An Empirical Study of Crash-inducing Commits in Mozilla Firefox

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

ABSTRACT

Software crashes are feared by software organisations and end users. Many software organisations have embedded automatic crash reporting tools in their software systems to help development teams track and fix crash-related bugs. Previous techniques, which focus on the triaging of crash-types and crash-related bugs, can help software practitioners increase their debugging efficiency on crashes. But, these techniques can only be applied after the crashes occurred and already affected a large population of users. To help software organisations detect and address crash-prone code early, we conduct a case study of commits that would lead to crashes, called "crash-inducing commits", in Mozilla Firefox. We found that crash-inducing commits are often submitted by developers with less experience. Developers perform more addition and deletion of lines of code in crash-inducing commits. We built predictive models to help software practitioners detect and fix crash-prone bugs early on. Our predictive models achieve a precision of 61.4% and a recall of 95.0%. Software organisations can use our proposed predictive models to track and fix crash-prone commits early on before they negatively impact users; increasing bug fixing efficiency and user-perceived quality.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Crash analysis, bug triaging, prediction model, mining software repositories.

1. INTRODUCTION

Software crashes refer to unexpected interruptions of software systems in users' environment. Frequent crashes can significantly decrease the overall user-perceived quality and even affect the reputation of a software organisation. Therefore, nowadays, many software organisations (*e.g.*, Mozilla, Microsoft, and Google) are deploying crash reporting tools in their software systems. Once the software crashes, the automatic crash reporting tool collects information on the crash event, then sends a detailed crash report to the software organisation. Crash reports are stored in a crash collecting system, where crashes with the same crashing signature (*i.e.*, the stack trace of the failing thread) are grouped into a crash-type. The crash collecting system analyses the impact of different crash-types and selects the top crash-types, which will be filed into bug tracking systems (*e.g.*, Bugzilla or Jira); enabling, quality assurance teams to focus their limited resources on fixing these important defects.

Khomh et al. [17] proposed an entropy-based crash triaging technique that computes the distribution of crash occurrences among users and assigns a higher priority to the bugs related to crashes that occur frequently and affect a large number of users. However, this approach can only capture crashes with high impact after the crash collecting system has gathered enough crash reports. During this period, the crashes may have affected a large number of users. Moreover, while time passes, the erroneous code becomes unfamiliar to developers, making it hard to correct.

To reduce the triaging period of crash-related bugs, in our previous study [2], we built statistical models in Mozilla projects to predict crash-related bugs that lead to frequent crashes, which impact a large user base. Although this improved approach can be applied at an early stage of development to detect crash-related bugs with a serious negative impact on users, software development teams still have to wait for a certain period, during which crashes are collected, triaged and filed into bug reports, before they can carry out their bug fixing activities. If software organisations could detect crash-prone code even earlier, at the time of commits, they would address the problems as soon as possible and prevent the unpleasant experience of crashes to the users, to a great extent. This approach is referred to as "Just-In-Time Quality Assurance" [16], which enables fine-grained defect predictions and allows quality assurance teams to identify error-prone code early on. By identifying error-prone commits quickly, quality assurance teams are also likely to make better decisions choosing developers to fix a bug.

In this paper, we investigate statistical models to predict commits that may introduce crashes in Mozilla Firefox. We are limited to Firefox because, at the time of this writing, no other organisation provides access to its crash reporting system. Software organisations can apply our proposed ap-

PROMISE '15, October 21 2015, Beijing, China
© 2015 ACM. ISBN 978-1-4503-3715-1/15/10…$15.00
DOI: http://dx.doi.org/10.1145/2810146.2810152

proach to detect crash-prone code early on before they affect a large number of users and address the defective code as soon as possible. We study Mozilla Firefox' crash reports between January 2012 and December 2012, as well as its commit logs from the beginning of the project until December 2012, and answer the following research questions:

RQ1: *What is the proportion of crash-inducing commits in Firefox?*

We analyse Firefox' crash reports and link them to the corresponding crash-related bugs. We then use the SZZ algorithm [32] to map these bugs to their related commits and identify the commits due to which the crash-related bugs occurred. We found that crash-inducing commits account for 25.5% in the studied version control system.

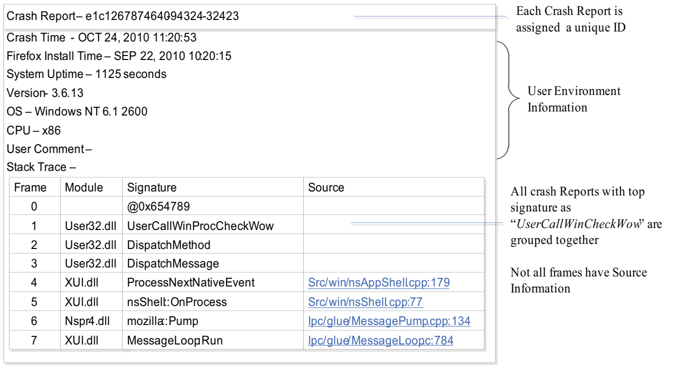
RQ2: *What characteristics do crash-inducing commits possess?*

By investigating the characteristics of crash-inducing commits and other commits, we found that, in general, crash-inducing commits are submitted by developers with less experience and are more often committed by developers from Mozilla. Developers change more files, add and delete more lines in crash-inducing commits. Compared to other commits, more crash-inducing commits fix a previous bug, and often, they lead to another bug. In terms of changed types, crash-inducing commits contain more unique changed types and the changed statements tend to be scattered in more changed types, while other commits tend to be changed on a specific changed type.

RQ3: *How well can we predict crash-inducing commits?*

Previous studies, which proposed statistical models to predict defects from bug reports, could be effective to some extent. However, before a certain type of crashes is filed into the crash collecting system, a large number of end users might have already suffered a negative experience. Moreover, during this period, developers may become less familiar with the code. In this case, they may spend more time identifying the erroneous lines to fix the problems. Therefore, statistical models that can predict error-prone code just-in-time are required to help software practitioners detect crash-inducing commits and effectively fix them early. We use GLM, Naive Bayes, C5.0, and Random Forest algorithms to predict whether or not a commit will induce future crashes. Our predictive models can reach a precision of 61.4% and a recall of 95.0%. Software organisations can apply our proposed technique to improve their defect triaging process and the satisfaction of their users.

