How to Group Crashes Effectively: Comparing Manually and Automatically Grouped Crash Dumps

Wei Le and Daniel Krutz
Rochester Institute of Technology
One Lomb Memorial Drive, Rochester NY 14623
{*wei.le,dxkvse*}*@rit.edu*

***Abstract*—Crash dumps have become an important source for software developers to learn quality issues in released software. Since a same bug can be repeatedly triggered by different users, an overwhelming number of crash dumps are returned daily. Techniques for automatically grouping crash dumps are mostly based on call stacks captured at crash sites; although fast, they can incorrectly group irrelevant crash dumps or miss related crash dumps. The goal of this paper is to compare manually and automatically grouped crash dumps and discover grouping criteria that are effectively used in manual diagnosis but lacking in automatic tools, from which we potentially enable more precise and capable crash diagnostic tools. In our study, we compared a total of 1,550 groups and 30,431 crash dumps from 5 Mozilla applications. We find that 1) call stacks are more dissimilar from each other in manual groups, as besides matching call stacks, developers frequently link multiple sources, such as reproduce steps and revision histories, for grouping crashes; 2) while automatic tools focus on grouping crashes based on the same root cause, developers also correlate crash dumps across different versions of programs or even different applications for related root causes; 3) both automatic and manual approaches are imprecise, but the two make different types of mistakes; and 4) for small applications, developers' domain knowledge on code works more effectively than the automatic approach for correlating crashes. From the study, we learn that to more effectively group crashes, our future tools should 1) enable multiple criteria and explore diverse uses of crash dump groups for prioritizing and fixing bugs, 2) correlate multiple sources of information or even multiple applications, and 3) uncover inherent relations between symptoms and source code.**

***Keywords*-Crash Dumps, Call Stacks, Grouping, Similarity**

I. Introduction

Due to the complexity, it is impractical to deliver perfect software in a single release. Thus, an important and challenging task during software maintenance is to capture and diagnose failures triggered at the client side. Examples of such feedback systems include Microsoft Dr. Watson [1] and Mozilla Crash Reporter [2]. In these systems, crash dumps, which typically contain call stacks recorded at the crash site, are returned for locating bugs in software. Since a same bug can be repeatedly triggered by different users and under different execution environments, the number of crash dumps sent in daily can reach millions [3]. Incorrectly grouping unrelated crash dumps or failing in identifying a new crash that actually belongs to an already fixed group can incur unacceptable manual overhead and potentially delay critical patches. Thus, it is demanding to find effective criteria for precisely correlating crash dumps, based on which, we then can develop fast and automatic tools.

Grouping crash dumps is challenging because the dumps mostly contain information that indicates dynamic symptoms of a bug, such as call stacks and register values. However, to locate a root cause and introduce a fix, we need to identify the part of the source code that is responsible for the crash. Since same symptoms are potentially caused by different pieces of wrong code or the same piece of problematic code can result in completely different call stacks at the crash, techniques purely based on dynamic symptoms for grouping crashes are insufficient [4], [5], [6], [7].

The goal of this work is to discover criteria and methodologies that are effectively applied in manual diagnosis for grouping crash dumps but lacking in automatic tools; from these, we aim to derive guidelines for designing new and more effective crash diagnostic tools. Specifically, our objectives are to determine 1) beyond the traditional criterion of grouping crash dumps based on the same root cause, whether we can find more types of connections between crash dumps; and 2) to precisely determine a group, what sources of information we should use beyond call stacks.

To achieve the goals, we perform a comprehensive comparison on *m-groups*, manually grouped crash dumps, and *a-groups*, automatically grouped crash dumps. Our dataset consists of a total of 1,550 groups and 30,431 individual call stacks from 5 Mozilla applications. For m-groups, we extract crash dumps from Bugzilla entries that are confirmed as *related* by developers. For *a-groups*, we chose groups automatically generated by Mozilla Crash Reporter [2]. Similar to most automatic tools [4], [5], [6], Mozilla Crash Reporter uses call stack information to determine crash dump groups. These crash dumps are reported from deployed, mature applications such as Firefox and Thunderbird, and the data are publicly available [3].

For comparing a-groups and m-groups, we define the four metrics: 1) what criteria are chosen to correlate the crash dumps? 2) what information is used to derive the groups? 3) is there any imprecision related to the grouping and why?

Table I
SUMMARY OF OUR FINDINGS

Research Questions	Comparison Metric	Automatic	Manual	Implications
Why we group crashes	Grouping criteria	Single: same cause	Multiple: same, related causes, who should fix	Exploit the uses of groups to better prioritize and fix bugs
How we group them	Grouping info	Limited: signatures, call stacks	Multi-sources: black-box + white-box info	Correlate multi-sources info, multi-versions, multi-applications
	Imprecision	Fundamental: based on symptoms	Ad-hoc: incorrectly parse and link info	Design better tools to reveal relations between symptoms and code
What are the capabilities	Call stack characteristics	Scalable: larger, more call stacks	Diverse: dissimilar between call stacks	Grouping based on call stacks is insufficient, especially for small apps

and 4) what are the characteristics of call stacks in a-groups and m-groups? To answer questions (1)–(3), we analyze developers' discussion logs on the 452 Bugzilla entries. These entries display the diagnostic process developers perform to correlate crash dumps from Bugzilla entries as well as ones from Mozilla Crash Reporter. To do the comparisons on (4) above, we automatically compute the sizes of grouped call stacks and also similarity measures between call stacks such as Brodie metrics [8] and longest common substrings.

