2015 IEEE International Conference on Software Quality, Reliability and Security

An Empirical Study of Highly-impactful Bugs in Mozilla Projects

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

***Abstract*—Bug triaging is the process that consists in screening and prioritising bugs to allow a software organisation to focus its limited resources on bugs with high impact on software quality. In a previous work, we proposed an entropy-based crash triaging approach that can help software organisations identify crash-types that affect a large user base with high frequency. We refer to bugs associated to these crash-types as highly-impactful bugs. The proposed triaging approach can identify highly-impactful bugs only after they have led to crashes in the field for a certain period of time. Therefore, to reduce the impact of highly-impactful bugs on user perceived quality, an early identification of these bugs is necessary. In this paper, we examine the characteristics of highly-impactful bugs in Mozilla Firefox and Fennec for Android, and propose statistical models to help software organisations predict them early on before they impact a large population of users. Results show that our proposed prediction models can achieve a precision up to 64.2% (in Firefox) and a recall up to 98.3% (in Fennec). We also evaluate the benefits of our proposed models and found that, on average, they could help reduce 23.0% of Firefox' crashes and 13.4% of Fennec's crashes, while reducing 28.6% of impacted machine profiles for Firefox and 49.4% for Fennec. Software organisations could use our prediction models to catch highly-impactful bugs early during the triaging process, preventing them from impacting a larger user base.**

***Index Terms*—Bug triaging, entropy analysis, crash report, prediction model, mining software repositories.**

I. Introduction

Today, many software organisations (*e.g.*, Microsoft, Mozilla) embed automatic crash reporting tools in their software systems. Whenever the software crashes (*i.e.*, terminates unexpectedly in the user environment), the automatic crash reporting tool collects information about the crash and sends a detailed crash report to the software vendor. A crash report usually contains a signature, the stack trace of the failing thread, runtime information, such as the crash time, and information about the user environment, *e.g.*, the operating system, the version, and the install age. Crashes with the same signature are grouped together and filed in the same bug report. Software quality managers and developers usually judge the priority and severity of a bug by looking at the frequency of its related crashes [14].

Indeed, crash frequency is an important factor to evaluate the severity of a bug, because a high crashing frequency represents a high number of crash occurrences. However, the frequency alone does not show the full picture because the crashes due to a bug may be concentrated on a limited user group, while the crashes due to another bug may affect most users of the software system. If the two bugs have the same number of crash occurrences, the second bug should be assigned a higher priority and severity because it impacts a larger user base. Khomh et al. [13] have proposed an entropy metric to capture the distribution of crash occurrences among the users of a software system. They also proposed an entropy-based crash triaging approach that assigns a high priority to the bugs related to crashes that occur frequently (*i.e.*, high frequency) and affect a large user base (*i.e.*, high entropy). In this work, we refer to crash-related bugs with both high crashing frequency and entropy as *highly-impactful bugs*. Although the entropy-based crash triaging approach proposed by Khomh et al. can successfully identify highly-impactful bugs, it takes a certain period of time to assess which crashes occur frequently with high entropy; a period during which the users of the system are negatively impacted by the crashes.

In this paper, we investigate models to predict highly-impactful bugs early on before the software is released. We propose models that software organisations can use to identify highly-impactful bugs at an early stage of development, *e.g.*, during alpha or beta-testing phases. Such prediction models allow software organisations to focus on highly-impactful bugs earlier and improve the overall quality of their software systems effectively. We analyse the crash reports of Firefox and Fennec for Android (referred as Fennec in the rest of this paper) during the period of January 2012 to December 2012 and answer the following research questions:

RQ1: What is the proportion of highly-impactful bugs?

We apply the algorithm proposed by Khomh et al. [13] to identify users by their machines' configuration, *i.e.*, CPU type, OS name, and OS version, and found that highly-impactful bugs account for 42.3% of all bugs in Firefox, and 37.9% in Fennec.

RQ2: Do highly-impactful bugs possess different characteristics than other bugs?

We study whether highly-impactful bugs possess different characteristics than other bugs (*i.e.*, bugs with low entropy and–or low frequency). Compared to other bugs, we observed that highly-impactful bugs are often fixed by more experienced developers and they generate larger amounts of comments. However, the proportion of highly-impactful bugs that are fixed and resolved is smaller in comparison to bugs with low entropy and–or low frequency.

RQ3: Can we predict highly-impactful bugs?

978-1-4673-7989-2/15 $31.00 © 2015 IEEE
DOI 10.1109/QRS.2015.45

We applied GLM, C5.0, ctree, randomForest, and cforest algorithms to predict whether or not a bug will become highly-impactful, *i.e.*, it will have both a high crashing frequency and a high entropy. Our predictive models can achieve a precision up to 64.2% (in Firefox) and a recall up to 98.3% (in Fennec). Software organisations can use our prediction models in the early stage of their new releases to identify and fix bugs before they become highly-impactful.

RQ4: Which benefits can be achieved by predicting highly-impactful bugs?

The identification and correction of highly-impactful bugs at an early stage of development reduces the number of users impacted by these bugs; resulting in an improvement of the user perceived quality of the software system. We calculate the date D_{pf} at which a highly-impactful bug, which is successfully predicted or transferred from a previous release, would be potentially fixed as follows. We compute the median fixing duration $Duration_{med}$ of fixed bugs that were assigned with the highest priority in the previous release and add it to the opening date (*i.e.*, D_o) of the highly-impactful bug, *i.e.*, $D_{pf} = D_o + Duration_{med}$. All the crashes that occurred after D_{pf} can be avoided if developers fix the bug without delay. Results show that a considerable amount of crash occurrences can be avoided with our prediction models. For Firefox and Fennec, on average, crash occurrences can be reduced by 23.0% and 13.4% respectively, and the number of unique machine profiles that are impacted by crashes can be reduced by 28.6% and 49.4%, respectively.

The remainder of the paper is organised as follows. Section II provides background information on Mozilla crash and bug triaging systems. Section III explains the identification of highly-impactful bugs. Section IV presents our data collection and processing. Section V describes and discusses the results of the four research questions. Section VI discusses threats to the validity and Section VII summaries related work. Section VIII concludes the paper.

