An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox

Foutse Khomh[1], Brian Chan[1], Ying Zou[1], Ahmed E. Hassan[2]

[1] Dept. of Elec. and Comp. Engineering, Queen's University, Kingston, Ontario, Canada
[2] School of Computing, Queen's University, Kingston, Ontario, Canada

E-mail: {foutse.khomh, 2byc, ying.zou}@queensu.ca, ahmed@cs.queensu.ca

***Abstract*—A crash is an unexpected termination of an application during normal execution. Crash reports record stack traces and run-time information once a crash occurs. A group of similar crash reports represents a crash-type. The triaging of crash-types is critical to shorten the development and maintenance process. Crash triaging process decides the priority of crash-types to be fixed. The decision typically depends on many factors, such as the impact of the crash-type, (*i.e.*, its severity), the frequency of occurring, and the effort required to implement a fix for the crash-type. In this paper, we propose the use of entropy region graphs to triage crash-types. An entropy region graph captures the distribution of the occurrences of crash-types among the users of a system. We conduct an empirical study on crash reports and bugs, collected from 10 beta releases of Firefox 4. We show that our proposed triaging technique enables a better classification of crash-types than the current triaging used by Firefox teams. Developers and managers could use such a technique to prioritize crash-types during triage, to estimate developer workloads, and to decide which crash-types patches should be included in a next release.**

***Keywords*-Crash, bug, triaging, entropy region graphs.**

I. Introduction

Software testing is the most widely used approach to detect bugs in software systems. It plays a central role in ensuring the quality and the success of a system. Nowadays, it is common to have built-in automatic crash reporting tools in software systems to collect crash reports directly from an end user's machine. The Windows OS, Internet Explorer, and Mozilla Firefox are few names that make use of automatic collection of field crash reports. For example, whenever Firefox closes unexpectedly, Mozilla Cash Reporter collects information about the event and sends a detailed crash report to the Socorro crash report server. The collected crash reports include a stack trace of the failing thread and other information about the user's environment to help developers replicate and fix the crash. A group of similar crash reports represents a crash-type. However, the built-in automatic crash reporting tools often collect a huge number of crash reports. For example, Firefox receives 2.5 million crash reports every day [1]. Triaging the collected crash-types is essential to allow developers and maintainers to focus their efforts more efficiently. During the triaging of crash-types, decisions are made about which crash-types should be fixed and when. These decisions typically depend on many factors, such as the impact of the crash-type (*i.e.*, its severity), the crash-type frequency, the effort required to implement a fix for the crash-type, and the risk of attempting to fix the crash-type. Currently, Firefox developers triage crash-types based on the daily frequency of the occurrence of a crash-type. For the top crash-types (*i.e.*, the crash-type with the maximum number of crash reports), Firefox developers file bugs in Bugzilla and link them to the corresponding crash-type in the Socorro server. Multiple bugs can be filed for a single crash-type and multiple crash-types can be associated with the same bug. The severity and priority of bugs are assigned manually by Mozilla triage teams, and are often modified later during the fixing process. Because bug classification depends on the personal judgment of triage team members, it is a subjective process and often results in resources being spent on non-essential issues [2]. Moreover, little estimation is provided to developers about the effort required to fix the crash-types and the risk of their attempts to fix the crash-types.

In this paper, we propose the use of crash entropy values to prioritize crash-types to fix during the triaging. More specifically, we propose the use of both entropy and frequency information during the triaging of a crash-type. The entropy of a crash-type quantifies the distribution of the occurrence of the crash-type to the users of the system. A high entropy value for a crash-type means that most users encountered that crash-type. The priority of such a crash-type should be raised by developers and quality managers. Currently, most triage teams sort crash-types based on frequency values, but do not consider the distribution of the crash-types among users. For example Firefox triage teams give an equal importance to the *Taskbar tab preview* crash-type (*i.e.*, "$mozilla :: widget :: WindowHook :: Lookup(unsignedint)$") and the *hang* crash-type (*i.e.*, "$hang|mozilla :: plugins :: PPluginInstanceParent :: CallNPP_Destroy(short*)$") which have frequency values of 17,485 and 17,417 respectively. Yet the impact of the two crash-types on the users is very different since the *Taskbar tab preview* crash-type affected only 7% of users while 21% of users experienced the *hang* crash-type. A good triaging should assign different levels of importance to the two crash-types. The use of frequency alone is not enough to show the full impact of a crash-type on the users of a system. Moreover, although Firefox triage teams assigned the same level of priority to the two crash-types, it took them 10,464 hours to fix the *Taskbar tab preview* crash-type compared to only 1,152 hours for the *hang* crash-type. This large difference in fixing time could be explained by the fact that because few users experienced the *Taskbar tab preview* crash-type; only limited information was available for developers to replicate and test the correction. We believe that the information on the entropy of the two crash-types would have enabled a better triaging of the crash-types and a better assessment of the efforts needed to fix the crash-types.

In this paper, we propose a triaging technique based on the frequency and the entropy of crash-types. We conduct an empirical study on crash reports and bugs, collected from 10 beta releases of Firefox 4, and show that the new proposed triaging technique enables a better classification of crash-types than the current technique used by Firefox teams.

