Why Did This Reviewed Code Crash?
An Empirical Study of Mozilla Firefox

Le An and Foutse Khomh
Polytechnique Montréal
{le.an, foutse.khomh}@polymtl.ca

Shane McIntosh
McGill University
shane.mcintosh@mcgill.ca

Marco Castelluccio
Mozilla Corporation and Università Federico II
mcastelluccio@mozilla.com

***Abstract*—Code review, *i.e.*, the practice of having other team members critique changes to a software system, is a pillar of modern software quality assurance approaches. Although this activity aims at improving software quality, some high-impact defects, such as crash-related defects, can elude the inspection of reviewers and escape to the field, affecting user satisfaction and increasing maintenance overhead. In this research, we investigate the characteristics of crash-prone code, observing that such code tends to have high complexity and depend on many other classes. In the code review process, developers often spend a long time on and have long discussions about crash-prone code. We manually classify a sample of reviewed crash-prone patches according to their purposes and root causes. We observe that most crash-prone patches aim to improve performance, refactor code, add functionality, or fix previous crashes. Memory and semantic errors are identified as major root causes of the crashes. Our results suggest that software organizations should apply more scrutiny to these types of patches, and provide better support for reviewers to focus their inspection effort by using static analysis tools.**

***Index Terms*—Crash analysis, Code review, Software maintenance, Mining software repositories**

I. Introduction

A software crash refers to an unexpected interruption of software functionality in an end user environment. Crashes may cause data loss and frustration of users. Frequent crashes can decrease user satisfaction and affect the reputation of a software organization. Practitioners need an efficient approach to identify crash-prone code early on, in order to mitigate the impact of crashes on end users. Nowadays, software organizations like Microsoft, Google, and Mozilla are using crash collection systems to automatically gather field crash reports, group similar crash reports into crash-types, and file the most frequently occurring crash-types as bug reports.

Code review is an important quality assurance activity where other team members critique changes to a software system. Among other goals, code review aims to identify defects at early stages of development. Since reviewed code is expected to have better quality, one might expect that reviewed code would tend to cause few severe defects, such as crashes. However, despite being reviewed, many changes still introduce defects, including crashes. For example, Kononenko et al. [21] find that 54% of reviewed code changes still introduce defects in Mozilla projects.

In this paper, we intend to understand the reasons why reviewed code still led to crashes. To achieve these goals, we mine the crash collection, version control, issue tracking, and code reviewing systems of the Mozilla Firefox project. More specifically, we address the following two research questions:

RQ1: What are the characteristics of reviewed code that is implicated in a crash?

We find that crash-prone reviewed patches often contain complex code, and classes with many other classes depending on them. Crash-prone patches tend to take a longer time and generate longer discussion threads than non-crash-prone patches. This result suggests that reviewers need to focus their effort on the patches with high complexity and on the classes with a complex relationship with other classes.

RQ2: Why did reviewed patches crash?

To further investigate why some reviewed code crashes, we perform a manual classification on the purposes and root causes of a sample of reviewed patches. We observe that the reviewed patches that crash are often used to improve performance, refactor code, address prior crashes, and implement new features. These findings suggest that software organizations should impose a stricter inspection on these types of patches. Moreover, most of the crashes are due to memory (especially null pointer dereference) and semantic errors. Software organizations can perform static code analysis prior to the review process, in order to catch these memory and semantic errors before crashes escape to the field.

The rest of the paper is organized as follows. Section II provides background information on Mozilla crash collection system and code review process. Section III describes how we identify reviewed code that leads to crashes. Section IV describes our data collection and analysis approaches. Section V discusses the results of the two research questions. Section VI discloses the threats to the validity of our study. Section VII discusses related work, and Section VIII draws conclusions.

II. The Mozilla Crash Collecting System and Code Review Process

In this section, we describe approaches of Mozilla on crash report collection and code review.

A. The Mozilla Crash Collection System

Mozilla integrates the Mozilla Crash Reporter, a crash report collection tool, into its software applications. Once a Mozilla

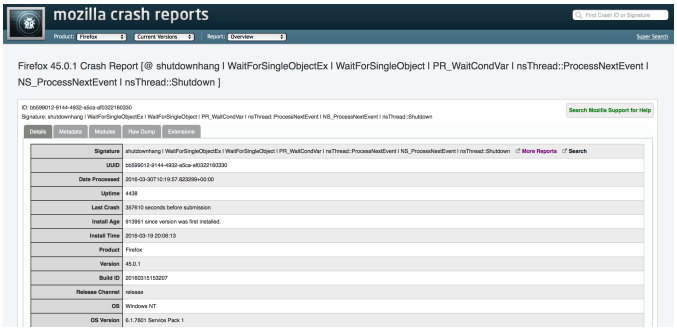

Figure 1: An example of crash report in Socorro.

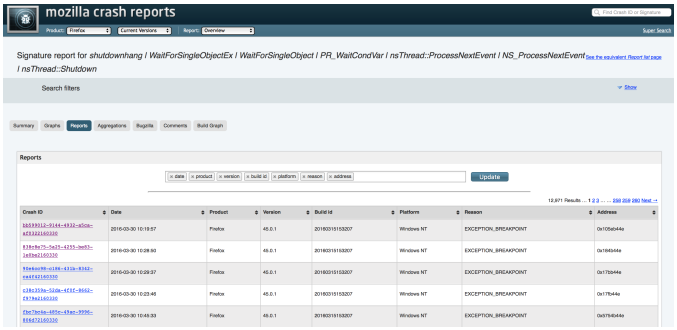

Figure 2: An example of crash-type in Socorro.

application, such as the Firefox browser, unexpectedly halts, the Mozilla Crash Reporter will generate a detailed crash report and send it to the *Socorro* crash report server [35]. Each crash report includes a stack trace of the failing thread and the details of the execution environment of the user. Figure 1 shows an example Socorro crash report. These crash reports are a rich source of information, which provide developers and quality assurance personnel with information that can help them to reproduce the crash in a testing environment.

