2011 27th IEEE International Conference on Software Maintenance (ICSM)

Classifying Field Crash Reports for Fixing Bugs : A Case Study of Mozilla Firefox

Tejinder Dhaliwal, Foutse Khomh, Ying Zou
Dept. of Electrical and Computer Engineering
Queen's University, Kingston
{tejinder.dhaliwal, foutse.khomh, ying.zou}@queensu.ca

Abstract— **Many software systems support automatic collection of field crash-reports which record the stack traces and other runtime information when crashes occur. Analysis of field crash-reports can help developers to locate and fix bugs. However, the amount of crash-reports collected is often too large to handle. To reduce the amount of data for the analysis, the existing approaches group similar crash-reports together. A bug can trigger a crash in different usage scenarios. Therefore, the crash-reports triggered by the same bug may not be identical. Using the existing approaches, the crash-reports triggered by the same bugs can be distributed into different groups and one group may contain crash-reports triggered by different bugs. We perform an empirical study of crash-reports collected for Mozilla Firefox to analyze the impact of crash-report grouping and identify the characteristics of an efficient grouping. We observe that when a group contains crash-reports triggered by multiple bugs, it takes longer time to fix the bugs in comparison to the bugs where crash-reports triggered by each bug are grouped separately. To effectively reduce the bug fixing time, we propose a grouping approach, such that, each group contains the crash-reports triggered by only one bug. The case study shows that an effective grouping can reduce the bug fix time by more than 5%.**

Keywords-Bug localization; Automatic crash reporting; Clustering

I. INTRODUCTION

A **crash** terminates an application unexpectedly in a natural setting. Automatic crash reporting tools are commonly built into software systems to collect crash-reports from a user environment and send them to a central repository. A crash-report usually contains a stack trace of the failing thread and other runtime information. A stack trace is an ordered set of frames and each frame refers to a method signature. The crash-reports can help the software developers to diagnose and fix the root cause of the crashes. The automatic collection of crash-reports in Mozilla Firefox improved the reliability of Mozilla Firefox by 40% from November 2009 to March 2010 [21]. Microsoft was able to fix 29% of all Windows XP bugs and third-party bugs due to the automatic collection and analysis of crash-reports [12]. However, the built-in automatic crash reporting tools often collect a large number of crash-reports. For example, Firefox receives 2.5 million crash-reports every day [21].

It is challenging for organizations to manage large amount of collected crash-reports effectively. In particular, due to the reoccurrence of the same bug, many of the crash-reports are redundant. To reduce the amount of crash-reports to handle, similar crash-reports are identified and grouped together in the central repository. We refer to a group of similar crash-reports as a **crash-type**. For example, Mozilla groups the crash-reports using the top method signature in the failing stack trace. However, such a grouping approach is not accurate since the crash-reports triggered by the same bug might not be identical. If multiple bugs have the same top method signature in their failing stack traces, the crash-reports triggered by these bugs are designated to the same crash-type. We observe that if the crash-reports caused by multiple bugs are grouped together it takes longer time to fix the bugs. The observation indicates that if each crash-type contains the crash-reports triggered by only one bug, it is easier for a developer to fix the bugs. A detailed comparison of the stack traces of two crash-reports can help to determine if the crash-reports are triggered by the same bug. Therefore, to ensure that the crash-reports triggered by different bugs are grouped separately, we propose a two-level grouping approach, where a crash-type is further divided based on the similarity of the stack traces of the crash-reports. We strive for the similar stack traces of the crash-reports within the same subgroup, such that each subgroup contains the crash-reports triggered by a single bug. The crash-reports are large in number, and a detailed comparison of stack traces could be an intensive computation, therefore we optimize our approach to handle the large amount of crash-reports.

We conduct an empirical study on crash-reports and bugs, collected from ten releases of Firefox. We want to understand the issues with the existing crash-report grouping approach and to evaluate if our proposed two-level grouping approach can improve the bug fixing process. We formulate the following research questions for our study:

978-1-4577-0664-6/11/$26.00 ©2011 IEEE

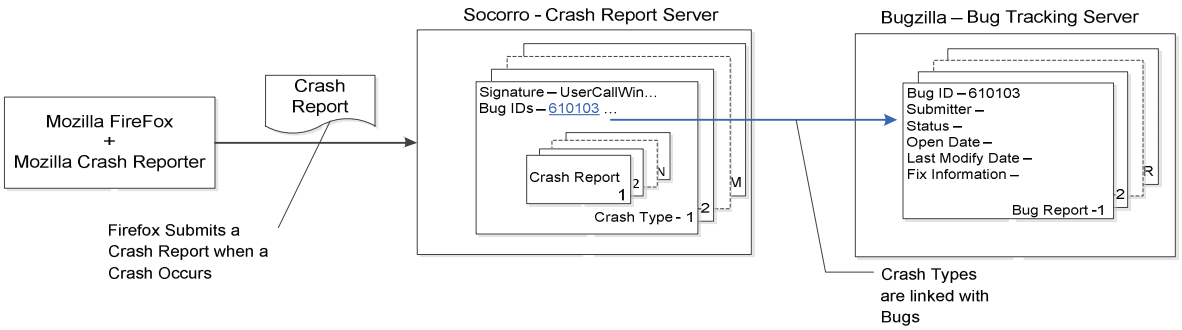

Figure 1: Mozilla Crash Report System

RQ1: Can stack traces in crash-reports help to locate bugs?

Our first research question evaluates the usefulness of the stack traces in the crash-reports for bug fixing activities. We want to verify if it is useful to analyze the stack traces in crash-reports to identify the bugs. We analyze the already fixed bugs and observe that 80% of the bugs are located in the faulty modules that appear in the stack traces of crash-reports.

RQ2: Does a crash-report grouping approach have an impact on bug fixing?