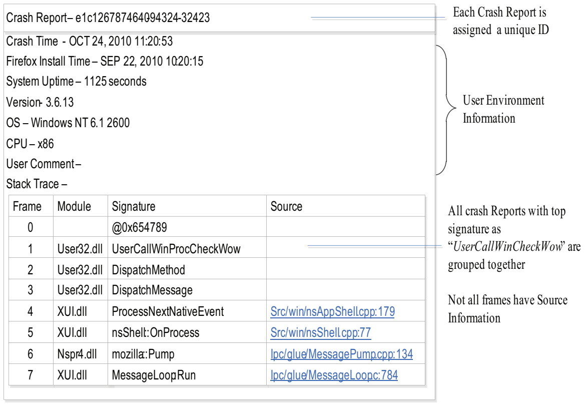
The rest of the paper is organized as follows. Section II describes the Mozilla crash triaging system, introduces the concept of crash-type entropy, and presents the proposed entropy based crash-type triaging approach. Section III describes the design of our case study and reports its results. Section IV discusses threats to the validity of our study. Section V discusses the related literature on triaging and entropy based analysis. Finally, Section VI concludes the paper and outlines future work.

II. Crash-type and Entropy

A. Mozilla Crash Triaging System

Firefox is delivered with a built-in crash reporting tool: Mozilla Crash Reporter. Whenever Firefox closes unexpectedly, Mozilla Crash Reporter collects information about the event and sends a detailed crash report to the Socorro crash report server. The crash reports include a stack trace of the failing thread and other information about the user's environment. A stack trace is an ordered set of frames. Each frame refers to a method signature and provides a link to the corresponding source code. Source code information is not always available in the frames; especially when a frame belongs to a third party binary. Figure 1 illustrates a sample crash report for Firefox.

Crash reports are sent to the Socorro crash report server [1]. The Socorro server assigns a unique id to each received report and groups the similar crash reports together. A group of similar crash reports is termed as a crash-type. The crash reports are grouped based on the top method signature of the stack trace. However, subsequent frames in the stack traces can vary for different crash reports in a crash-type. Figure 2 illustrates a sample crash-type. For each crash-type, the Socorro server provides a crash-type summary, a list of crash reports grouped under the crash-type and a set of bugs filed for the crash-type. Figure 2 shows the structure of the crash-type "UserCallWinProCheckWow" on the Socorro server.



Crash Report– e1c126787464094324-32423

Crash Time - OCT 24, 2010 11:20:53

Firefox Install Time – SEP 22, 2010 10:20:15

System Uptime – 1125 seconds

Version- 3.6.13

OS – Windows NT 6.1 2600

CPU – x86

User Comment –

Stack Trace –

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

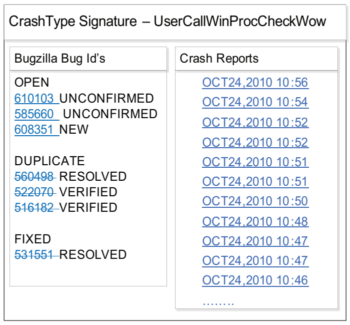
Each Crash Report is assigned a unique ID

User Environment Information

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

Not all frames have Source Information

Figure 1. A sample crash report for the crash-type "UserCallWinProCheck-Wow"



CrashType Signature – UserCallWinProcCheckWow

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

Figure 2. A sample crash-type summary

The Socorro server provides a rich web interface for developers to analyze crash-types. Developers triage the crash-types by prioritizing the top crash-types (*i.e.*, crash-type with the maximum number of crash reports) to analyze and fix the bugs responsible for the crash.

Mozilla uses Bugzilla for tracking bugs and maintains a bug report for each filed bug. For the most frequent crash-types (*i.e.*, crash-type with the maximum numbers of crash reports), Firefox developers file new bugs in Bugzilla and link them to the corresponding crash-type in the Socorro server. Multiple bugs are sometimes linked to a single crash-type. Multiple crash-types can also link to the same bug. The Socorro server and Bugzilla are integrated, developers can directly navigate to the linked bugs from a crash-type summary in the Socorro server. Figure 3 presents a bird view of the Firefox crash triaging system.

Mozilla quality assurance teams triage bug reports and assign severity levels to the bugs [3]. When a developer fixes a bug, he or she often submits a patch to Bugzilla. A patch includes source code changes, test code and other configuration file changes. Once approved, the patch code is

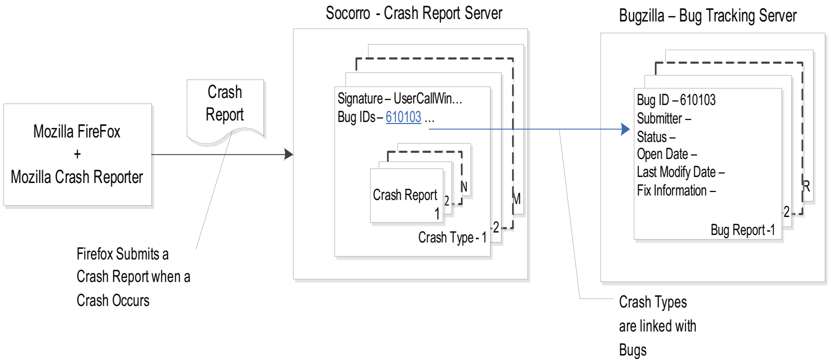


Figure 3. Mozilla crash triaging system

integrated into the source code of Firefox.

B. *Entropy of a Crash-type*

In this study we apply the normalized Shannon's entropy measure [4] to crash-types. We aim at capturing the distribution of a crash-type among the users of a system. We compute the entropy of a crash-type following Equation (1):

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

Where CT is a crash-type; p_i is the probability of a specific user i reporting CT ($P_i \geq 0$, and $\sum_{i=1}^{i=n} p_i = 1$); and n is the total number of unique users of the system.