II. Mozilla Crash and Bug Triaging Systems

Mozilla ships software with a built-in automatic crash reporting tool, *i.e.*, the Mozilla Crash Reporter. When a Mozilla product crashes unexpectedly, the user receives a dialog box from the Mozilla Crash Reporting tool that suggests to submit the crash report to developers for improving the product's quality. A crash event and a detailed crash report are generated and sent to the Socorro server [21]. The crash report provides a stack trace for the failing thread and other 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. Socorro collects crash reports from end users, assigns a unique ID to each report and groups similar crash reports together by the top method signatures of their stack trace. Such a group of crash reports in which all the stack traces share a common top frame is called a crash-type. The Socorro server is an open-source project, and its data are also open. It provides a rich web interface for software practitioners to analyse crash-types. Developers can file bugs for crash-types with high crash occurrences in Bugzilla. Multiple crash-types can be linked to the same bug and multiple bugs can also be linked to the same crash-type. In Soccoro, the list of bugs filed in Bugzilla is provided for each crash report. The Socorro server and Bugzilla are integrated, *i.e.*, developers can directly navigate to the linked bugs (in Bugzilla) from a crash-type summary in Socorro. Developers use the information contained in crash reports to debug and fix bugs.

III. Identification of Highly-Impactful Bugs

We identify highly-impactful bugs following the approach proposed by Khomh et al. [13]. The approach is composed of three parts. The first part consists in identifying the list of unique user profiles that are impacted by each bug. The second part is the computation of the entropy and frequency of the bugs. The third part is the classification of bugs based on entropy and frequency values. The following subsections elaborate on each of these parts.

Part 1: Identification of Unique User Profiles Impacted by Each Bug

Mozilla crash reports do not contain personal information to identify users reporting the crashes for privacy reasons, but we can identify user profiles with the following information from crash reports:

- *crash signature*: top method signature of a crash. Crashes with the same crash signature are grouped into one crash type in Socorro [13].
- *OS name*: name of the operating system on which the crash occurred.
- *OS version*: version of the operating system on which the crash occurred.
- *CPU type*: family model of the CPU of the machine on which the crash occurred.
- *Bug list*: list of bugs related to the crash.

For each crash report, we combine the contained users' environment information (*i.e.*, OS name, OS version, CPU type) to build a vector of unique profiles where each profile represents a unique machine configuration [1]. Identifying unique machine configurations is important to compute the entropy of a bug. We associate each unique profile with the list of bugs for which crash reports contain information corresponding to the profile. The detailed mapping algorithm between bugs and machine profiles is described in our previous work [1].

Part 2: Computation of the Entropy and Frequency of Bugs

We compute the entropy of a bug using the normalised Shannon's entropy [17] defined in Equation (1):

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

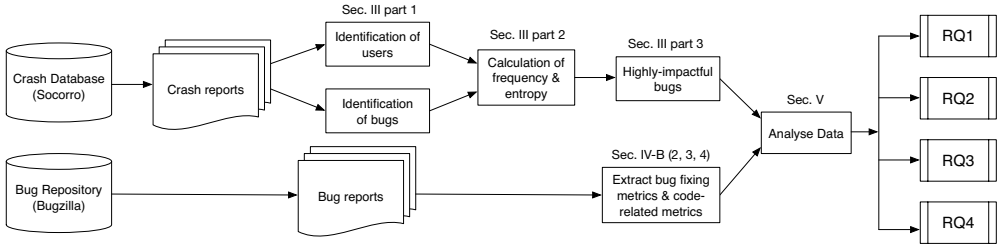

Fig. 1: Overview of our approach to identify highly-impactful bugs and extract bug fixing metrics

where b is a bug; p_i is the probability of a specific machine profile i reporting the bug b ($p_i \geq 0$, and $\sum_{i=1}^{n} p_i = 1$); and n is the total number of unique machine profiles running the software. For a bug b, where all the machine profiles have the same probability of reporting b, the entropy is maximal (*i.e.*, 1). On the other hand, if a bug b is reported by only one machine profile i, the entropy of b is minimal (*i.e.*, 0). Bugs with high entropy values are reported by more unique machine profiles. Therefore, a high entropy value for a bug means that the bug is experienced by a high percentage of users.

We compute the frequency of a bug b following Equation (2).

$$Fq(b) = \frac{Ncr(b)}{max_{j \in B}(Ncr(j))} \quad (2)$$

where b is a bug; $Ncr(j)$ is the number of crash reports linked to the bug j, and B is the total number of bugs. We implement the entropy and frequency computation algorithm in Python and share the code in the following repository: https://github.com/swatlab/highly-impactful.

Part 3 : Classification of Bugs Based on Entropy and Frequency Values

Similar to Khomh et al [13], we use the median values of entropy and frequency to classify bugs into the following four categories:

- **Highly-impactful Bugs:** bugs with frequency and entropy values above the median. These bugs impact a large proportion of users.
- **Skewed Bugs:** bugs with a high frequency value (*i.e.*, above the median) but a low entropy (*i.e.*, below or equal to 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 highly-impactful among the users that report them (*i.e.*, entropy value above to the median) but do not occur very often to the majority of users (*i.e.*, frequency value below or equal to the median).
- **Isolated Bugs:** bugs with frequency and entropy values below or equal to the median. These bugs cause crashes rarely and affect a small number of users.

IV. Study Design

This section presents the data collection and data processing of our case study, which aims to address the following four research questions:

1) What is the proportion of highly-impactful bugs?
2) Do highly-impactful bugs possess different characteristics than other bugs?
3) Can we predict highly-impactful bugs?
4) Which benefits can be achieved by predicting highly-impactful bugs?

A. Data Collection

We mine crash reports to compute the frequency and entropy of bugs as well as bug reports to study the characteristics of highly-impactful bugs and to build predictive models. We select the two following open-source software systems: *Firefox* and *Fennec for Android*. Firefox is a popular web browser developed by Mozilla. Fennec for Android is the codename of Firefox for Android. We analyse crash reports in both systems from January 2012 to December 2012 and the corresponding bug reports. These crashes and bugs are extracted from copies of Socorro and Bugzilla databases obtained from the Mozilla corporation. Table I shows the numbers of crash reports, extracted bugs reports, related releases, and detected users in the two subject systems.