The Socorro server automatically clusters the collected crash reports into *crash-types* according to the similarity of the top method invocations of their stack traces. Figure 2 shows an example Mozilla crash-type. The Socorro server ranks crash-types according to their frequency, *e.g.*, Socorro publishes a daily top 50 crash-types, *i.e.*, the crash-types with the maximum number of crash reports, for each of the recent releases of Firefox.

Socorro operators file top-ranked crash-types as issue reports in the *Bugzilla* issue tracking system. Quality assurance teams use Socorro to triage these crash-related issue reports and assign severity levels to them [1]. For traceability purposes, Socorro crash reports provide a list of the identifiers of the issues that have been filed for each crash-type. This link is initiated from Bugzilla. If a bug is opened from a Socorro crash, it is automatically linked. Otherwise, developers can add Socorro signatures to the bug reports. By using these traceability links, software practitioners can directly navigate to the corresponding issues (in Bugzilla) from the summary of a crash-type in the web interface of Socorro. Note that different crash-types can be linked to the same issue, while different issues can also be linked to the same crash-type [19].

B. The Mozilla Code Review Process

Mozilla manages its code review process using issue reports in Bugzilla. After writing a patch for an issue, the developer can request peer reviews by setting the `review?` flag on the patch. At Mozilla, the reviewers are often chosen by the patch author herself [16]. If the patch author does not know who should review her patch, they can consult a list of module owners and peers. Senior developers can also often recommend good reviewers. The designated reviewers need to inspect a patch from various aspects [29], such as correctness, style, security, performance, and compatibility. Once a developer has reviewed the patch, they can record comments with a review flag, which also indicates their vote, *i.e.*, in support of (+) or in opposition to (−) the patch. Mozilla applies a two-tiered code review process, *i.e.*, *review* and *superreview*. A *review* is performed by the owner of the module or peer who has expertise in a specific aspect of the code of the module [7]; while a *superreview* is required for certain types of changes, such as significant architectural refactoring, API or pseudo-API changes, or changes that affect the interactions of modules [38]. Therefore, to evaluate patches, there are four possible voting combinations on a reviewed patch: `review+`, `review-`, `superreview+`, and `superreview-`.

A code review may have several iterations. Unless the patch receives only positive review flags (`review+` or `superreview+`), it cannot be integrated into the VCS of Mozilla. In this case, the patch author needs to provide a revised patch for reviewers to consider. Some Mozilla issues are resolved by a series of patches. Since the patches are used to address the same issue, reviewers need to inspect the entire series of patches before providing a review decision. In the trial review platform of Mozilla, ReviewBoard, the patches of an issue are automatically grouped together [28]. Thus, in this paper, we examine the review characteristics at the issue level. Finally, the Tree Sheriffs [41] (*i.e.*, engineers who support developers in committing patches, ensuring that the automated tests are not broken after commits, and monitoring intermittent failures, and reverting problematic patches) or the patch authors themselves will commit the reviewed patches to the VCS.

III. Identifying Reviewed Code that Crashes

In this section, we describe our approach to identify reviewed code that is implicated in a crash report. Our approach consists of three steps: identifying crash-related issues, identifying commits that are implicated in future crash-related issues, and linking code reviews to commits. Below, we elaborate on each of these steps.

A. Identifying Crash-related Issues

Mozilla receives 2.5 million crash reports on the peak day of each week. In other words, the Socorro server needs to process around 50GB of data every day [36]. For storage capacity and

privacy reasons, Socorro only retains those crash reports that occurred within the last six months. Historical crash reports are stored in a crash analysis archive[1]. We mine this archive to extract the *issue list*, which contains issues that are linked to a crash, from each crash event. These issues are referred as to *crash-related issues* in the rest of this paper.

B. Identifying Commits that are Implicated in Future Crash-related Issues

We apply the SZZ algorithm [34] to identify commits that introduce crash-related issues. First of all, we use Fischer et al.'s heuristic [13] to find commits that fixed a crash-related issue I by using regular expressions to identify issue IDs from commit messages. Then, we extract the modified files of each crash-fixing commit with the following Mercurial command:

```
hg log --template {node},{file_mods}
```

By using the CLOC tool [6], we find that 51% of the Firefox codebase is written in C/C++. Although JavaScript and HTML (accounts for respectively 20% and 14% in the code base) are the second and third most used languages. Code implemented by these languages cannot directly cause crashes because it does not have direct hardware access. Crash-prone Javascript/HTML changes are often due to the fault of parsers, which are written in C/C++. Therefore, in this paper, we focus our analysis on C/C++ code. Given a file F of a crash-fixing commit C, we extract C's parent commit C', and use the `diff` command of Mercurial to extract F's deleted line numbers in C', henceforth referred to as *rm_lines*. Next, we use the `annotate` command of Mercurial to identify the commits that introduced the *rm_lines* of F'. We filter these potential crash-introducing candidates by removing those commits that were submitted after I's first crash report. The remaining commits are referred to as *crash-inducing commits*.

As mentioned in Section II-B, Mozilla reviewers and release managers consider all patches together in an issue report during the review process. If an issue contains multiple patches, we bundle its patches together. Among the studied issues whose patches have been approved by reviewers, we identify those containing committed patches that induce crashes. We refer to those issues as *crash-inducing issues*.

IV. Case Study Design

In this section, we present the selection of our studied system, the collection of data, and the analysis approaches that we use to address our research questions.

A. Studied System

We use Mozilla Firefox as the subject system because at the time of writing of this paper, only the Mozilla Foundation has opened its crash data to the public [42]. It is also the reason why in most previous empirical studies of software crashes (*e.g.*, [20], [19]), researchers analyzed data from the Mozilla Socorro crash reporting system [35]. Though Wang et al. [42] studied another system, Eclipse, they could obtain crash information from the issue reports (instead of crash reports). However, the exact crash date cannot be obtained from the issue reports, which hampers our ability to apply the SZZ algorithm. Dang et al. [11] proposed a method, ReBucket, to improve the current crash report clustering technique based on call stack matching. The studied collection of crash reports from the Microsoft Windows Error Reporting (WER) system is not accessible for the public.