The remainder of the paper is organised as follows. Section 2 provides background information on Mozilla crash collecting system. Section 3 explains the identification technique of crash-inducing commits. Section 4 describes data collection and processing for the empirical study. Section 5 presents and discusses the results of the three research questions. Section 6 discusses threats to the validity. Section 7 summaries related work. Section 8 draws conclusions.



Crash Report– e1c126787464094324-32423

Crash Time - OCT 24, 2010 11:20:53
Firefox Install Time – SEP 22, 2010 10:20:15
System Uptime – 1125 seconds
Version- 3.6.13
OS – Windows NT 6.1 2600
CPU – x86
User Comment –
Stack Trace –

Frame	Module	Signature	Source
0		@0x654789	
1	User32.dll	UserCallWinProcCheckWow	
2	User32.dll	DispatchMethod	
3	User32.dll	DispatchMessage	
4	XUI.dll	ProcessNextNativeEvent	Src/win/nsAppShell.cpp:179
5	XUI.dll	nsShelt:OnProcess	Src/win/nsShell.cpp:77
6	Nspr4.dll	mozilla:Pump	Ipc/glue/MessagePump.cpp:134
7	XUI.dll	MessageLoop:Run	Ipc/glue/MessageLoopc:784

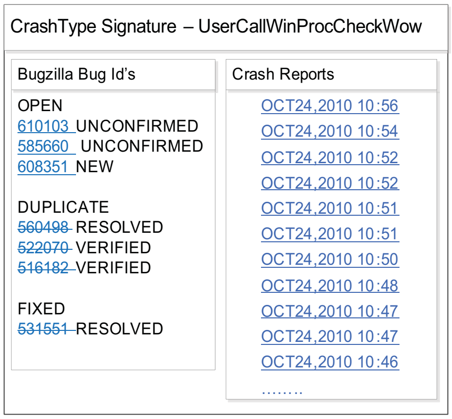
Each Crash Report is assigned a unique ID

User Environment Information

All crash Reports with top signature as "*UserCallWinCheckWow*" are grouped together

Not all frames have Source Information

Figure 1: A sample crash report from Firefox



CrashType Signature – UserCallWinProcCheckWow

Bugzilla Bug Id's	Crash Reports
OPEN 610103 UNCONFIRMED 585660 UNCONFIRMED 608351 NEW DUPLICATE ~~560498~~ RESOLVED ~~522070~~ VERIFIED ~~516182~~ VERIFIED FIXED ~~531551~~ RESOLVED	OCT24,2010 10:56 OCT24,2010 10:54 OCT24,2010 10:52 OCT24,2010 10:52 OCT24,2010 10:51 OCT24,2010 10:51 OCT24,2010 10:50 OCT24,2010 10:48 OCT24,2010 10:47 OCT24,2010 10:47 OCT24,2010 10:46 ....

Figure 2: A sample crash-type from Firefox

2. MOZILLA CRASH COLLECTING SYSTEM

Mozilla delivers software with a built-in automatic crash reporting tool, *i.e.*, the Mozilla Crash Reporter. When a Mozilla product, such as Firefox, terminates unexpectedly, Mozilla Crash Reporter will generate and send a detailed crash report to the Socorro crash report server [33]. The crash report provides a stack trace for the failing thread and information about the user's environment. A stack trace is an ordered set of frames where each frame refers to a method signature and provides a link to the corresponding source code. Different stakeholders, quality managers and developers, can use crash reports to allocate development resources. Figure 1 illustrates a sample crash report from Mozilla Firefox.

Socorro collects crash reports from end users and groups similar crash reports together by the top method signatures in their stack traces. Such a group of crash reports where all the stack traces possess the common top frames is termed as a *crash-type*. However, the subsequent frames in the stack traces might be different. Figure 2 shows a sample crash-type from Firefox.

Socorro server's data are open and provide a rich Web interface for software practitioners to analyse crash-types. In the Socorro server, crash-types are automatically ranked based on the frequency of their occurrences. Software managers can file crash-types with high crashing frequency into Bugzilla, *i.e.*, Mozilla's bug tracking system. Different crash-types can be linked to the same bug, while different bugs can

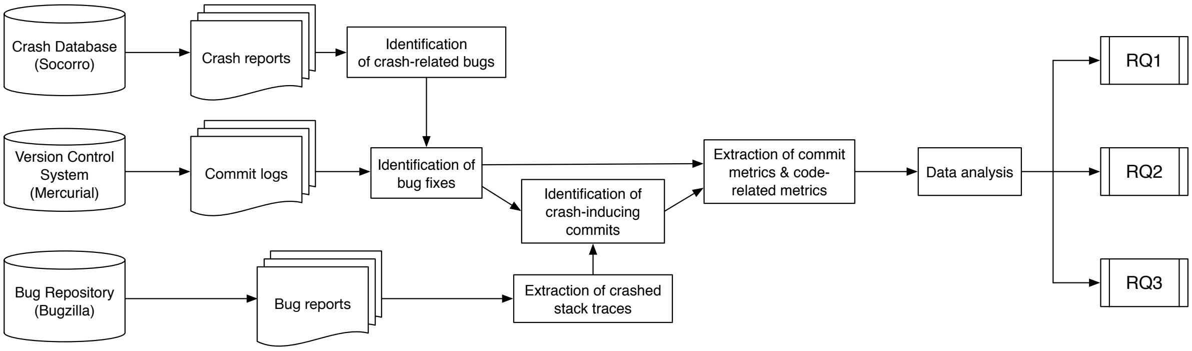

Crash Database (Socorro)
Crash reports
Identification of crash-related bugs
Version Control System (Mercurial)
Commit logs
Identification of bug fixes
Identification of crash-inducing commits
Extraction of commit metrics & code-related metrics
Data analysis
RQ1
RQ2
RQ3
Bug Repository (Bugzilla)
Bug reports
Extraction of crashed stack traces