TABLE I: Numbers of crash reports, extracted bugs, related releases, and detected users in the studied systems

System	Crash reports	Bug reports	Releases*	Machine profiles
Firefox	132,484,824	6,636	22	40,942
Fennec	6,239,077	2,565	8†	11,488

** We only count the number of official main releases, e.g., Firefox 10.0.1 and 10.0.2 are considered as minor releases of Firefox 10.*

† Mozilla did not release Fennec 11, 12, and 13. The next release of Fennec 10.0 is Fennec 14.0. We consider all minor releases of Fennec 11, 12, and 13 as Fennec 10.

B. Data Processing

Figure 1 shows a general view of our data processing approach. First, we mine crash reports to compute bug frequency and entropy values. Then, we mine bug reports to analyse the characteristics of highly-impactful bugs and build statistical models for their prediction. The remainder of this section elaborates more on these steps. All the data and scripts used in this study are available at: https://github.com/swatlab/highly-impactful.

1) Mining Crash Reports: We parse crash reports using a Python script and extract the following information: *crash signature*, *OS name*, *OS version*, *CPU type*, *bug list*, and *uptime*. We use *uptime* as a predictor to answer RQ3, while using the rest of information to identify the list of unique machine profiles that are associated to each bug (see Section III).

2) Mining Bug Reports: Similar to crash reports, we parse bug reports using a Python script and extract information about bug opening date, bug fixing date, bug assigned date, reporter name, assignee name, involved fixer name(s), bug title, bug comments, and re-opened times. We also identify patches from bugs' attachments using the keyword "patch", *i.e.*, if the type of an attachment is marked as "patch", we consider it as a bug fixing patch.

3) Computing Code Complexity Metrics: We localise the faulty file(s) of each bug by analysing its crashing stack trace. In most crash-related bug reports, the top crashing frames are provided in the comments. We parse these crashing frames to extract crashed files or crashed function signatures, which are then mapped to the corresponding files in the source code of a specific release on which the crashes occurred. Using the SLOCCount tool [20], we found that, in both subject systems, C and C++ code accounts for more than 90%. Hence, in this study, we only take C and C++ files into account. We analyse every detected main source code release of Firefox and Fennec using the Understand tool [16], which generates its results into an Understand database (UDB) and provides a Python API for further analysis. We apply a Python script to extract five code complexity metrics for each faulty file identified: lines of code, average cyclomatic complexity, number of functions, maximum nesting, and ratio of comment lines over code lines. Detailed characteristics of these metrics are described in Section V.

4) Social Networking Analysis Metrics: From the Understand databases generated in Section IV-B3, we extract all C and C++ files and combine each .c or .cpp file with its corresponding .h file as a class node while merging their dependencies. To represent the relationship among these nodes, we build an adjacency matrix. Using this matrix and a Python script based on the network analysis package, igraph [10], we compute the following social networking analysis (SNA) metrics for each class node: PageRank, betweenness, closeness, indegree, and outdegree (detailed characteristics of these metrics are described in Section V). We map the SNA metric values of each class node back to their corresponding bugs. Bugs that do not contain stack trace or could not be mapped to any file in the source code (*e.g.*, only crashed memory addresses are given in the stack trace [1]) are not mapped to any SNA or complexity metric value.

V. Case Study Results

This section presents and discusses the results of our four research questions. For each question, we present the motivation, the approach followed to answer the questions, and the findings.

RQ1: What is the proportion of highly-impactful bugs?

Motivation. This question is preliminary to the other questions. It provides quantitative data on the proportion of highly-impactful bugs in our subject systems. This result will clarify the prevalence of highly-impactful bugs in Mozilla Firefox and Fennec to help understand the importance of their identification early on during the development process.

Approach. We identify highly-impactful bugs following the approach described in Section III and compute their percentage over the total number of reported bugs.

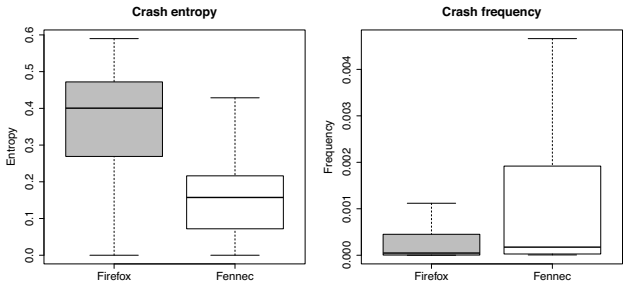

Fig. 2: Distribution of bugs' crashing entropy and frequency in the subject systems

Findings. Figure 2 shows the distribution of entropy and frequency values for all bugs in our subject systems. Table II shows the detailed distribution of bugs in the four categories presented in Section III.

Highly-impactful bugs account for respectively 42.3% and 37.9% of all crash related bugs in the two studied systems.

By focusing their maintenance effort on highly-impactful bugs, software organisations will significantly improve the user perceived quality of their software systems, *i.e.*, it will reduce the proportion of crash occurrences in the field.

TABLE II: Distribution of highly-impactful bugs, and other bugs in the subject systems

System	Highly	Skewed	Moderately	Isolated
Firefox	2806 (42.3%)	510 (7.7%)	512 (7.7%)	2808 (42.3%)
Fennec	972 (37.9%)	302 (11.8%)	301 (11.7%)	990 (38.6%)

RQ2: Do highly-impactful bugs possess different characteristics than other bugs?

Motivation. Highly-impactful bugs crash very frequently in the field and impact a large proportion of users. Once such bugs are discovered, developers must fix them as soon as possible, to reduce their negative impact. Khomh et al. [13] recommend assigning the highest priority to these bugs. However, to effectively fix highly-impactful bugs quickly and avoid them being tossed several times, *i.e.*, passed around from one developer to another, it is important to assign them to the right developers. Previous work [11] found that tossing bugs results in longer bug fixing time. Highly-impactful bugs also must be fixed completely, *i.e.*, without any re-opening, because re-opened bugs negatively impact software quality [2]. In this research question, we study the bug fix characteristics of highly-impactful bugs, and compare them to non highly-impactful bugs (*i.e.*, all other remaining bugs).

