An Empirical Analysis of Crash Dump Grouping

Daniel E. Krutz and Samuel Malachowsky

Rochester Institute of Technology, Rochester NY 14623, USA
{dxkvse, samvse}@rit.edu

Abstract. Software developers are involved in a constant struggle to produce high quality software. Crash dumps are an important resource for gathering invaluable information about software defects in the field. 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. In our study, we compared a total of 1,550 groups and 30,431 crash dumps from 5 Mozilla applications. We found that 1) call stacks are more dissimilar from each other in manual groups, in addition to matching call stacks, developers frequently use multiple resources, such as reproduction 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. From this study, we determined that to more effectively group crashes, 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

1 Introduction

Due to the complexity of applications, it is impractical to expect perfect software on the initial 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's Dr. Watson [14] and Mozilla Crash Reporter [16]. In these systems, crash dumps which typically contain call stacks recorded at the crash site, are used for locating bugs in software. Since the same bug can be repeatedly triggered by different users and under different execution environments, the number of crash dumps submitted daily can reach millions [6]. 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 imperative to find effective criteria for precisely correlating crash dumps, from which we can use to develop fast and automatic tools.

Grouping crash dumps is challenging because the dumps primarily 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 the 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 [3, 4, 11, 15].

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 determine what sources of information should be used beyond call stacks.

We performed 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 [16]. Similar to most automatic tools [3, 4, 15], 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 [6].

For comparing a-groups and m-groups, we define 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? and 4) what are the characteristics of call stacks in a-groups and m-groups? To answer these questions, we analyzed developer 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 conduct 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 [12] and longest common substrings. We found that:

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.

This paper is organized as follows: Section 2 explains how we collect the dataset and perform the study. Section 3 presents our comparison results analyzed from 5 Mozilla applications. In Section 4, we summarize the related work, and in Section 5, we discuss the limitations of our study. Section 6 concludes our work.

2 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.

2.1 Data Collection Process

In order to collect the necessary data for our study, we built a collection and analysis tool called Mozilla Call Stack Grouper (MCSG) which consisted of several stages. The initial step of MCSG is the data collection phase where the user first selects a group of crash stacks to be analyzed, which may be for a specific Mozilla application and version. For example, the user may choose to analyze all call stacks for Mozilla Firefox 46.0a1. MCSG will then pull the *top crasher* call stacks for the specified application and version from the Crash Stats Mozilla website[1]. A *top crasher* is a crash type with one of the highest number of crash reports on the Mozilla website [7].

The data collection process begins by gathering the call stacks and developer discussion logs for these top crashers, which is done using HttpUnit[2]. Each of the collected call stacks are stored locally in a .txt file in the folder with all the other call stacks from the same top crasher group. Once all of the folders and call stacks have been collected, the comparison process may begin. There are several existing techniques for measuring call stack similarity [3, 15]. MCSG measures call stack similarity using the call stack similarity metrics as defined in these works.

Some of the primary challenges which needed to be overcome in the creation of MCSG was the ability to accurately collect the necessary information, especially with the format of the crash source frequently changing. A second significant problem to be overcome was the implementation of the crash comparison mechanisms. While the implementation of the comparison algorithms was a fairly straightforward task, the number of comparisons and evaluations which needed to be completed was very large, and therefore creating an analysis mechanism which would operate accurately and efficiently was a challenging task. We spent a large amount of time optimizing our comparison algorithm and verifying its accuracy.

2.2 Dataset: a-groups and m-groups

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(MCR) primarily includes: 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.

[1] https://crash-stats.mozilla.com/
[2] http://httpunit.sourceforge.net

MCR 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 [5], 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) code segment 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.

Figure 1 demonstrates 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 across five applications including *Firefox*, *Thunderbird*, *Fennec*, *FennecAndroid* and *SeaMonkey*. For a-groups, we collected 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 this 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 developer discussions on how to diagnose the crash dumps.

