

# An Empirical Study of Crash-inducing Commits in Mozilla Firefox

Le An · Foutse Khomh · Yann-Gaël Guéhéneuc

Received: date / Accepted: date

**Abstract** Software crashes are dreaded 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 software has been crashing for a certain period of time. To help software organisations detect and fix crash-prone code early, we examine the characteristics of commits that lead to crashes, called *crash-inducing commits*, in Mozilla Firefox. We find 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, but they make less effort to fix bugs caused by these commits. We also characterise commits that would lead to frequent crashes, which impact a large population of users (dubbed *highly-impactful crash-inducing commits*). Compare to other crash-related bugs, bugs due to highly-impactful crash-inducing commits were less reopened, and tend to be fixed by a single commit. We built predictive models to help software practitioners detect and fix crash-prone bugs early on. Our predictive models achieve a precision of 61.2% and a recall of 94.5% to predict crash-inducing commits and can achieve a precision of 60.9% and a recall of 91.1% to predict highly-impactful crash-inducing commits. Software organisations can use

---

Le An  
SWAT Lab, DGIGL, Polytechnique Montréal, Montréal, QC, Canada  
E-mail: le.an@polymtl.ca

Foutse Khomh  
SWAT Lab, DGIGL, Polytechnique Montréal, Montréal, QC, Canada  
E-mail: foutse.khomh@polymtl.ca

Yann-Gaël Guéhéneuc  
PTIDEJ Team, DGIGL, Polytechnique Montréal, Montréal, QC, Canada  
E-mail: yann-gael.gueheneuc@polymtl.ca

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.

**Keywords** Crash analysis · Bug triaging · Prediction model · Mining software repositories

## 1 Introduction

Software crashes refer to unexpected interruptions of software systems in users' environments. 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. When and if the system crashes, the automatic crash reporting tool collects information on the crash event and 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 as faults into bug tracking systems (*e.g.*, Bugzilla or Jira) to enable quality-assurance teams to focus their limited resources on fixing these important faults.

Khomh et al. [?] 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 identify crashes with high impact after the crash collecting system has gathered enough crashes. Until enough crashes are received, the crashes may have affected a large number of users. Moreover, while time passes, the faulty code becomes unfamiliar to developers, making it harder to correct.

To reduce the triaging period of crash-related bugs, in our previous study [?], we built statistical models to predict crash-related bugs that lead to frequent crashes and which impact a large user base. Although these models can be applied at an early stage of development to detect crash-related bugs with a serious negative impact on users, software organisations still must wait for a period of time during which crashes are collected, triaged, and filed into bug reports, before they can be fixed.

We argue that, if software organisations could detect crash-prone code even earlier, at the time of commits, *i.e.*, before the software is built and released, they could address the faults quicker and prevent the unpleasant experience of crashes to their users. Such an approach is referred to as “Just-In-Time Quality Assurance” [?] and it enables fine-grained fault predictions and allows quality-assurance teams to identify error-prone code at commit time. By identifying error-prone commits sooner, quality-assurance teams are also likely to make better decisions in choosing developers that can fix these bugs.

In this paper, we investigate statistical models to predict commits that may introduce crashes in Mozilla Firefox. 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 [?] to map these bugs to their related commits to identify the commits that introduced the fault responsible for the crash. We found that crash-inducing commits account for 25.5% of all commits in the studied version control system and that 37.1% of the commits that change C/C++ code would lead to crashes.

*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 than the average. Also, they are more often committed by developers from Mozilla than from outside. Developers change more files and add and delete more lines in crash-inducing commits. Compared to other commits, crash-inducing commits fix more previous bugs but, often, they lead to other bugs. 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. In addition, we observed that the bugs caused by crash-inducing commits require less supplementary fixes than other bugs and they are reopened less often. Also, 43.7% of crash-related bugs are without any resolution; implying that developers do not specifically target these (severe) bugs during bug fixing activities, which is a bit surprising.

We also investigated commits that lead to frequent crashes impacting a large user base, referred to as *highly-impactful crash-inducing commits* and, compared to other crash-related bugs, the fixes of highly-impactful bugs require less reworking (*i.e.*, supplementary fixes) than other bugs: developers seem to be very careful when fixing these bugs.

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

Previous studies, which used statistical models to predict faults from bug reports, are effective to some extent. However, before a certain type of crash is filed into the crash collecting system, a large number of users might have already suffered it. Moreover, during this period, developers may become less familiar with the code and thus may have to spend more time identifying the faulty lines to fix the faults. Therefore, statistical models that can predict fault-prone code just-in-time have the potential to help developers detect crash-inducing commits as soon

as they are introduced 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 61.2% and a recall of 94.5%.

*RQ4: How well can we predict commits that lead to frequent crashes that impact a large user base?*

Crash-related bugs have different impact on end-users. Mozilla prioritises these bugs by their crash frequency. Though frequency is an important metric, it does not capture the full picture of the severity of a crash-related bug. Khomh et al. [?] have proposed a combination of frequency and entropy measurements to capture the severity of crash-related bugs. We leverage their proposed entropy-based classification [?] and apply the best statistical algorithm from **RQ3** (*i.e.*, Random Forest) to predict commits that can lead to bugs with high crashing frequency and impacting a large user base, *i.e.*, *highly-impactful crash-inducing commits*. Despite the low percentage of highly-impactful crash-inducing commits (23.7% in commits that change C/C++ code) in the studied data-set, our model can still achieve a precision of 60.9% and a recall of 91.1%. Software organisations could apply our model to improve their fault triaging process and their users' satisfaction.

*RQ5: What are the characteristics of commits that are misclassified by our prediction models?*

Sometimes, our models misclassify some *clean* commits as crash-inducing commits (false positives) and some crash-inducing commits as *clean* commits (false negatives). We studied these misclassified faults and we observed that our models tends to classify commits with less developers' experience, higher numbers of changed files and lines of code as "crash-inducing commits". In addition, we observed that false positive commits contain more complex code and more changed types. These commits changed more lines of code and files, and were often submitted by less experienced developers which is why they are misclassified by our models.

