Improving Bug Localization using Correlations in Crash Reports

Shaohua Wang
School of Computing
Queen's University
Kingston, ON, Canada
shaohua@cs.queensu.ca

Foutse Khomh
SWAT Lab, DGIGL
École Polytechnique de Montréal
Montréal, QC, Canada
foutse.khomh@polymtl.ca

Ying Zou
Electronical and Computer Engineering
Queen's University
Kingston, ON, Canada
ying.zou@queensu.ca

***Abstract*—Nowadays, many software organizations rely on automatic problem reporting tools to collect crash reports directly from users' environments. These crash reports are later grouped together into crash types. Usually, developers prioritize crash types based on the number of crash reports and file bugs for the top crash types. Because a bug can trigger a crash in different usage scenarios, different crash types are sometimes related to a same bug. Two bugs are correlated when the occurrence of one bug causes the other bug to occur. We refer to a group of crash types related to identical or correlated bugs, as a crash correlation group.**

In this paper, we propose three rules to identify correlated crash types automatically. We also propose an algorithm to locate and rank buggy files using crash correlation groups. Through an empirical study on Firefox and Eclipse, we show that the three rules can identify crash correlation groups with a precision of 100% and a recall of 90% for Firefox and a precision of 79% and a recall of 65% for Eclipse. On the top three buggy file candidates, the proposed bug localization algorithm achieves a recall of 62% and a precision of 42% for Firefox and a recall of 52% and a precision of 50% for Eclipse. On the top 10 buggy file candidates, the recall increases to 92% for Firefox and 90% for Eclipse. Developers can combine the proposed crash correlation rules with the new bug localization algorithm to identify and fix correlated crash types all together.

***Keywords*-Bug Localization; Bug Correlation; Crashes; Crash Reports; Stack Traces; Automatic Problem Reporting Tools.**

I. Introduction

Nowadays, many big software vendors such as Microsoft[1] embed automatic problem reporting tools in their software systems. Whenever the software crashes (*i.e.*, terminates unexpectedly) in a user's environment, the automatic problem reporting tool collects information about the crash and sends a detailed crash report to the software vendor. A *crash report* usually contains the stack trace of the failing thread and other runtime information. A *stack trace* is an ordered set of frames; each frame referring to a method signature. Crash reports are used by several stakeholders such as developers fixing crashes and product managers allocating development resources. Using crash reports, Microsoft developers were able to fix 29% of the bugs found in Windows XP SP1, and more than 50% of the Office XP SP2 bugs [1]. The automatic collection of crash reports helped Mozilla developers to improve the reliability of Firefox by 40% from November 2009 to March 2010 [2].

Built-in automatic crash reporting tools often collect large amounts of crash reports. For example, Mozilla Firefox receives 2.5 million crash reports every day [3]. To reduce the amount of crash reports to handle, similar crash reports are identified and grouped together based on the similarity of their stack traces. We refer to a group of similar crash reports as a *crash type*. The *signature* of a crash type is usually the top method signature of the stack traces. The crash types are sorted based on the number of crash reports and developers usually file bugs for the top crash types, *i.e.*, crash types with high numbers of crash reports. Later, stack traces from the failing threads, contained in crash reports, are used by developers to diagnose and fix the bugs.

A bug can frequently trigger crashes in different usage scenarios, causing different crash types to be linked to the same bug. Also, a crash type can be linked to multiple duplicate or correlated bugs. A duplicate bug report describes a problem already filed. Two bugs are considered to be correlated if the occurrence of one bug causes the other bug to occur. We refer to a group of crash types related to identical or correlated bugs, as a *crash correlation group (CCG)*. A crash type can belong to one or several crash correlation groups. For example, if a crash type CT_1 shares a bug with a crash type CT_2 and another bug with a crash type CT_3. CT_1 belongs to two crash correlation groups, *i.e.*, $\{CT_1, CT_2\}$ and $\{CT_1, CT_3\}$.

Many studies have been performed on the use of stack traces in crash reports to locate and fix bugs. Schroter *et al.* [4] examined stack traces in bug reports and found that bugs are fixed faster when their reports contain at least one stack trace. Brodie *et al.* [5] proposed a method based on a comparison of stack traces to identify similar bugs using historical information on known bugs. Dhaliwal *et al.* [6] examined the use of stack traces for bug fixing and identified some limitations in the crash grouping process of Mozilla Firefox. They also proposed a grouping approach for crash reports, based on a comparison of failing stack traces using the Levenshtein distance [7]. Despite this body of work, to the best of our knowledge, no previous study has proposed a method to identify correlated crash types, *i.e.*, crash types belonging to a same crash correlation group.

1. http://www.microsoft.com/en-ca/default.aspx

In this paper, we investigate the possibility to identify correlated crash types using stack traces. The identification of crash correlation groups can help developers fix bugs more efficiently; crash types in a crash correlation group should be analyzed together when fixing bugs. Crash correlation groups provide a diversity of crashing scenarios that could help developers identify the root cause of the bugs more efficiently.

We conduct our study using Firefox and Eclipse stack traces and address the following three research questions:

RQ1) Can an analysis of crash signature help identify correlated crash types?

We examine the signatures of crash types and generate a rule to identify crash correlation groups automatically. The rule does not require a detailed analysis of failing stack traces and can identify crash correlation groups with a precision of 100% and a recall of 68% for Firefox. On Eclipse, the rule achieves a precision of 69% and a recall of 46%.

RQ2) Can a detailed analysis of stack traces improve the identification of correlated crash types?