Approach. To answer this research question, we first apply the approach described in Section III to classify bugs from

TABLE III: Metrics used to compare the characteristics of highly-impactful bugs and other bugs

Metric	Description and Rationale
Fixing time	Duration (in seconds) of the period between the bug opening date and the last modification date. We use the fixing time as a proxy for fixing effort.
Triaging time	Duration (in seconds) of the period between the bug opening date and the first bug assignment date. Low triaging time may imply an efficient work on bug classification.
Patch number	Number of patches submitted to fix a bug. A high number of patches means a high fixing effort (multiple attempts were made to fix the bug).
Comment size	Number of words in the comments contained in a bug report. A high number of words would mean that an intensive discussion took place.
Reporter experience	The number of bugs filed by the reporter of a bug in the past. A reporter who filed a high number of bugs is likely to gain recognition for the relevance of her bug reports. Quality managers may decide to pay more or less attention to her reported bugs.
Assignee experience	The number of bugs fixed by the bug assignee in the past. Assignees with a high experience are likely to fix the bug quickly.
Fixer number	The number of unique developer names in the bug fixing history. A high number of fixers means that the bug was tossed around or required the participation of multiple developers.
CC number	Number of developers who were interested in the bug. These developers may not have played a direct role in fixing the bug. However, a high CC number indicates a high interest in a bug.
Re-opened frequency	Number of time that a bug is re-opened. Frequent bug re-openings increase development costs and decrease users' satisfaction [18].
Closed percentage	Percentage of bugs from a category (*i.e.*, highly, skewed, moderately, or isolated) whose final status is "resolved" or "verified". A high closed percentage for a category may suggest a prioritisation of the bugs from the category.

our subject systems in four categories: Highly-impactful Bugs (highly), Skewed Bugs (skewed), Moderately-impactful Bugs (moderately), and Isolated Bugs (isolated). Next, for each bug, we parse the corresponding bug report using a Python script available at https://github.com/swatlab/highly-impactful, and compute the metrics shown in Table III. Finally, we test the following 10 null hypotheses to compare the bug fix characteristics of highly-impactful bugs and bugs from the other three categories.

Comparing the effort required to fix highly-impactful bugs vs. other bugs.

H_{01}^1: *the fixing time is the same for highly-impactful bugs and other bugs.*

H_{01}^2: *the triaging time is the same for highly-impactful bugs and other bugs.*

H_{01}^3: *the number of patches is the same for highly-impactful bugs and other bugs.*

H_{01}^4: *the comment size is the same for highly-impactful bugs and other bugs.*

H_{01}^5: *the re-opened frequency is the same for highly-impactful bugs and other bugs.*

Comparing people involved in filing and fixing highly-impactful bugs vs. other bugs

H_{02}^1: *reporters' experience is the same for highly-impactful bugs and other bugs.*

H_{02}^2: *assignees' experience is the same for highly-impactful bugs and other bugs.*

H_{02}^3: *the number of fixers is the same for highly-impactful bugs and other bugs.*

H_{02}^4: *the number of developers interested in highly-impactful bugs is the same as other bugs.*

Comparing the bug fix rate of highly-impactful bugs vs. other bugs

H_{03}^1: *the percentage of highly-impactful bugs that are closed is the same as other bugs.*

We apply the Wilcoxon rank sum test [9] to accept or reject the first 9 hypotheses. For H_{03}^1, we compare the values of *closed percentage* obtained for the four categories. The Wilcoxon rank sum test is a non-parametric statistical test used for assessing whether two independent distributions have equally large values. We use this test to compare the characteristics of highly-impactful bugs with other bugs. We also apply the Kruskal-Wallis test [9] to compare the characteristics of bugs from all the four categories (*i.e.*, highly, skewed, moderately, and isolated). The Kruskal-Wallis test is an extension of the Wilcoxon rank sum test. It is used to assess whether two or more samples originate from the same distribution. It does not assume a normal distribution since it is a non-parametric statistical test. For all statistical tests, we use a 95% confidence level (*i.e.*, p-value < 0.05) to decide whether to reject a null hypothesis. As we are investigating 9 characteristics, we apply the Bonferroni correction [7], which consists in dividing the threshold p-value by the number of tests (*i.e.*, we consider that there is a statistically significant difference only if the *p-value $< 0.05/9 = 0.0056$*).

Because highly-impactful bugs are defined using median (*i.e.*, 50th percentile) values of entropy and frequency (see Section III). We perform a sensitivity analysis to assess the impact of this chosen threshold on the results. Precisely, we repeat the identification of highly-impactful bugs using the 70th, and 90th percentiles, and repeat testing the 10 null hypotheses mentioned above.

Findings. Table IV shows the mean values of metrics described in Table III, for highly-impactful bugs and bugs from the other three categories (*i.e.*, skewed, moderately, and isolated), as well as the p-values for Wilcoxon and Kruskal-Wallis tests. Statistically significant p-values are bolded in Table III. On the one hand, the results of the Wilcoxon rank sum test are statistically significant for comment size and reporter experience in both studied systems. Therefore we reject H_{01}^4 and H_{02}^1. Although in general highly-impactful bugs have longer comments and are reported by more experienced developers, in Firefox, their comment size and developers' experience are lower than those of skewed bugs. On the other hand, there is no statistically significant difference between highly-impactful bugs and other bugs for the *fixing time*, *triaging time* and the *fixer number* in both systems, hence we cannot reject H_{01}^1, H_{01}^2 and H_{02}^3. For H_{01}^3, H_{01}^5, H_{02}^2, and H_{02}^4 the results are system dependant. We discuss them in detail in the following paragraphs.