Moreover, we observed that 9.2% of commits do not lead to crashes but still caused a bug. Hence, developers still must carefully check the code contained in these commits before integrating it into the version control system.

Finally, we observed that developers performed a high proportion of renaming operations on the code of these commits. Inappropriate or incomplete renaming operations can lead to missing runtime variable and-or mismatching errors, which can crash a software system. Our model could not classify correctly these commits because none of our metrics capture fine-grained source code changes between commits. In the future, we plan to include some fine-grained metrics of source code changes into our model to improve their recall. Hence, it seems that

renaming operations are risky and that developers should be careful when performing renaming operations in the code.

We are limited to Firefox because, at the time of writing, no other software organisation provides access to its crash reporting system. Software organisations could apply our proposed approach internally to detect crash-prone code early and address the faulty code as soon as possible, before such affects a large number of users.

This paper is an extension of an earlier conference paper [?]. Our original work:

1. calculated the percentage of crash-inducing commits in Mozilla Firefox.
2. compared crash-inducing commits against other commits in various aspects.
3. predicted crash-inducing commits using statistical models, and identified the most important predictors.

which we extend as follows:

1. We have readjusted the classification of changed types and rebuilt our predictive models.
2. We have examined whether bugs caused by crash-inducing commits require supplementary bug fixes more often than other bugs and whether they are re-opened more frequently than other bugs.
3. We have proposed models to predict commits that lead to frequent crashes that impact a large user base.
4. We have also examined the reason behind the false positives and false negatives of our prediction models.

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 our five research questions. Section 6 discusses threats to the validity of our results. Section 7 summaries related work. Section 8 draws conclusions and suggests future work.

## 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 [?]. 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 identify and fix faults in the system. They can also use information from crash

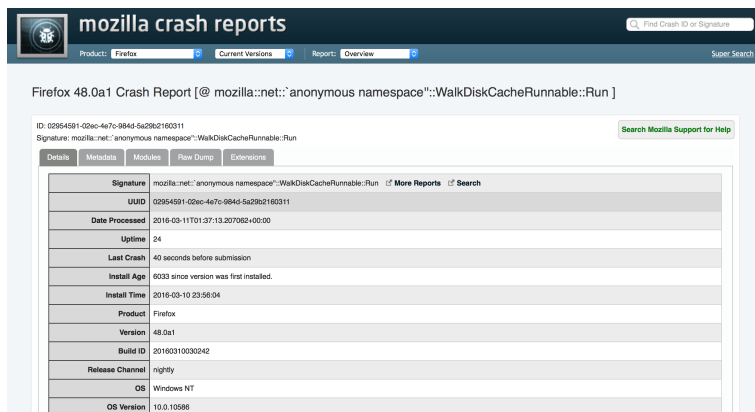


Fig. 1: A sample crash report from Firefox

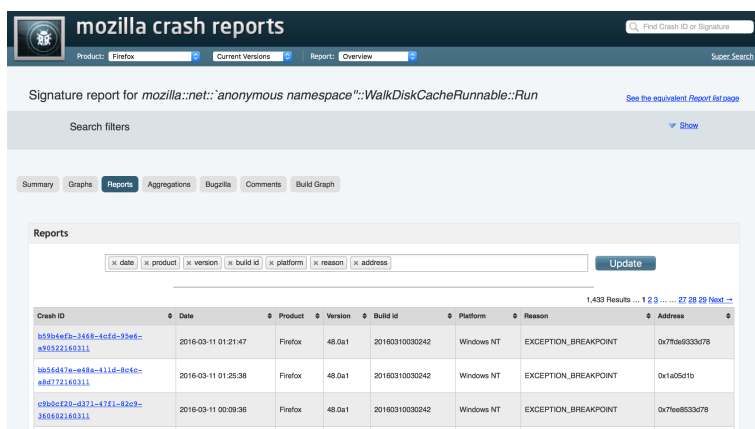


Fig. 2: A sample crash-type from Firefox

reports to allocate development resources. Figure 1 presents 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. Developers and quality assurance teams 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 also be linked to the same

crash-type [? ]. 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 [? ]. 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 our data and analytic scripts are available at: <https://github.com/swatlab/crash-inducing>.

Applying the SZZ algorithm [? ], 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 fault 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. [? ] introduced the SZZ algorithm, a plethora of studies (such as [? ? ? ]) 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. [? ] 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. Some commits that fixed a previous bug fix (called supplementary bug fixes [? ]), often lack information about the fixed bug in their message, *i.e.*, only a commit ID (*i.e.*, a SHA1 string) of a previous fix is provided. In this case, we track the commit IDs back to their original commits and check whether these original commits could be mapped to a bug report. Hence, 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 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 contained in 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 also find out the first crash date  $D_{first}$  of the bug  $B_{crash}$  and the last crash date  $D_{last}$  before the opening of the bug. We decide crash-inducing commits using the following rules:

- *Rule 1:* If  $D_{candidate}$  is earlier than  $D_{first}$ , this candidate commit is identified as a “crash-inducing commit”.
- *Rule 2:* If  $D_{candidate}$  is later than  $D_{first}$  but earlier than the last crash date  $D_{last}$ , we consider this candidate commit as a “crash-inducing commit” if it changed any of the files appearing in the crashed stack trace of  $B_{crash}$ .