For a crash-type CT where all the users have the same probability of reporting CT (*i.e.*, $p_i = \frac{1}{n}, \forall i \in 1, 2, \ldots, n$), the entropy is maximal (*i.e.*, 1). On the other hand, if a crash-type CT is reported by only one user i, the entropy of CT is minimal (*i.e.*, 0). Crash-types with high entropy values are reported by more users. Therefore, such crash-types are likely to be easier to replicate than crash-types with low entropy values. When the entropy of a crash-type is low, it indicates that there is a gross propensity for a certain subset of users to report the crash-type, while other users rarely do so. Crash-types that have low entropy values may indicate that the anomaly is on the specific users side and not from the software system. Entropy values of crash-types could help developers and quality assurance teams identify problems with higher negative impact on users. The entropy of crash-types can also help developers identify crash-types that need better coverage, *i.e.*, crash-types for which the recruitment of more users with a specific profile is needed to help developers replicate and fix their associated bug.

C. *Entropy Analysis*

We propose an entropy graph that can be used to triage crash-types. The entropy of a crash-type shows the distribution of the occurrences of crash-type among a set of users. However, the entropy of a crash-type solely doesn't capture the magnitude of the occurrence of a crash-type. We propose to combine the entropy and the frequency values to better capture the overall effect of a crash-type on the users of a system. For example, crash-type CT_1 was reported by 3 users: A (40 crash reports), B (100 crash reports), and C (60 crash reports) out of a group of 100 users. Crash-type CT_1 has 200 reports in total. A crash-type CT_2 was reported by all the 100 users, with 2 reports each. Similarly, crash-type CT_2 has 200 reports as well. We define the probability distribution of a crash-type reported by users as the probability that a user i reports the crash-type. For each user, we count the total number of crash reports of a crash-type reported by the user and divide it by the total number of crash reports of the crash-type that are reported by all the users. Hence, for our example, the probability of $UserA$ is $p(UserA) = \frac{40}{200} = 0.2$. Similarly, the probabilities of $UserB$ and $UserC$ are $p(UserB) = \frac{100}{200} = 0.5$, and $p(UserC) = \frac{60}{200} = 0.3$, respectively. The normalized Shannon Entropy of CT_1 is 0.22. Since all the 100 users reported an equal number of reports for CT_2, the Shannon Entropy of CT_2 is 1. Although the frequencies of CT_1 and CT_2 are equal, their entropy values are very different. We propose to categorize crash-types of systems by the level of their entropy values (*i.e.*, above a certain threshold), as well as the total frequency of their occurrence. Because the number of reported crash-types can be numerous, we propose to visualize all crash-types as points according to their entropy distribution value and frequency value on a region graph as shown in Figure 4. This makes it easier to see the general disposition of crash-types among all the users of a system. More specifically, each point on the graph represents a crash-type characterized by its entropy and frequency values. The x-axis represents the normalized frequency of crash-types. We compute the normalized frequency of a crash-type by dividing its frequency by the frequency of the most reported crash-type. The y-axis represents the entropy distribution of the crash-types. The maximum value on both axes is 1. As the frequency and entropy values increase across both axes, the probability that the crash-type covers a larger population of users increases.

Table I
SUMMARY OF ENTROPY REGIONS

Region	Entropy	Frequency	Priority	Description
Highly Distributed	High	High	High	The crash-type is reported with high frequency and is well distributed among users, indicating that more users can report it consistently. More information on failing stack traces are available and developers will be able to easily replicate by selecting users from the testing base.
Skewed	Low	High	Medium	The crash-type is reported frequently, however the distribution is skewed towards certain users. The diversity of the reported stack traces is low. Developers may have to rely on certain beta users to replicate the crash.
Moderately Distributed	High	Low	Low	The crash-type is only reported by a selected number of users but they report it evenly. This indicates a specific user cluster that developer has to recruit from in order to replicate the crash.
Isolated	Low	Low	Very Low	This configuration is the least desirable from developers point of view. The crash-type is obscure and hard to replicate on any system. Very few information are provided on failing stack traces. Developers will have the hardest time resolving this crash.

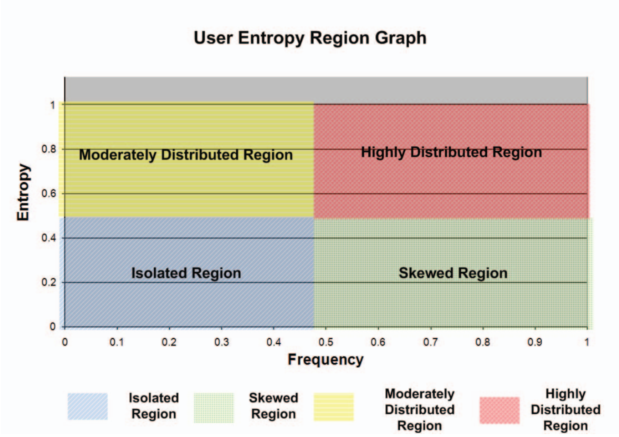


Figure 4. Entropy distribution graph regions

D. *Crash-type Triaging*

Having all the points on the graph can tell if a system is prone to isolated users reporting infrequent crash-types or whether it contains crash-types reported by the majority of users. We propose to divide the entropy graph into regions with different priorities as illustrated by Figure 4. Table I summarizes the possible regions that a crash-type can land and their different characteristics. The boundaries of the illustrated regions varies based on the context and maturity of systems. Developers and quality managers can make use of historical data from the field testing of previous versions of their systems to compute optimal boundaries. In this work, we use the median of frequencies and entropy values to identify the boundaries illustrated in Figure 4.

