39] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, pages 257–267. IEEE, 2000.