In the original SZZ algorithm [? ], Śliwerski et al. filtered bug-inducing commits by bug opening date, which however cannot be directly applied to filter crash-inducing commits. We use *Rule 1* to select commit candidates submitted prior to the first crash occurrence. But this rule may omit some crash-inducing commits. Because a crash-related bug may derive from different crash-types. A crash-type contains crashes that have the same top method signatures (in their stack traces). However, their subsequent method signatures could be different. So, crashes with different stack traces which were induced by different commits can be filed into the same bug report. Figure 3 illustrates an example:  $Commit_1$ , which submission date is  $D_1$ , induced  $Crash_1$ ;  $Commit_2$ ,



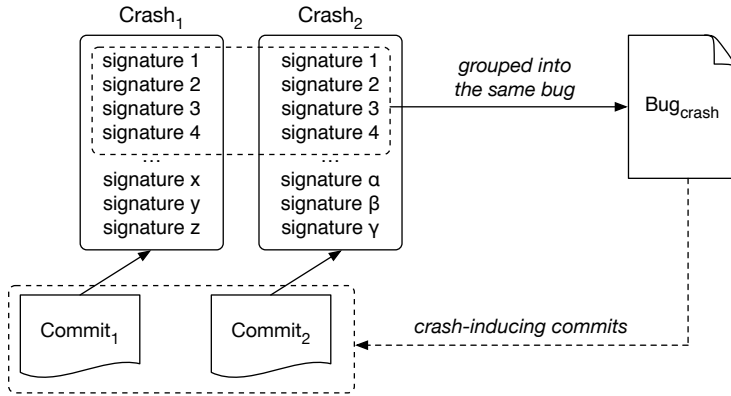


Fig. 3: Different crashes can be classified into the same bug report

which submission date is  $D_2$ , induced  $Crash_2$ , where  $D_2$  is later than  $D_1$ .  $Crash_1$  and  $Crash_2$  have common top method signatures, but have different method signatures in the rest of their crashing stack traces. If both crashes were filed into the bug  $B_{crash}$  and we apply only *Rule 1* on this bug,  $Commit_2$  would be omitted. Therefore, we also apply *Rule 2* to discover all commits that introduced crashes related to  $B_{crash}$ .

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 replication studies.

## 4 Case Study Design

This section describes the data collection and processing for our case study, which aims to answer the following five 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. How well can we predict commits that lead to frequent crashes that impact a large user base?
5. What are the characteristics of commits that are misclassified by our prediction models?

### 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

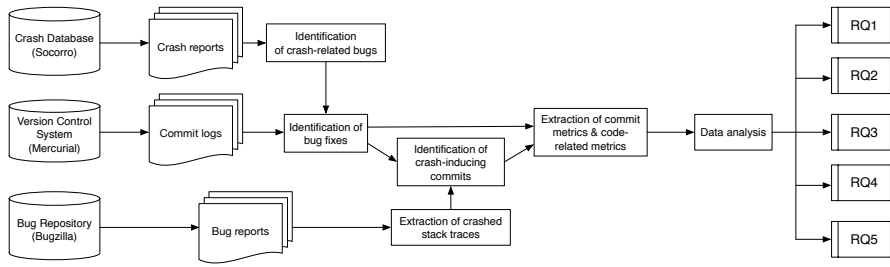


Fig. 4: Overview of our approach to identify crash-inducing commits and extract their characteristic metrics

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 4 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 [? ], we apply the source code analysis tool *Understand* [? ] to compute the code-related metrics of the analysed files and identify the relationship among these files. Developers can either use *Understand*'s graphical interface or its command line tool<sup>1</sup> to generate an *Understand* database (UDB), from which we program against the *Understand* Python API<sup>2</sup> to extract five metrics on code complexity for

<sup>1</sup> <https://scitools.com/feature/automation-using-the-command-line>

<sup>2</sup> <https://scitools.com/new-python-api/>

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++ [?], 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 dependencies among different files in Firefox to 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* [?] 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 commit 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. We assume that changes on comments 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* [?] 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<sup>3</sup>. As we detected more than 80 unique srcML tags from the studied changed

<sup>3</sup> 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)
Access modifier	<i>super, public, private, protected, extern</i>
C++ template	<i>template, typename</i>
Class	<i>class, class_decl, member_list, constructor, constructor_decl, destructor, destructor_decl</i>
Code block	<i>block, expr, expr_stmt</i>
Comment	<i>comment</i>
Control flow	<i>while, do, if, else, break, goto, label, for, foreach, continue, then, switch, case, return, condition, incr, default</i>
Data structure	<i>enum, struct, struct_decl, typedef, union, union_decl</i>
Declaration	<i>asm, decl, decl_stmt, using, namespace, range, specifier</i>
Function	<i>function, function_decl</i>
Initialisation	<i>init</i>
Invocation	<i>call</i>
Operator	<i>escape, index, sizeof</i>
Parameter	<i>param, parameter_list, argument, argument_list</i>
Preprocessor	<i>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:derecive, macro</i>
Renaming	<i>renaming, name</i>
Variable type	<i>type</i>

files, we group the srcML tags with similar semantic functions into a *changed type*, while ignoring trivial srcML tags, such as “@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 [? ], defined as:

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

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). On the contrary, if a commit has only one changed type, the entropy is minimal (*i.e.*, 0).

#### 4.2.5 Identifying Bugs requiring Supplementary Fixes and Reopened Bugs

In our previous research [? ], we studied two kind of bugs that need additional effort to get fixed than other bugs:

- Bugs requiring Supplementary Fixes: bugs are fixed by not only one commit, but by multiple commits.
- Reopened Bugs: bugs that have been reopened.

We used the approach described in [?] to identify these bugs. Concretely, we apply regular expressions to parse Mozilla commit messages, if a bug ID is mentioned in the messages of more than one commit, we consider it as a bug that requires supplementary fixes. Next, we parse Mozilla bug reports, if we find a “REOPENED” tag in a bug’s history, we consider it as a reopened bug.

## 5 Case Study Results

This section presents and discusses the results of our five research questions. For each question, we discuss the motivation, the approach designed to answer the question, 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 their limited resources to resolve faults 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 as crash-inducing commits. Figure 5 illustrates the proportion of crash-inducing commits and other commits (referred to as *crash-free commits* in the rest of this paper). If we consider commits that changed at least one C/C++ file, crash-inducing commits account for 37.1% of all the commits (with changes on C/C++ code).

One out of every four commits would cause subsequent crashes, which are considered to be severe faults [?], because crashes can unexpectedly stop users’ running processes, leading 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 fix 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 on.