Figure 3: Overview of our approach to identify crash-inducing commits and extract their characteristic metrics

also be linked to the same crash-type [1]. Socorro provides a list of bugs for each crash report whose crash-type has been filed into Bugzilla. The Socorro server and Bugzilla are integrated, *i.e.*, developers can directly navigate to the corresponding bugs (in Bugzilla) from a crash-type's summary in Socorro's Web interface. Developers use the information contained in crash reports to debug and fix bugs. Mozilla quality assurance teams triage bug reports and assign severity levels to the bugs [5]. Developers port patches to fix a bug. Once approved, the patches will be integrated into the source code.

3. IDENTIFICATION OF CRASH-INDUCING COMMITS

In this section, we describe the identification procedure for crash-inducing commits. All of our data and analytic scripts are available at:
https://github.com/swatlab/crash-inducing.

Applying the SZZ algorithm [32], we identify crash-inducing commits in two steps: identification of crash-related bugs and identification of commits that induce those bugs. The remainder of this section elaborates on each of these steps.

3.1 Identification of Crash-related Bugs

We extract the bug list from each of the studied crash reports. For each of the crash-related bug, we use regular expressions to identify the crashed stack trace from the bug's title and comments, then extract crash-related files or methods from the stack trace. We record the identified files or methods as defective locations of the crash-related bugs, which will be used to identify crash-inducing commits in the next step. Each crash-related bug may be linked to multiple crash occurrences. We sort these crashes by time and record the dates of the first and the last crash occurrences before the bug was opened.

3.2 Identification of Crash-inducing Commits

Since Śliwerski et al. [32] introduced the SZZ algorithm, a plethora of studies (such as [19, 28, 36]) have leveraged this approach to identify the commits that induce subsequent commits, especially bug fixes, in version control systems. In this paper, we use the SZZ algorithm to identify the commits that lead to crash-related bugs as follows.

3.2.1 Extraction of Crash-related Changed Files

We use heuristics proposed by Fischer et al. [11] to map the crash-related bug IDs to their corresponding bug fixes. We use regular expressions to detect bug IDs from the message of each commit. We then manually eliminate the false positives from the results. Some commits, which fix previous bug fixes (called supplementary bug fixes [3]), may lack bug identifier in the commit messages, where only a commit ID of a previous fix is provided. We track these commit IDs back to their original commits and check whether these original commits could be mapped to a bug report. Thus, we ensure that every crash-related bug can be mapped to all possible corresponding commits. As Mozilla's revision history is managed by Mercurial, for each of the identified bug fixes, we run a Mercurial command to extract its modified files and deleted files:

```
hg log --template {rev}, {file_mods}, {file_dels}
```

Here, we do not take added files into account, because only modified and deleted files could be changed by preceding commits.

3.2.2 Identification of the Previous Commits of the Changed Files

The changed files identified in Section 3.2.1 (*i.e.*, modified and deleted files) are considered as files that address the crash-related bugs. For each of the changed files in a certain commit C to the bug B_{crash}, if its previous commit C' is dated before the bug's first crash occurrence date, C' would be considered as a "crash-inducing commit". Concretely, to seek out the previous commits of each changed file to a specific commit, we use Mercurial's `annotate` command to track the previous commit ID of each line in this file. Among the identified commit IDs, we first remove those related to white spaces and comment lines. The remaining commit IDs are candidates of crash-inducing commits. Then, for each of the IDs, we record its committed date as $D_{candidate}$. We find out the first crash date D_{first} of the bug B_{crash}, which is extracted in Section 3.1, and compare it with $D_{candidate}$. If $D_{candidate}$ is earlier than D_{first}, this candidate commit is identified as a "crash-inducing commit". Otherwise, if $D_{candidate}$ is later than D_{first}, but earlier than the last crash date D_{last} before the opening of the bug, we will check whether this candidate commit contains any of the files appearing in the crashed stack trace of B_{crash}. If yes, we also include this commit into the set of crash-inducing commits.

All of the above steps have been implemented in Python scripts. Future researchers can use our scripts to validate our data analysis process or conduct their replication studies.

4. CASE STUDY DESIGN

This section describes the data collection and processing for our case study, which aims to address the following three research questions:

1. What is the proportion of crash-inducing commits in Firefox?

2. What characteristics do crash-inducing commits possess?

3. How well can we predict crash-inducing commits?

4.1 Data Collection

We analyse crash reports of Mozilla Firefox from January 2012 until December 2012. Since a crash-inducing commit cannot be submitted later than any of its related crashes, we select the revision history of Mozilla Firefox from the beginning of the project until December 2012. In summary, there are in total 132,484,824 crash reports (grouped into 2,210,126 crash-types) and 127,212 commits selected in this research.

4.2 Data Processing

Figure 3 shows an overview of our data processing steps for the case study. The corresponding data and Python scripts are available at:
https://github.com/swatlab/crash-inducing.

4.2.1 Mining Crash Reports

To identify crash-inducing commits and investigate the characteristics of these commits, we extract the following metrics from each crash reports: *bug list*, *crash date*, and *release number*. We use the bug IDs in the bug list to map a crash report to its bug reports. We then use crash dates to find the earliest and the latest crash occurrence dates before the opening of each bug (see Section 3.1). We use the source code of all detected releases to compute code complexity metrics and social network analysis metrics.

4.2.2 Computing Code Complexity Metrics

For each studied commit, we use the Mercurial `log` command to extract all of its changed files. Then, as in our previous work [2], we apply the source code analysis tool *Understand* [29] to compute the code-related metrics of the analysed files and identify the relationship among these files. This tool generates an Understand database (UDB), which provides a Python API[1] to allow researchers to write their own scripts and create custom data. We use a Python script to extract five metrics on code complexity for the files in each subject commit: lines of code (LOC), average cyclomatic complexity, number of functions, maximum nesting, and ratio of comment lines over all lines in a file. Because more than 90% of Firefox' code is written in C or C++ [2], in this step, we only take C and C++ files into consideration. Details of the selected code complexity metrics are discussed in Section 5.