The contributions of our paper are summarized in Table I. The first column lists the research questions we aim to explore. The second column summarizes a set of comparison metrics we define for answering the research questions. Under *Automatic* and *Manual*, we list a set of observations discovered from analyzing a-groups and m-groups. Under *Implications*, we provide our insights on how to design effective crash diagnostic tools learned from the study. Our findings include:

1) Developers correlate crash dumps not only for the shared root causes, but also for the related causes and for who can fix them. Diagnosis based on a group of crash dumps is sometimes more informative than diagnosis based on a single crash dump;
2) Developers coordinate multiple sources of information, multiple versions of programs and sometimes different applications to determine the groups, which enables many dissimilar call stacks to be correctly grouped;
3) Mistakes in grouping crash dumps could take months to be corrected. Automatic approaches based on dynamic symptoms are fundamentally imprecise, while developers make ad-hoc mistakes. An effective tool should establish precise relations between symptoms and code; and
4) Automatic approach is scalable; however, manual diagnosis is more effective in grouping crash dumps in small applications.

This paper is organized as follows. Section II explains how we collect the dataset and perform the study. Section III presents our comparison results analyzed from 5 Mozilla applications. In Section IV, we summarize the related work, and in Section V, we discuss the limitations of our study. Finally, in Section VI we conclude our work.

II. DATASET AND METHODOLOGIES

In this section, we present how our dataset is constructed and how the information regarding a-groups and m-groups is collected for comparison.

A. Dataset: a-groups and m-groups

Our study subject is *crash dumps*. When a crash occurs at the client side, a crash dump is generated by the deployed crash management system. Typically, it captures the program state at the crash as well as static information about the system and software. For example, a crash dump returned by Mozilla Crash Reporter mainly include: 1) call stacks of each active thread at the crash, 2) exception types captured at the crash such as EXCEPTION_ACCESS_VIOLATION_WRITE, and 3) operating system, software and their versions, as well as the building, installation and crashing dates and time. When a user clicks the *submit crash report* button, the crash dumps are sent back to the server for postmortem analysis. Mozilla Crash Reporter system organizes groups of crash dumps based on applications. For each application, it ranks most frequently occurred crash dump groups within a certain time window.

A user can also send in crash information manually through Bugzilla [9], in which case, the crash information is constructed in ad-hoc based on the users' judgment. Compared to crash dumps generated automatically, the report in Bugzilla may contain additional information, such as 1) reproduce steps that lead to crashes, 2) piece of code that is suspected to cause the crash, 3) the URL visited when the application crashes, 4) expected results, 5) the other bugs that may be relevant, and 6) any connections to groups of crash dumps reported by Mozilla Crash Reporter.

In Figure 1, we show how a-groups and m-groups in our study are constructed. The groups of crash dumps are selected from both Mozilla Crash Reporter and Bugzilla, dated between 2/16-4/16/2012, across five applications including *Firefox*, *Thunderbird*, *Fennec*, *FennecAndroid* and *SeaMonkey*. For a-groups, we collect maximally 300 top crash dump groups for each application.

For m-groups, we inspect the summary page Mozilla presents for each a-group, shown in Figure 2. From the page, we find Bugzilla entries that are correlated to the

a-group, shown in Figure 2. We recursively search each correlated Bugzilla entry to find more Bugzilla entries that are confirmed to be relevant. The crash dumps reported in these Bugzilla entries constitute an m-group. As an example, in Figure 1, b_1 and b_2 are the two Bugzilla entries found relevant to *a-group* and b_3 is identified by recursively inspecting b_1 and b_2 for related bugs. The Bugzilla entries also contain the developers' discussions on how to diagnose the crash dumps.

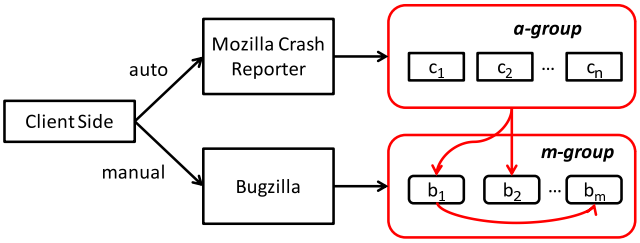


Figure 1. Collect a-Groups and m-Groups

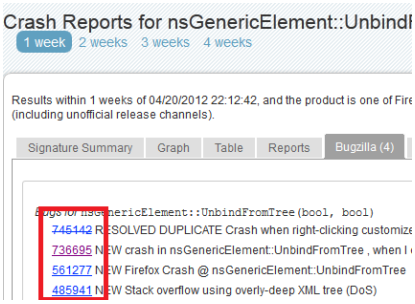


Figure 2. Link a-group and m-group

Table II summarizes the dataset we collected. Under T_g, we display the number of a-groups and m-groups studied for each application. The first four applications contain more than 300 a-groups and by setting a threshold 300, we successfully collected 297 to 299 a-groups. For each application, we randomly select 20 a-groups, based on which we construct m-groups, as explained in Figure 1. Under T_b, we count the total number of Bugzilla entries associated with the 20 m-groups. In each Bugzilla entry, users can report multiple crash dumps related to the bug. Under T_c, we list the total number of crash dumps in the m-groups. The data indicate that on average, each Bugzilla entries contain 2–3 crash dumps.