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

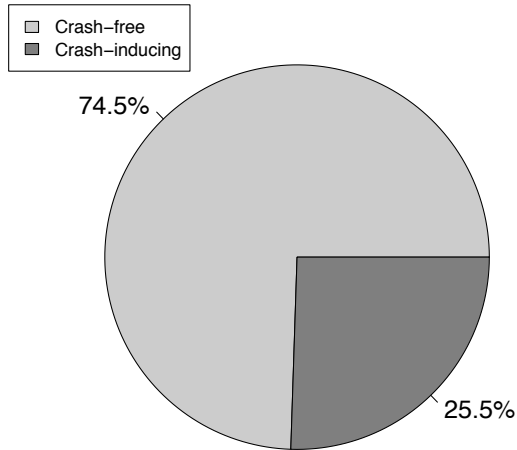


Fig. 5: Proportion of crash-inducing commits and crash-free commits in Firefox

RQ2: What characteristics do crash-inducing commits possess?

**Motivation.** Crash-inducing commits can lead to a dreadful user experience. Moreover, if a crash-related bug is not fixed promptly and properly, and re-appear later on, developers may have a hard time finding the source of the bug since they would have to re-understand the context of some past code changes. Understanding the characteristics of crash-inducing commits can help software practitioners be aware of factors that lead to crashes of a software system, and build predictive models to identify crash-prone code just-in-time.

In addition, different crashes can affect end-users to different extent. Mozilla uses crashing frequency to prioritise their crash-related bugs. Khomh et al. [?] proposed an entropy-based approach to classify crash-types along two dimensions: crashing frequency and entropy, where the latter represents the distribution of a crash-type in the user base. In our previous work, we applied this idea to classify crash-related bugs into four categories, as shown in Figure 6. We refer to bugs that crash frequently and affect a large number of users as *highly-impactful bugs*. In this research question, we will also compare the characteristics of commits that lead to highly-impactful bugs (refer to as *highly-impactful crash-inducing commits*) against other commits.

**Approach.** For each of the commits identified either as crash-inducing commit or crash-free commit, we parse the commit log to extract the metrics presented in Table 2. We test the following 8 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 crash-free commits.

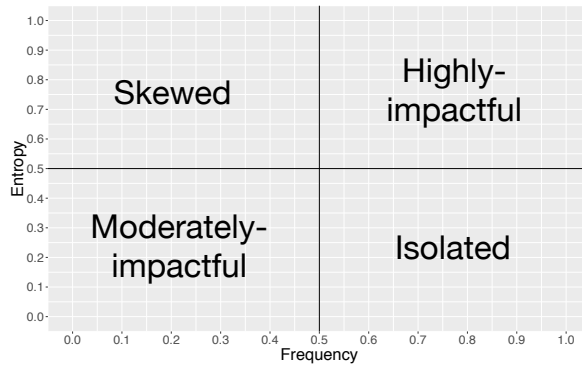


Fig. 6: Categories of bugs based on crashing frequency and entropy

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

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.
Entropy of changes	Measurement of the dispersion of changed code among files 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).

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

$H_{01}^5$ : the entropy of changes is the same for crash-inducing commits and crash-free commits.

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

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

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

**Comparing the people factor of crash-inducing commits vs. crash-free commits.**

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

We use the Wilcoxon rank sum test [?] to accept or reject the 8 null hypotheses. This test is a non-parametric statistical test, which is used for measuring whether two independent distributions have equally large values. We use a 95% confidence level (*i.e.*,  $p$ -value  $< 0.05$ ) to decide whether to reject a null hypothesis. Since we will conduct 8 null hypothesis tests, to counteract the problem of multiple comparisons, we apply the Bonferroni correction [?] which 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  $< 0.05/8 = 0.006$ .

We will also compare crash-inducing commits and other commits in terms of the following aspects:

1. Percentage of Mozilla committers
2. Percentage of bug fixing commits

In addition, different bugs require different effort to get fixed. We use the approach described in Section 4.2.5 to investigate whether bugs caused by crash-inducing commits and bugs caused by other commits required the same effort from developers. More specifically, we will investigate the following aspects:

1. Percentage of bugs that require supplementary fixes (*i.e.*, bugs that were fixes by more than one commit) [?].
2. Percentage of reopened bugs (*i.e.*, bugs that have been reopened).

To identify highly-impactful crash-inducing commits, we applied the approach described in [?], to compute the crashing entropy value (from 0 to 1) of each crash-related bug. A high entropy value means a high distribution of the crash-type in the user base (*i.e.*, the bug impacts a large population of users), and vice versa.

Based on our previous study [?], we use the median value of frequency and entropy to decide whether a crash-related bug has high crashing frequency and entropy values, as illustrated in Figure 6. Then, we classify all crash-related bugs into the following categories, which are sorted by their priority in descending order:

- **Highly-impactful Bugs:** bugs with frequency and entropy values above or equal to the median. These bugs impact a large number of users.
- **Skewed Bugs:** bugs with a high frequency value (*i.e.*, above or equal to the median) but a low entropy (*i.e.*, below the median). These bugs only seriously affect a small proportion of users and are more likely to be specific to the users' systems.
- **Moderately-impactful Bugs:** bugs that are widely distributed in the user base (*i.e.*, entropy value above or equal to the median) but do not occur very often (*i.e.*, frequency value below the median).
- **Isolated Bugs:** bugs with frequency and entropy values below the median. These bugs rarely occur and affect a small number of users.