The triaging of crash-types is a process that typically depends on factors, such as the severity of crash-types, the frequency of crash-types and the effort required to fix the crash-type. Crash-types are sometimes purposely left unfixed for some of the following reasons: the crash-type occurs rarely and affects only few users and the effort required to fix the crash-type is large and expensive. Often, some crash-types are left unfixed because of the risk in attempting to fix them. Sliwerski *et al.* [5] observed that code changes that fix bugs in systems are up to two times more likely to introduce new bugs than other kinds of changes. An effective triaging of crash-types should allow developers to focus their time and effort on crashes that they are actually able to fix.

Taking into account the aforementioned triaging criteria, we make the following recommendations for the triage of crash-types using an entropy graph:

- **Highly Distributed Region:** crash-types with high frequency and entropy values (*i.e.*, values above the median threshold) should be given a “high” priority.
- **Skewed Region:** a crash-type with a high frequency value but a low entropy, should be given a “medium” priority since it means that the crash-type only seriously affects a small proportion of users and is more likely to be specific to the user’s systems.
- **Moderately Distributed Region:** conversely, a crash-type with a high entropy value but low frequency means that it is well distributed among the users that report it, but does not occur very often to the majority of users and therefore should be given a “low” priority.
- **Isolated Region:** crash-types with low frequency and low entropy values should be given a “very low” priority since they are very rare and affect only a small number of users.

If additional information on the user perception of a crash-type is available, the priority of crash-types with low entropy values that are considered “critical”, “major”, or “blocker” should be raised to “high”. Developers should start fixing them as early as possible since they are likely to be hard to fix. This is due to the limited information provided by their failing threads and the difficulty of their replication.

Entropy graphs could also be used to assess the reliability of a population of testers. For example if a system has most of its crash-types in the skew region, the population of testers in general can be considered fairly reliable, since not every tester reports the same issues. If the majority of crash-types is in the Isolated region, the testing population as a whole would be fairly useless. There is a very limited number of bugs reported by testers and it may not be critical to investigate it.

III. Case Study

The *goal* of this case study is to assess the usefulness of entropy region graph in triaging crash-types. The *motivation* is the improvement and the automation of existing approaches for prioritizing crash-types. Developers and quality assurance teams could make use of entropy region graphs to automate the prioritization of their systems crash-types and reduce triaging time and effort. Quality managers could also use entropy region graphs to better plan testing activities of their systems and allocate support resources to reduce servicing costs.

Table II
Descriptive statistics of our data set

version	release date	number of crash reports	median uptime (sec)	average crash/day
4.0b1	6-Jul-10	608,912	388	28,996
4.0b2	27-Jul-10	350,387	321	23,359
4.0b3	11-Aug-10	348,531	373	26,810
4.0b4	24-Aug-10	530,624	427	37,902
4.0b5	7-Sep-10	455,091	52	65,013
4.0b6	14-Sep-10	1,698,763	535	29,803
4.0b7	10-Nov-10	1,848,378	686	44,009
4.0b8	22-Dec-10	871,047	644	37,872
4.0b9	14-Jan-11	608,949	572	55,359
4.0b10	25-Jan-11	329,195	776	47,028

The *context* of this study consists of data derived from the field testing of ten beta releases of Firefox, ranging from Firefox-4.0b1 to Firefox-4.0b10. Firefox is a free and open source web browser from the Mozilla Application Suite, managed by Mozilla Corporation. Since March 2011, Firefox has become the second most widely used browser in the world, with approximately 30% of usage share of web browsers [6]. Firefox runs on various operating systems including Microsoft Windows, GNU/Linux, Mac OS X, FreeBSD, and many other platforms. For each beta release, we downloaded all the reported crash-types and their associated bug-reports from the Socorro server and Bugzilla. Table II reports the descriptive statistics of our data set. The uptime in Table II is the duration in seconds for which Firefox was running before it crashed.

In the following subsections, we describe the data collection for our study. We present our research questions, and describe our analysis method. Then we present and discuss the results of our study.

A. Data Collection

When Firefox crashes on a user's machine, the Mozilla Crash Reporter collects information about the event and sends a detailed crash report to the Socorro server [1].

A crash report includes the stack trace of the failing thread and other information about a user environment, such as operating system, Firefox version, install time, and a list of plug-ins installed. The Socorro server groups crash reports based on the top method signature of a stack trace to create a crash-type. Developers file bugs in Bugzilla and link them to the corresponding crash-types in Socorro server. A bug report contains detailed information about a bug, such as the bug open date, the last modification date, the bug severity, and the bug status. We mine the Socorro server to identify users reporting crash-types. We also mine Bugzilla repository to identify bug fix information. For each crash-type and its associated bugs, we compute several metrics to assess the effort needed to fix the crash-type. In the following, we discuss the details of each of these steps.

Identification of Users. Crash reports in the Socorro server do not contain personal information to identify unique users reporting the crashes due to privacy concerns. To identify users reporting crashes, we have to use heuristics. We parse the downloaded crash reports and extract the following available information on the crash events:

- the install age (in seconds) since the installation or the last update of the user's system;
- the date at which the crash was processed on the server;
- the client crash date, *i.e.*, the time on the user's system when the crash occurred (this value can shift around with clock resets);
- the uptime (in seconds) since the user's system was launched;
- the last crash of the user.