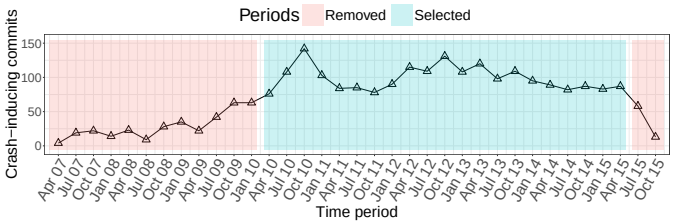

Figure 3: Number of crash-inducing commits during each three months from March 2007 to September 2015. Periods with low number of crash-inducing commits are removed.

B. Data Collection

We analyze the Mozilla crash report archive. We collect crash reports that occurred between February 2010 (the first crash recorded date) until September 2015. We collect issue reports that were created during the same period. We only take closed issues into account. We filter out the issues that do not contain any successfully reviewed patch (*i.e.*, patch with a review flag `review+` or `superreview+`). To select an appropriate study period, we analyze the rate of crash-inducing commits throughout the collected timeframe (March 2007 until September 2015). Figure 3 shows the rate of crash-inducing commits over time. In this figure, each time point represents one quarter (three months) of data. We observe that the rate of crash-inducing commits increases from January 2007 to April 2010 before stabilizing between April 2010 and April 2015. After April 2015, the rate suddenly drops. Since the last issue report is collected in September 2015, there is not enough related information to identify crash-inducing commits during the last months. Using Figure 3, we select the periods between April 2010 and April 2015 as our study period and focus our analysis on the crash reports, commits, and issue reports during this period. In total, we analyze 9,761,248 crash-types (from which 11,421 issue IDs are identified), 41,890 issue reports, and 97,840 commits. By applying the SZZ algorithm from Section III-B, we find 1,202 (2.9%) issue reports containing reviewed patches that are implicated in crashes.

C. Data Extraction

We compute metrics for reviewed patches and the source code of the studied system. Figure 4 provides an overview of our data extraction steps. To aid in the replication of our study, our data and scripts are available online.[2]

[1] https://crash-stats.mozilla.com/api/

[2] https://github.com/swatlab/crash_review

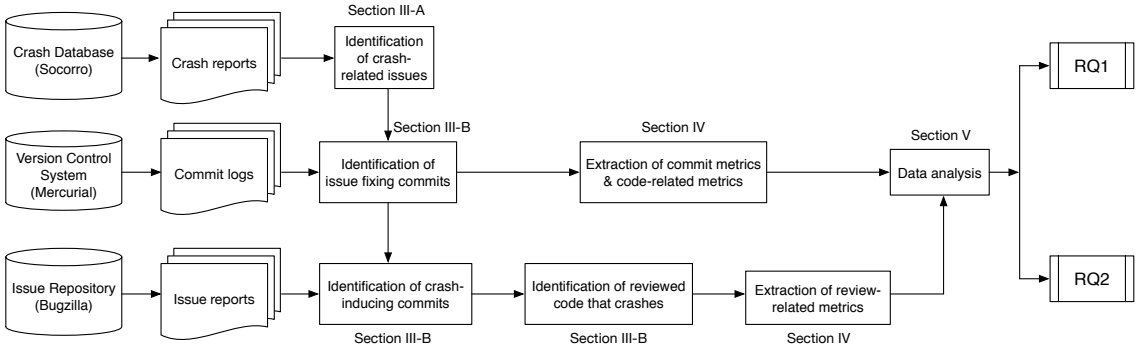

Figure 4: Overview of our approach to identify and analyze reviewed code that crashed in the field.

1) Review Metrics: For each reviewed patch, we extract the names of the author and reviewer(s), as well as its creation date, reviewed date, patch size, and the votes from each of the review activities. We also extract the list of modified files from the content of the patch. Although main review activities of Mozilla are organized in Bugzilla attachments, we can also extract additional review-related information from Bugzilla comments and transaction logs. If a comment is concerned with an attachment like a patch, Bugzilla provides a link to the attachment in the comment. We can use this to measure the review discussion length of a patch. Bugzilla attachments only contain votes on review decisions, such as `review+` and `review-`. To obtain the date when a review request for a patch was created, we search for the `review?` activity date in the issue discussion history. As we consider all of the patches of an issue together, we use the mean to aggregate patch-specific values to the issue-level. Unlike other systems, such as Qt [40], Mozilla does not allow self-review, *i.e.*, the author of a patch cannot act as a reviewer of that patch. However, Mozilla patch authors may set the `review+` score themselves, from time to time, when reviewers are generally satisfied with the patch with the exception of minor changes. Thus in this paper, we remove the patch author from the reviewer list of each of the studied issues. More details on our review metrics are provided in Section V.

2) Code Complexity Metrics: To analyze whether reviewed code that crashed in the field is correlated with code complexity, we compute code complexity metrics using the *Understand* static code analysis tool [32]. We wrote a script to compute five code complexity metrics for each C/C++ file using Understand, *i.e.*, Lines Of Code (LOC), average cyclomatic complexity, number of functions, maximum nesting level, and the proportion of comment lines in a file. More details on our complexity metrics are provided in Section V.

3) Social Network Analysis Metrics: To measure the relationship among classes, we apply Social Network Analysis (SNA) [14] to measure the centrality [33] of each C/C++ class, *i.e.*, the degree to which other classes depend on a certain class. A high centrality value indicates that a class is important to a large portion of the system, and any change to the class may impact a large amount of functionality. We compute centrality using the class-to-class dependencies that are provided by Understand. We combine each `.c` or `.cpp` file with its related `.h` file into a *class node*. We use a pair of vertices to represent the dependency relationship between any two mutually exclusive class nodes. Then, we build an adjacency matrix [2] with these vertex pairs. By using the *igraph* network analysis tool [10], we convert the adjacency matrix into a call graph, based on which we compute the PageRank, betweenness, closeness, indegree, and outdegree SNA metrics.

V. Case Study Results

In this section, we present the results of our case study. For each research question, we present the motivation, our data processing and analysis approaches, and the results.

RQ1: What are the characteristics of reviewed code that is implicated in a crash?