Our second question investigates the impacts of a crash-report grouping approach on the bug fixing time. In Firefox crash-reports, we find that when multiple bugs are filed for the same crash-type, it takes longer time to fix the bugs in comparison to fixing the bugs, which are uniquely filed for a crash-type. Moreover, when a bug is filed and related to multiple crash-types, it takes even shorter time to fix the bug. The result indicates that it takes longer time to fix bugs when crash-reports triggered by multiple bugs are grouped together. Furthermore, we compare the similarity among the stack traces of different crash-reports within a crash-type and observe that lower similarity indicates that the crash-reports within a crash-type are triggered by different bugs.

RQ3: Does a detailed comparison of stack trace help improve the grouping?

The third question evaluates our proposed grouping approach. This question verifies if the approach can group the crash-reports triggered by different bugs separately. Using our approach, we found that 88% of the identified groups contain the crash-reports triggered by a single bug. On average, 98% of the crash-reports in an identified group are triggered by the same bug.

Organization of the paper: Section 2 gives an overview of the Mozilla crash reporting system. Section 3 introduces the proposed enhancements in crash-report grouping approach. Section 4 describes our experimental setup, data collection and analysis approach. Section 5 presents the results of our study. Section 6 discusses the limitations and threats to validity of the study. Section 7 discusses the related work. Finally, Section 8 concludes our work and discusses future work.

II. Overview of Mozilla Crash Reporting System

Firefox is the second most popular web browser with 27% usage share worldwide [6]. Firefox supports automatic collection of the crash-reports. It is delivered with a built in crash reporting tool, Mozilla Crash Reporter [3]. Figure 1 presents an overview of the Mozilla crash reporting system. When Firefox is terminated unexpectedly, the Mozilla Cash Reporter sends a detailed crash-report to the Socorro crash report server [17]. A crash-report includes the stack trace of the failing thread and other information about the user environment, such as operating system, Firefox version, install time, and a list of plug-ins installed.

Socorro groups the crash-reports based on the top method signature of the stack trace. However, the subsequent frames in the stack trace might be different for different crash-reports in a crash-type. For each crash-type, Socorro server provides a crash-type summary, i.e., a list of the crash-reports of the crash-type and a set of bugs filed for the crash-type. Socorro provides a rich web interface for the developers to analyze the crash-types. Developers prioritize the top crash-types (i.e., the crash-type with the maximum number of crash-reports) to analyze and fix the bugs responsible for the crash.

Mozilla uses Bugzilla to track bugs and maintains a bug report for each filed bug. A bug report contains detailed information about a bug, such as the bug open date, the last modification date, and the bug status. When a developer fixes a bug, he often submits a patch to Bugzilla. The patch includes source code changes and other configuration file changes. Once approved, the patch code is integrated into the source code of Firefox. Patches can be used to identify where a bug is fixed. For the top crash-types, 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. Web interfaces of the Socorro server and Bugzilla are integrated, developers can navigate from a crash-type summary in the Socorro server to the bugs filed for the crash-type in Bugzilla.

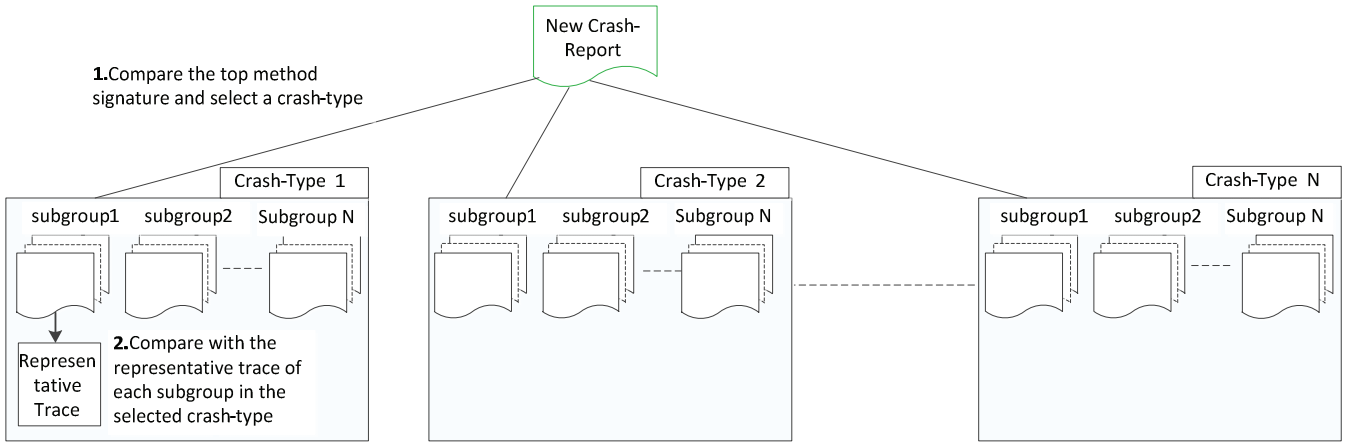

Figure 2: The Tow-Level grouping of crash reports using representative trace

III. TWO-LEVEL GROUPING APPROACH OF CRASH-REPORTS

To group the crash-reports triggered by different bugs separately, we propose to enhance the existing crash-report grouping approach of Socorro. The existing approach is performance efficient as it compares only the top method signature of the stack traces. However, using this approach, a crash-type might contain the crash-reports caused by multiple bugs. We suggest a two level grouping of the crash-reports. The first level of grouping leverages the existing approach used by Socorro. It clusters the crash-reports based on the top method signature of the stack traces to form crash-types. Furthermore, we use a detailed comparison of stack traces to divide the crash-reports in a crash-type into subgroups. The subgroups within a crash-type create the second level grouping. Each subgroup is intended for developers to analyze and file bugs, instead of using the crash-types. Our approach subdivides crash-reports that have greater dissimilarities among stack traces within a crash-type. If the first level crash-types contain very similar crash-reports, such crash-types remain intact. Figure 2 shows the structure of the two- level grouping of crash-reports.