Other user's environment information provided in crash reports includes: product, version, build, development branch, operating system name, operating system version, architecture (*e.g.*, x86) + CPU family model and stepping, user comments, addons checked, flash version, and app notes (*i.e.*, graphics card vendor id and device id).

For each crash report, we subtract the "install age" from the crash time to identify the point in time when the user reporting the crash have installed Firefox. We combine the user installation time with the information available on the user's environment and the last crash times from the crash reports, to build a vector of unique profiles; each profile representing a user.

Identifying unique users reporting crash-types is important to compute the entropy of a crash-type. We associate each unique profile with the list of crash-types for which crash reports contain information corresponding to the profile.

Metrics Extraction. For all the bugs filed for all the crash-types from our 10 beta releases of Firefox 4 on the Socorro server, we retrieve bug reports from Bugzilla. Overall, $1,329$ crash-types in our data set are linked to at least one bug. The total number of bugs is $1,733$; where 519 bugs are fixed, 253 bugs are fixed duplicated bugs, and 961 bugs are left unfixed. We parse each of the bug reports to extract information about the bug open and modified dates. We compute the duration of the fixing period for each bug fixed (*i.e.*, the difference between the bug open time and the last modification time). We compute the number of comments for each bug. The number of comments in a bug report reflects the level of discussions between developers about the bug. It has been used in previous studies [2] to measure developers effort to fix a bug. We extract additional information on severity, priority, and status of bugs that are provided by Mozilla quality teams and are available in bug reports. We use the extracted priorities to compare our proposed triaging technique to the existing prioritization approach of Mozilla quality teams. For each crash-type with associated bugs with status "FIXED" and "CLOSED", we compute the duration of the fixing period of the crash-type by differentiating the earliest bug open time of its associated bugs and the latest modification time of the associated bugs. We sum the number of comments of associated bugs to compute the effort needed to fix the crash-type. We also count the number of bugs linked to the crash-type to assess the complexity of the crash-type.

B. Research Questions

To assess the benefits of entropy region graphs for crash-types triaging, we aim at answering the following research questions:

1) **RQ1:** *Can an analysis of the distribution of crash-types entropy help classify crash-types by the level of difficulty?*
2) **RQ2:** *Do crash-types belonging to different regions of an entropy graph possess different characteristics?*
3) **RQ3:** *Do entropy graphs help improve the triaging of crash-types?*

C. Analysis Method

RQ1. We study whether an analysis of the distribution of crash-types entropy values could help classify the crash-types. This question is preliminary and aimed at providing quantitative evidence to support the intuition behind our study that the entropy of crash-types affects the difficulty to fix the crash-types. To assess the difficulty to fix a crash-type, we use the number of bugs associated to the crash-type, the duration of the fixing period of the crash-type as well as the number of comments exchanged by developers about the crash-type. These metrics have been used in previous studies on bug fixing [2].

We answer this research question in three steps: first, we investigate if crash-types with more bugs mapped to them have significantly different entropy values compared to crash-types that are associated with a single bug. We test the following null hypothesis:

H^1_{01} : *the distribution of entropy values is the same for crash-types associated with many bugs and crash-types associated with single bug.*

Second, we compare the duration of the fixing period of crash-types with high entropy values to the duration of the fixing period of crash-types that have low entropy values. We test the following null hypothesis:

H^2_{01} : *the distribution of the duration of a crash-type fixing period is the same for crash-types with high entropy values and crash-types that have low entropy values.*

We use the median to decide on high and low entropy values. Third we compare the number of comments exchanged by developers about crash-types with high entropy values to the number of comments of crash-types that have low entropy values. We test the following null hypothesis:

H^3_{01} : *the distribution of the number of comments exchanged about a crash-type is the same for crash-types with high entropy values and crash-types that have low entropy values.*

We use the Wilcoxon rank sum test [7] to accept or reject H^1_{01}, H^2_{01}, and H^3_{01}. The Wilcoxon rank sum test is a non-parametric statistical test used for assessing whether two independent distributions have equally large values. For example, we compute the Wilcoxon rank sum test to compare the distribution of the duration of the fixing period for crash-types with high entropy values and crash-types with lower entropy values.

RQ2. We want to understand to what extent the different regions of an entropy graph are able to discriminate crash-types of different characteristics. Similar to **RQ1**, we use the duration of the fixing period of a crash-type and the number of comments exchanged by developers about the crash-type to characterize the crash-type. We use the Kruskal-Wallis rank sum test to investigate if the distributions of durations of crash-types fixing period and the number of comments exchanged about crash-types are the same across the regions of the entropy graph. The Kruskal-Wallis rank sum test is a non-parametric method for testing the equality of the population medians among different groups. It is an extension of the Wilcoxon rank sum test to 3 or more groups. We therefore test the two following null hypothesis:

H^1_{02} : *the distribution of the duration of a crash-type fixing period is the same for all crash-types across the regions of the entropy graph.*

H^2_{02} : *the distribution of the number of comments exchanged about a crash-type is the same for all crash-types across the regions of the entropy graph.*

RQ3. The third research question evaluates the entropy based crash-type triaging approach presented in Section II-D. We extract severity and priority information from bug reports associated to crash-types from our data set. A preliminary analysis of bug reports revealed that the priority field is rarely used by the Mozilla quality team. Only 7% of bug reports contain a priority value. Therefore, in cases of absence of values in the priority field, we rely solely on severity values to recover the priority of crash-types. We use the following rule to estimate the priority of crash-types based on the severity levels of their associated bugs:

- We consider a crash-type to be of high priority if at least one of its associated bugs has a severity level either "critical", "major", or "blocker".