B. Comparison Metrics and Approaches

Given dataset a-groups and m-groups, our goals are to determine whether manual and automatic approaches apply consistent criteria and information to group crashes, and whether the two approaches correlate same types of call stacks. As shown in Figure 3, we compare the groups based on the four metrics, including:

Table II
DATASET FROM 5 MOZILLA APPLICATIONS

Program	a-Groups		m-Groups		
	T_g	T_c	T_g	T_b	T_c
Firefox 14.0a1	298	18316	20	110	233
Thunderbird 10.0	299	3151	20	106	237
Fennec 2.1.2	299	2567	20	87	155
Fennec Android 13.0a1	297	4925	20	90	189
SeaMonkey 2.7.2	257	514	20	59	144

- *Grouping criteria*: why do we group crash dumps and how the groups can be used?
- *Grouping information*: what information shall we use, and how shall we use it, for more effectively grouping crash dumps?
- *Imprecision in the groups*: when do we group unrelated crash dumps or when do we fail to find the relevant crash dumps?
- *Characteristics of call stacks*: what are the patterns in call stacks in a-groups and m-groups? How do the characteristics imply the capabilities of the two approaches?

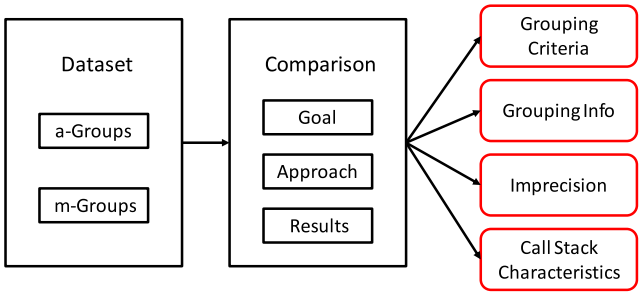


Figure 3. Compare a-Groups and m-Groups

To collect information for performing comparisons, we take the following three steps.

Identifying grouping criteria and information. For a-groups, we studied documentation related to Mozilla Crash Reporter to understand the criteria and algorithms used. To determine grouping criteria and information used for m-groups, we analyzed 452 Bugzilla entries where the m-groups are determined. For learning grouping criteria, our focus is to determine what relations are established between crash dumps in the group and how developers compare and contrast crash dumps in a group for prioritizing, diagnosing and fixing the code. To obtain information important for correlating crash dumps, we define a set of keywords, representing potential sources of information, e.g., *call stacks*, *reproduce steps*, *regression windows*, *URL*, *plugin* and *libraries*. Based on these key words, we extract relevant text from the Bugzilla entries. We then count the frequencies at which each type of information is used and also the frequencies at which different types of information

are combined in determining a group.

Determining imprecision in a-groups and m-groups. In this step, we aim to learn the existence, causes and consequences of imprecision for both of the approaches. To determine imprecision in a-groups, we randomly selected 100 a-groups from our dataset and for each a-group, we analyze the relevant Bugzilla entries that contain diagnostic information for the a-groups. We determine an a-group is imprecise if the developers confirm that the a-group: 1) contains a corrupted signature or call stack, 2) includes crash dumps that should not be correlated, and 3) fails to group crash dumps that should be correlated. Imprecision in m-groups is caused by developers' mistakes. We inspect developers' discussion logs in the Bugzilla entries and identify cases where developers believe unrelated crash dumps are grouped by mistakes.

Comparing call stack characteristics. To compare the capabilities of manual and automatic approaches, we study the patterns in call stacks for a-groups and m-groups. To determine if there is a pattern, we also construct *r-groups*; each of the r-groups contains the random number of crash dumps randomly selected from a-groups. We compare a-groups and m-groups regarding 1) the sizes of groups, including the number of call stacks in each group and the length of the call stacks, and 2) the similarity between call stacks. We compute the metrics in string matching algorithms to measure the similarity between the call stacks, including the *Brodie value* (*B-Value*), *Brodie weight* (*B-Weight*) [5], [8], longest common substrings (*LCS*), and the percentage of identical call stacks in a group (*C-Identical*). In the following, we explain how each of the measure is calculated.

Suppose m is the number of *matching lines* between two call stacks, and l_1 and l_2 are the lengths of the two call stacks. Brodie value bv is computed as:

$$bv = \begin{cases} m/l_1 & l_1 == l_2 \\ m/((l_1 + l_2)/2) & l_1 \neq l_2 \end{cases}$$

We consider a *matching line* between two call stacks if at location i from the top of the call stacks C_1 and C_2, the function names $C_1[i]$ and $C_2[i]$ are identical.

Brodie weight is a string similarity metric improved from the Brodie value. It distinguishes the weight of the functions in call stacks for determining similarity. The assumption is that functions located at the top of the call stack should have more weight than ones located at the bottom. The detailed algorithm of how to compute Brodie weight between two call stacks is given in [5].

We obtain the Brodie value/weight for a group by adding up the Brodie value/weight for every two call stacks in the group and then dividing the times of comparisons. Finally, we get an average across groups for an application.