We use the Levenshtein distance [11] to evaluate the similarity between stack traces. The Levenshtein distance is used for comparing two sequences. It measures the amount of differences between the sequences. The Levenshtein distance between two stack traces is the number of changes needed to transform one stack trace into the other. A change can be inserting a frame, deleting a frame or replacing a frame. We evaluate the Levenshtein distances for every pair of stack traces within a crash-type by comparing the top 10 frames of the stack traces. We limit the comparison of stack traces to the top 10 frames of each stack trace, since previous study [1] found that bugs are in general fixed in one of the top 10 frames from the failing stack trace.

The average value of Levenshtein distance is calculated by comparing the differences between every pair of stack traces in a crash-type. The average value of Levenshtein distance for a crash-type indicates the diversity among all the stack traces of the crash-type. We refer to it as the *Trace Diversity* of the crash-type. Similarly, we can compute the *Trace Diversity* of a subgroup, i.e., the average Levenshtein distance among the stack traces of all the crash-reports in a subgroup. We measure the trace diversity for existing crash-types and determine the threshold value, i.e., the maximum value of the trace diversity for crash-types where the crash-reports are triggered by the same bug. We suggest that each subgroup must have a trace diversity value less than the threshold value, such that we can ensure that all crash-reports in a subgroup are triggered by the same bug.

In the top ranked crash-types (i.e., the most frequently occurring crash-types), the number of crash-reports is very large. As a consequence, the detailed comparison of a large amount of stack traces could be computation intensive. Therefore, the grouping approach must be performance efficient. Moreover, it is a continuous process to collect the field crash-reports and assign them to appropriate crash-types. If a grouping approach is applied to all the crash-reports collected for a crash-type, the organization of existing sub-groups might be changed each time when a new crash-report is added into a crash-type. However, it is critical to maintain the subgroups over time. In particular, stable subgroups allow developers to analyze the crash-reports within a subgroup, file bugs for each subgroup and refer back to the subgroup. To address these issues we use incremental grouping at the second level of our approach. When a new crash-report is added to a crash-type, the report is assigned to a subgroup without changing the grouping structure of existing crash-reports in the crash-type.

To improve the performance of the detailed comparison and maintain the structure of the already formed subgroups within a crash-type, we assign a representative trace for each sub-group (as shown in Figure 3). When a crash-report is received at the central repository, it is assigned to a crash-type based on the top method signature. In the selected crash-type, the new crash-report is compared with the existing subgroups. To compare a crash-report with a

subgroup, it is not compared with every report in the subgroup. Instead, the stack trace of the new report is compared with the representative trace of the subgroup. The new report is added to the subgroup with the minimum Levenshtein distance between the stack trace of the new crash-report and the representative trace of the subgroup. However, the Levenshtein distance value must be less than the threshold value; otherwise a new subgroup is created for the crash-report.

In particular, a representative trace is a sequence presenting the number of appearance of the modules in each of the top 10 frames of the stack traces. More specifically, the i^{th} frame of a representative stack trace presents the number of appearance of each module that appears in the i^{th} frame of any stack trace from the subgroup. Figure 3 shows an example subgroup with four crash-reports. In this example, three crash-reports have the module *B* in the second frame of their stack trace and one crash-report has the module *C* in the second frame of its stack trace. Therefore, the second frame of the representative stack trace has a value "F_B =3, F_C =1". F_B in the second frame denotes the number of appearance for module *B* in the second frame of the stack traces from the subgroup.

The Levenshtein distance measures the amount of difference between two sequences. More specifically, Levenshtein algorithm incrementally combines the distances of individual nodes to compute the difference. If both sequences are of the same type, for any pair of nodes from each sequence, the distance is 0 if the nodes are the same; and the distance is 1 if the nodes are different. In our work, we compare the stack trace of a crash-report with the representative trace, which are not of the same type, because the stack trace is a sequence of frames and the representative trace is a sequence of set of frequencies. For a representative trace *R* and a stack trace *S*, the difference between any pair of nodes r and s, selected from *R* and *S* respectively, is defined in Equation (1). For example, the distance between a stack trace frame containing module *B*, and the second frame of the representative stack trace shown in Figure 3 i.e., (F_B =3, F_C =1) would be 1-3/4 = 0.25.

$$dist(r,s) = 1 - \frac{Freq(r,s)}{RC} \quad (1)$$

Where Freq(r,s) is the frequency value of the module M in r, where M is the module appearing in s ; and RC is the total number of crash-reports in a subgroup.

The representative trace gives higher preference to more frequent frames, therefore only the new crash-reports containing the stack traces with frames similar to the frequent frames in a subgroup are added to the subgroup. We assume this way that the representative trace bootstrap the similarity among the crash-reports of a subgroup, we evaluate the approach on Mozilla crash-report dataset to verify the effectiveness of the grouping approach.

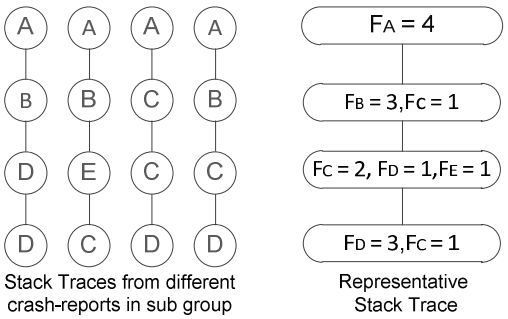

Figure 3: Representative stack trace for a Subgroup

IV. Experimental Setup

This section introduces the data collection process, outlines the steps of our data analysis, and discusses the techniques used to evaluate the proposed two-level grouping approach.

A. Data Collection

We sample crash-reports from ten beta releases of Firefox, ranging from Firefox-4.0b1 to Firefox-4.0b10. The beta releases are used for field testing. We download the summaries of the available crash-types and select the crash-types for which at least one bug is filed. For each selected crash-type we download 100 crash-reports (randomly sampled). We download all the available crash-reports for the crash-types which have less than 100 crash-reports. We parse the sampled crash-reports and extract the failing stack traces. Table 1 reports the descriptive statistics of our dataset.