Motivation. We intend to compare the characteristics of the reviewed patches that lead to crashes (Crash) with those that did not lead to crashes (Clean). Particularly, we want to know whether patch complexity, centrality, and developer participation in the code review process are correlated with the crash proneness of a reviewed patch. The result of this research question can help software organizations improve their code review strategy; focusing review efforts on the most crash-prone code.

Approach. We extract information from the source code to compute code complexity and SNA metrics and from issue reports to compute review metrics. Tables I to III provide descriptions of each of the studied metrics.

We assume that changes to complex classes are likely to lead to crashes because complex classes are usually more difficult to maintain. Inappropriate changes to complex classes may result in defects or even crashes. The SNA metrics are used to estimate the degree of centrality (see Section IV-C3) of a class. Inappropriate changes to a class with high centrality may impact dependent classes; thus causing defects or even crashes. For each SNA metric, we compute the mean of all class values for the commits that fix an issue. Regarding the review metrics, we assume that patches with longer review duration and more

Table I: Code complexity metrics used to compare the characteristics of crash-inducing patches and clean patches.

Metric	Description	Rationale
Patch size	Mean number of lines of the patch(s) of an issue. We include context lines and comment lines because reviewers need to read all these lines to inspect a patch.	The larger the code changes, the easier it is for reviewers to miss defects [21].
Changed file number	Mean number of changed C/C++ files in the issue fixing commit(s).	If a change spreads across multiple files, it is difficult for reviewers to detect defects [21].
LOC	Mean number of the lines of code in the changed classes to fix an issue.	Large classes are more likely to crash [20].
McCabe	Mean value of McCabe cyclomatic complexity [22] in all classes of the issue fixing commit(s).	Classes with high cyclomatic complexity are more likely to lead to crashes [20].
Function number	Mean number of functions in all classes in the issue fixing commit(s).	High number of functions indicates high code complexity [3], which makes it difficult for reviewers to notice defects.
Maximum nesting	Mean of maximum level of nested functions in all classes in the issue fixing commit(s).	Code with deep nesting level is more likely to cause crashes [20].
Comment ratio	Mean ratio of the lines of comments over the lines of code in all classes of the issue fixing commit(s)	Reviewers may have difficulty to understand code with low ratio of comment [17], thus miss crash-prone code.

Table II: Social network analysis (SNA) metrics used to compare the characteristics of crash-inducing patches and clean patches. We compute the mean of each metric across the classes of the fixing patch(es) within an issue. *Rationale*: An inappropriate change to a class with high centrality value [33] can lead to malfunctions in the dependent classes; even cause crashes [20].

Metric	Description
PageRank	Time fraction spent to "visit" a class in a random walk in the call graph. If an SNA metric of a class is high, this class may be triggered through multiple paths.
Betweenness	Number of classes passing through a class among all shortest paths.
Closeness	Sum of lengths of the shortest call paths between a class and all other classes.
Indegree	Numbers of callers of a class.
Outdegree	Numbers of callees of a class.

review comments have higher risk of crash proneness. Since these patches may be more difficult to understand, although developers may have spent more time and effort to review and comment on them. We use the review activity metrics that were proposed by Thongtanunam et al. [40]. In addition, we also take *obsolete* patches into account because these patches were not approved by reviewers. The percentage of the obsolete patches that fix an issue can help to estimate the quality and the difficulty of the patches on an issue, as well as developer participation.

We apply the two-tailed *Mann-Whitney U test* [15] to compare the differences in metric values between crash-inducing patches and clean patches. We choose to use the Mann-Whitney U test because it is *non-parametric*, *i.e.*, it does not assume that metrics must follow a normal distribution. For the statistical test of each metric, we use a 95% confidence level (*i.e.*, $\alpha = 0.05$) to decide whether there is a significant difference among the two categories of patches. Since we will investigate characteristics on multiple metrics, we use the Bonferroni correction [12] to control the familywise error rate of the tests. In this paper, we compute the adjusted p-value, which is multiplied by the number of comparisons.

For the metrics that have a significant difference between the crash-inducing and clean patches, we estimate the magnitude of the difference using *Cliff's Delta* [5]. Effect size measures report on the magnitude of the difference while controlling for the confounding factor of sample size [8].

To further understand the relationship between crash proneness and reviewer origin, we calculate the percentage of crash-inducing patches that were reviewed by Mozilla developers,

Table III: Review metrics used to compare the characteristics of crash-inducing patches and clean patches. We compute the mean metric value across the patches within an issue.

Metric	Description	Rationale
Review iterations	Number of review flags on a reviewed patch.	Multiple rounds of review may help to better identify defective code than a single review round [40].
Number of comments	Number of comments related with a reviewed patch.	Review with a long discussion may help developers to discover more defects [40].
Comment words	Number of words in the message of a reviewed patch.	
Number of reviewers	Number of unique reviewers involved for a patch.	Patches inspected by multiple reviewers are less likely to cause defects [30].
Proportion of reviewers writing comments	Number of reviewers writing comments over all reviewers.	Reviews without comments have higher likelihood of defect proneness [40], [23].
Negative review rate	Number of disagreement review flags over all review flags.	High negative review rate may indicate a low quality of a patch.
Response delay	Time period in days from the review request to the first review flag.	Patches that are promptly reviewed after their submission are less likely to cause defects [30].
Review duration	Time period in days from the review request until the review approval.	Long review duration may indicate the complexity of a patch and the uncertainty of reviewers on it, which may result in a crash-prone patch.
Obsolete patch rate	Number of obsolete patches over all patches in an issue.	High proportion of obsolete patch indicates the difficulty to address an issue, and may imply a high crash proneness for the landed patch.
Amount of feedback	Quantity of feedback given from developers. When a developer does not have enough confidence on the resolution of a patch, she would request for feedback prior to the code review.	The higher the amount of feedback, the higher the uncertainty of the patch author, which can imply a higher crash proneness.
Negative feedback rate	Quantity of negative feedback over all feedback.	High negative feedback rate may imply high crash proneness for a patch.