Table 3: Median value of hypothesis testing 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	$< 2.2e-16$
Entropy of changes	0.58	0	$< 2.2e-16$
Number of changed types	4	3	$< 2.2e-16$
Entropy of changed types	0.43	0.35	$< 2.2e-16$

Table 4: Median value of hypothesis testing metrics for highly-impactful crash-inducing commits (HICI) and other commits, as well as the  $p$ -value of the Wilcoxon rank sum test

Metric	HICI	Other	$p$ -value
Committer's experience	177	243	$< 2.2e-16$
Message size	12	11	$< 2.2e-16$
Changed files	3	2	$< 2.2e-16$
Added lines	11	5	$< 2.2e-16$
Deleted lines	39	14	$< 2.2e-16$
Entropy of changes	0.63	0.21	$< 2.2e-16$
Number of changed types	5	3	$< 2.2e-16$
Entropy of changed types	0.45	0.37	$< 2.2e-16$

Table 5: Median value of hypothesis testing metrics for highly-impactful crash-inducing commits (HICI) and other crash-inducing commits (OCIC), as well as the  $p$ -value of the Wilcoxon rank sum test

Metric	HICI	OCIC	$p$ -value
Committer's experience	177	218	$< 2.2e-16$
Message size	12	12	0.06
Changed files	3	3	$< 2.2e-16$
Added lines	11	7	$< 2.2e-16$
Deleted lines	39	27	$< 2.2e-16$
Entropy of changes	0.63	0.45	$< 2.2e-16$
Number of changed types	5	4	$< 2.2e-16$
Entropy of changed types	0.45	0.39	$< 2.2e-16$

We will perform the same (hypothesis and proportional) analyses to compare highly-impactful crash-inducing commits against other commits. Moreover, we will also compare highly-impactful crash-inducing commits against other crash-inducing commits (*i.e.*, crash-inducing commits with less impact).

### ***Finding.***

**Hypothesis tests:** Table 3 shows the median values of crash-inducing commits and crash-free commits for the metrics listed in Table 2, as well as the  $p$ -values of the Wilcoxon rank sum tests. 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 [? ?] where researchers found that relative code churn measures can indicate faults in modules. In addition, crash-inducing commits have higher entropy of changes values, *i.e.*, their changed code tend to be equally distributed among the changed files (mean and median values of 0.45, and 0.58 respectively); while in the case of crash-free commits, mean and median values of the entropy of change metric are respectively 0.36 and 0. 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 across 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.

In light of results from Table 3, we reject null hypotheses  $H_{01}^1 \sim H_{01}^5$ ,  $H_{02}^1 \sim H_{02}^2$ , and  $H_{03}^1$ . In other words, for all metrics listed in Table 2, there exist statistically significant differences between crash-inducing commits and crash-free commits.

Table 4 compares highly-impactful crash-inducing commits with other commits. We observe the similar results as in Table 3, *i.e.*, highly-impactful crash-inducing commits were submitted by less experienced developers with longer commit messages. These commits changed significantly more lines of code and contain more changed types.

Table 5 shows the comparison between highly-impactful crash-inducing commits and other crash-inducing commits. Developers who submitted highly-impactful crash-inducing commits have significantly lower experience. More lines of code were changed (and these changes tend to equally distributed in multiple files) in highly-impactful crash-inducing commits, which possess more changed types.

**Proportional analysis:** Table 6 summarizes the results of our proportion analysis between crash-inducing commits and crash-free commits. Interestingly, we observe that crash-inducing commits are 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. It could also mean that Mozilla developers handle the more complex changes on the code. In addition, 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

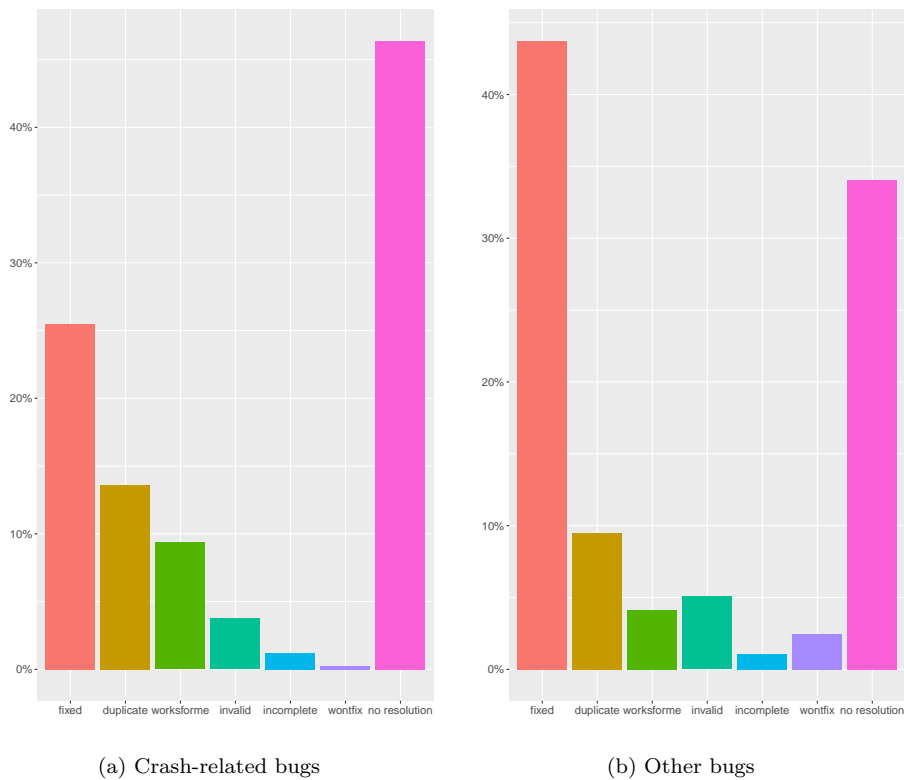


Fig. 7: Frequency of resolutions of crash-related bugs and other bugs

in software development [?]. A higher proportion of crash-inducing commits are aimed at fixing 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 [?], that legacy code becomes difficult to maintain.