For all the bugs filed for the crash-types in our data set, we retrieve the bug reports from Bugzilla. If a patch is submitted for the bug, the bug report includes the patch. For every patch found in a bug report, we perform a syntactical analysis to retrieve information about what changes are made to fix the bug. We map this information on source code change locations to the stack trace in the crash-reports. Moreover, for each fixed bug we compute the bug fixing time, i.e., the difference between the bug open time and the last modification time. In the case of a bug resolved as DUPLICATE, if the original bug is filed for the same crash-type, we ignore the duplicate bug. If the original bug is filed for some other crash-type, we link the original bug to where the duplicate bug was linked.

Table 1: Descriptive Statistics of the Data Set

The number of crash-types with at least one bug filed	1,329
The total number of crash-reports sampled	82,156
The total number of bugs linked to crash-types	1,733
The number of fixed bugs	519
The number of duplicated bugs	253
The number of open bugs	961
The number of fixed bugs with a patch	231

B. Data Analysis

In this study, we examine the usefulness of stack traces for bug fixing activities and evaluate the current grouping approach used in Firefox.

RQ1: We investigate if stack traces contained in crash-reports can help developer to locate the bugs. We map the modules changed for bug fixing to the stack traces of the crash-reports. If the faulty module appears in any of the stack traces from the crash-type for which the bug is filed, we call it a bug *fixed in the linked stack trace*. If the faulty module appears in a stack trace from other crash-type, i.e. a crash-type not linked with the bug, we call it a bug *fixed in other stack traces*. If a bug is fixed in a module that has never appeared in a failing stack traces from any crash-type, we call it a bug *fixed elsewhere*. We compute the bug fixing time for the bugs and test the following null hypothesis:

H_{01}: the lifetime of a bug is the same for the bugs fixed in the linked stack traces, the bugs fixed in other stack traces and the bugs fixed elsewhere.

We use the Kruskal-Wallis rank sum test to investigate if the distribution of fixing times is the same for the bugs fixed in the linked stack traces, the bugs fixed in other stack traces and the bugs fixed elsewhere. 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.

RQ2: We investigate if the grouping of crash-reports has an impact on the bug fixing time. First we categorize the bugs by checking if the crash-reports triggered by a bug are grouped separately or if the crash-reports triggered by multiple bugs are grouped together. When the bugs are uniquely linked with one or more crash-types, it indicates that the crash-reports triggered by the bug are grouped separately. If multiple bugs are collectively linked with a crash-type, it indicates that the crash-reports triggered by multiple bugs are grouped together. Figure 4 presents the categories we defined for the bugs filed for the crash-types. We subdivide the bugs for which the crash reports are grouped separately, by checking if the crash-reports triggered by a bug are grouped together in a single crash-type, or split in multiple crash-types. We compare the fixing time for the categories and test the following two null hypotheses:

H^1_{02}: the lifetime of a bug is the same for the bugs for which crash-reports triggered by multiple bugs are grouped together and for the bugs for which the crash-reports triggered by every individual bug are grouped separately.

H^2_{02}: the lifetime of a bug is the same for bugs for which the crash-reports are grouped in a single crash-type or crash-reports are split in multiple groups.

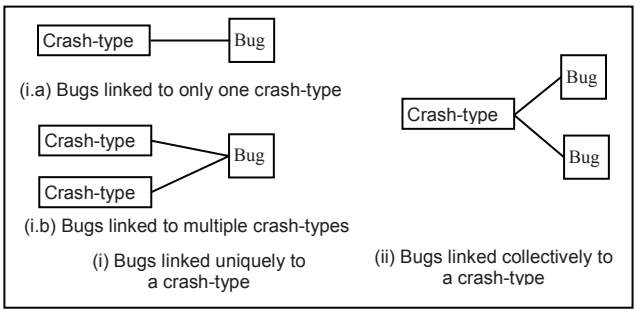

Figure 4: Categories of bugs based on the number of bugs linked to the corresponding crash-type

We use the Wilcoxon rank sum test [15] to accept or reject H^1_{02} and H^2_{02}. 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 fixing time for the bugs linked to multiple crash–types and the bugs linked to a single crash-type.

Furthermore, we analyze the trace diversity of the crash-types, as discussed in Section 3. We analyze the relation between the trace diversity of crash-types and the number of bugs linked with the crash-types.

RQ3: The third research question evaluates the two-level grouping approach presented in Section 3. We use the silhouette validation technique to evaluate the two-level grouping algorithm. The silhouette validation [19] is a technique to measures the goodness of a grouping approach. Using silhouette validation, we compare the dissimilarity of a crash-report with other crash-reports from the same subgroup and the similarity of the crash-report with other subgroups in the same crash-type. For a crash-report *i* the silhouette value *S(i)* is defined in Equation (2).

$$S(i) = \frac{b(i) - a(i)}{\max(b(i), a(i))} \quad (2)$$

Where a(i) is the average dissimilarity of the crash-report i to all other crash-reports in the same subgroup and b(i) is the minimum of average dissimilarity of the crash-report i to the crash-reports in other subgroups in the same crash-type.

We compute the similarity (or dissimilarity) of two crash-reports by comparing the top ten frames of the stack traces from the crash-report, as discussed in Section 3. The average silhouette value for all crash-reports is the silhouette value for the crash-type. The silhouette value has a range from -1 to 1, where the value -1 implies misclassified and a value close to 1 implies well clustered.

Furthermore, we assess the effectiveness of the two-level grouping approach to group the crash-reports triggered by the same bug. We select the crash-types for which at least one bug is fixed and the bug has the patch information. We apply our proposed grouping approach to the selected crash-types, and build subgroups. For each bug, we identify

modules that are changed to fix the bug. We map this information to the stack traces contained in crash-reports from a subgroup. If a bug fix location appears in the stack trace of any of the crash-report from the subgroup, we link the bug with the subgroup. As a result, a subgroup can be linked to a single bug, to multiple bugs, or to no bug. It is desirable to have a subgroup linked with a single bug, since it suggests that crash-reports in the subgroup are triggered by the same bug.