Table IV: Median metric value of crash-inducing patches (Crash) and clean (Clean) patches, adjusted p-value of Mann-Whitney U test, and Cliff's Delta effect size.

Metric	Crash	Clean	p-value	effect size
Code complexity metrics				
Patch size	406	111	**<0.001**	**0.53 (large)**
Changed files	4.8	2.0	**<0.001**	**0.49 (large)**
LOC	1259.3	1124.5	0.2	–
McCabe	3.0	3.0	0.5	–
Function number	45.8	43.0	0.3	–
Maximum nesting	3.0	3.0	1	–
Comment ratio	0.3	0.2	**<0.001**	**0.24 (small)**
Social network analysis metrics				
PageRank	4.4	3.2	**<0.001**	**0.17 (small)**
Betweenness	50,743.5	22,011.3	**<0.001**	**0.16 (small)**
Closeness	2.2	2.1	**<0.001**	0.12 (negligible)
Indegree	12.0	7.5	**<0.001**	**0.15 (small)**
Outdegree	27.3	26.0	**0.02**	0.05 (negligible)
Review metrics				
Review iterations	1.0	1.0	**0.001**	0.03 (negligible)
Number of comments	0.5	0	**<0.001**	**0.15 (small)**
Comment words	2.5	0	**<0.001**	**0.16 (small)**
Number of reviewers	1.0	1.0	1	–
Proportion of reviewers writing comments	1	1	**<0.001**	0.10 (negligible)
Negative review rate	0	0	**0.03**	0.01 (negligible)
Response delay	14.2	8.1	**<0.001**	0.14 (negligible)
Review duration	15.2	8.2	**<0.001**	**0.15 (small)**
Obsolete patch rate	0	0	1	–
Amount of feedback	0	0	**0.03**	0.02 (negligible)
Negative feedback rate	0	0	1	–

external developers, and by both Mozilla and external developers. Previous work, such as [26], used the suffix of an email address to determine the affiliation of a developer. However, many Mozilla employees use an email address other than `mozilla.com` in Bugzilla, when they review code. To make our results more accurate, an author of the paper, who is working at Mozilla, used a private API to examine whether a reviewer is a Mozilla employee.

Results. Table IV compares the reviewed patches that lead to crash (Crash) to those that do not crash (Clean). Statistically significant p-values and non-negligible effect size values are shown in bold. Figure 5 visually compares crash-inducing and clean patches on the metrics (after removing outliers because they can bury the median values), where there is a statistically significant difference and the effect size is not negligible. In this figure, the red bold line indicates the median value on the crash-inducing patches (or clean patches) for a metric. The dashed line indicates the overall median value of a metric. The width variation in each plot shows the variation of the data density.

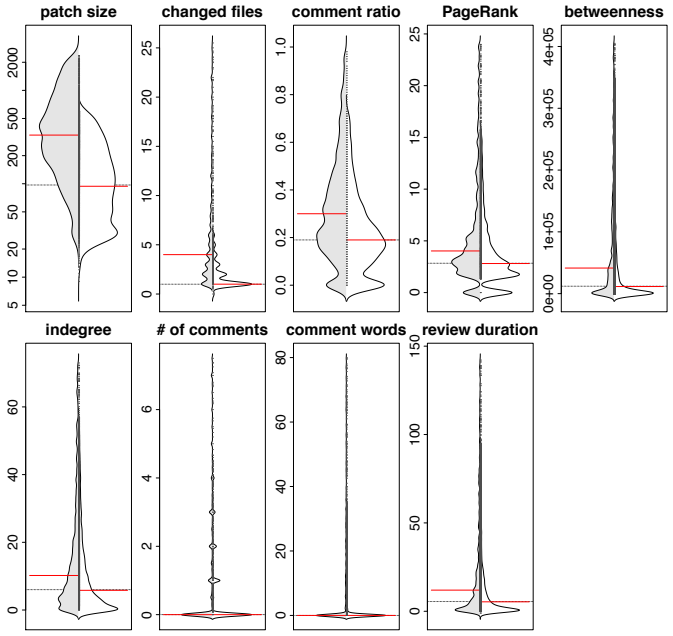

Figure 5: Comparison between crash-inducing patches (left part, grey) *vs.* clean patches (right part, white). Since we removed outliers from the plots, the median values may not correspond to the values in Table IV, which includes the outliers.

For the code complexity metrics, crash-inducing patches have a significantly larger patch size, higher number of changed files, and higher comment ratio than clean patches. The magnitude of the differences on patch size and changed files is large; while the magnitude of the differences on comment ratio is small. This result implies that the related files of the reviewed patches that crash tend to contain complex code. These files have higher comment ratio because developers may have to leave more comments to describe a complicated or difficult problem. Our finding suggests that reviewers need to double check the patches that change complex classes before approving them. Investigators also need to carefully approve patches with intensive discussions because developers may not be certain about the potential impact of these patches.

In addition, crash-inducing patches have significantly higher centrality values than clean patches on all of the social network analysis metrics. The magnitude of closeness and outdegree is negligible; while the magnitude of PageRank, betweenness, and indegree is small. This result suggests that the reviewed patches that have many other classes depending on them are more likely to lead to crashes. Reviewers need to carefully inspect the patches with high centrality.

Regarding the review metrics, compared to clean patches, crash-inducing patches have significantly higher number of comments and comment words. This finding is in line with the results in [21], where the authors also found that the number of comments have a negative impact on code review quality. The response time and review duration on crash-inducing patches

Table V: Origin of the developers who reviewed clean patches and crash-inducing patches.

Origin	Total	Crash	Crash rate
Mozilla	38,481	1,094	2.8%
External	2,512	55	2.2%
Both	897	53	5.9%
Total	41,890	1,202	2.9%

tend to be longer than clean patches. These results are expected because we assume that crash-inducing patches are harder to understand. Although developers spend a longer time and comment more on them, these patches are still more prone to crashes. In terms of the magnitude of the statistical differences, crash-inducing and clean patches that have been reviewed only have a small effect size on number of comments, comment words, and review duration; while the effect sizes of other statistical differences are negligible.