We analyzed the fixes of bugs in Firefox and found that developers tend to use a single commit to fix crash-related bugs. We also observed that crash-related bugs are reopened less often, in comparison to bugs that do not crash the system, which may be an indication that developers are more careful when fixing crash-related bugs. Bugs that require supplementary fixes and/or bugs that are reopened are costly for software organisations. To get a deeper insight of the bug correction process of Firefox, we parsed all the bug reports that were submitted in Mozilla Bugzilla between January 2012 and December 2012. For each of these bugs, we checked the resolution status. Figure 7 shows the resolution frequency of crash-related bugs and other bugs during the studied period. We observe that 46.4% of crash-related bugs have no resolution, and only 25.5% of crash-related bugs have been resolved. Regarding crash-free bugs, 34% have no resolution, and 43.7% of these bugs have been resolved. Moreover, 9.4% of crash-related bugs were resolved as “worksforme”, whereas

Table 6: Median value of proportional metrics for crash-inducing commits and other commits

<b>Metric</b>	<b>Crash-inducing</b>	<b>Crash-free</b>
Using Mozilla email	41.8%	36.7%
Is bug fix	91.4%	83.5%
Supplementary fixes	15.5%	38.3%
Bug reopening	3.8%	6.7%

Table 7: Median value of proportional metrics for highly-impactful crash-inducing commits (HICI) and other commits

<b>Metric</b>	<b>HICI</b>	<b>Other</b>
Using Mozilla email	42.8%	37.2%
Is bug fix	89.7%	84.8%
Supplementary fixes	4.7%	38.3%
Bug reopening	1.1%	6.7%

only 4.1% of crash-free bugs have this resolution. In a previous study, Joorabchi et al. [?] found that 66% of “closed” non-reproducible reports (*i.e.*, bugs resolved with “worksforme”) can be eventually reproduced and fixed. In our previous work, we also found that some bugs are prematurely closed with the “worksforme” resolution. Therefore, the “worksforme” resolution may be a mislabelling and could reflect developers’ negative attitude towards a difficult problem. The above statistics suggest that developers do not resolve many crash-related bugs even though they work on them carefully when they choose to fix them. This outcome is surprising given the fact that crash-related bugs can lead to users’ frustration and affect a software organisation’s reputation. We explain this surprising result by the fact that Firefox being an open source software system, developers can choose the bugs that they wish to fix. This flexibility may result in a majority of developers choosing easy bugs, that are not crashing the system.

Table 7 shows the comparison between highly-impactful crash-inducing commits and other commits. We observe the similar results as in Table 6, *i.e.*, highly-impactful crash-inducing commits tend to be submitted by Mozilla developers. A higher percentage of these commits aim to fix bugs than other commits. The bugs caused by these commits are also re-opened less often and fixed in fewer commits (in general a single commit) in comparison to other bugs (including crash-inducing commits that crash less frequently and affect less users). This result suggests that when developers fix a highly-impactful bug, they are very careful to ensure that their fix is correct. Moreover, we found that only 26.2% of highly-impactful bugs have been fixed. This result is similar to the “fixed” proportion of other crash-related bugs.

*In general, crash-inducing commits are submitted by less experienced developers. They contain longer commit messages, change more files and more lines of code than crash-free commits. Crash-inducing commits contain more changed types, their changed statements tend to be scattered across different changed types. Many crash-inducing commits are aimed at fixing a previous bug. Crash-inducing commits are often submitted by developers using Mozilla email accounts (i.e., Mozilla developers). Developers are careful when fixing crash-related bugs; fixes of crash-related bugs require less re-working (i.e., supplementary fixes) in comparison to the fixes of other bugs.*

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 faulty 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 [? ], 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 25 metrics along 4 dimensions from respectively the studied commit logs and the corresponding source code of Firefox. Table 8 to Table 11 show our selected metrics (i.e., independent variables for the prediction models) and their rationales. Since we compute code complexity, SNA, and changed type metrics only for C/C++ code, we only consider commits that change C/C++ code in the prediction.

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 [? ]. In a previous

Table 8: Commit log metrics

Attribute	Explanation and Rationale
Hour	Hour (0-24). Code committed at certain hours may lead to crashes ( <i>e.g.</i> , hours around quitting time).
Week day	Day of week (from Mon to Sun). Code committed on certain week days may be less carefully written ( <i>e.g.</i> , Friday) [? ? ], and would lead to crashes.
Month day	Day in month (1-31). Code committed on certain days may be less carefully written ( <i>e.g.</i> , 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. ( <i>e.g.</i> , December, during Christmas holidays).
Day of year*	Day of year (1-366). Combined the rationales of 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.
Entropy of changes	Measurement of the dispersion of changes among files in a commit. In RQ2, we found that commits with higher entropy value tend to induce 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.

bug prediction study, Shihab et al. [?] 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. It runs faster and uses less memory than than C4.5 [? ]. Developed by Leo Breiman and Adele Cutler, Random Forest [?] 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 [? ]. 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 [? ], 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 8 to Table 11, removed metrics are marked with \*.

Table 9: 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>i.e.</i> , 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.

Table 10: 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 11: 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.

We use ten-fold cross validation [?] 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

Table 12: Accuracy, precision, recall, and F-measure (in %) obtained from GLM, Naive Bayes, C5.0, and Random Forest when predicting crash-inducing commits and crash-free commits

Metric	GLM	Bayes	C5.0	Random Forest
Accuracy	67.4	43.3	70.0	73.5
Crash-inducing precision	58.9	38.9	57.2	61.2
Crash-inducing recall	38.8	94.5	76.8	76.7
Crash-inducing F-measure	46.8	55.0	65.5	68.0
Crash-free precision	70.1	78.8	83.0	83.8
Crash-free recall	84.1	13.2	66.2	71.7
Crash-free F-measure	76.6	22.6	73.4	77.3

variables (prediction metrics) to identify the top predictors for the algorithm with the best prediction results.

**Finding.** Table 12 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 61.2% and a recall up to 94.5%. 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 modified 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 61.2%, and a recall of 94.5%. 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.*

RQ4: How well can we predict commits that lead to frequent crashes that impact a large user base?