Statistically significant differences: In Firefox, the number of patches proposed for highly-impactful bugs is significantly lower than other bugs. The experience of developers assigned to highly-impactful bugs and the number of developers indirectly involved (*i.e.*, CC number) in fixing highly-impactful

TABLE IV: Mean value of characteristic metrics for highly-impactful bugs and other bugs, as well as the p-values of the Wilcoxon and Kruskal-Wallis tests

Metric	Firefox						Fennec					
	Highly	Skewed	Moderately	Isolated	Wilcoxon	Kruskal.	Highly	Skewed	Moderately	Isolated	Wilcoxon	Kruskal.
Fixing time	1.22e+7	1.40e+7	1.52e+7	1.55e+7	0.126	0.031	8.87e+6	6.63e+6	8.34e+6	1.33e+7	0.41	0.042
Triaging time	4.51e+6	5.54e+6	7.37e+6	3.89e+6	0.212	0.028	3.12e+6	5.86e+6	1.36e+6	6.63e+6	0.197	0.34
Patch number	0.48	0.52	0.61	0.68	**4.3e-14**	**<2.2e-16**	0.59	0.43	0.80	0.68	0.066	**2.0e-6**
Comment size	595.0	728.3	451.9	421.2	**<2.2e-16**	**<2.2e-16**	585.7	538.2	484.3	428.6	**7.4e-11**	**9.0e-11**
Reopened freq.	0.063	0.061	0.065	0.069	0.645	0.858	0.122	0.08	0.070	0.066	**9.7e-6**	**1.4e-4**
Reporter exp.	202.0	224.5	123.8	109.8	**<2.2e-16**	**<2.2e-16**	152.3	149.1	97.8	97.0	**1.7e-8**	**2.3e-13**
Assignee exp.	959.8	1238.4	747.6	776.9	**5.0e-15**	**<2.2e-16**	341.7	459.4	276	284.1	0.036	**4.8e-9**
Fixer number	7.0	6.7	6.8	6.6	0.709	0.588	6.8	5.8	7.0	6.8	0.112	**8.0e-4**
CC number	6.7	6.5	6.1	5.7	**1.6e-5**	**1.0e-5**	6.5	6.0	6.4	6.2	0.174	0.25
Closed %	59.2%	55.1%	73.5%	73.9%	–	–	63.7%	53.3%	73.2%	74.8%	–	–

bugs are significantly higher than other bugs. In Fennec, the bug re-opening frequency of highly-impactful bugs is significantly higher than other bugs. In both systems, the proportion of highly-impactful bugs that are closed is slightly higher than the proportion of bugs from the skewed category, but significantly lower than the proportion of bugs from moderately-impactful and isolated categories that are closed. This result suggests that Mozilla developers do not necessarily prioritise highly-impactful bugs during bug fixing activities; a finding consistent with previous result by Kim et al. [14], that Mozilla developers judge the priority and severity of a bug mostly by looking at the frequency of its related crashes. The sensitivity analysis confirms these findings for highly-impactful bugs detected using the 70th percentile. However, if we identify highly-impactful bugs using the 90th percentile, the differences between the reporter experience and assignee experience of highly-impactful bugs and other bugs is not statistically significant in Firefox. In Fennec, the difference between the reporter experience of highly-impactful bugs and other bugs is not statistically significant. We attribute this result to the low number of highly-impactful bugs found in these systems when using the 90th percentile (*i.e.*, 4.2% in Firefox and 3% in Fennec).

Non statistically significant differences: The results from Table IV show lower fixing time values in Firefox for highly-impactful bugs in comparison to other bugs. However, the Wilcoxon test was not statistically significant. In our previous work [13], we found that highly-impactful crash-types required longer fixing time. However, a crash-type is often related to more than one bug (and vice-versa), which could explain the different result obtained here. Also, we rely on machine profiles to identify highly-impactful bugs instead of user profiles [1] as in our previous work [13]. This choice was dictated by the fact that data provided to us by the Mozilla corporation did not contained references to users (because of privacy restrictions). The result of the sensitivity analysis shows that the number of developers involved in the correction of bugs (*i.e.*, the fixer number) with frequency and entropy values above the 70th percentile is statistically significantly higher than other bugs in Fennec. If we identify highly-impactful bugs using the 90th percentile, their fixing time is significantly higher than the fixing time of other bugs in Firefox. However, they are in small number (*i.e.*, 4.2% of total crash-related bugs).

All these results suggest that Mozilla quality assurance teams do not prioritise highly-impactful bugs (a lower proportion of these bugs are fixed) albeit they impact a large user base and do not seem to be more difficult to fix than other bugs. If developers could predict these highly-impactful bugs early on and fix them, they would significantly reduce their negative impact and improve the user perceived quality of the system.

RQ3: Can we predict highly-impactful bugs?

Motivation. Khomh et al. [13] have proposed an entropy-based approach (described in Section III) that can be used to identify highly-impactful bugs. However, this approach requires a certain period of time to assess which bug occurs frequently with high entropy. Table IV shows that the average triaging time for highly-impactful bugs in Firefox and Fennec is respectively 52.2 days and 36.2 days. During those periods, the users of the systems are impacted by crashes that can lead to data loss and–or frustration. In this research question, we investigate strategies to predict highly-impactful bugs. Specifically, our goal is to determine whether a bug is highly-impactful at the moment it is reported (*i.e.*, once the bug report is filed). Such a prediction can be applied by software organisations to identify highly-impactful bugs at an early stage of development, *e.g.*, during alpha or beta-testing phases. This approach may allow developers to focus on highly-impactful bugs earlier and improve the overall quality of the software more efficiently.

Approach. We mine bug reports and crash reports and compute the metrics described in Table V. We select these metrics because they have been successfully used in bug prediction studies (*e.g.*, Shihab et al. [18] used *Week day*, *Month day*, and *Day of year* to predict re-opened bugs) and they are available once a bug is submitted. We choose several regression and classification algorithms in R to build predictive models: General Linear Model (GLM), C5.0, ctree, randomForest, and cforest. GLM is an extension of linear multiple regression for

a single dependent variable. It is extensively used in regression analysis. Decision tree is a widely used classification approach to predict a binary result. C5.0 and ctree are two different implementations of decision tree. They are respectively from the R packages "C50" and "party". Based on decision tree, Leo Breiman and Adele Culter developed Random Forest, which uses a majority voting of decision trees to generate classification (predicting, often binary, class label) or regression (predicting numerical values) results [5]. In our configuration, we build 50 trees with five randomly selected attributes in each tree. We use two implementations of Random Forest in this paper: randomForest and cforest, which are respectively from the R packages "randomForest" and "party".