Table V shows the percentage of the patches that were reviewed by Mozilla developers, external developers, and by both Mozilla and external developers. Regarding the crash-inducing rate of the studied patches, the patches reviewed by both Mozilla and external developers lead to the highest rate of crashes (5.9%). On the one hand, there are few patches that were reviewed by both Mozilla and external developers, this result may not be representative. One the other hand, Mozilla internal members and external community members do not have the same familiarity on a specific problem, such collaborations may miss some crash-prone changes. We suggest patch authors to choose reviewers with the same level of familiarity on the changed module(s) and the whole system. In the future, we plan to further investigate the relationship between crash proneness and the institution that the reviewers represent.

Reviewed patches that crash tend to be related with large patch size and high centrality. These patches often take a long time to be reviewed and are involved with many rounds of review discussions. More review effort should be invested on the patches with high complexity and centrality values.

RQ2: Why did reviewed patches crash?

Motivation. In **RQ1**, we compared the characteristics of reviewed code that crashes with reviewed code that does not crash. To more deeply understand why reviewed patches can still lead to crashes, we perform a qualitative analysis on the purposes of the reviewed patches that crash and the root causes of their induced crashes.

Approach. To understand why developers missed the crash-inducing patches, we randomly sample 100 out of the 1,202 issues that contain reviewed patches that crash. If we use a confidence level of 95%, our sample size corresponds to a confidence interval of 9%. Inspired by Tan et al.'s work [39], we classify the purposes of patches (patch reasons) into 13 categories based on their (potential) impact on users and detected fault types. The "incorrect functionality" category defined by Tan et al. is too broad, so we break it into more detailed patch reasons: "incorrect rendering", "(other) wrong functionality", and "incompatibility". In addition, since we do not only study defect-related issues as Tan et al., we add more categories about the reason of patches, such as "refactoring", "improvement", and "test-only problem". Table VI shows the patch reasons used in our classification. We conduct a card sorting on the sampled issues with the following steps: 1) examine the issue report (the title, description, keywords, comments of developers, and the patches). Two authors of this paper individually classified each issue into one or more categories; 2) created an online document to compare categories and resolved conflicts through discussions; 3) discussed each conflict until a consensus was reached.

Then, from the results of the SZZ algorithm, we find the crash-related issues caused by the patches of the sampled

Table VI: Patch reasons and descriptions (abbreviation are shown in parentheses).

Reason	Description
Security	Security vulnerability exists in the code.
Crash	Program unexpectedly stops running.
Hang	Program keeps running but without response.
Performance degradation (perf)	Functionalities are correct but response is slow or delayed.
Incorrect rendering (rendering)	Components or video cannot be correctly rendered.
Wrong functionality (func)	Incorrect functionalities besides rendering issues.
Incompatibility (incompt)	Program does not work correctly for a major website or for a major add-on/plug-in due to incompatible APIs or libraries, or a functionality, which was removed on purpose, but is still used in the wild.
Compile	Compilation errors.
Feature	Introduce or remove features.
Refactoring (refactor)	Non-functional improvement by restructuring existing code without changing its external behaviour.
Improvement (improve)	Minor functional or aesthetical improvement.
Test-only problem (test)	Errors that only break tests.
Other	Other patch reasons, *e.g.*, data corruption and adding logging.

Table VII: Crash root causes and descriptions.

Reason	Description
Memory	Memory errors, including memory leak, overflow, null pointer dereference, dangling pointer, double free, uninitialized memory read, and incorrect memory allocation.
Semantic	Semantic errors, including incorrect control flow, missing functionality, missing cases of a functionality, missing feature, incorrect exception handling, and incorrect processing of equations and expressions.
Third-party	Errors due to incompatibility of drivers, plug-ins or add-ons.
Concurrency	Synchronization problems between multiple threads or processes, *e.g.*, incorrect mutex usage.

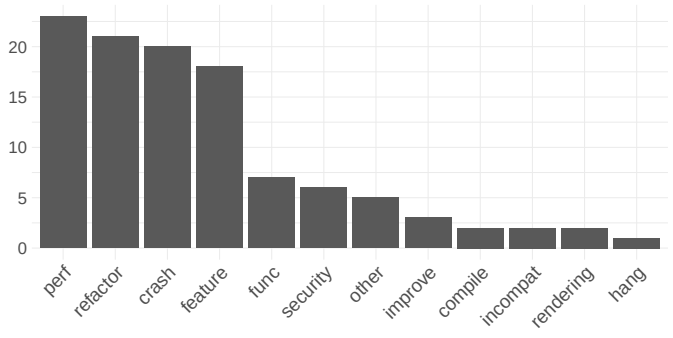

Figure 6: Distribution of the purposes of the reviewed issues that lead to crashes.

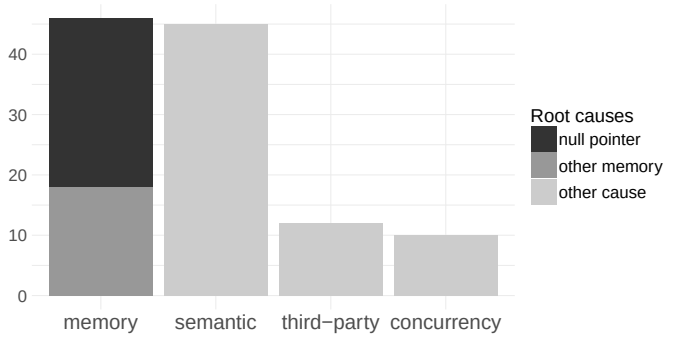

Figure 7: Distribution of the root causes of the reviewed issues that lead to crashes.

issues. Following the same card sorting steps, we classify the root causes of these crash-related issues into five categories, as shown in Table VII.

Results. Figure 6 shows the distribution of patch reasons obtained from our manual classification. Among the reviewed patches that lead to crashes, we find that most patches are used for improving Firefox' performance, refactoring code, fixing previous crashes, and implementing new features. These results imply that: 1) improving performance is the most important purpose of the reviewed patches that crash; 2) some "seemingly simple" changes, such as refactoring, may lead to crashes; 3) fixing crash-related issues can introduce new crashes; 4) many crashes were caused by new feature implementations. The classification suggests that reviewers need to scrutinize patches due to the above reasons, and software managers can ask a *super review* inspection for these types of patches.