We compute the accuracy of our grouping algorithm as defined in Equation (3):

$$Accuraccy = \frac{N(s) + N(z)}{N(s) + N(z) + N(m)} \quad (3)$$

Where N(s) is the number of crash-reports in the subgroups linked to a single bug, N(z) is the number of crash-reports in the subgroups linked to no bug, and N(m) is the number of crash-reports in the subgroups linked to multiple bugs.

The accuracy metric assesses the ability of the approach to group the crash-reports triggered by the same bug. When a subgroup is linked to multiple bugs, it's likely that the crash-reports in the subgroup are triggered by multiple bugs. If the crash-reports in the subgroups are not linked with any bugs, such crash-reports are triggered by a bug which is not identified and not filed by the developers.

We compute the precision of a subgroup as defined in Equation (4). The precision of a subgroup measures the percentage of crash-reports in the subgroup triggered by the bug linked to the subgroup. If the faulty module, where the bug is fixed, appears in the stack trace of a crash-report, we consider that the crash-report is caused by the same bug.

$$Precision = \frac{N(t)}{N(t) + N(f)} \quad (4)$$

Where N(t) is the number of crash-reports in a subgroup for which the faulty module appears in the stack trace; and N(f) is the number of crash-reports in a subgroup for which the faulty module does not appear in the stack trace.

V. RESULTS

In this section, we present the results of our case study on the research questions and discuss our findings.

A. RQ1: Can stack traces in crash-reports help to locate bugs?

This research question investigates the use of the crash-reports for bug fixing. More specifically, we aim to assess if stack traces contained in the crash-reports are useful to fix bugs. We analyze all fixed bugs which have available patches, and extract the corresponding patches to identify modules that are changed to fix the bug. The patches are available for 231 bugs and can be mapped to the source code. We map the bug fix locations to the stack traces of the crash-reports, as described in Section 4.B. We compute the percentages of bugs belonging to each of the following three

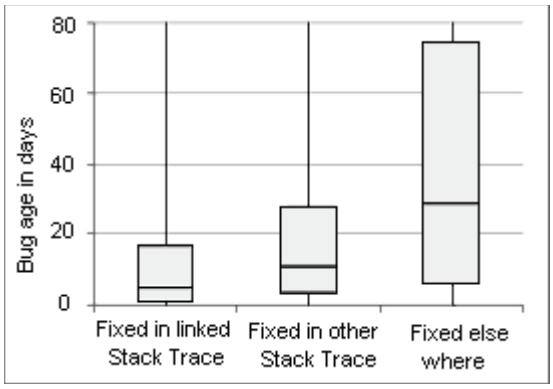

Figure 5: Boxplots comparing lifetimes of the bugs, based on bug fix locations.

categories: (1) bugs that are fixed in the linked stack traces; (2) bugs that are fixed in other stack traces; and (3) bugs that are fixed elsewhere. On average, 57% of bugs are fixed in the linked stack traces; 23% of the bugs are fixed in other stack traces; and the remaining 20% of bugs are fixed elsewhere. Figure 5 presents the boxplots of the lifetime of bugs in the three categories.

As shown in Figure 5, the bugs fixed in the linked stack traces are fixed quicker than the bugs classified in the other two categories. The mean and median values of the time to fix bugs in the linked stack traces category are 19 days and 5 days, respectively. The mean and median values are 23 days and 11 days, respectively for bugs fixed in other stack traces and 48 days and 29 days, respectively for the bugs that are fixed elsewhere.

We perform the Kruskal-Wallis rank sum test on the lifetimes of bugs from the three categories and obtain a statistically significant result (i.e., p-value is less than 0.01). Therefore, we reject hypothesis H_{01}. We conclude that the lifetime of a bug is significantly shorter when the faulty module appears in the stack traces of the crash-reports. The lifetime of a bug can be further reduced when the stack traces containing the faulty modules are correctly linked to the bug.

It indicates that bugs fixed in the linked stack traces take shorter time to get fixed, since developers can locate the bugs easily by analyzing the failing stack traces of the linked crash-reports. However, it is surprising that bugs fixed in other stack traces take shorter time than bugs fixed elsewhere. Since we sample only 100 crash-reports for each crash-type, the shorter bug fixing time observed for the bugs fixed in other stack traces indicates that there may be other crash-reports in the crash-types with stack traces containing the faulty module. Overall, our results suggest that in general for 57% to 80% of the bugs, stack traces in the crash-reports can help to locate the bugs. We answer positively our research question that the stack traces in crash-reports can help the localization and correction of bugs. Moreover, crash-reports triggered by the bug can be identified by analyzing the stack traces in the crash-reports.

B. RQ2: Does the grouping of crash-reports impacts bug fixing?

We observe in our data set that for some of the crash-type, multiple bugs are filed and some bugs are linked to multiple crash-types. We assume that if crashes triggered by multiple bugs are grouped together, this creates ambiguity for developers to analyze the crash-type. We also assume that if a crash-type contains the reports triggered by a single bug, developers can fix the bug more efficiently. To verify our assumptions and answer the research question, we compare the bug fixing times for different bug categories based on bug crash-type relations.

1) Crash-types Linked to Multiple Bugs

Our data set contains 519 fixed bugs. 74% of the fixed bugs are uniquely linked to the corresponding crash-types. The remaining 26% of bugs are linked to the crash-types where other OPEN or FIXED bugs are also linked to the same crash-type; it indicates that crash-reports triggered by different bugs are grouped together.