- When the highest severity level of the associated bugs is "normal", we consider the priority of the crash-type to be "medium".
- When the highest severity level of the associated bugs is "trivial", we consider the priority of the crash-type to be "very low".
- Otherwise the priority of the crash-type is considered "low".

We compute the similarity between the priority levels assigned by our entropy based crash-type triaging approach and the priority levels of crash-types obtained from bug reports following Equation (2).

$$Similarity(C) = \frac{N_T}{N} \tag{2}$$

Where, C is a set of crash-types; N_T is the number of crash-types in C for which the priority level assigned by the entropy based triaging approach is the same as the priority extracted from bug reports; and N is the total number of crash-types in C.

We use the status of bugs associated to crash-types (examples of status include FIXED, INVALID or WORKSFORME) and the durations of crash-types fixing period to further assess the benefits of our proposed triaging approach.

D. Study Results

This section reports and discusses the results of our study.

RQ1: Can an analysis of the distribution of crash-types entropy help classify crash-types by the level of difficulty? In this research question we are interested in assessing the relevance of entropy analysis of crash-types to identify the difficulty levels of crash-types. To test our first null hypothesis, we organize crash-types in two groups: the group of crash-types associated to single bug and the group of crash-types associated to multiple bugs. We compute the entropy value of each crash-type from the two groups following Equation (1). Results show that on average, the entropy value of a crash-type associated with multiple bugs is twice the entropy value of a crash-type with a single bug. We perform a Wilcoxon rank sum to verify the statistically significance of this difference and obtain a p-value $< 2.039e-08$. Therefore, we reject H_{01}^1. Crash-types linked to multiple bugs affect more users than crash-types linked to a single bug.

To test our second and third null hypothesis, we group fixed crash-types by the level of their entropy values. We use the median of entropy values as our threshold to build two groups: a group of crash-types with high entropy values and a group of crash-types with low entropy values. The entropy value of a crash-type is considered high if it is greater than the median of the entropy values of all the crash-types. Otherwise, it is considered low.

For each crash-type from the two groups, we compute the duration of the fixing period of the crash-type. We observe that in average, crash-types with high entropy values take longer to get fixed compared to crash-types with low entropy values. We perform a Wilcoxon rank sum and obtain a p-value of 0.006. Therefore, we reject H_{01}^2. We compute the number of comments exchanged about each crash-type from our two groups. In average, 55.5 comments where exchanged for crash-types with high entropy values compared to 17.65 comments for crash-types with low entropy values. We perform a Wilcoxon rank sum and obtain a p-value of 0.001. Hence, we reject H_{01}^3. Crash-types with high entropy values take longer to get fixed and more comments are exchanged during their fixing period.

Although one could have expected crash-types with low entropy values to take a longer time to get fixed and spark more discussion, because of the potential difficulty of their replication, our result shows the opposite. We explain this finding by the fact that many crash-types with low entropy values are left unfixed. In fact, 20% of crash-types with low entropy values in our data set never got fixed and in our analysis we have only considered crash-types whose underlying bugs are eventually fixed.

We conclude that an entropy analysis can help developers and quality managers identify crash-types that will be particularly difficult to fix, because they would likely be related to many bugs. A situation that will likely result in longer fixing periods and more contributions efforts from developers. Therefore, we answer positively our research question.

RQ2: Do crash-types belonging to different regions of an entropy graph possess different characteristics? To answer this research question, we compute for each fixed crash-type from our data set, the duration of its fixing period and the number of comments exchanged by developers about the crash-type. We also compute the entropy and frequency values of the crash-type and map the crash-type into a region of the entropy graph. We organize the crash-types in four groups corresponding to the four regions of the entropy graph that are illustrated on Figure 4.

We observe that crash-types from the Skewed region take the longest time to get fixed. Their average fixing time is 21,169 hours. Followed by crash-types from the Moderately Distributed region with an average of 8,475 hours. The average fixing time of crash-types from the Highly Distributed region is 4,299 hours. Crash-types from the Isolated region take in average 3,562 hours to get fixed. We perform the Kruskal-Wallis rank sum test on the durations of crash-types fixing periods from the four regions and obtain a p-value of 0.005. Therefore, we reject H_{02}^1.

The number of comments exchanged by developers about the crash-types is significantly different across the groups formed by the regions of the entropy graph. We obtain a p-value $< 2.2e-16$ for the Kruskal-Wallis rank sum test.

We observe that the average comments rate for a crash-type in the Highly Distributed region is 22 comments and the average comments rate in the Moderately Distributed region is 20 comments. The average comment rate is the highest for crash-types from the Skewed region (67 comments) and the lowest for crash-types from the Isolated region (9 comments). Hence, we reject H_{02}^{2}.

Crash-types from the Skewed region appear to be harder to fix. The duration of their fixing period is in average five time the duration of the fixing period of a crash-type from the Highly Distributed region. The average comments rate in the Skewed region is three times the average comments rate in the Highly Distributed region. This result is expected. Because of the low entropy of crash-types from the Skew region, developers are likely to have difficulties finding enough information to replicate and fix the crash. This result complements our findings from (**RQ1**) that crash-types with low entropy values require less effort to get fixed. In fact, we observe now that when a crash-type with a low entropy value has a high frequency, the effort required to fix the crash-type becomes very high. A finding that also confirms our intuition from Section II-C that a combination of entropy and frequency values provides a better assessment of the overall impact of a crash-type than either frequency or entropy solely.