4.2.3 Computing Social Network Analysis Metrics

From the Understand database generated in Section 4.2.2, we identify the dependency among different files in Firefox and compute Social Network Analysis (SNA) metrics for each file. Concretely, from the studied C and C++ files, we combine each *.c* or *.cpp* file and its corresponding *.h* file into a class node. We then build an adjacency matrix to represent the relationship among these nodes. We use the network analysis tool *igraph* [15] to convert the adjacency matrix into a call graph, by which we compute the following social network analysis metrics: PageRank, betweenness, closeness, indegree, and outdegree. Details of the selected SNA metrics are discussed in Section 5.

In Section 4.2.2 and Section 4.2.3, we compute the code-related metrics for each of the releases detected from Section 4.2.1. For a given commit C whose committed date is D_c, we search the latest release R whose release date D_r is satisfied: $D_r < D_c$. We map all the files in the commit C to the release R, and record the code complexity and SNA metrics for each of the successfully mapped files.

4.2.4 Identifying Changed Types

In a commit, different types of changes affect a software system to different extents in terms of crashes. For example, comment changes and refactorings may have little probability to trigger subsequent crashes. Yet, if parameters or function calls are not appropriately modified (or added/deleted) in a commit, crashes would probably happen when the commit is integrated into the version control system. We use the source code analysis tool *srcML* [34] to convert C or C++ code into XML files where each syntactic statement will be converted into an XML node, in which an XML tag labels its type. For a given changed file F in a certain commit C, we use the following Mercurial command to check it out:

```
hg cat -r C F
```

Then, we also check out the file with the same name F' in the previous commit C'. After converting F and F' into XML format, we use a Python script to recursively compare the difference on each of the corresponding srcML tags[2]. As we detected more than 80 unique srcML tags from the studied changed files, we group the srcML tags with similar semantic functions into a "changed type", while ignoring trivial srcML tags, such as "block" and "@format". Table 1 shows all of changed types and their corresponding srcML tags.

Besides counting the number of changed types in a commit, we also investigate the distribution of the changed types in the commit. We compute the value of the normalised Shannon entropy [30], defined as:

$$H_n(C) = -\sum_{i=1}^{n} p_i \times log_n(p_i) \qquad (1)$$

where C is a commit; p_i is the probability of C possessing a specific changed type CT_i ($p_i \geq 0$, and $\sum_{i=1}^{n} p_i = 1$); n is the total number of unique changed types listed in Table 1. So, for a commit, if all changed types have the same occurrences, *i.e.*, the changed types are equally distributed, the entropy is maximal (*i.e.*, 1). If a commit only has one changed type, the entropy is minimal (*i.e.*, 0).

[1] https://scitools.com/new-python-api/

[2] For all srcML tags, please refer to: http://www.srcml.org/doc/srcMLGrammar.html

Table 1: Changed types identified from Firefox' source code

Changed type	srcML tag(s)
Class	*class, class_decl, member_list*
Comment	*comment*
Constructor	*constructor, constructor_decl*
Control flow	*while, do, if, else, break, goto, for, foreach, continue, then, switch, case, return, condition, incr, default*
Data structure	*enum, struct, struct_decl, typedef, union, union_decl*
Declaration	*asm, decl, decl_stmt, using, namespace, range, specifier*
Destructor	*destructor, destructor_decl*
Function	*function, function_decl*
Initialisation	*init*
Invocation	*call*
Access modifier	*super, public, private, protected, extern*
C++ feature	*template*
Parameter	*param, parameter_list, argument, argument_list*
Preprocessor	*cpp:define, cpp:elif, cpp:else, cpp:endif, cpp:error, cpp:file, cpp:if, cpp:ifdef, cpp:ifndef, cpp:include, cpp:line, cpp:pragma, cpp:undef, cpp:value, cpp:derective, macro*
Refactoring	*name, typename, label*
Variable Type	*type*

5. CASE STUDY RESULTS

This section presents and discusses the results of our three research questions. For each question, we discuss the motivation, the approach designed to answer the questions, and the findings.

RQ1: What is the proportion of crash-inducing commits in Firefox?

Motivation. This question is preliminary to the other questions. It provides quantitative data on the prevalence of commits that induce subsequent crashes in Mozilla Firefox. The results of this question will help software managers realise the prevalence of the crash-inducing commits and adjust their bug triaging strategy to focus the resources to resolve defects causing the crashes as soon as possible.
Approach. We identify crash-inducing commits using the technique presented in Section 3, then calculate their percentage over the total number of studied commits.
Finding. Among the 127,212 analysed commits, 32,463 are identified to result in future crashes. Figure 4 illustrates the proportion of crash-inducing commits and other commits (referred to as crash-free commits in the rest of this paper).

> *Crash-inducing commits account for more than 25% of the total number of studied commits in Firefox.*

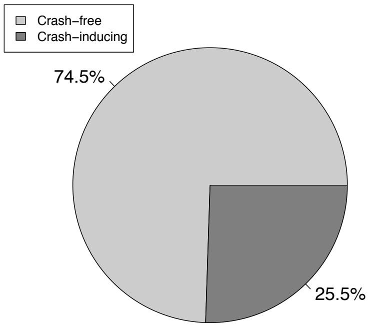


Figure 4: Proportion of crash-inducing commits and crash-free commits in Firefox

Table 2: Metrics used to compare the characteristics between crash-inducing commits and crash-free commits

Metric	Description and rationale
Committer's experience	Number of prior submitted commits.
Message size	Number of words in a commit message.
Changed files	Number of changed files (including added, deleted, and modified files) in a commit.
Added lines	Number of added lines of code in a commit.
Deleted lines	Number of deleted lines of code in a commit.
Number of changed types	Number of unique changed types in a commit.
Entropy of changed types	Measurement of the dispersion of different changed types in a commit (see Section 4.2.4).
Using Mozilla email	Whether a committer uses a Mozilla email address.
Is bug fix	Whether a commit is aimed to fix a bug.