To compute LCS across all the call stacks in a group, we first detect a set of common substrings between two call stacks. We then determine whether other call stacks in the group contain the same common substrings. The comparison is performed between the average LCS across a-groups and the average LCS across m-groups for an application.

The *C-Identical* identifies the maximum number of crash dumps in a group that are actually identical, calculated by n_i/n. In an a-group or m-group, we may find several subgroups, within which, crash dumps are identical. n_i here is the number of identical call stacks in the largest subgroup and n is the size of the a-group or m-group.

III. Results

In this section, we present our comparison results and our insights for designing better grouping tools.

A. Comparison based on Grouping Criteria

RQ1: Why do we group crash dumps?

1) Besides grouped based on the same root cause, crash dumps can also be correlated if the patch to one crash is the cause for another crash.
2) Developers also group crash dumps if they can be conveniently fixed together, for example, by a same developer.
3) Diagnosing a group of crash dumps can be beneficial: first, a group of symptoms help determine general manifestation of a bug; second, a group of similar call stacks potentially enable automatic bug localizations.

Table III
Grouping Criteria for m-Groups

Grouping Criteria	Goals
Same Root Cause	Fix one to fix all in the group
	Determine if a given new crash is fixed
	Localize bugs via comparing similar stacks
	Learn bug manifestation to prioritize them
Related Root Cause	Localize root causes if fix to previous crashes is the cause for the current crash
Who Fix the Bug	Find experts who can fix a group of bugs from the same code region or of same types

For automatic approaches that compare call stack similarity for grouping crash dumps, the implied grouping criterion is that the crash dumps in a group should share the root cause. However, we find that the grouping criteria applied in manual diagnosis are more diverse and flexible.

In Table III, we summarize our discoveries. Similar to automatic approaches, in many cases, developers correlate crash dumps if they believe these crashes are originated from the same bug in the code. In another word, if we introduce a fix based on any crash in the group, we can fix all the crash dumps grouped, and if a new crash dump is determined to belong to the group, we do not need to further diagnose it.

Interestingly, although the goal of grouping crash dumps is to avoid repeatedly diagnosing crash dumps caused by the same root cause, developers sometimes still compare

multiple crash dumps in the same group for determining the types and locations of a bug. It indicates that a group of crash dumps can be more informative than individual crash dumps in helping diagnose failures. For example, a same bug may cause completely different symptoms, and we find a case where developers prioritize a group of crash dumps because one of the crashes have the serve symptom related to a security attack.

As shown in the table, developers also correlate crash dumps if the root causes that lead to the crashes have a temporal or locality relationship. In our study, we find cases where one crash is caused by an incomplete/incorrect fix to the other crashes. Developers believe that correlating these crashes can help quickly identify the cause and the fix for the newly generated crashes.

In addition, developers group crash dumps using the criterion of who can fix them. Here, the components where the crashes occur, together with the bug types, are used to determine the groups. The goal is to better distribute the debugging tasks to people who are most familiar with the code and the bug types.

B. Comparison based on Grouping Information

RQ2: What information shall we use to group crashes?

1) Call stacks, build information and reproduce steps are the three main sources for developers to group crashes.
2) Developers frequently correlate multiple sources of information for determining the groups. For example, developers once linked call stacks and reproduce steps by comparing call stacks in crash dumps with the call stack in a non-crash thread.
3) Developers correlate crash dumps across applications to determine if the root cause is located in the shared libraries or plugins.
4) Different versions of programs can contain the same piece of problematic code, and thus we should enable the grouping across different versions of the code.

Mozilla Crash Reporter automatically groups crash dumps by matching 1) the version of software where the crash dumps are originated; and 2) the function call on the top of the call stack at the crash, called *signature*. On the contrarily, developers use a variety of information. No systematic way is applied to choose which source of information should be used for determining a particular group of crash dumps.

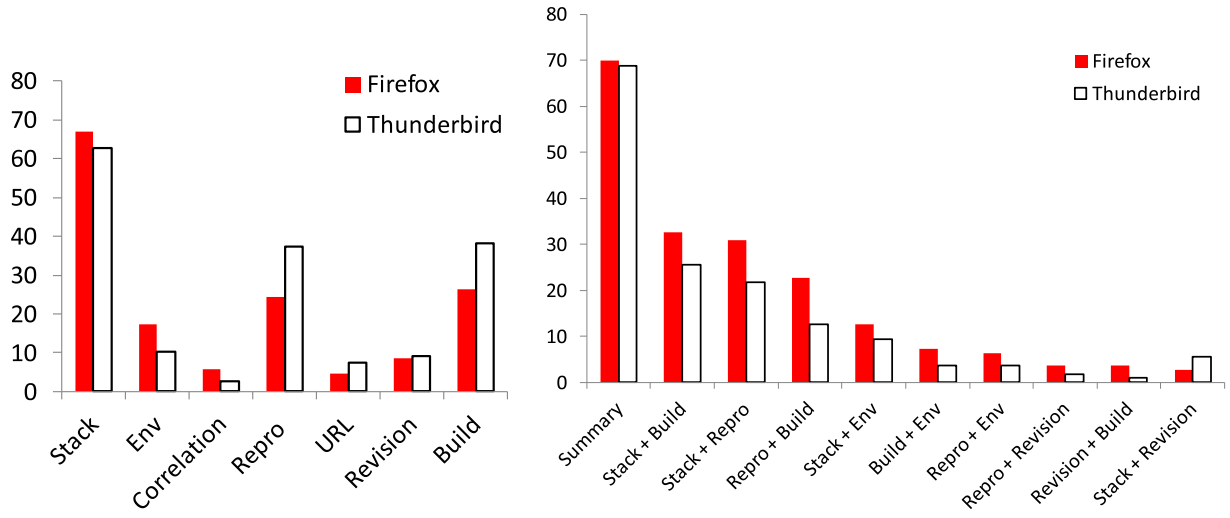
We studied a total of 40 m-groups from Firefox and Thunderbird. We find that developers generally apply three types of information for grouping crash dumps: 1) white-box information, i.e., crash symptoms related to code including call stacks and their signatures, 2) black-box testing information, e.g. steps on triggering the bug, and 3) information related to software versions, such as build time and revision histories.