When both frequency and entropy values of a crash-type are low (*i.e.*, the crash-type belongs to the Isolated region), we observe that it takes less time for developers to fix the crash-type. We explain this result by the fact that crash-types from the Isolated region are likely to be more simpler, since they occur infrequently and are encountered by few users. In some cases, as discussed in Section II-B these crash-types may be the results of some anomalies on users' side and not from the system. Moreover we observed in our data set that 19.2% of crash-types from the Isolated region are purposely left unfixed.

We conclude from above results that crash-types from the four regions of our entropy graph possess very different characteristics and require different levels of effort from developers. This answers our second research question in the positive.

RQ3: Do entropy graphs help improve the triaging of crash-types? To assess the benefits of using entropy graphs for crash-types triaging, we compute the similarity between the priority levels assigned by our entropy based crash-type triaging approach and the priority levels of crash-types obtained from bug reports following Equation (2). Table III summarizes the obtained results. Except for crash-types from the Isolated region, the priorities assigned by the entropy based triaging approach are the same as the priority levels obtained from bug reports.

Table III
SIMILARITIES BETWEEN PRIORITY LEVELS ASSIGNED BY THE ENTROPY BASED CRASH-TYPE TRIAGING APPROACH, AND PRIORITY LEVELS FROM BUG REPORTS

Region	Similarity
Highly Distributed	100%
Skewed	100%
Moderately Distributed	100%
Isolated	19%

We investigated crash-types from the Isolated region and found that although they occurred infrequently and affected only a small number of users, the Mozilla quality team assigned a "critical" severity level to 80% of bugs linked to crash-types from the Isolated region. Overall 89.3% of bugs in our data set were found with a "critical", "major", or "blocker" severity value. A very high number that hints at a potential inaccuracy in the manual triaging process of Mozilla quality teams. Moreover, we observed that 17% of crash-types from the Isolated region that are assigned a high priority by the Mozilla quality teams are left unfixed. The age of bugs linked to unfixed crash-types with high priority values ranges from 3 months to 7 years and 9 months, while the median age of a fixed bug in our data set is only 2.3 months. For the remaining 83% of crash-types with high priority values in the Isolated region, they required in average 4,620.74 hours from Mozilla developers, with a median fixing duration of 1,680.5 hours. A slower fixing process if compared to the time spent by the same developers to fix "low" priority crash-types (which is 2,353.73 hours in average, with a median of 1,104 hours). From these observations, we conclude that although Mozilla quality teams sometimes assign high priorities to crash-types from the Isolated region, they do not fix these crash-types in the same timely manner associated with other high priority crash-types in different regions. These results suggest that the crash-type triaging process currently used by Mozilla quality teams should be improved to better reflect the concrete levels of attention paid by developers when fixing crashes.

We answer positively our research question and conclude that entropy graphs provide a better triaging of crash-types than current Mozilla triage teams. We suggest that developers and quality assurance teams can use our proposed automatic entropy based triaging approach to speed up and improve their crash-types triaging.

IV. THREATS TO VALIDITY

We now discuss the threats to validity of our study following the guidelines for case study research [8].

Construct validity threats concern the relation between theory and observation. In this work, the construct validity threats are mainly due to measurement errors. We extract crash and bugs information by parsing their corresponding html (crash reports) and xml (bug reports) files. We use a heuristic based on "install age", "last crash times", configuration and architecture of crashing systems to identify the unique users of our studied versions of Mozilla Firefox. Since our study critically relies on the identification of users reporting the crash-types. Prior to this study, we have exchanged with members of the quality assurance team at the Mozilla Fondation to confirm our user identification heuristic. We also randomly sampled 10 of our identified users profiles and manually verified the consistency of information in the crash reports of their reported crash-types. In two cases, we had to merge the profiles because of high similarities in their respective crash reports. We found the 8 other profiles to be consistent and unique. More validations of our identified users set are needed to strengthen the findings of this study. Another construct validity threat concerns missing priority information in bug reports, we use a heuristic based on severity levels to estimate priorities. A severity level indicates the seriousness of a bug. Developers refer to severity levels when fixing bugs.

Threats to internal validity do not affect this study since we do not claim causation [8]. We simply report our observations and try to provide explanations to these observations.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. We used non-parametric tests that do not require making assumptions about the distribution of data sets.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. Both the Socorro crash server and Bugzilla are made publicly available. Interested researchers could obtain the same data for the same releases of Firefox.

Threats to external validity concern the possibility to generalize our results. Although this study is limited to 10 releases of Firefox, we obtain results consistent with previous findings [2] that current triaging process fail to ensure that developers' time is spent on more critical bugs that will actually get fixed. Nevertheless, further studies with different systems and different crash triaging systems are desirable to make our findings more generic.

V. Related Work

In this section, we introduce related literature on triage in software testing and discuss entropy analysis in software engineering studies.