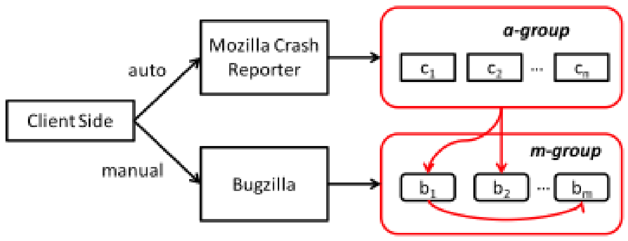
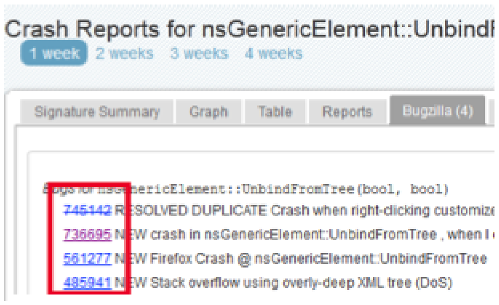


Fig. 1. Collect a-Groups and m-Groups

Table 1. Dataset from 5 Mozilla Applications

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



ericElement::UnbindFromTree(bool, bool)

~~745142~~ RESOLVED DUPLICATE Crash when right-clicking customiz

736695 NEW crash in nsGenericElement::UnbindFromTree , when I

561277 NEW Firefox Crash @ nsGenericElement::UnbindFromTree

485941 NEW Stack overflow using overly-deep XML tree (DoS)

Fig. 2. Link a-group and m-group

Table 1 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.

2.3 Comparison Metrics and Approaches

Using our dataset of a-groups and m-groups, our goal is 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:

- *Grouping criteria*: How can 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?

The following steps are used to collect information for performing comparisons:

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

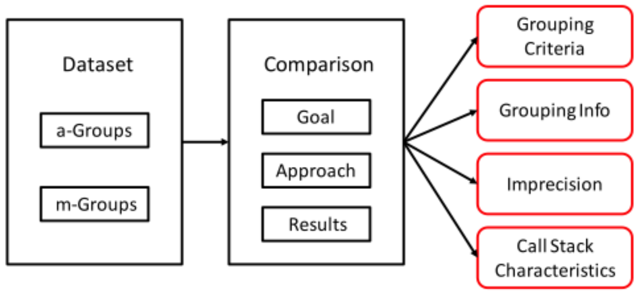


Fig. 3. Compare a-Groups and m-Groups

crash dumps, we defined a set of keywords, representing potential sources of information, e.g., *call stacks*, *reproduce steps*, *regression windows*, *URL*, *plugin* and *libraries*. These keywords were chosen due to our observed frequency in crash data. Based on these keywords, 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 developer error. We inspect developer 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*) [4, 12], 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 [4].

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.

3 Results

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

3.1 Comparison based on Grouping Criteria

RQ1: How can we group crash dumps?

Our first research question was how crash dumps are grouped to associate related crashes with one another. 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 2, 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 still frequently compare multiple crash dumps in the same group for determining the types and locations of a bug. This 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

Table 2. 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 Fixed the Bug	Find experts who can fix a group of bugs from the same code region or of same types

attack. As shown in the Table 2, 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. 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.

In summary to our research question about crash dump grouping, we found that:

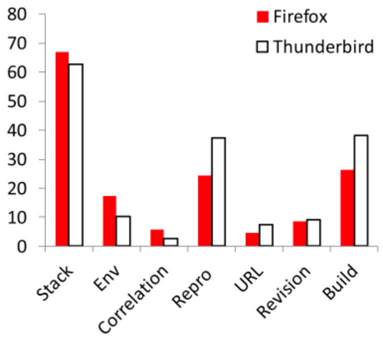
1. Besides grouped based on the same root cause, crash dumps can also be correlated if the patch to one crash is the cause for another crash.
2. Diagnosing a group of crash dumps can be beneficial: first, a group of symptoms help determine general manifestation of a bug; second, a group of similar call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

3.2 Comparison based on Grouping Information

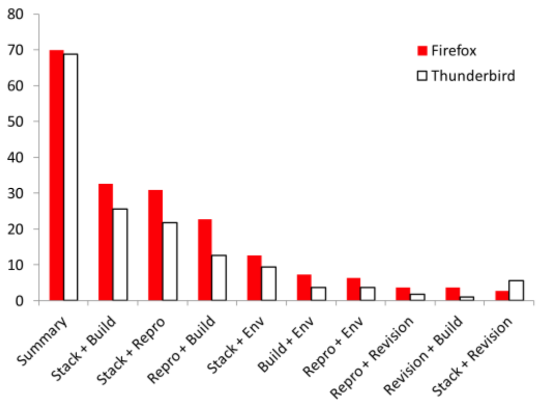
RQ2: What information shall we use to group crashes?

We next sought to determine the best information to effectively and efficiently group crashes. 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 contrary, 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 and found 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



(a) List of Information Source



(b) Linking Multiple Sources

Fig. 4. Information Used for Manually Grouping Crash Dumps

presents the percentage of crash dumps that are grouped using the specific source(s) of information.