We summarize our analysis results in Figure 4. On the left, we rank a list of information source developers frequently use in determining crash dump groups, and on the right we show the correlation of these sources applied in manual diagnosis. The y-axis presents the percentage of crash dumps that are grouped using the specific source(s) of information.

In Figure 4(a), along the x-axis, *Stack* represents call stacks. *Env* indicates the plugins and libraries involved in the crash. *Correlation*, used by Mozilla Crash Reporter, specifies how often a signature and a .dll component are simultaneously witnessed in a crash. For example, given a set of call stacks, we count that a certain signature occurs 10 times, and among the 5 times, a specific .dll also occurs; in this case, the correlation ratio between the signature and the component is 50%. *Repro* and *URL* indicate two relevant types of testing information. *Repro* represents the steps taken to reproduce the bug and *URL* refers to a list of URL visited before an application crashes. *Revision* is a list of changes made to the application shortly before a crash. Finally, *Build* gives the date and time to indicate when the application is built and installed, as well as the version of operating system where the application runs.

Figure 4(a) shows that call stack, build information and reproduce steps are the three most frequently used sources. For Firefox, 67% of the crash dumps were grouped using call stacks, 26.4% applied build information and 24.5% were determined using reproduce steps. Similarly, for Thunderbird, 62.7% used call stacks, 38.2% used the build information and 37.3% applied reproduce steps. The secondary important source is *Revision*, based on which, 8.5% of the Firefox crash dumps and 9.1% of the Thunderbird crash dumps are grouped. By inspecting m-groups determined using *Revision* information, we find that when crashes occur shortly after a new release of an application, plugin or operating system, developers give the priority to consider whether the crash dumps can be caused by the same bug in the new code and thus can be grouped together. Sometimes, the developers suspect that the crash dumps are caused by bugs in a certain plugin or operating system, and thus they correlate crash dumps from different applications running with the same plugin or operating system.

In Figure 4(b), we show that developers often make decisions by coordinating multiple sources of information. The leftest bars in the figure indicate that 70% of the crash dumps from Firefox and 68.8% from Thunderbird are grouped by using more than one source of information. Consistent with Figure 4(a), developers most frequently link call stacks, build information and reproduce steps to determine the groups. As shown in Figure 4(b), the top three combined-sources include 1) call stack with build information, 2) call stack with reproduce steps and 3) reproduce steps with build information. Correlating Figure 4(a) and 4(b), we can obtain interesting findings. For example, we can derive that 67% of the Firefox crash dumps are grouped using call stacks, among which, about a half are used with the reproduce steps and a half are used with build information. The three of the



(a) List of Information Source (b) Linking Multiple Sources

Figure 4. Information Used for Manually Grouping Crash Dumps

sources might be used together in some cases.

Our inspection in m-groups shows that developers often use build information as the first step to reduce the scope of the grouping. However, grouping crash dumps of the same version sometimes is not ideal because different versions of a program may contain the same problematic code and any crash dumps caused by this code should be grouped.

We also discover that coordinating call stacks and reproduce steps can be challenging. We find a case where developers compare a crash dump with call stacks in non-crash threads to determine what steps in testing can produce a specific sequence of calls in the call stacks.

C. Comparison based on Imprecision

RQ3: when do we group unrelated crash dumps and when do we fail to find the relevant crash dumps?

1) Imprecision in m-groups is caused by developers' mistakes, and sometimes, a simple mistake can take months to recover.
2) Grouping purely based on the similarity of call stacks is insufficient; especially some applications can generate very dissimilar call stacks at the crash.

We first show our results on imprecision in m-groups. In Table IV, we list the four examples confirmed as developers' mistakes. Under *Bug ID*, we list the identifications of the Bugzilla entries where the mistakes are found. Under *Developers' Mistakes*, we give the descriptions of the mistakes. Under *Time*, we show the time period from when the mistake is firstly introduced to the Bugzilla post to when the developers confirm the problems. In the first two cases, developers misread the call stacks and mismatched the code. In the third case, developers only used signatures, the approach implemented in Mozilla Crash Reporter, rather than performed a more depth analysis to determine the group. In the fourth case, developers made a wrong judgment and believed the crash is a regression of a previous a bug. The results show that these mistakes can be difficult to discover, and the time that takes to find the problems is not always proportional to how complicated a mistake is about. The implication of this result is that we need to develop automatic tools to help avoid these simple but expensive mistakes from developers.