A. Triage in Software Testing

Several researchers have developed techniques and tools to help developers and quality assurance teams improve their triaging activities. Jeong *et al.* [9] have investigated the reassignment of bug reports and propose the use of tossing graphs to support bug triaging activities. Anvik *et al.* [3] propose a semi-automated approach to assign developers to bug reports. This approach is based on a machine learning algorithm that learns the kinds of reports resolves by each developers in the past and suggests a small number of best candidate developers for each new bug. Canfora and Cerulo [10] propose a semi-automatic method that suggests the set of best candidate developers suitable to resolve new change requests. The method retrieves the candidate developers using the textual description of the change requests. Menzies and Marcus [11] propose SEVERIS, an automated method to assist triage teams in assigning severity levels to bug reports. A machine learning algorithm is used to learn the severity levels from existing sets of bugs reports. Weiss *et al.* [12] introduce an approach to help triage teams automatically predict the durations of bug fixing period. This enables them to perform early effort estimations and to better assign the issues. Different from aforementioned approaches, which have focused on bug triage, our study analyzes crash triage and proposes a new triaging approach for crash-types. Since crash-types are linked to bugs, triage teams could combine the results of our entropy based crash-types triaging approach with previous bug triaging techniques to assign developers to crash-types, or to make a decision on when to fix a crash-type (*e.g.*, a crash-type that would take too long to get fixed may be purposely left unfixed until a future release).

B. Entropy Analysis in Software Engineering Studies

Entropy measures are extensively used in software engineering studies. Hassan *et al.* [13] in their investigation of the complexity of software development processes use the Shannon entropy to measure the complexity of systems and conclude that systems with higher entropy rates are more complex and decay overtime. Similarly, Zaman *et al.* [14] in their comparative study of security and performance bugs apply the same normalized Shannon entropy on bug fixing patches to assess the complexity of bug fixes. Bianchi *et al.* [15] propose the use of entropy metrics to monitor the degradation of software systems. They develop a tool based on software representation models to automatically compute entropy metrics before and after every maintenance intervention. Hafiz *et al.* [16] treat a software system as an information source and used Shannon, Hartley and Renyi entropy measures to extract different types of information from the system. They remarked that files that are more functional and descriptive provide a larger amount of entropy. Kim *et al.* [17] propose new software complexity metrics (*i.e.*, class complexity and inter-object complexity) for object oriented software systems based on the traditional Shannon entropy. Chapin *et al.* [18] analyze entropy metrics of software systems and conclude that by observing any abrupt change in the entropy of a software system, one can gather a good insight on how the maintenance of the software system should be performed.

Another interesting take on entropy analysis is the work by Unger *et al.* [19] which use the Shannon entropy measure to quantify information content in databases and propose a measure of the general vulnerability of databases based on entropy values. Similar to our study, they analyzed large software repositories and propose a technique based on entropy analysis.

VI. Conclusion

Triaging crash-types is a crucial software maintenance activity. A good triaging of crash-types is essential to allow developers and maintainers to focus their efforts more efficiently. Our study proposes a new triaging approach based on an entropy analysis of crash-types. The new approach introduces a concept of entropy graph regions to assign priority values to crash-types.

Quantitatively, we have showed that entropy analysis help classify crash-types by their level of difficulty. We have also showed the ability of entropy regions to discriminate crash-types of different characteristics. We evaluate the proposed entropy based crash-type triaging approach by comparing the similarity between its assigned priority levels and the priority levels of crash-types obtained from bug reports. We obtain that for all the regions except the Isolated region, the priorities assigned by the entropy based triaging approach are the same as the priority levels obtained from bug reports. A further analysis of the priorities in the Isolated region reveals that although Mozilla quality teams sometimes assign high priorities to crash-types from the Isolated region they do not fix these crash-types in priority. Our result suggests the necessity to improve the current crash-type triaging process of Mozilla quality teams. Therefore, the priorities assigned based on entropy values better reflect the priorities that are applied by developers when fixing the crashes. Developers and quality assurance teams could make use of our proposed automatic entropy based triaging approach to improve their crash-types triaging and better plan their testing and maintenance activities. In future work, we plan to perform further validation of our approach on different systems using different crash triaging process.

References

[1] "Socorro: Mozilla's crash reporting system," *Accessed on March 29, 2011*. [Online]. Available: http://blog.mozilla.com/webdev/2010/05/19/socorro-mozilla-crash-reports/

[2] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 495–504. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806871

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[4] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, pp. 3–55, January 2001. [Online]. Available: http://doi.acm.org/10.1145/584091.584093

[5] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[6] G. W. Stats, "W3counter (2010-10-31)," *Retrieved on 2010-11-09*.

[7] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods*, 2nd ed. John Wiley and Sons, inc., 1999.

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

[9] 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*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595715

[10] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 1767–1772. [Online]. Available: http://doi.acm.org/10.1145/1141277.1141693

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

[12] 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*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.13

[13] A. E. Hassan and R. C. Holt, "The chaos of software development," in *Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 84–. [Online]. Available: http://portal.acm.org/citation.cfm?id=942803.943729

[14] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: A case study on firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. IEEE Computer Society, 2011, pp. 93–102.

[15] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio, "Evaluating software degradation through entropy," in *IN ELEVENTH INTERNATIONAL SOFTWARE METRICS SYMPOSIUM*, 2001, pp. 210–219.

[16] S. K. Abd-El-Hafiz, "Entropies as measures of software information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 110–. [Online]. Available: http://portal.acm.org/citation.cfm?id=846228.848671

[17] K. Kim, Y. Shin, and C. Wu, "Complexity measures for object-oriented program based on the entropy," in *Proceedings of the Second Asia Pacific Software Engineering Conference*, ser. APSEC '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 127–. [Online]. Available: http://portal.acm.org/citation.cfm?id=785406.785448

[18] N. Chapin, "An entropy metric for software maintainability," *Twenty-Second Annual Hawaii International Conference on System Sciences, Software Track*, pp. 522–523, January 1995.

[19] E. Unger, L. Harn, and V. Kumar, "Entropy as a measure of database information," *Proceedings of the Sixth Annual Computer Security ApplicationsConference*, pp. 80–87, December 1990.