We compare the fixing time of bugs that are uniquely linked to one or more crash-types with the fixing time of bugs that are collectively linked to the same crash-type. Figure 6 shows the boxplots of bug fixing times for both cases. The mean and median values of the time to fix bugs uniquely linked with one or more crash-types are 26 days and 10 days, respectively. The mean and median values are 43 days and 17 days respectively, for bugs collectively linked to a same crash-type. If the crash-reports triggered by each bug are grouped separately, the bug takes on average 17 days lesser to be fixed than fixing the bugs for which crash-reports are grouped together. This finding validates our assumption that it is difficult to locate and fix the bug when crash-reports triggered by different bugs are grouped together. We perform a Wilcoxon rank sum test to verify the statistically significance of this result and obtained a p-value of 0.04. Therefore, we reject $H^{l}_{02.}$

In summary, when multiple bugs are collectively linked to the same crash-type, it takes a longer time to have the bugs fixed than fixing the bugs that are uniquely linked to one or more crash-types. We answer our research question positively: the grouping of crash-reports has an impact on the bug fixing time.

2) Bugs Linked to Multiple Crash-types

In our data set, 384 fixed bugs are uniquely linked to one or multiple crash-types; 40% of the bugs are uniquely linked to multiple crash-types and the remaining 60% of bugs are uniquely linked to a single crash-type. We compute the lifetimes of bugs and observe that the bugs linked to multiple crash-types take on average 3 days less to be fixed than fixing the bugs linked to a single crash-type. It hints that the bug is assigned a high priority when a bug is linked with multiple crash-types. Moreover, when bugs are linked to multiple crash-types, the crash-types provide rich information on different scenarios of the bug occurrences. Thus, it helps developers better understand the issues.

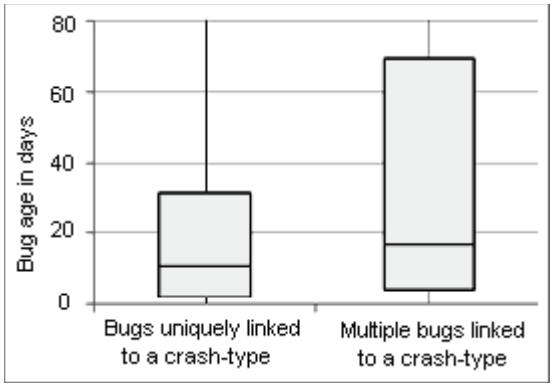

Figure 6: Boxplots comparing lifetimes of bugs uniquely linked to a crash-type vs bugs collectively linked to a crash-type

However, the Wilcoxon rank sum test reveals no statistically significant difference between the lifetimes of the bugs linked to multiple crash-types and the lifetimes of the bugs linked to a single crash-type (i.e., p-value equal to 0.08).

3) Trace Diversity of Crash-types

We analyze the current grouping approach of crash-reports from Socorro to understand the diversity of the stack traces contained in the crash-reports of a crash-type. As aforementioned, the existing approach groups crash-reports based on the top method signature of the failing stack trace. The stack traces are not identical for all the crash-reports of a crash-type. We quantify the diversity of the stack trace in crash-reports from a crash-type using the trace diversity as discussed in Section 3. We categorize the crash-types based on the number of bugs filed for each crash-type. For each category, we compute the average trace diversity of the crash-types. Table 2 lists the detailed results for the categories.

As shown in Table 2, if a single bug is filed for a crash-type, the crash-type has relatively lower trace diversity than the crash-types that have multiple bugs filed. We statistically verify the result as the Spearman's rank correlation value between the trace diversity values and the number of bugs linked to the crash-type is 0.95 (i.e., p-value equal to 4.96e-05). The result shows that higher trace diversity indicates that crash-reports in a crash-type are triggered by multiple bugs. The effectiveness of a crash-report grouping approach can be improved by controlling the magnitude of the trace diversity value when grouping crash-reports together.

Table 2: Average Trace Diversity Values for All Crash-Types

Number of bugs linked to each crash-type	Average Trace Diversity
1	4.82
2	5.81
3	5.88
4	6.67
5	8.22

C. *RQ3: Does a detailed comparison of stack trace help improve the grouping?*

We perform a case study to assess the effectiveness of the two-level grouping approach presented in Section 3. We select the 231 bugs from our data set that have patches available. The 231 bugs are linked to 277 crash-types which consist of 18,498 crash-reports. We apply the two-level grouping approach to regroup the crash-reports. We set the trace diversity threshold value to 5, since in Section 5.B we observe that the crash-types for which a single bug is filed have a trace diversity value close to 5. Table 3 lists the descriptive statistics of the data set used for our evaluation. Table 4 shows the result of the evaluation of the two-level grouping approach.

The average trace diversity of the subgroups created using the two-level grouping approach is low, i.e., 3.8. We measure the goodness of our grouping by computing silhouette values. The average silhouette value for each crash-type is 0.81. A high value (i.e., 0.81) suggests a good clustering of crash-types.

For the subgroups, we compute the accuracy as described in Section 4.B. As shown in Table 4, the accuracy of the two-level grouping approach is 0.88. It shows that 88% of the newly created subgroups are linked to only one bug or no bug.

We compute the precision for the 512 subgroups that are linked to a single bug, using Equation (4). The average precision of the subgroups is 0.98, meaning that on average 98% of crash-reports in each subgroup are triggered by the same bug, which is linked to the subgroup.

Despite 88% of accuracy and 98% precision, one can question that the number of subgroups created are 3 times more than the number of crash-types. But our approach maintains the existing crash-types, so at the first level, the number of groups is the same as currently in Socorro. However, when developers analyze a crash-type, the subgroups provide more detailed information. If two subgroups are related to different bugs, the subgroups improve the bug fixing process by separating the crash-reports caused by each bug. Even if two subgroups are caused by the same bug, both subgroups represent significantly different stack traces. As discussed in Section 5.B when a developer selects one crash-report from each subgroup, the selected crash-reports provide better information than randomly selected reports. The 512 accurately created subgroups are linked to 220 bugs, i.e., on average 2.3 subgroups are created for each bug.