Table IV
IMPRECISION IN M-GROUPS: EXAMPLE MISTAKES

Bug ID	Developers' Mistakes	Time
716232	Mismatch call stacks	4.6 hours
695505	Inspect wrong version of code	3.0 months
524921	Match signature only	2.7 days
703133	Incorrectly link to a patch	4.7 days

Table V
IMPRECISION IN A-GROUPS

Total	Corrupted Stack	Group Unrelated	Fail to Group Related
100	4	2	4

In Table V, we present the set of data that demonstrate the imprecision in a-groups. Our approach is to inspect the Bugzilla entries that document the manual diagnosis for the a-groups and find imprecision in a-groups confirmed by developers. Under *Total*, we show that we studied the Bugzilla entries related to a total of 100 a-groups in our dataset. Under *Corrupted Stack*, *Group Unrelated*, and *Fail to Group Related*, we list the number of instances for the three types of imprecision: 1) call stacks are corrupted at the crash and thus the signature returned is no longer the top function where the actual failure occurs; 2) crash dumps

Call Stack 1	Call Stack 2	Match
_SEH_prolog	InternalCallWinProc UserCallWinProcCheckWow	×
CallWindowProcAorW	CallWindowProcAorW	✓
CallWindowProcW	CallWindowProcW	✓
mozilla::plugins::PluginInstance Child::PluginWindowProc	mozilla::plugins::PluginInstance Child::PluginWindowProc	✓
InternalCallWinProc	InternalCallWinProc	✓

Figure 5. Same Cause, under Different Versions of OS, result in Different Signatures

grouped in an a-group are actually irrelevant; and 3) crash dumps of the same/related root causes are not grouped in the same a-group.

In our study, we find that all of the three types of imprecision exist in a-groups. The actual instances may even be higher than the ones listed in the table, as developers may only discover a small portion of such problems. In the following, we present two examples discovered during inspecting these mistakes. The examples indicate that it is neither sound nor complete to group crash dumps only based on the equivelance of the signatures or the similarity of the call stacks.

In Figure 5, the two crash dumps are generated from the same cause in the same version of the program. However, because the crashes are triggered in different versions of Windows, the crash dumps contain different signatures (see the first row in the table) and thus were not grouped by Mozilla Crash Reporter.

In Figure 6, we show that by only comparing the similarity between call stacks, we can fail to distinguish a legitimate and illegal correlation between the call stacks. On the left in Figure 6, the two call stacks have the same signatures and 19 out of 26 calls in the first call stack have appeared in the second call stack. On the right of the figure, the two call stacks also contain the same signatures; however, only 10 out of 30 calls in the first call stack have appeared in the second. In fact, developers confirm that the first pair have irrelevant root causes, while the second pair should be correlated. Any techniques using call stack similarity for grouping crash dumps could fail to correctly group the call stacks in the two cases.

D. Comparison based on Call Stack Characteristics

RQ4: Does automatic and manual approaches group a similar set of crash dumps?

1) Automatic approach is scalable in that it can correlate hundreds of crash dumps and group stacks with thousands of calls; however, the developers' knowledge about the code is more effective in grouping crash dumps in small applications.
2) Call stacks in an m-group are more dissimilar from each other than ones in an a-group, suggesting manual diagnosis can group more varieties of crash dumps that are not able to be grouped by automatic approaches.
3) Call stacks correlated in a-groups and m-groups are more similar among each other than the ones randomly grouped, suggesting call stack similarity can be an indicator of sharing a same or related cause and can be used as one factor to determine groups of crash dumps.

In this section, we present our results on comparing sizes of grouped crash dumps and the similarity of call stacks in the groups. From the characteristics of grouped call stacks, we aim to find implications on capabilities of the two approaches in grouping crash dumps.

In Table VI, under G_{max}, G_{min} and G_{ave}, we report the maximum, minimum and average numbers of call stacks in the a-groups and m-groups under study. Under L_{max}, L_{min} and L_{ave}, we show the maximum, minimum and average lengths of the call stacks that are grouped. Comparing the data under G_{max} and G_{ave}, as well as L_{max} and L_{ave} for the a-groups and m-groups, we find that automatic approach is more scalable; it groups more crash dumps and the crash dumps handled are generally much larger.

An exception is the smallest program *SeaMonkey*. The automatic approach is only able to correlate 2 crash dumps on average in a group, suggesting that the signatures of call stacks are different among most of the crash dumps. On the other hand, manual diagnosis is able to construct groups containing 7 crash dumps on average. By inspecting developers' discussion logs on constructing m-groups, we found that the developers' domain knowledge on code and revision histories play an important role in grouping dissimilar call stacks. For small applications, the number of bugs is limited and thus the number of crash dumps are relatively low. Therefore, it is easier to find the problem that automatic approaches fail to group related but dissimilar call stacks. Considering the number of groups for small applications still reach hundreds, we need more effective automatic approaches that can group dissimilar call stacks.

In Table VI, under L_{min}, we see that the minimum length of call stacks in some a-groups can be as low as 2–4. We manually inspect these call stacks and found these are corrupted stacks, an imprecision source in a-groups discussed before.