Before building our models, we use the Variance Inflation Factor (VIF) analysis to remove correlated variables. We set the threshold to 5. Variables with VIF result over the threshold are considered as correlated. In Table V, * stands for the eliminated metrics in Firefox while † stands for the eliminated metrics in Fennec.

We cluster the extracted bugs of the different releases; grouping bugs from minor releases into their main release (*e.g.*, bugs from releases 10.0.1 and 10.0.2 are grouped with those of the major release 10). We intended to use consecutive releases to test the performance of our prediction models but observed that a high proportion of bugs is transferred across the releases. In some releases, the "transferred bugs" account for more than 80% of all bugs. This high bug transfer rate is due to the fact that Mozilla follows a rapid release cycle since 2011 [12]. With short release cycles, many bugs are transferred from an old release to new releases before getting fixed. If only few new bugs are discovered in a new release there is no need for a prediction model since developers can manually triage the bugs. Hence, to test the performance of our proposed models, we consider releases with at least 25% of new bugs. More specifically we test our models on releases of Firefox and Fennec containing respectively 25%, 30%, and 35% of new bugs (in fact, there are merely two releases of Firefox in which new bugs account for more than 40%). For each of these thresholds, we select Firefox and Fennec releases with amounts of new bugs greater than the threshold. We use the new bugs to create a testing set. We use the bugs from the preceding release to train the models. We compute the accuracy, precision, recall and F-measure metrics for each of the classification algorithms using only the new bugs from the testing set (to avoid overfitting). We use the variable importance function (*e.g.*, varimp) in R to discover the top predictors for each of the algorithms.

Findings. Table VI shows the average prediction accuracy, precision, recall, and F-measure for the five classification algorithms in Firefox and Fennec.

cforest achieves the best recall when predicting highly-impactful bugs in both studied systems. In general, our predictive models can obtain a precision of 64.2% in Firefox and 45.4% in Fennec, as well as a recall of 71.8% in Firefox and 98.3% in Fennec.

TABLE V: Prediction metrics

Metric	Description and Rationale
Bug report metrics	
Week day	Day of week (from Mon to Sun). Bug reports created on certain week days may be overlooked for fixing; resulting into large numbers of crashes. (*e.g.*, Friday) [19], [3].
Month day	Day in month (1-31). Bug reports created on certain days may be overlooked for fixing; resulting into large numbers of crashes. (*e.g.*, some dates before holidays).
Month	Month of year (1-12). Bug reports created in some seasons may be overlooked for fixing; resulting into large numbers of crashes. (*e.g.*, December, during Chrismas holidays)
Day of year*†	Day of year (1-366). Combined the rationales of month day and month.
Description Size	Number of words in a bug description. A too short message (due to hasty work) or too long message (due to the difficulty) may lead to fixing failure and late resolution of the bug, which may lead to large numbers of crashes.
Component	Component where a bug is located. Bugs occurring in complex or central (*i.e.*, highly coupled with other parts of the system) components may be difficult to resolved, which may lead to large numbers of crashes.
Reporter experience	Number of bugs filed by the reporter of a bug in the past. A reporter who filed a high number of bugs is likely to gain recognition for the relevance of her bug reports. Quality managers may decide to pay more or less attention to her reported bugs, which may result into large or low numbers of crashes.
Crash report metrics	
Uptime	Median uptime of crashes related to a bug. The uptime of a crash is the duration (in seconds) of the period during which Firefox or Fennec was running on a user's machine before the occurrence of the crash. Bugs related to low uptime values may cause large numbers of crashes (a user may restart its system and–or the software multiple times in hope that a rejuvenation will suppress the crash).
Pre-opening daily crashes	Average daily crash occurrences for a bug before the bug report is filed (data extracted from crash reports during the period of February 2010 to December 2011). High pre-opening crash occurrences may imply high post-opening crash occurrences.
Pre-opening daily impacted users†	Average daily number of machine profiles impacted by a bug before the bug report is filed (calculated from the same dataset as pre-opening daily crashes). High pre-opening daily rate of impacted machine profiles is likely to translate into high post-opening rate of impacted machine profiles.
Code complexity metrics	
LOC	Lines of code of the bug-related class. A large class may be hard to maintain and prone to crashes.
Number of functions*†	Number of functions in the bug-related class. Same rationale as LOC.
Cyclomatic complexity	Average cyclomatic complexity of the functions in the bug-related class. Complex code is hard to maintain and prone to crashes.
Max nesting†	Maximum level of nested functions. A high level of nesting increases the conditional complexity and is likely to increase the crashing probability.
Comment ratio	Ratio of the number of comments to the total lines of code in the bug-related class. A code with few comments may not be easy to understand, and may consequently lead to large numbers of crashes.
Social networking analysis metrics **(other selected metrics share the same rationale as PageRank)**	
PageRank*†	Time fraction spent to "visit" the bug-related class in a random walk in the call graph. If an SNA metric of a class is high, a bug in that class may be triggered through multiple paths and the bug is likely to appear frequently, because multiple paths lead to that class.
Betweenness	In the call graph, number of classes passing through the bug-related class among all shortest paths.
Closeness	Sum of lengths of the shortest call paths between the bug-related class and all other classes.
Indegree	Numbers of callers of the bug-related class.
Outdegree	Numbers of callees of the bug-related class.

TABLE VI: Accuracy, precision, recall and F-measure (in %) obtained from GLM, C5.0, ctree, randomForest, and cforest to predict highly-impactful bugs (the proportion of new bugs is $> 35\%$ (testing set))

System	Metric	GLM	C5.0	ctree	randomForest	cforest
Firefox	Accuracy	40.2	64.2	**66.2**	64.9	63.1
	precision	31.7	60.3	**64.2**	61.7	58.8
	recall	23.2	70.6	64.0	66.9	**71.8**
	F-measure	26.8	**65.0**	64.1	64.2	64.7
Fennec	Accuracy	**52.2**	45.2	46.2	47.3	45.4
	precision	32.6	43.8	45.1	**45.5**	44.9
	recall	6.6	80.1	94.9	91.7	**98.3**
	F-measure	10.9	56.6	61.2	60.9	**61.6**

TABLE VII: Number of training/testing pairs, precision and recall (in %) of cforest with different proportions of new bugs (testing sets)

New bugs	Firefox			Fennec		
	# pairs	Precision	Recall	# pairs	Precision	Recall
25%	66	49.4	84.5	21	42.3	97.2
30%	34	49.2	83.1	17	43.2	97.4
35%	12	58.8	71.8	12	44.9	98.3

Table VII shows how the precision, recall, and F-measure of the cforest algorithm varies with the size of the testing set (*i.e.*, the amount of new bugs). Precision and recall increase with the size of the testing set in Fennec; in Firefox, the precision increases while the recall decreases.