Figure 7 shows the distribution of our manually classified root causes. According to the results, most crashes are due to memory and semantic errors. To further understand the detailed causes of the memory errors, we found that 61% of these errors are as a result of null pointer dereferences. By studying the issue reports of the null pointer crashes, we found that most of them were eventually fixed by adding check for NULL values, *e.g.*, the issue #1121661.[3] This finding is interesting because some memory faults can be avoided by static analysis. Mozilla has planned to use static analysis tools, such as *Coverity* [9] and *Clang-tidy* [4], to enhance its quality assurance. We suggest that software organizations can perform static analysis on a series of memory faults, such as null pointer dereference and memory leaks, prior to their code review process. Our results suggest that static code analysis can not only help to mitigate crashes but also certain security faults. Even though the accuracy of the static analysis cannot reach 100%, it can help reviewers to focus their inspection efforts on suspicious patches. In addition, semantic errors are also an important root cause of crashes. Many of these crashes are eventually resolved by modifying the `if` conditions of the faulty code. Semantic errors are relatively hidden in the code, we suggest reviewers to focus their inspections on changes of control flow, corner cases, and exception handling to prevent potential crashes. Software organizations should also enhance their testing effort on semantic code changes.

Reviewers should focus their effort on patches that are used to improve the performance of the software, refactor source code, fix crashes, and introduce new features, since these types of patches are more likely to lead to crashes. If possible, a super review or inspection from additional reviewers should be conducted for these patches. Memory and semantic errors are major causes of the crashes; suggesting that static analysis tools and additional scrutiny should be applied to semantic changes.

VI. Threats to Validity

Internal validity threats are concerned with factors that may affect a dependent variable and were not considered in the study. We choose steady periods for the studied commits by analyzing the distribution of crash-inducing commit numbers. We eliminate the periods where the numbers of crash-inducing commits are relatively low because some crash-inducing code has not been filed into issues at the beginning and at the end of our collected data.

The SZZ algorithm is a heuristic to identify commits that induce subsequent fixes. To mitigate the noise introduced by this heuristic, we removed all candidates of crash-introducing commits that only change comments or whitespace. We validate the accuracy of the algorithm by comparing changed files of a crash-inducing commit with the information in its corresponding crash-related issue report. As a result, 68.1% of our detected crash-inducing commits changed at least one file mentioned in the crashing stack trace or comments of their corresponding issues. The remaining commits might change a dependent class of the code in the stack trace, or developers do not provide any stack trace in their corresponding issue reports. Therefore, we believe that the SZZ algorithm can provide a reasonable starting point for identifying crash-prone changes.

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=1121661#c1

Finally, in **RQ1**, we use some time-related metrics (*e.g.*, review duration), which measures the period since a review for a patch was requested until the patch was approved. Although a review duration of two months does not mean that developers really spent two months to review a patch, it can reflect the treatment time of a development team (including pending time, understanding time, and evaluation time) to the patch. For example, when the review queue of a reviewer is long, her assigned patches may be pending for a long time before she begins to inspect them [37].

Conclusion validity threats are concerned with the relationship between the treatment and the outcome. We paid attention not to violate the assumptions of our statistical analyses. In **RQ1**, we apply the non-parametric test, the Mann-Whitney U test, which does not require that our data be normally distributed.

In our manual classifications of root causes of the reviewed patches that crashes, we randomly sampled 100 reviewed issues and the crashes that were induced. Though a larger sample size might yield more nuanced results, our results clearly show the most crash-prone types of patches, and the major root causes of the reviewed patches that crash.

Reliability validity threats are concerned with the replicability of the study. To aid in future replication studies, we share our analytic data and scripts online: https://github.com/swatlab/crash_review.

External validity threats are concerned with the generalizability of our results. In this work, we study only one subject system, mainly due to the lack of available crash reports and code review data. Thus, our findings may not generalize beyond this studied system. However, the goal of this study is not to build a theory that applies to all systems, but rather to empirically study the relationship between review activities and crash proneness. Nonetheless, additional replication studies are needed to arrive at more general conclusions.

VII. Related Work

In this section, we discuss the related research on crash analysis and code review analysis.

A. Crash Analysis

Crashes can unexpectedly terminate a software system, resulting in data loss and user frustration. To evaluate the importance of crashes in real time, many software organizations have implemented automatic crash collection systems to collect field crashes from end users.

Previous studies analyze the crash data from these systems to propose debugging and bug fixing approaches on crash-related defects. Podgurski et al. [27] introduced an automated failure clustering approach to classify crash reports. This approach enables the prioritization and diagnosis of the root causes of crashes. Khomh et al. [19] proposed an entropy-based approach to identify crash-types that frequently occurred and affect a large number of users. Kim et al. [20] mined crash reports and the related source code in Firefox and Thunderbird to predict top crashes for a future release of a software system. To reduce the efforts of debugging crashing code, Wu et al. [43] proposed a framework, ChangeLocator, which can automatically locate crash-inducing changes from a given bucket of crash reports.

In this work, we leverage crash data from the Mozilla Socorro system to quantitatively and qualitatively investigate the reasons why reviewed code still led to crashes, and make suggestions to improve the code review process.

B. Code Review & Software Quality

One important goal of code review is to identify defective code at early stages of development before it affects end users. Software organizations expect that this process can improve the quality of their systems.

Previous studies have investigated the relationship between code review quality and software quality. McIntosh et al. [23], [24] found that low code review coverage, participation, and expertise share a significant link with the post-release defect proneness of components in the Qt, VTK, and ITK projects. Similarly, Morales et al. [25] found that code review activity shares a relationship with design quality in the same studied systems. Thongtanunam et al. [40] found that lax code reviews tend to happen in defect-prone components both before and after defects were found, suggesting that developers are not aware of problematic components. Kononenko et al. [21] observed that 54% of the reviewed changes are still implicated in subsequent bug fixes in Mozilla projects. Moreover, their statistical analysis suggests that both personal and review participation metrics are associated with code review quality. In a recent work, Sadowski et al. [31] conducted a qualitative study on the code review practices at Google. They observed that problem solving is not the only focus for Google reviewers and only a few developers said that code review have helped them catch bugs.