The comparison on similarity of call stacks is shown in Table VII. Under *B-Value*, *B-Weight*, *LCS* and *C-Identical*, we list the data collected for the four similarity measures discussed in Section II. For all the data, the larger values indicate the more similar among the call stacks within the groups. The data computed for a-, m- and r-groups are displayed under *a-*, *m-* and *r-* respectively.

Comparing columns of *a-* and *m-*, we find that the four similarity measures consistently imply crash dumps in m-groups is more dissimilar from each other than the ones in a-groups. Thus, we should include criteria and sources of information used in manual diagnosis but still lacking in automatic tools for more effectively grouping the crash

Match 19/26, Different Causes		
js_DestroyScriptsToGC	js_DestroyScriptsToGC	✓
thread_purger	PurgeThreadData	×
...	...	...
XRE_main	XRE_main	✓
main	CloseHandle	✓

Match 10/30, Same Causes		
JSObject::nativeSearch	JSObject::nativeSearch	✓
js::LookupPropertyWithFlags	js_LookupProperty	×
...	...	...
nsINode::DispatchEvent	@0xffffff81	×
nsContentUtils::DispatchTrustedEvent	nsArrayCC::Release	×

Figure 6. Call Stack Similarity Fail to Distinguish Legitimate and Illegal Groups

Table VI
SIZES OF CRASH DUMP GROUPS AND CALL STACKS IN M-GROUP AND A-GROUP

Program	a-Groups						m-Groups					
	G_{max}	G_{min}	G_{ave}	L_{max}	L_{min}	L_{ave}	G_{max}	G_{min}	G_{ave}	L_{max}	L_{min}	L_{ave}
Firefox 14.0a1	8645	13	61.5	1854	46	1925.5	39	3	11.7	1035	61	412.2
Thunderbird 10.0	355	2	10.5	8864	6	333.6	33	3	11.9	1024	93	376.1
Fennec 2.1.2	2233	1	8.6	13832	2	415.5	24	3	7.8	1456	46	306.2
Fennec Android 13.0a1	993	2	16.6	12697	4	659.2	24	3	9.5	1456	82	360.2
SeaMonkey 2.7.2	15	1	2.0	686	2	76.5	16	3	7.4	596	6	237.9

dumps.

The data under *LCS* and *C-Identical* indicate that the similarity of call stacks can also differ based on applications. For example, among the five applications, Thunderbird reports the maximum LCS and C-Identical values for a-groups. Firefox which contains many JavaScript related crash dumps show lower LCS and C-Identical values. We find that in general, crash dumps involved with JavaScript modules are more dissimilar from each other, although the dumps are confirmed to share the root causes. The rationale is that it takes very different paths to trigger the failures, implying 1) function calls in the JavaScript modules are generally small, and/or 2) the bugs that lead to the crashes are interprocedural. This finding suggests that for different applications, we cannot apply a fixed threshold on call stack similarity in determining groups of crash dumps.

Compared the data under *a-*, *m-* and *r-*, we learn that correlated crash dumps generally contain more similar call stacks than the ones randomly grouped. Thus, although it is imprecise to only use call stack similarity to group crash dumps, call stack similarity is still a valuable source of information for determining correlations between call stacks.

E. Discussions

We have inspected a few a-groups to determine if there exist interesting patterns in call stacks that potentially help quickly diagnose the bugs. In one group, we find a sequence of calls repeatedly invoked on the call stacks, indicating the potential problems in recursive calls. In another group, two call stacks contain a few different calls, but invoked on the same object, shown in Figure 7. The diagnosis thus should start with comparing the different calls `js::types::TypeSet::addCall`, `js::types::TypeSet::addSubset` and `js::types::TypeSet::addArith` and also inspecting the state of the object `js::types::TypeSet` under these calls. We also find patterns where the call stacks in the group contain a similar set of function calls; however, the orders in which the functions are invoked on the stacks are different in each call stack.

IV. RELATED WORK

In this section, we present the related work in grouping and diagnosing crash dumps. The work we found for grouping crash dumps is mostly based on the similarity of call stacks. The motivation is to match a given failure in the problem database[5], [6], [8], [10].

Bartz et al. [4] applied a machine learning similarity metric for grouping Windows failure reports. This is done using information from clients when the users describe the symptoms of failures. The primary mechanism for measurements is an adaptation of the Levenshtein edit distance process, which is deemed to be one of the less costly string matching algorithms [11].

Lohman et al. [10] developed the techniques of normalizing strings based on length before comparing them. They applied metrics commonly used in string matching algorithms, including *edit distance*, *longest common subsequence* and *prefix match*.

Brodie et al. [5], [8] proposed that similar bugs are likely to produce stacks which resemble one another. To determine if a new failure is originated from the same cause documented in the database, they developed the metrics of Brodie weight for determining similarities between call stacks. The idea is that when measuring similarity, a higher weight is placed upon items that match between the top of two stacks. The assumption is that the closer to the top of a stack a function call is, the more relevant it is to the matching process [8].