The best results are obtained when the proportion of new bugs represents more than 35% of all bugs. We computed the top predictors for the different models and found that ***pre-opening daily impacted user number*** **is the most important predictor for four algorithms in Firefox; meaning that the pre-opening (*i.e.*, before the bug report is opened) impact of a bug on users is a good indicator of its future impact on users (*i.e.*, after bug opening)**, which is an expected result. In Fennec, *component of bug*, *bug opened month*, and *median crashing uptime* are the top predictors for at lease two algorithms. The best predictor is not obviously identified for this system. One explanation may be the small size of Fennec.

RQ4: Which benefits can be achieved by predicting highly-impactful bugs?

Motivation. Results from **RQ3** show that highly-impactful bugs can be predicted early on before a new release, with a recall of 71.8% in Mozilla and 98.3% in Fennec. Therefore, rather than waiting for a large number of crashes to occur, developers can identify and address highly-impactful bugs without delay. To quantify the benefits that can be achieved by predicting highly-impactful bugs, we simulate the application of our proposed cforest model (the best predictive model of the case study presented in RQ3) to the 12 pairs of releases that contain at least 35% of new bugs, and assess the amount of crash occurrences and unique machine profiles that can be avoided.

Approach. In the studied training releases, we compute the median fixing time (*i.e.*, the period between the bug opening date and the last modification date) of all resolved bugs with the priority "P1". Those bugs are assigned the highest priority and are expected to be fixed earlier than other bugs. We refer to it as $duration_{med}$. For the bugs in the studied testing releases, we apply our cforest model to predict their categories (*i.e.*, highly, skewed, moderately, or isolated). We use Equation (3) to compute the simulated fixed date of a bug:

$$D_{pf} = D_o + Duration_{med} \quad (3)$$

Where D_{pf} stands for the date at which a highly-impactful bug, which is successfully predicted or transferred from a training release, would be potentially fixed; D_o stands for the opened date of the bug; and $Duration_{med}$ stands for the median fixing duration of fixed bugs that were assigned with the highest priority (*i.e.*, P1). We consider all the crashes (related to the predicted or transferred highly-impactful bug) that occurred after the simulated fixing date, as crashes that can be avoided, if developers fix the bug without delay. We identify machine profiles impacted by these crashes by applying the heuristic described in Section III.

To calculate the proportion of crash occurrences that can be reduced, we sum the crashes that can be avoided for all successfully predicted and transferred highly-impactful bugs in each testing release, then divide this number by the total number of crashes in the release. To calculate the proportion of unique machine profiles that can be reduced, for every testing release, we subtract each successfully predicted or transferred bug's related machine profiles that are impacted before the simulated fixing date from the total unique machine profiles impacted by this bug. We divide this number by the total number of machine profiles to obtain the percentage of reduced machine profiles for the bug. Next, we compute the average percentage of reduced machine profiles for all bugs in the testing release. Finally, we compute the average percentage of crash occurrences and unique machine profiles that can be reduced for Firefox and Fennec releases respectively.

Since developers' time and resource is limited, we also compute the amount of time that developers would spend fixing false positives (*i.e.*, wrongly predicted highly-impactful bugs). We divide the result by the total time spent on bug fixing activities to calculate the percentage of time lost to false positives.

Finding. **A considerable amount of crash occurrences can be avoided by our "early triaging technique". On average, the number of crash reports can be reduced by 23.0% in Firefox and by 13.4% in Fennec. The number of unique machine profiles that are impacted by crashes can be reduced by 28.6% in Firefox and by 49.4% in Fennec.** In addition, false positive highly-impactful bugs would consume on average 6.3% of the total bug fixing time in Firefox (respectively 29.6% in Fennec). We manually investigated these false positive highly-impactful bugs and found that 96.4% of these bugs in Firefox (respectively 95.5% in Fennec) are assigned a severity level of "blocker" or "critical". Also,

51.2% of them eventually get fixed in Firefox (respectively 41.8% in Fennec). This suggests that even though these false positives are not highly-impactful bugs in the sense that they do not have both high entropy and high frequency, they are nonetheless important and should be fixed in priority. Therefore, the amount of time spent on these bugs is not completely wasted.

In conclusion, these results show that triaging and predicting highly-impactful bugs early (before a new release of a software system) can help reduce a large amount of crashes experienced by users, which could improve the overall quality of the software system in a more cost-effective manner.

VI. Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [24].

Construct validity threats concern the relation between theory and observation. In this study, the construct validity threats are mainly due to measurement errors. We parse crash and bug reports from copies of Socorro and Bugzilla databases obtained from the Mozilla corporation. Khomh et al. [13] found in a previous study that highly-impactful crash-types require longer fixing time. However, our result in this study show that the fixing time of highly-impactful bugs in Firefox is slightly lower in comparison to other bugs. We attribute this difference to the fact that a crash-type is often related to more than one bug (and vice-versa). Also, in this study, we rely on machine profiles to identify highly-impactful bugs instead of users' profiles inferred from installation time as in our previous work [13]. A choice dictated by the data provided to us by the Mozilla corporation, which do not contain references to users. In **RQ4**, we estimate the date at which a successfully predicted or transferred highly-impactful bug is fixed by adding the median fixing duration of fixed bugs that are assigned the highest priority in our training data set to the bug opening date. This estimation may not be accurate. However, our goal in **RQ4** is only to provide a simulation of the proportion of crash occurrences that can be avoided.