One out of every four commits would cause subsequent crashes, which are considered as severe defects [37], because crashes can unexpectedly stop users' running process, lead to negative user experience and even decrease the reputation of a software organisation. Therefore, software practitioners should capture crash-inducing commits quickly, *i.e.*, when they are submitted into the version control system in order to address them as soon as possible. In the rest of this section, we will investigate the characteristics of crash-inducing commits and examine how to effectively predict them early.

RQ2: What characteristics do crash-inducing commits possess?

Motivation. Crash-inducing commits lead to bad user experience. If such a problem was not addressed promptly, developers would have to re-understand the code to locate the erroneous lines. Understanding the characteristics of crash-inducing commits can help software practitioners be aware of the factors that lead to crashes of a software system, and build predictive models to prevent them just-in-time.
Approach. For each of the commits identified either as crash-inducing commit or crash-free commit, we parse its commit log to extract the metrics presented in Table 2. We test the following 9 null hypotheses to statistically compare the characteristics between crash-inducing commits and crash-free commits.

Comparing the extents of changes in crash-inducing commits vs. crash-free commits.

H_{01}^{1}: *the number of words is the same for crash-inducing commits and for crash-free commits.*

H_{01}^{2}: *the number of changed files is the same for crash-inducing commits and for crash-free commits.*

H_{01}^{3}: *the number of added lines is the same for crash-inducing commits and crash-free commits.*

H_{01}^{4}: *the number of deleted lines is the same for crash-inducing commits and crash-free commits.*

Table 3: Median value of characteristic metrics for crash-inducing commits and crash-free commits, as well as the p-value of the Wilcoxon rank sum test

Metric	**Crash-inducing**	**Crash-free**	**p-value**
Committer's experience	190	246	<2.2e-16
Message size	12	11	<2.2e-16
Changed files	3	2	<2.2e-16
Added lines	9	5	<2.2e-16
Deleted lines	34	13	<2.2e-16
Number of changed types	3	2	<2.2e-16
Entropy of changed types	0.339	0.23	<2.2e-16
Using Mozilla email	41.8%	36.7%	–
Is bug fix	91.4%	83.5%	–

Comparing the changed types of crash-inducing commits vs. crash-free commits.

H^1_{02}: *the number of unique changed types is the same for crash-inducing commits and for crash-free commits.*

H^2_{02}: *the entropy value of changed types is the same for crash-inducing commits and for crash-free commits.*

Comparing the people and bug-related factors of crash-inducing commits vs. crash-free commits.

H^1_{03}: *committers' experience is the same for crash-inducing commits and for crash-free commits.*

H^2_{03}: *the percentage of Mozilla committers is the same for crash-inducing commits and for crash-free commits.*

H^3_{03}: *the percentage of bug fixing commits is the same for crash-inducing commits and for crash-free commits.*

We use the Wilcoxon rank sum test [14] to accept or reject the 7 first null hypotheses. This test is a non-parametric statistical test, which is used for measuring whether two independent distributions have equally large values. As for H^2_{03} and H^3_{03}, we simply compare the percentage values between crash-inducing commits and crash-free commits. We use a 95% confidence level (*i.e.*, p-value < 0.05) to decide whether to reject a null hypothesis. Since we will conduct 7 null hypothesis tests, to counteract the problem of multiple comparisons, we apply the Bonferroni correction [9] that consists in dividing the threshold p-value by the number of tests. Thus, our threshold to decide whether a result is statistically significant is p-value $< 0.05/7 = 0.007$.

Finding. Table 3 shows the median values of crash-inducing commits and crash-free commits on the metrics listed in Table 2, as well as the p-value of the Wilcoxon rank sum test. According to the results, crash-inducing commits are submitted by developers with less experience, suggesting that novice developers tend to write error-prone code. The message size of crash-inducing commits is significantly longer than crash-free commits. It is possible that crash-inducing commits are more complex and hence developers need longer comments to describe these changes. In crash-inducing commits, developers change significantly more files, and add and delete more lines than crash-free commits. This result is consistent with previous studies [21, 22] where researchers found that relative code churn measures can indicate defect modules. In terms of changed types, crash-inducing commits possess more unique changed types, and their changed types' entropy is higher than crash-free commits. In other words, the changed statements are distributed in more changed types in crash-inducing commits than in crash-free commits. This observation suggests that it is preferable to make semantically coherent changes (*i.e.*, changes of the same type) in commits. When developers modify the code with a lot of changed types (with the modifications equally distributed across the changed types), these modifications have a higher probability to induce subsequent crashes.

Another interesting finding is the fact that crash-inducing commits were mostly submitted by developers using Mozilla email accounts. This situation may be due to the fact that commits from outside contributors receive more scrutiny (through code review sessions) than those from Mozilla developers. Finally, most of our studied commits (either crash-inducing or crash-free) are bug fixing attempts. This finding confirms that bug fixing has become the major activity in software development [23]. A higher proportion of crash-inducing commits are aimed to fix bugs; meaning that modifying code to fix an existing bug is a risky task that can induce other bugs; confirming arguments from previous studies, such as [24], that legacy code becomes difficult to maintain.

In light of results from Table 3, we reject null hypotheses $H^1_{01} \sim H^4_{01}$, $H^1_{02} \sim H^2_{02}$, and H^1_{03}. In other words, for all metrics listed in Table 2, there exist statistically significant differences between crash-inducing commits and crash-free commits.

> *In general, crash-inducing commits are submitted by less experienced developers. They contain longer commit messages, more changed files and changed lines than crash-free commits. Crash-inducing commits contain more changed types, their changed statements tend to be scattered in different changed types. More crash-inducing commits are aimed to fix previous bugs. And more crash-inducing commits are submitted by developers using Mozilla email accounts (i.e., Mozilla developers).*

RQ3: How well can we predict crash-inducing commits?