The results of [23], [24], [40], [21], [31] suggest that despite being reviewed, many changes still introduce defects. Therefore, in this study, we investigate the relationship between the rigour of the code review that a code change undergoes and its likelihood of inducing a software crash – a type of defect with severe implications. We draw inspiration from these prior studies to design our set of metrics [40], [18]. We also draw inspiration from Tan et al.'s work [39] to conduct a qualitative study by identifying the root causes of the reviewed patches that induce crashes and the purpose of these patches.

VIII. Conclusion

The code review process helps software organizations to improve their code quality, reduce post-release defects, and collaborate more effectively. However, some high-impact defects, such as crash-related defects, can still pass through this process and negatively affect end users. In this paper, we compare the characteristics of reviewed code that induces crashes and clean reviewed code in Mozilla Firefox. We observed that crash-prone reviewed code often has higher complexity and centrality, *i.e.*, the code has many other classes depending on it. Compared to clean code, developers tend to

spend a longer time on and have longer discussions about the crash-prone code; suggesting that developers may be uncertain about such patches (RQ1). Through a qualitative analysis, we found that the crash-prone reviewed code is often used to improve performance of a system, refactor source code, fix previous crashes, and introduce new functionalities. Moreover, the root causes of the crashes are mainly due to memory and semantic errors. Some of the memory errors, such as null pointer dereferences, could be likely prevented by adopting a stricter organizational policy with respect to static code analysis (RQ2). In the future, we plan to investigate to which extent static analysis can help to mitigate software crashes. We are also contacting other software organizations in order to study their crash reports to validate the results obtained in this work.

REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 361–370, New York, NY, USA, 2006. ACM.

[2] N. Biggs. *Algebraic graph theory*. Cambridge university press, 1993.

[3] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality?: an empirical case study of windows vista. *Communications of the ACM*, 52(8):85–93, 2009.

[4] Clang-Tidy tool. http://clang.llvm.org/extra/clang-tidy, 2017. Online; Accessed March 31st, 2017.

[5] N. Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[6] CLOC. http://cloc.sourceforge.net, 2017. Online; Accessed May 22nd, 2017.

[7] Firefox code review. https://wiki.mozilla.org/Firefox/Code_Review, 2016. Online; Accessed March 31st, 2016.

[8] R. Coe. It's the effect size, stupid: What effect size is and why it is important, 2002.

[9] Coverity tool. http://www.coverity.com, 2017. Online; Accessed March 31st, 2017.

[10] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.

[11] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1084–1093. IEEE Press, 2012.

[12] A. Dmitrienko, G. Molenberghs, C. Chuang-Stein, and W. Offen. *Analysis of Clinical Trials Using SAS: A Practical Guide*. SAS Institute, 2005.

[13] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the 29th International Conference on Software Maintenance (ICSM)*, pages 23–32. IEEE, 2003.

[14] R. A. Hanneman and M. Riddle. Introduction to social network methods, 2005.

[15] M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric statistical methods*. John Wiley & Sons, 3rd edition, 2013.

[16] How to submit a patch at Mozilla. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/How_to_Submit_a_Patch, 2017. Online; Accessed May 31st, 2017.

[17] H. Hulkko and P. Abrahamsson. A multiple case study on the impact of pair programming on product quality. In *Proceedings of the 27th International Conference on Software Engineering (ICSM)*, pages 495–504. IEEE, 2005.

[18] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[19] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan. An entropy evaluation approach for triaging field crashes: A case study of Mozilla Firefox. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 261–270. IEEE, 2011.

[20] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park. Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering*, 37(3):430–447, 2011.

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

[22] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[23] 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)*, pages 192–201. ACM, 2014.

[24] 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*, 21(5):2146–2189, 2016.

[25] R. Morales, S. McIntosh, and F. 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)*, pages 171–180. IEEE, 2015.

[26] M. Pinzger and H. C. Gall. Dynamic analysis of communication and collaboration in oss projects. In *Collaborative Software Engineering*, pages 265–284. Springer, 2010.

[27] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 465–475. IEEE, 2003.

[28] Creating Commits and Submitting Review Requests with ReviewBoard. http://mozilla-version-control-tools.readthedocs.io/en/latest/mozreview/commits.html, 2017. Online; Accessed May 31st, 2017.

[29] Mozilla Reviewer Checklist. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Reviewer_Checklist, 2017. Online; Accessed May 31st, 2017.

[30] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 541–550. ACM, 2008.

[31] 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*, pages 181–190. ACM, 2018.

[32] Understand tool. https://scitools.com, 2016. Online; Accessed March 31st, 2016.

[33] J. Scott. *Social network analysis*. SAGE publications, 2012.

[34] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[35] Socorro: Mozilla's Crash Reporting Server. https://crash-stats.mozilla.com/home/products/Firefox, 2016. Online; Accessed March 31st, 2016.

[36] Socorro: Mozilla's crash reporting system. https://blog.mozilla.org/webdev/2010/05/19/socorro-mozilla-crash-reports/, 2016. Online; Accessed March 31st, 2016.

[37] Mozilla discussion on speeding up reviews. https://groups.google.com/forum/?hl=en#!msg/mozilla.dev.planning/hGX6vy5k35o/73b3Vw9GmS8J, 2017. Online; Accessed May 31st, 2017.

[38] Super-review policy. https://www.mozilla.org/en-US/about/governance/policies/reviewers/, 2016. Online; Accessed March 31st, 2016.

[39] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[40] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*, pages 168–179, 2015.

[41] Mozilla Tree Sheriffs. https://wiki.mozilla.org/Sheriffing, 2017. Online; Accessed February 1st, 2017.

[42] S. Wang, F. Khomh, and Y. Zou. Improving bug management using correlations in crash reports. *Empirical Software Engineering*, 21(2):337–367, 2016.

[43] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*, pages 1–35, 2017.