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. In **RQ3**, we imposed a minimum size to our testing sets, *i.e.*, we considered releases with at least 25% of new bugs. However, to avoid biasing our results with this threshold, we performed additional evaluations of our proposed models using respectively 25%, 30%, and 35% of new bugs. In Bugzilla, all time stamps are reported in UTC timezone. Therefore, reported *week day*, *month day*, and *month* might not precisely reflect developers' local time. However, from all these metrics, only *month* contributed significantly to the models (see the results about top predictors). This metric (*i.e.*, *month*) is less likely to be biased by time zone conversions.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. To determine the cut-off of bugs with high crash entropy and high crash frequency, we conducted a sensitivity analysis. We applied different thresholds of 50%, 70%, and 90% of percentiles to verify the characteristics of highly-impactful bugs. Results show that different percentiles do not affect the conclusion. For any detail about the sensitivity analysis, please check our data at: https://github.com/swatlab/highly-impactful. We used non-parametric tests that do not require making assumptions about the data set distribution. In **RQ2**, we did not investigate the characteristics of the four categories of bugs with respect to priority and severity assigned in Bugzilla because, in our previous work [13], we found that the priority and severity labels in Mozilla's bug reports do not reflect the concrete levels of attention paid by developers when fixing the bugs.

External validity threats concern the possibility to generalise our results. Although we only conduct our case study with two Mozilla subsystems, because only the Mozilla Foundation has opened their crash collecting database to the public [22] to date, most of our findings are consistent with previous studies [13], [25]. We share our data and scripts at: https://github.com/swatlab/highly-impactful. Further studies with different systems are required to verify our results and make our findings more generic.

VII. Related Work

In this section, we introduce some related literature on bug triaging and prediction models.

A. Bug triaging

In previous studies, researchers proposed different defect triaging techniques to help software organisations improve their triaging activities. Anvik et al. [4] introduced a semi-automated approach to ease the assignment of reports to a developer. They applied a supervised machine learning algorithm to learn the kinds of reports resolved by each developer in the bug repository, then suggested a small number of suitable developers to resolve each new bug. Canfora and Cerulo [6] also proposed a semi-automatic approach to select the best candidate set of developers to resolve new change requests. This approach identifies the candidate developers using the textual description of the change requests. Menzies and Marcus [15] proposed an automated approach, SEVERIS, to help triage teams assign severity levels to bug reports. Their approach is based on standard text mining and machine learning techniques applied to existing sets of bug reports. Weiss et al. [23] proposed an approach to help triage teams automatically predict the fixing effort (*i.e.*, bug fixing time). This approach allows for early effort estimation, to help triage teams better assign issues. Jeong et al. [11] studied bug tossing (*i.e.*, reassignment of bug reports) and found that tossing bugs lead to longer bug fixing time. They proposed a tossing graph model, which captures past tossing history, to reduce tossing steps and improve the accuracy of their automatic bug assignment approach. Khomh et al. [13] proposed an entropy based technique to triage crash-types in Firefox. Their proposed approach achieves a better classification of crash-types than the current technique applied by Firefox teams. In

this research, we apply the entropy based approach by Khomh et al. [13] to triage bug reports, since we can map crash-types to their corresponding bugs. Our approach can identify bugs that affect a large user base with high crashing frequency.

B. Prediction models

In previous studies, researchers found that some factors may link to software defects or failures. Hassan et al. [8] created decision trees to predict ahead of time the certification result of a build for a large software project in IBM Toronto Lab. Zimmermann et al. [25] used Logistic Regression model to predict bug re-opening in Windows. Shihab et al. [18] compared 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, yields the best prediction results. In our previous work [2], we used GLM, C5.0 (the improved version of C4.5), ctree, randomForest, and cforest to predict bug re-opening in supplementary bug fixes. We found that randomForest outperforms C5.0 and other algorithms. In this study, cforest, another implementation of Random Forest, achieves the best prediction results.

VIII. Conclusion

Bug triaging guides software practitioners to focus their effort to address bugs with high priority when resources are limited. Current bug triaging approaches only take bugs' crash frequency into account while ignoring the impact of bugs on end users. Although previous studies used entropy analysis to improve the current bug triaging approaches, these approaches were applied only after end users have already suffered crashes for a certain period of time. In this paper, after examining the prevalence and characteristics of highly-impactful bugs, *i.e.*, bugs with high crashing frequency and entropy, in Mozilla Firefox and Fennec, we built predictive models to help software organisations predict them early before they impact a large population of users. Our proposed models can predict highly-impactful bugs with a precision up to 64.2% (in Firefox) and a recall up to 98.3% (in Fennec). Using a simulation to evaluate the benefit of our best predictive model, cforest, we found that, on average, our early prediction technique can effectively prevent 23.0% of crash occurrences in Firefox (respectively 13.4% in Fennec) and reduce 28.6% of unique machine profiles that are impacted in Firefox (respectively 49.4% in Fennec). Software organisations could use our suggested predictive models to identify highly-impactful bugs and improve the satisfaction of their users. In the future, we plan to implement our approach in a tool and validate our results on different software systems. We also appeal to other software organisations to share their crash report databases with the public to help generalise the results of our study.

References

[1] L. An and F. Khomh. Challenges and issues of mining crash reports. In *Proceedings of the 1st International Workshop on Software Analytics (SWAN)*, Montreal, QC, Canada, March 2015.

[2] L. An, F. Khomh, and B. Adams. Supplementary bug fixes vs. re-opened bugs. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Victoria, BC, Canada, September 2014.

[3] 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.

[4] 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.

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

[6] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1767–1772, New York, NY, USA, 2006. ACM.

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

[8] 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.

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

[10] igraph. http://igraph.org/redirect.html, 2014. Online; accessed October 13th, 2014.

[11] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 111–120, New York, NY, USA, 2009. ACM.

[12] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou. Understanding the impact of rapid releases on software quality. *Empirical Software Engineering*, pages 1–38, 2014.

[13] 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.

[14] 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.

[15] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 346 –355, 28 2008-oct. 4 2008.

[16] Understand tool. https://scitools.com, 2014. Online; accessed October 13th, 2014.

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

[18] 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.

[19] 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.

[20] SLOCCount. http://www.dwheeler.com/sloccount/, 2014. Online; accessed October 13th, 2014.

[21] Socorro: Mozilla's crash reporting system. https://crash-stats.mozilla.com/home/products/Firefox, 2014. Online; accessed October 13th, 2014.

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

[23] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

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

[25] 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.