Motivation. Crash-inducing commits may negatively impact users' experience, decrease the overall software quality and even the reputation of the software organisation. If we can predict these defective commits early on, we will not only increase the satisfaction of users, but also shorten the period between the introduction of these crash-related bugs in the system and their detection and correction. In fact, if the detection of a bug is done long time after its introduction in the system, developers are likely to have a hard time identifying the root cause of the bug since their knowledge of the code tends to decrease overtime. Hence, a delayed detection of bugs is likely to augment maintenance overhead. In our previous work [2], we extracted metrics from bug reports to predict highly impactful crash-related bugs. Although this approach can shorten bug triaging time to some extent, developers still have to wait for a certain period, during which crashes are collected, triaged and filed into bug reports, before they can carry out their bug fixing activities. During this period, end users (possibly in large numbers) may have suffered unexpected aborts of the software. A just-in-time detection of crash-inducing commits will enable developers to act immediately on crash-prone commits before they can negatively impact users.

Approach. We extract 24 metrics along 4 dimensions from respectively the studied commit logs and the corresponding source code of Firefox. Table 4 to Table 7 show our selected metrics (*i.e.*, independent variables for the prediction models) and their rationales.

Table 4: Commit log metrics

Attribute	**Explanation and Rationale**
Hour	Hour (0-24). Code committed at certain hours may lead to crashes (e.g., hours around quitting time).
Week day	Day of week (from Mon to Sun). Code committed on certain week days may be less carefully written (*e.g.*, Friday) [32, 4], and would lead to crashes.
Month day	Day in month (1-31). Code committed on certain days may be less carefully written (*e.g.*, before and during public holidays); resulting into subsequent crashes.
Month	Month of year (1-12). Code committed in some seasons may be less carefully written; resulting into crashes. (*e.g.*, December, during Chrismas holidays).
Day of year*	Day of year (1-366). Combined the rationales of month day and month.
Message Size	Number of words in a commit message. In RQ2, we found that crash-inducing commits are correlated with longer commit messages.
Experience	Number of prior submitted commits. In RQ2, we found that crash-inducing commits tend to be submitted by less experienced developers.
From Mozilla	Whether a committer uses a Mozilla email address. In RQ2, we found that crash-inducing commits are often submitted by Mozilla's developers.
Number of changed files	Number of changed files in a commit. In RQ2, we found that commits with more changed files tend to cause subsequent crashes.
Is bug fix	Whether a commit aimed to fix a bug. In RQ2, we found that crash-inducing commits are correlated with bug fixing code.
Is supplementary fix	Whether a commit is to fix a prior fixed bug. Supplementary fixes may enhance previous fixes and may be less likely to cause crashes.
Before crashed files	Percentage of a commit's files that caused crashes in prior commits. Crashed code may be difficult to fix, and still lead to future crashes.

Table 5: Code complexity metrics

Attribute	**Explanation and Rationale**
LOC	Median lines of code in all classes in a commit. In RQ2, we found that crash-inducing commits have higher code churn (*i.e.*, added/deleted lines).
Number of functions	Median number of classes' functions in a commit. A huge class may be difficult to understand or modify, and lead to crashes.
Cyclomatic complexity	Median cyclomatic complexity of the functions in all classes in a commit. Complex code is hard to maintain and may cause crashes.
Max nesting*	Median maximum level of nested functions in all classes in a commit. A high level of nesting increases the conditional complexity and may increase the crashing probability.
Comment ratio	Median ratio of the lines of comments over the total lines of code in all classes in a commit. Codes with lower ratio of comments may not be easy to understand, and may result in crashes.

To predict whether or not a commit will cause subsequent crashes, we apply multiple regression and machine learning algorithms: General Linear Model (GLM), Naive Bayes, decision tree, and Random Forest. GLM is an extension of multiple linear regression for a single dependent variable. It is extensively used in regression analyses. Naive Bayes are a set of logistic regression algorithms based on applying Bayes' theorem with strong independence assumptions between the features. Although independence is normally a poor assumption, in practice, this algorithm often performs well [26]. In a previous bug prediction study, Shihab et al. [31] used the C4.5 decision tree algorithm to predict re-opened bugs and obtained good prediction results. In this research, we use C5.0 model, the improved version of C4.5, which can obtain a higher accuracy, perform faster, and have less memory usage than C4.5 [7]. Developed by Leo Breiman and Adele Cutler, Random Forest [6] uses a majority voting of decision trees to generate classification (predicting, often binary, class labels) or regression (predicting numerical values) results. This algorithm yields an ensemble that can achieve both low bias and low variance [8]. In this study, we build 100 trees, each of which are with 5 randomly selected metrics.

To deal with collinearity in the data, before building the predictive models, we apply the Variance Inflation Factor (VIF) analysis to eliminate correlated metrics. As recommended in [27], we set the threshold to 5, *i.e.*, metrics with VIF values over this threshold are considered as correlated and will be removed from the predictive models. In Table 4 to Table 7, removed metrics are marked with *.

We use ten-fold cross validation [10] to compute the accuracy, precision, recall, and F-measure for crash-inducing commits and crash-free commits. In the cross validation, we randomly split the subject commits into ten disjoint sets. Nine sets are used as training data and the remaining set as testing data. We repeat the process for ten times and report median results for accuracy, precision, recall and F-measure. Because crash-inducing commits and crash-free commits are imbalanced in our data set, we under-sample the majority class instances, *i.e.*, we randomly deleted instances from the data set of crash-free commits to make the data sets of crash-inducing commits and crash-free commits to have the same number of instances. We do this under-sampling only during the training phase. We rank the importance of the independent variables (prediction metrics) to identify the top predictors for the algorithm with the best prediction results.