**Motivation.** In RQ2, we characterised commits that would lead to frequent crashes, impacting a large user base. These commits can tarnish the brand



Table 13: Accuracy, precision, recall, and F-measure (in %) obtained from GLM, Naive Bayes, C5.0, and Random Forest when predicting highly-impactful crash-inducing commits

Metric	GLM	Bayes	C5.0	Random Forest
Accuracy	76.7	36.9	79.1	81.1
Crash-inducing precision	58.0	26.5	56.8	60.9
Crash-inducing recall	7.1	91.1	48.5	54.9
Crash-inducing F-measure	12.7	40.9	52.4	57.6
Crash-free precision	77.1	87.2	84.6	86.4
Crash-free recall	98.5	19.3	88.4	89.0
Crash-free F-measure	86.6	31.7	86.6	87.6

of a software organisation since they result in many users experiencing frequent crashes. In this research question, we intend to build statistical models that can enable an early detection of highly-impactful crash-inducing commits.

**Approach.** We use the same predictive algorithms as in **RQ3** to build our statistical models. As we found that only 23.7% of commits (that changed C/C++ files) would lead to highly-impactful bugs, when under-sampling our data-set of commits that are not highly-impactful, we adjust the probability value from 0.5 to 0.3, in order to balance our training data-set.

**Finding.** Table 13 shows prediction results for highly-impactful bugs. In general, our models can predict commits that induce highly-impactful bugs with with a precision of 60.9% and a recall of 91.1%. As in **RQ3**, Random Forest also outperforms the other algorithms and the *closeness* metric is still the best predictor.

*Our models can achieve a precision of 60.9%, and a recall of 91.1% when predicting highly-impactful crash-inducing commits. The Random Forest algorithm achieves the best prediction performance. The closeness metric is ranked as the best predictor by this algorithm (i.e., Random Forest).*

RQ5: What are the characteristics of commits that are misclassified by our prediction models?

**Motivation.** Although our statistical models achieve a good performance in **RQ3** and in **RQ4**, we intend to investigate the reasons why some clean commits are misclassified as faulty (false positives), and some faulty commits are misclassified as clean (false negatives). A good understanding of the characteristics of false positives and false negatives can help improve our statistical models.

**Approach.**

We extract false positive and false negative commits from the results of our

Table 14: Median metric values of false positive commits and other commits.

Metric	False positive	Other	$p$ -value
Committer’s experience	197	238	$< 2.2e-16$
Message size	12	11	$< 2.2e-16$
Changed files	3	2	$< 2.2e-16$
Inserted lines	15	9	$< 2.2e-16$
Deleted lines	28	25	0.003
Entropy of changes	0.6	0.4	$< 2.2e-16$
Number of changed types	4	3	$< 2.2e-16$
Entropy of changed types	0.4	0.4	$< 2.2e-16$
Closeness	3.5	3.5	$< 2.2e-16$
LOC	822	694	$8.0e-09$

Table 15: Median metric values of false negative commits and other commits.

Metric	False negative	Other	$p$ -value
Committer’s experience	261	226	$< 2.2e-16$
Message size	11	11	$< 2.2e-16$
Changed files	2	3	$< 2.2e-16$
Inserted lines	6	10	$< 2.2e-16$
Deleted lines	21	26	0.003
Entropy of changes	0.1	0.4	$3.795e-10$
Number of changed types	3	4	$< 2.2e-16$
Entropy of changed types	0.3	0.4	$< 2.2e-16$
Closeness	3.4	3.5	$< 2.2e-16$
LOC	497	730	$< 2.2e-16$

Random Forest classifier (built in **RQ3** and **RQ4**), and conduct the following analyses:

**False positive:** We statistically compare false positive commits against other studied commits in terms of the metrics described in Table 2 as well as closeness (our best predictor) and LOC (a popularly metric used to assess software maintenance effort, *e.g.*, [? ]). We also analyse bug reports created between January 2012 and December 2013, to examine whether false positive commits lead to other kinds of bugs (other than crash-related bugs). As in **RQ2**, we also use a 95% confidence level and the Bonferroni correction to decide whether a results is statistically significant, *i.e.*,  $p$ -value  $< 0.05/10 = 0.005$ .

**False negative:** First of all, we apply the aforementioned statistical approach to compare false negative commits against other studied commits. Then, we examine the characteristics of crash-inducing commits that are misclassify by our predictions models by comparing their changed types with those of other commits. For each studied changed type listed in Table 1, we will report the percentage of its occurrences in false negative commits and in other commits.

### ***Finding.***

**False positive:** Table 14 shows median metric values for false positive and other commits when predicting crash-inducing commits. For all the studied metrics, false positive commits are significantly different than other commits.

Table 16: Percentage (%) of changed type occurrences in false negative commits and other commits.

Changed type	False negative	Other
Renaming	46.2	39.3
code block	23.6	22.2
Parameter	8.5	10.4
Comment	6.1	7.3
Preprocessor	5.4	7.7
Declaration	2.8	3.7
Control flow	2.8	3.6
Function	1.8	2.3
Invocation	1.5	1.9
Type	0.6	0.6
Data type	0.3	0.4
Class	0.2	0.3
Initialisation	0.1	0.1
Access	0.1	0.1
Operator	0	0
C++ template	0	0

In general, false positive commits are often submitted by less experienced developers, they have higher complexity in term of lines of code, and changed more lines of code. Their changed code tends to equally distributed among multiple files. This is the reason why these commits are misclassified. In fact, we observed that the Random Forest model tends to classify commits with less developers’ experience, higher number of changed files and lines of code as “crash-inducing commits”.

In addition, 1,211 out of 13,093 false positive commits (9.2%) led to other bugs (that did not crashed the system). Therefore, although our Random Forest model wrongly classified them as crash-inducing commits, developers should still pay attention to them because they are likely to introduce a fault in the system, even though the fault does not crash the system. Developers should double check these commits (*e.g.*, during code review sessions) before integrating them into the version control system.

**False negative:** Table 15 shows median metric values of false negative and other commits. False negative commits are misclassified, because they were submitted by more experienced developers, changed less files and less lines of code. Their entropy of changes is also lower than other commits. Table 16 shows the percentage of changed type occurrences in false negative commits and in other commits. These two kinds of commits have a very close percentage (less than 2%) of all changed types except renaming, where false negative commits have higher percentage. Surprisingly, renaming is the most frequent changed type that leads to crashes; implying that inappropriate or incomplete renaming can lead to runtime variable missing or mismatching errors (*e.g.*, “variable does not exist” `From Foutse ▶ is this an example? ◀`), which crash a software system. Especially, when developers use a tool to perform a renaming operation, the tool may not rename all variables in all related files.