In Figure 4(a) 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 reproduction 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 left 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), demonstrates interesting findings. For example, we can derive that 67% of the Firefox crash dumps are grouped using call stacks.

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 is often 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 found 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.

In summary to our research question about the information used to group crashes, we found that:

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.

3.3 Comparison based on Imprecision

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

For our third research question, we analyzed common mistakes in crash dump grouping. We first show our results on imprecision in m-groups. In Table 3, we list the four examples confirmed as developer mistakes. Under *Bug ID*, we list the identifications of the Bugzilla entries where the mistakes are found. Under *Developer 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 the complexity of the mistake. The implication of this result is that we

Table 3. Imprecision in m-Groups: Example Mistakes

Bug ID	Developer 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 4. Imprecision in a-Groups

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

need to develop automatic tools to help avoid these simple but expensive mistakes from developers.

In Table 4, 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 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 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 based only on the equivalence 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.

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	✓

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

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.

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	×

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

In summary to our research question about common mistakes in crash grouping, we found that:

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

3.4 Discussions

We have inspected several a-groups to determine if there are interesting patterns in call stacks that may potentially help to 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. As shown in Figure 7, in another group two call stacks contain a few different calls, but are invoked on the same object. 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 order in which the functions are invoked on the stacks are different in each call stack.

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

Fig. 7. Interesting Patterns in Call Stacks

4 Related Work

There is a wide variety of existing work on crash dump grouping and analysis. Bartz et al. [3] 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 [2]. Han et al. [9] created StackMine to mine callstack traces to discover high impact performance bugs. Subsequent works have used a variety of mechanisms, such as genetic algorithms [18] to improve call stack analysis.

There has also been substantial work done on how to better collect and organize call stacks. Wu et al. [19] created Casper, a LL(1) grammar model based tool to represent all possible call traces and a grammar model to represent every possible logged call trace. They found that collecting call traces with minimal implementation was an NP-hard problem, and proposed an efficient alternative approach for collecting suboptimal results. Glerum et al. [8] reported on their experiences using the Windows Error Reporting (WER) tool. This work described Microsoft's error collection system for the Windows OS including how the collected data is grouped into 'buckets' and how the system helps to prioritize developer effort, recognize error trends, and find related defects.

Lohman et al. [13] 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. [4, 12] 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 [12].

Kim et al. [11] developed crash graphs to aggregate a set of crash dumps into a graph, which was shown to be able to more efficiently identify duplicate bug reports and predict if a given crash will be fixed. We also found related work which aims to prioritize and reproduce crash dumps. Kim et al. [10] 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. [1] developed techniques for creating unit tests for reproducing crash dumps. The approach consists of a 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. [7] found that it takes longer to fix bugs when the group contains crashes caused by multiple bugs. Schroter et al. [17] concluded that stack traces in the bug reports indeed help developers fix bugs.

5 Limitations

Although our findings are profound, they are not without their limitations. First, for a-groups, we mainly studied groups of crash dumps automatically generated from Mozilla applications, as the data is 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.

6 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. 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*, ECOOP ’08, 2008.
2. 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.

3. K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. Finding similar failures using callstack similarity. In *Proceedings of the Third conference on Tackling computer systems problems with machine learning techniques*, SysML'08, pages 1–1, Berkeley, CA, USA, 2008. USENIX Association.
4. M. Brodie, S. Ma, L. Rachevsky, and J. Champlin. Automated problem determination using call-stack matching. *Journal of Network and Systems Management*, 13(2):219–237, 2005.
5. Bugzilla. http://www.bugzilla.org/, 2012.
6. Crash Dumps for Mozilla Applications. https://crash-stats.mozilla.com/products/, 2012.
7. T. Dhaliwal, F. Khomh, and Y. Zou. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 333–342, Washington, DC, USA, 2011. IEEE Computer Society.
8. K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.
9. S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 145–155, Piscataway, NJ, USA, 2012. IEEE Press.
10. 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.
11. 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.
12. 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*, ICAC '05, pages 101–110, Washington, DC, USA, 2005. IEEE Computer Society.
13. 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.
14. Microsoft Dr. Watson. https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx?mfr=true, 2012.
15. 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.
16. Mozilla Crash Reporter. https://support.mozilla.org/en-US/kb/Mozilla\%20Crash\%20Reporter, 2012.
17. 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.
18. D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 270–281, New York, NY, USA, 2015. ACM.
19. R. Wu, X. Xiao, S.-C. Cheung, H. Zhang, and C. Zhang. Casper: An efficient approach to call trace collection. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 678–690, New York, NY, USA, 2016. ACM.