Finding. Table 8 shows the median accuracy, precision, recall, and F-measure for the four algorithms used to predict whether a commit will cause crashes in Firefox. According to the results, our models can predict crash-inducing commits with a precision up to 61.4% and a recall up to 95.0%. Random Forest is the best prediction algorithm, which obtains the best F-measure when predicting either crash-inducing commits or crash-free commits. Among the 22 selected metrics, the SNA metric *closeness* is ranked as the most important predictor in all the 10 phases of the cross validation. This metric evaluates the degree of centrality of a class in the whole project. Our obtained result suggests that when many other classes depend on a class, a change to this (central) class is likely to induce crashes. Moreover, *message size*, *number of changed files*, *outdegree*, and *percentage of before crashed files* are ranked as the second important predictors; meaning that the length of comments in a commit, the number of changed files, the number of callees of classes modified by a commit, and the crashing history of files mod-

Table 6: Social network analysis metrics (other metrics in this dimension share the same rationale with PageRank. We compute median value of each metric for all classes in a commit.)

Attribute	Explanation and Rationale
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. An inappropriate change to the class may lead to malfunctions in the dependent classes; resulting into crashes.
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.

Table 7: Changed type metrics

Attribute	Explanation and Rationale
Number of changed types	Number of unique changed types in a commit. In RQ2, we found that crash-inducing commits tend to contain more changed types.
Entropy of changed types	Distribution of changed types in a commit (see Section 4.2.4). In RQ2, we found that crash-inducing commits tend to have higher entropy of changed types.

ified in a commit are good indicators of the risk of crashes related to the integration of a commit in the code repository.

> *Our predictive models can achieve a precision of 61.4%, and a recall of 95.0%. The Random Forest algorithm achieves the best prediction performance. Closeness is ranked as the best predictor in this algorithm. Software organisations can make use of the proposed predictive models to track crash-prone commits as soon as they are submitted for integration in the code repository, for example, during code review sessions.*

6. THREATS TO VALIDITY

In this section, we discuss the threats to validity of our study following the guidelines for case study research [38].

Construct validity threats concern the relation between theory and observation. In this research, the construct validity threats are mainly due to measurement errors. We used the source code of the previous release to a commit to compute complexity and SNA metrics. More specifically, for a given file F in a commit C, we found the previous release R of C, and computed the code complexity and SNA metrics of F in the context of the release R. Although the new commit C could slightly affect the values of these metrics, we observed that in most cases there is no noticeable change. Also, computing the metrics every time a new commit is submitted would delay the detection of the crash-inducing commits (since the computation of the metrics takes some time). In this paper, as a compromise, we use the files in the previous release to estimate a current commit's code complexity and SNA metrics. In the future, we will experiment with parallel algorithms to compute these metrics in real time.

Table 8: Accuracy, precision, recall, and F-measure (in %) obtained from GLM, Naive Bayes, C5.0, and Random Forest to predict crash-inducing commits and crash-free commits

Metric	GLM	Bayes	C5.0	Random Forest
Accuracy	67.5	41.7	69.9	73.3
Crash-inducing precision	59.5	38.6	57.2	61.4
Crash-inducing recall	37.3	95.0	76.6	76.5
Crash-inducing F-measure	45.8	54.7	65.4	68.1
Crash-free precision	69.8	77.8	82.6	83.8
Crash-free recall	84.8	10.0	66.4	71.4
Crash-free F-measure	76.7	17.7	73.5	77.4

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. In Section 3.2.2, although we removed all candidates of crash-inducing commits that only changed comments and–or white space lines, our "crash-inducing commits" may still contain some false positives. Concretely, in a fix of a crash-related bug, not all of the changes are aimed to address defects. Some lines may be added because of a refactoring or an addition of a new feature. These changes are hard to identify with an automatic approach. In our future work, we plan to manually examine a sample of the identified crash-inducing commits, and report its precision and recall.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate the assumptions of the constructed statistical models. In RQ2, we used non-parametric tests which do not require making assumptions about the distribution of the data set. When mapping crash-related bugs to their bug fixes, we manually checked false positives from the results. In addition, we manually grouped different srcML tags into changed types as shown in Table 1.

External validity threats concern the possibility to generalise our results. In this paper, we analysed only Mozilla Firefox. Although many software organisations are using crash collecting systems, to the best of our knowledge, only the Mozilla corporation has opened its crash reports to the public [35]. In our previous work [2], we used another Mozilla project, Fennec for Android, as a subject system to study crash-related bugs. However, the code of Firefox and Fennec are both managed by a Mercurial release branch, in which, the two sub-systems share some common components; making it hard to separate the two systems at the level of commits. We look forward to generalise our proposed approach to more software systems. We share our data and scripts at https://github.com/swatlab/crash-inducing. Researchers and software practitioners can use these data and scripts to validate our results and replicate our technique to other systems.

7. RELATED WORK

In this section, we introduce some related studies on crash analysis, traditional defect prediction techniques, and Just-In-Time defect prediction techniques.

7.1 Crash Analysis

Crashes stop a software system unexpectedly, causing data loss and frustration to users. Today, many software organisations have deployed automatic crash collecting systems to gather and triage crash occurrences. Researchers intend to study the crash reports from these systems to facilitate the debugging and bug fixing process for software practitioners. Podgurski et al. [25] proposed an automated failure clustering approach for the classification of crash reports to facilitate their prioritisation and the diagnostic of their root causes. Khomh et al. [17] mined crash reports in Mozilla Firefox, and proposed an entropy-based approach that can be used to identify crash-types with high impact, *i.e.*, crash-types that occur frequently and impact a large number of users. Based on the approach proposed by Khomh et al., Wang et al. [35] studied crash information in Firefox and Eclipse, and proposed an algorithm that can locate and rank defective files, as well as a method that can identify duplicate and related bug reports. Kim et al. [18] analysed crash reports and the related source code in Firefox and Thunderbird to predict top crashes before a new release of a software system.