To further assess the benefit of our proposed grouping approach, we compare the estimated bug fixing time when crash-types are divided in subgroups using the two-level grouping approach and the actual bug fixing time that we compute from the bug reports. The collective time for fixing all the 231 bugs is 6540 days. As discussed in Section 5.B, on average the bug fixing time for a bug uniquely linked with a crash-type is 26 days; and the bug fixing time for the bugs collectively linked with a crash-type is 43 days.

Table 3: Descriptive Statistics of Evaluation Data Set

	# of crash-type	**# of crash-reports**	**# of Bugs Linked**	**fix time (days)**	**Avg. *TD***
All crash-types	277	18498	231	***6540***	6.5
Crash-types linked to a single bug	225	14244	204	5212	4.6
Crash-types linked to multiple bugs	52	4254	27	1328	14.7

Table 4: Descriptive Statistics of Result

	# of subgroups	**# of crash-reports**	**# of Bugs Linked**	**Est. fix time (days)**	**Avg. *TD***
All subgroups	941	18498	231	***6193***	3.8
subgroups linked to a single bug	512	10812	220	5720	3.6
subgroups linked to zero bug	297	5547	0	0	3.9
subgroups linked to multiple bugs	132	2139	11	473	4.3

Avg. TD – Average Trace Diversity
Est. fix time – Estimated Bug Fixing Time

Using these average values of bug fixing time, we estimate the collective bug fixing time for the 231 bugs when developers use the proposed two-level grouping approach. The estimated time is (220*26 + 11*43) = 6193 days. We can conclude that the two-level grouping approach can reduce the bug fixing time by 5.3%.

VI. THREATS TO VALIDITY

We now discuss the threats to validity of our study following the guidelines for case study research [20].

Construct validity threats concern the relation between theory and observation. In this study, the construct validity threats are mainly due to measurement errors. We extract stack trace and bug information by parsing the html and xml files and map the bug fix location to the stack traces by applying string matching. The techniques we use are similar to the techniques used by previous studies [1][16].

Threats to internal validity do not affect this study since we do not claim causation [20]. We simply report our observations, although our discussion tries to explain 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 data set distribution.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. Moreover, both the Socorro crash server and Bugzilla are available publicly[17], to obtain the same data for the same releases of Firefox.

Threats to external validity concern the possibility to generalize our results. Nevertheless, our study is limited to 10 releases of Firefox, further studies with different systems and different automatic crash-reporting systems are desirable to make our findings more generic.

VII. Related Work

This section discusses the related literature on field crash-reports, bug correlation, and analysis of stack trace.

A. Bug correlation and localization.

Grouping of field crash-reports is similar to bug correlation, where we try to find which two crash-reports are correlated. There has been an extensive research on automatic bug correlation and bug localization. Lee and Soffa [26] introduced a bug correlation algorithm to identify causal relationships among bugs in a system. Ball et al. [24] developed a localization technique for error traces generated from a model checker. The aim of their technique was to identify the transitions that only appear in failing traces but not correct traces. Liblit et al. [2] analyzed predicate patterns in correct and incorrect executions traces and proposed an algorithm to separate the effects of different bugs in order to identify predictors associated with individual bugs. They claim that their algorithm is able to detect a wide variety of both anticipated and unanticipated causes of failure. Jones et al. [8] [10] examined the execution traces of successful and fail test cases and proposed Tarantula, a technique based on visualization to assist developers locates errors and bugs in their systems. Nessa et al. [22] proposed a bug localization algorithm based on N-gram analysis, to rank the executable statements of a software by level of suspicion. Their new algorithm was able to outperform Tarantula on three case studies. Wong and Debroy [25] propose a comprehensive survey of existing bug localization techniques. Similar to our study, the above works emphasize the importance of stack trace for bug localization. However, none of the techniques mentioned in these works can be used to analyze stack trace from crash-reports. These techniques are all dependent on instrumentation, predicates, and coverage reports or successful traces. This needed information is not available in crash-reports.

B. Analysis of stack trace

Schroter et al. [1] investigated the use of stack trace for bug fixing through an empirical study of the bugs in Eclipse. They observed that for 60% of crashes that had at least one stack trace available, bugs were fixed in one of the frame from the stack trace. Our study confirms the result and we use this result as base for our grouping algorithm. Chan and Zou [4] proposed the use of visualization for bug correlation and the identification of relation between different crashes. But given the large number of crash-reports (2.5 M crash-reports every day), visualization cannot be used to comprehend all the crash-reports. However, when crash-reports are grouped together correctly, the visualization of representative reports from each group can be used to find correlation between different bugs. The most closely related work to our study is the work by Brodie et al. [14][13], they used the stack-trace comparison to identify similar bugs. But, their approach makes use of historical data of already known problems. From a collection of different stack-traces of an already known problem, they develop a stack-trace pattern for each problem. Whenever a new problem is reported, it is compared with existing pattern of known problems and if a match is found, support staff can use this knowledge to handle the issue. However, the problem we address in this study is fundamentally different, as we propose an approach to identify similar crashes without having a prior knowledge of the bug or any pattern related to that bug.

C. Crash-report grouping

WER [12] is a system developed by Microsoft for handling field error reports. WER predates other crash-reporting tools and has a very large user base compared to Socorro since it is used with all Windows, IE and Microsoft Office applications. WER performs a progressive data collection of field errors; whenever a crash occurs on user's side, only a crash label is sent to the server. Developers need to configure the server if they wish to receive detailed crash-reports for a crash label. WER server groups detailed crash-reports using a bucketing algorithm. The Bucketing algorithm uses multiple heuristics specific to the application supported by WER and updated by developers manually. Whereas the system studied in this paper uses the open sources libraries, Breakpad [3] for the collection of client side data and Socorro [21] for processing field crash-reports on the server side. In comparison with WER, we propose a simpler and application independent approach. The suggested approach does not require any intervention from developers. Moreover, crash graphs, which are aggregated views of multiple crashes, proposed by Kim et al. [23] to identify fixable crashes in advance can also be applied with our grouping approach. The bucketing algorithm of WER can be easily replaced with our simpler and application independent grouping approach to predict fixable crashes.