Table VII
SIMILARITY OF CALL STACKS IN A-GROUPS AND M-GROUPS

Program	B-Value			B-Weight			LCS			C-Identical		
	a-	m-	r-	a-	m-	r-	a-	m-	r	a-	m-	r-
Firefox	1.01	0.04	0.11	0.91	0.02	0.01	4.36	0.55	0	0.54	0.15	0.08
Thunderbird	0.62	0.05	0.35	0.53	0.04	0.04	14.01	0.60	0	0.71	0.16	0.22
Fennec	0.52	0.03	0.12	0.53	0.03	0.04	8.84	7.05	0	0.33	0.24	0.13
FenecAndroid	0.66	0.07	0.17	0.76	0.05	0.04	10.05	9.75	0	0.40	0.31	0.15
SeaMonkey	0.04	0.03	0.25	0.03	0.03	0.02	4.82	1.85	0	0.33	0.30	0.14

Call Stack 1	Call Stack 2	Call Stack 3	Match
js::types::TypeSet::add	js::types::TypeSet::add	js::types::TypeSet::add	✓
js::types::TypeSet::addArith	js::types::TypeSet::addSubset	js::types::TypeSet::addCall	×
js::analyze::ScriptAnalysis::analyzeTypesBytecode	js::analyze::ScriptAnalysis::analyzeTypesBytecode	js::analyze::ScriptAnalysis::analyzeTypesBytecode	✓
js::analyze::ScriptAnalysis::analyzeTypes	js::analyze::ScriptAnalysis::analyzeTypes	js::analyze::ScriptAnalysis::analyzeTypes	✓
JSScript::ensureRanInference	JSScript::ensureRanInference	JSScript::ensureRanInference	✓

Figure 7. Interesting Patterns in Call Stacks

Besides the above work in grouping crash dumps using call stack similarity, Kim et al.[7] developed crash graphs to aggregate a set of crash dumps into a graph, which demonstrated to be able to more efficiently identify duplicate bug reports and predict if a given crash will be fixed.

We also find related work which aims to prioritize and reproduce crash dumps. Kim et al. [12] proposed that we should focus on diagnosing top crashes, as the observations on Firefox and Thunderbird show that 10 to 20 top crashes account for above 50 % of total crash reports. They develop a machine learning technique to predict the top crashes in new releases based on the features of top crashes in the past releases.

Artzi et al. [13] developed techniques for creating unit tests for reproducing crash dumps. The approach consists of monitoring phase and test generation phase. The monitoring phase stored copies of the receiver and arguments for each method and the test generation phase restores the method and arguments.

The work on crash dumps also includes empirical studies. Dhaliwal et al.[14] found that it takes longer to fix bugs when the group contains crashes caused by multiple bugs. Schroter [15] et al. concluded that stack traces in the bug reports indeed help developers fix bugs.

V. LIMITATIONS

The limitations of our study are threefold. First, for a-groups, we mainly studied groups of crash dumps automatically generated from Mozilla applications, as the data are publicly available. Although we believe Mozilla Crash Reporter implements the state-of-the-art automatic techniques, some of our conclusions drawn based on the Mozilla system may not be generalized for all existing automatic grouping techniques. Second, some of our results are obtained from analyzing Bugzilla entries. Information documented in Bugzilla can be ambiguous and our interpretation for the information can be imprecise. Third, some of the similarity measures we applied may not accurately reflect the similarity in grouped crash dumps and thus we applied multiple metrics. For example, in Table VII, we found that the r-groups report larger Brodie values than the m-groups. The reason is that r-groups constructed from call stacks in a-groups contain much larger crash dumps than ones in m-groups. Brodie weight is a more accurate measure for comparing call stacks with different sizes.

VI. CONCLUSIONS

This paper studies manual and automatic approaches in grouping crash dumps. We compare the two approaches regarding the grouping criteria, the information used for grouping the crash dumps, the imprecision in the groups, as well as characteristics of grouped call stacks. We find that automatic approaches purely based on call stack similarity are not sufficient, especially for small applications. To more effectively group the crash dumps, we need to: 1) design grouping criteria that can enable better uses of the group information in diagnosing failures; 2) correlate multiple sources of information and connect related applications; 3) design tools that can establish precise relations between symptoms and code.

REFERENCES

[1] Microsoft Dr. Watson, https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx?mfr=true, 2012.

[2] Mozilla Crash Reporter, https://support.mozilla.org/en-US/kb/Mozilla\%20Crash\%20Reporter, 2012.

[3] Crash Dumps for Mozilla Applications, https://crash-stats.mozilla.com/products/, 2012.

[4] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle, "Finding similar failures using callstack similarity."

[5] M. Brodie, S. Ma, L. Rachevsky, and J. Champlin, "Automated problem determination using call-stack matching." *J. Network Syst. Manage.*, 2005.

[6] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems," in *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*, 2007.

[7] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, 2011.

[8] M. Brodie, S. Ma, G. M. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn, "Quickly finding known software problems via automated symptom matching." in *ICAC'05*, 2005.

[9] Bugzilla, http://www.bugzilla.org/, 2012.

[10] G. Lohman, J. Champlin, and P. Sohn, "Quickly finding known software problems via automated symptom matching," in *Proceedings of the Second International Conference on Automatic Computing*, 2005.

[11] G. V. Bard, "Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric," in *Proceedings of the fifth Australasian symposium on ACSW frontiers - Volume 68*, 2007.

[12] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung, and S. Park, "Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *Software Engineering, IEEE Transactions on*, 2011.

[13] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *Proceedings of the 22nd European conference on Object-Oriented Programming*, ser. ECOOP '08, 2008.

[14] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of mozilla firefox," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011.

[15] A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 2010.