7.2 Traditional Defect Prediction Techniques

Traditional defect prediction techniques used coarse-grained metrics, such as bug report metrics, to identify defect-prone modules or specific types of bugs. By using social factors, technical factors, coordination factors, and prior-certifications factors, Hassan et al. [13] created decision trees to predict ahead of time the certification result of a build for a large software project at IBM Toronto Lab. Shihab et al. [31] extracted metrics from bug reports and built models using C4.5, Zero-R, Naive Bayes and Logistic Regression algorithms, to predict bug re-opening in three open-source projects. In their study, the decision tree model, C4.5, yielded the best prediction results. As a complementary work, Zimmermann et al. [39] used Logistic Regression models to predict bug re-opening in Windows. In our previous work [2], we used GLM, C5.0 (the improved version of C4.5), ctree, randomForest, and cforest to predict crash-related bugs with high crashing frequency and which impact a large population of users.

7.3 Just-In-Time Defect Prediction Techniques

Though traditional defect prediction techniques can help software organisations prevent defects to some extent, developers can only identify the error-prone modules responsible for these defects after the defects have been filed into bug reports. During the period between the integration of the defective code into the version control system and the opening of the bug report, a defective commit could negatively impact a large user base. Just-In-Time defect prediction techniques are designed to predict defects in commits, in order to allow developers to track and fix defects as soon as they are submitted for integration in version control systems. Kamei et al. [16] used a wide range of source code metrics to predict defect-prone commits in six open-source systems and five commercial systems. Fukushima et al. [12] applied Just-In-Time defect prediction techniques to cross-project defect predictions and found them viable for projects with little historical data. Using a number of code and process factors extracted at change level, Misirli et al. [20] built statistical models to predict high impact fix-inducing changes. In this paper, we use change level metrics to predict crash-inducing commits.

8. CONCLUSION

Crashes, which are unexpected interruptions of a software system, are one of the major source of frustration for users. Frequent crashes of a software system can significantly decrease user-perceived quality and even affect the overall reputation of a software organisation. To help software practitioners identify crash-prone code early on, we conduct a study of crash-inducing commits in Mozilla Firefox. We found that crash-inducing commits account for more than 25% of all studied commits. We also found that, compared to other commits, crash-inducing commits are often submitted by developers with less experience and contain longer comments, more changed files and changed lines, as well as more changed types.

To help software practitioners track and fix crash-inducing commits as soon as possible, we built predictive models using various regression and machine learning algorithms. These predictive models achieved a precision up to 61.4% and a recall up to 95.0%.

Software organisations can use our proposed predictive models to detect crash-prone code as soon as they are submitted for integration in the source code repository. They could then correct the code quickly to avoid users from experiencing the crashes. In the future, we plan to generalise our approach to other software systems and implement it into tools for different programming languages.

9. ACKNOWLEDGEMENT

This work is supported in part by grants from Hydro-Québec and NSERC (Natural Sciences and Engineering Research Council of Canada).

10. REFERENCES

[1] L. An and F. Khomh. Challenges and issues of mining crash reports. In *Software Analytics (SWAN), 2015 IEEE 1st International Workshop on*, pages 5–8. IEEE, 2015.

[2] L. An and F. Khomh. An empirical study of highly-impactful bugs in Mozilla projects. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015.

[3] L. An, F. Khomh, and B. Adams. Supplementary bug fixes vs. re-opened bugs. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 205–214. IEEE, 2014.

[4] P. Anbalagan and M. Vouk. Days of the week effect in predicting the time taken to fix defects. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 29–30. ACM, 2009.

[5] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[6] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[7] C5.0 algorithm. http://www.rulequest.com/see5-comparison.html, 2015. Online; accessed June 13th, 2015.

[8] R. Díaz-Uriarte and S. A. De Andres. Gene selection and classification of microarray data using random forest. *BMC bioinformatics*, 7(1):3, 2006.

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

[10] B. Efron. Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983.

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

[12] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181. ACM, 2014.

[13] A. E. Hassan and K. Zhang. Using decision trees to predict the certification result of a build. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 189–198. IEEE, 2006.

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

[15] igraph. http://igraph.org/redirect.html, 2015. Online; accessed June 13th, 2015.

[16] 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. *Software Engineering, IEEE Transactions on*, 39(6):757–773, 2013.

[17] 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 *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 261–270. IEEE, 2011.

[18] 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. *Software Engineering, IEEE Transactions on*, 37(3):430–447, 2011.

[19] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[20] A. T. Misirli, E. Shihab, and Y. Kamei. Studying high impact fix-inducing changes. *Empirical Software Engineering*, pages 1–37, 2015.

[21] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 181–190. IEEE, 2008.

[22] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[23] N. I. of Standards & Technology. The economic impacts of inadequate infrastructure for software testing, May 2002. US Dept of Commerce.

[24] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

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

[26] I. Rish. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46. IBM New York, 2001.

[27] P. A. Rogerson. *Statistical methods for geography: a student's guide*. Sage Publications, 2010.

[28] B. A. Romo, A. Capiluppi, and T. Hall. Filling the gaps of development logs and bug issue data. In *Proceedings of The International Symposium on Open Collaboration*, page 8. ACM, 2014.

[29] Understand tool. https://scitools.com, 2015. Online; accessed June 13th, 2015.

[30] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5:3–55, January 2001.

[31] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, 18(5):1005–1042, 2013.

[32] 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.

[33] Socorro: Mozilla's crash reporting system. https://crash-stats.mozilla.com/home/products/Firefox, 2015. Online; accessed June 13th, 2015.

[34] srcML. http://www.srcml.org, 2015. Online; accessed June 13th, 2015.

[35] S. Wang, F. Khomh, and Y. Zou. Improving bug management using correlations in crash reports. *Empirical Software Engineering*, pages 1–31, 2014.

[36] C. Williams and J. Spacco. SZZ revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.

[37] R. Wu. Diagnose crashing faults on production software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 771–774. ACM, 2014.

[38] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, 3rd edition, 2002.

[39] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1074–1083. IEEE, 2012.