VIII. CONCLUSION AND FUTURE WORK

It has become the norm to embed automatic collection of crash-reports in software systems. However, limited studies investigated the use of the collected crash-reports by developers in their maintenance activities. In this work, we studied the use of field crash-reports during the beta testing of Firefox-4. We summarize the key findings of our study as follows:

1) We analyze the use of failing stack traces in crash-reports by developers when performing bug fixing activities and find that 80% of bugs are fixed in modules appearing in failing stack traces of crash-reports. Therefore, stack traces in crash-reports can be used to identify the crash-reports that are triggered by the same bug.

2) We investigate the crash-report grouping approach used by Mozilla. We observe that in average it takes 17 days

longer to fix the bugs when crash-reports triggered by multiple bugs are grouped together in comparison to fixing the bugs for which the crash-reports are grouped separately.

3) We identify the limitation of the current grouping approach and propose a *Trace Diversity* metric which could help improve the efficiency of groupings. The result shows that if the trace diversity of a crash-type is greater than 5, the crash-type is likely to contain crash-reports triggered by multiple bugs.

4) We suggest a detailed comparison of stack traces to group the crash-reports. This limits the trace diversity of a crash-report group and it is easier for developers to locate and fix bugs. Our grouping approach limits the trace diversity of a subgroup to less than 5 and 88% of the subgroups contain crash-reports triggered by a single bug. This improvement to the existing Mozilla crash reporting system can help to reduce the bug fixing time by more than 5%.

We create a representative trace to identify the crash-reports caused by the same bug. In a way, the representative stack trace reflects the stack trace pattern of the bug. In the future, we plan to optimize the representative trace to further improve the crash report grouping. The representative trace can also be used for bug correlation and bug localization.

ACKNOWLEDGMENT

We are very thankful to Chris Hofmann from the Mozilla Foundation, for his valuable suggestions and support for this empirical study.

REFERENCES

[1] A. Schroter, N. Bettenburg, R.Premraj "Do The Stack Traces Help Developers Fix Bugs?" MSR 2010: 7th IEEE Working Conference on Mining Software Repositories, Waterloo, Ontario, Canada, May 2010

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 15-26, Chicago, Illinois, USA, June 2005

[3] Breakpad: Client side bug reporting in Mozilla https://wiki.mozilla.org/Breakpad (accessed March 29, 2011)

[4] B. Chan, Ying zou, A. E. Hassan and A. Sinha "Visualizing the Results of Field Testing." Internaional Symposium on Software Relibality Engimeering, Mysuru, India, November 2009

[5] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical Debugging: A Hypothesis Testing-based Approach," IEEE Transactions on Software Engineering, October 2006

[6] Global stat counter, http://gs.statcounter.com/#browser (accessed March 29, 2011)

[7] H. Shah, C. Görg, and M. J. Harrold, "Why do developers neglect exception handling?" in Procs. of the Int. Workshop on Exception Handling. ACM, 2008, pp. 62–68.

[8] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," IEEE/ACM Conference on Automated Software Engineering, December, 2005

[9] J. Goerzen, "Finding stubborn bugs with meaningful debug info," Linux J., vol. 2005, no. 129, p. 7, 2005.

[10] J. Jones, M. J. Harrold, and J. Stasko. "Visualization of test information to assist fault localization." In Proceedings of the International Conference on Software Engineering, pages 467{477, Orlando, Florida, May 2002

[11] J. B. Kruskal. "An Overview of Sequence Comparison: Time Warps, String Edits, and Macromolecules," SIAM Review.Vol. 25, No. 2 (Apr., 1983), pp. 201-237

[12] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle and G. Hunt. "Deugging in the (Very) Large: Ten Years of Implementation and Experience." In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles 2009

[13] M. Brodie, S. Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, P. Shon. "Quickly Finding Known Software Problems via Automated Symptom Matching." In Proceedings of the Second International Conference on Autonomic Computing 2005

[14] M. Brodie, S. Ma, L. Rachevsky and J. Champlin. "Automatic Problem Determination Using Call-Stack Matching." Journal of Network and System Management, Vol. 13, No 2, June 2005.

[15] Mathematical statistics with application, K. M. Ramchandrab, Chris P Tsokos, 2009

[16] M. Fischer, M. Pinzger, and H. Gall. 2003. "Populating a Release History Database from Version Control and Bug Tracking Systems". In Proceedings of the International Conference on Software Maintenance (ICSM '03). IEEE Computer Society, Washington, DC, USA.

[17] Mozilla crash reporting server, http://crash-stats.mozilla.com/products/Firefox (accessed March 29, 2011)

[18] N. Bettenburg, S. Just, A. Schr¨oter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in Procs of FSE. ACM, 2008, pp. 308–318.

[19] P. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis" Journal of Computational and Applied Mathematics archive,Volume 20 Issue 1, Nov. 1987

[20] R. K. Yin. Case Study Research: Design and Methods Third Edition. SAGE Publications, London, 2002

[21] Socorro: Mozilla's Crash Reporting System, http://blog.mozilla.com/webdev/2010/05/19/socorro-mozilla-crash-reports/ (accessed March 29, 2011)

[22] S. Nessa, M. Abedin, W. Eric Wong, L. Khan, and Y. Qi, Software Fault Localization Using N-gram Analysis, WASA 2008, LNCS 5258, pp. 548–559, 2008.

[23] Sunghun Kim, Thomas Zimmermann, Nachiappan Nagappan. Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage (Practical Experience Report). In Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011), Hong Kong, China, June 2011.

[24] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2003.

[25] W. Eric Wong and Vidroha Debroy "Software Fault Localization?" IEEE Reliability Society 2009, Annual Technology Report.

[26] W. Le and M. L. Soffa, "Path-Based Fault Correlations," Proceeding of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE '10), SantaFe, New Mexico, USA, November 2010