This finding suggests that developers should be careful when performing this apparently “simple” operation. Nowadays, although some IDEs support automatic renaming, they cannot guarantee that all related or dependent variables (functions, or classes) are correctly renamed `From Foutse` `do you have a reference for this?`. In the future, we will empirically evaluate whether renaming or refactoring-related metrics can help improve the recall of our models. We also plan to study the relationship between code refactorings and fault-proneness.

In addition, we found similar results of the false positive and false negative commits yielded by our Random Forest model in **RQ4**. Software researchers and practitioners can refer to our detailed results at: <https://github.com/swatlab/crash-inducing>.

*It is worthy to spend time examining false positive commits because although they do not lead to crashes, some of them cause other types of bugs. False negative commits have higher percentage of renaming operations than other commits; suggesting that developers should be careful when performing renaming operations.*

## 6 Threats to Validity

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

*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, the original SZZ algorithm [?] selects bug-inducing commit candidates that were submitted before a bug’s opening date. But we cannot directly apply this principle to filter crash-inducing commit candidates, because a bug may derive from different crashes, which have the common top signatures but differ from each other in the remaining signatures. Thus, we take both the first crash occurrence date and bug opening date into account, and match a candidate’s changed files with the crash-related bug’s crash-signatures. In addition, computing the code complexity and SNA 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 design a parallel algorithm to compute these metrics in real time.

*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 re-

moved 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 the bug. 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. In this extension, we readjust the classification for some srcML tags to group them into the correct changed types. Moreover, in this paper, we also investigate the reason behind false positives and false negatives from the prediction models.

*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 [? ]. In our previous work [? ], 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 central 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 on other systems.

## 7 Related Work

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

### 7.1 Crash Analysis

Crashes stop a software system unexpectedly, causing data loss and users’ frustration. Today, many software organisations have deployed automatic crash collecting systems to gather and triage crash occurrences. Researchers intend to study crash reports from these systems to facilitate the debugging and bug fixing process for software practitioners. Podgurski et al. [? ] 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. [? ] 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. [?] studied crash information in Firefox and Eclipse, and proposed an algorithm that can locate and rank faulty files, as well as a method that can identify duplicate and related bug reports. Kim et al. [?] 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 Fault Prediction Techniques

Traditional fault prediction techniques used coarse-grained metrics, such as bug report metrics, to identify fault-prone modules or specific types of bugs. By using social factors, technical factors, coordination factors, and prior-certifications factors, Hassan et al. [?] 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. [?] 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. [?] used Logistic Regression models to predict bug re-opening in Windows. In our previous work [?], 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 Fault Prediction Techniques

Though traditional fault prediction techniques can help software organisations prevent faults to some extent, developers can only identify the error-prone modules responsible for these faults after the faults have been filed into bug reports. During the period between the integration of the faulty code into the version control system and the opening of the bug report, a faulty commit could have negatively impacted a large user base. Just-In-Time fault prediction techniques are designed to predict faults in commits, in order to allow developers to track and fix faults as soon as they are submitted for integration in version control systems. Kamei et al. [?] used a wide range of source code metrics to predict fault-prone commits in six open-source systems and five commercial systems. Fukushima et al. [?] applied Just-In-Time fault prediction techniques to cross-project fault 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. [?] built statistical models to predict high impact fix-inducing changes. In this paper, we use change level metrics to predict crash-inducing commits. Some crashes would frequently occur and affect a large population of users, we also predict commits that lead to these

highly-impactful crashes. Moreover, we perform statistical analyses to understand the reason of false positive and false negatives errors in our prediction. We found that though false positive commits do not lead to crash-related bugs, some of them lead to other bugs and other possess latent peril. We are also aware of the risk of renaming changes and suggest software practitioner to pay attention to this “simple” operation.

## 8 Conclusion

Crashes, which are unexpected terminations of software systems, are one of the major sources of frustration for users. The frequent crashes of software systems 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 to answer five research questions pertaining the proportion of crash-inducing commits in Firefox (RQ1), the characteristics do crash-inducing commits (RQ2), the prediction of crash-inducing commits (RQ3), the prediction of commits that lead to frequent crashes and that impact a large user base (RQ4), and, finally, the characteristics of commits that are misclassified by our prediction models (RQ5).

In summary, we found that crash-inducing commits account for more than 25% of all the commits that we studied in Firefox. We also found that, compared to other commits, crash-inducing commits are often submitted by developers with less experience and that they contain longer comments, more changed files and changed lines as well as more change types. In addition, compared to other crash-related bugs, bugs that yield to frequent crashes and that impact a large user base were less reopened and tended to be fixed by a single commit.

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 61.2% and a recall up to 94.5% to predict crash-inducing commits and achieved a precision up to 60.9% and a recall up to 91.1% to predict commits that lead to highly-impactful bugs, *i.e.*, bugs that yield to frequent crashes impacting a large user base. By analysing the prediction errors, we observed that renaming is the most frequent change type in Firefox and that crash-inducing commits have a higher percentage of renaming changes. This observation suggests that developers are not fully aware of the latent risks of their renaming operations and should double-check their renaming operations for correctness and completeness.

Software organisations can use our predictive models to identify crash-prone code as soon as it is committed in the source code repository and, more generally, our approach to build models adapted to their context. They could then correct their code quickly to avoid that users experience crashes and, thus, reduce users’ frustrations.

In the future, we plan to generalise our approach to other software systems when and if they open their crash collecting systems to researchers. We also want to implement our models into tools for different programming languages and integrate them into interactive development environments to warn developers as soon as they commit of the potential crash that could yield from their commits. We also want to study in more details the code and developers' characteristics related to crashes to propose mitigating measures even before the code is committed by developers.

**Acknowledgements** This work is supported by NSERC (Natural Sciences and Engineering Research Council of Canada).