To improve on the results of **RQ1**, we examine failing stack traces and propose two additional rules to detect correlated crash types automatically. When executed together, our three rules identify crash correlation groups in Firefox with an average precision of 100% and an average recall of 90%. On Eclipse, the three rules achieve an average precision of 79% and an average recall of 65%. The average execution time of the three rules is in the order of 128 seconds. The scalability is preserved.

RQ3) Can an analysis of correlated crash types help identify buggy files?

We propose an algorithm, using our proposed crash correlation group identification rules, to locate and rank suspicious files using the stack traces of correlated crash types. When considering only the top three buggy file candidates, our algorithm achieves a recall of 62% and a precision of 42% on Firefox; and a recall of 52% and a precision of 50% on Eclipse. The top ten candidate files reported by our algorithm can recover up to 92% of buggy files in Firefox and up to 90% of buggy files in Eclipse.

The rest of this paper is organized as follows. Section II presents an overview of the Mozilla Crash Reporting System. Section IV introduces the experimental setup. Section V presents and discusses the results of our study. Section VI discusses threads to the validity. Section VII summarizes the related literature. Finally, Section VIII concludes the paper and outlines some avenues for future work.

II. Crash Reporting

Many software organizations use a bug tracking system (*e.g.*, Eclipse's Bugzilla) to store and track bugs. When a crash occurs on a user's machine, the software generates a failing stack trace that developers can use to fix bugs related to the crash. Users usually file bug reports in bug tracking systems to report these failing stack traces as well as other information that can help developers to reproduce and fix the bugs.

However, not all users file bugs or report failing stack traces. To ensure that developers get the necessary information to fix bugs, more software organizations now ship their product to users with an embedded problem reporting tool that can collect failing stack traces automatically (*e.g.*, the Mozilla Crash Reporter embedded in the Firefox browser). When a crash occurs, the failing stack trace is automatically collected by the problem reporting tool and a crash report containing information related to the crash is sent to a crash report repository (*e.g.*, the Mozilla Socorro crash report server) maintained by the software organization. A crash report usually contains a signature, the stack trace of the failing thread, some runtime information such as the crash time, and information about the user environment, *e.g.*, the operating system, the version, and the install time. Crash reports are grouped into crash types and ranked based on their frequency of occurrence. We discuss the grouping of crash reports in Section III. For the top crash types, bug reports are created in a bug tracking system and linked to their corresponding crash type. Multiple bug reports can be filed for a single crash type and multiple crash types can be associated with the same bug report. A bug report contains detailed semantic information about a bug, such as the bug open date and the bug status. Some bug reports also contain stack traces (*e.g.*, Eclipse's bug reports). Bugs are triaged and assigned to developers for fixing.

III. Stack Traces and Crash Types

A stack trace is an ordered set of frames $\langle F_1, F_2, \ldots, F_n \rangle$. Each frame F_i is composed of a method signature which we denote by *methSign* and a fully qualified file name which we denote by *qfileName*. $F_i = methSign_i | qfileName_i$, where $i \in \{1 \ldots n\}$ is the position of the frame F_i in the stack trace, and n is the total number of frames in the stack trace. F_1 is the top frame of the stack trace. Figure 1 presents an example of stack trace extracted from a crash report of Firefox.

Frame	*<Method Signature>*	*<Fully Qualified File Name>*
F_1	OnWriteSegmentt	http/nsHttpConnection.cpp
F_2	DispatchMethod	xpcom/io/nsPipe3.cpp
F_3	DispatchMessage	xpcom/io/nsPipe3.cpp
F_4	ProcessNextNativeEvent	Src/win/nsAppShell.cpp
F_5	nsShell::OnProcess	Src/win/nsShell.cpp
F_6	mozilla::Pump	Ipc/glue/MessagePump.cpp
F_7	MessageLoop:Run	Ipc/glue/MessageLoop.c
...	...	...

Fig. 1: Example of Stack Trace from Firefox

On the Mozilla Socorro server, crash reports are grouped into crash types based on the similarity of the top frames (*i.e.*, F_1) of their stack traces [6]. The top frames of all the stack traces in a crash type are identical. The method signature (*i.e.*, *methSign*) from the common top frame is used as the *signature* of the crash type. In the following, we refer to the top frame common to all the stack traces of a crash type as the *top frame* of the crash type. However, the subsequent frames in a stack trace might be different for different crash reports in a crash type.

A crash type signature S has the following structure: $S = P_1|P_2|\ldots|P_n$, where each element P_i is composed of $\langle File\rangle\langle Op\rangle\langle Method\rangle\langle Parameter\rangle\langle Memory\ location\rangle$. $File$, Op, $Method$, and $Parameter$ are respectively the name of a file, an operator, a method, and a parameter.

In a crash type signature, at least one P_i should be $\neq NULL$. In a P_i, the attributes $File$, Op, $Method$, and $Parameter$ can be $NULL$. However, a P_i cannot be formed using only the name of an operator (*i.e.*, Op). The value of Op depends on the programming language, *e.g.*, for a system written in C++, Op is generally either the scope operator "::" or _. Figure 2 shows an example of signature from the Mozilla Socorro server. This signature is composed of two elements. In the first element, Op, $Method$, and $Parameter$ are $NULL$ while in the second element only the memory location is $NULL$.

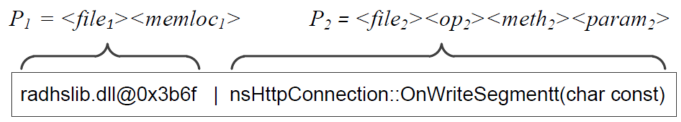
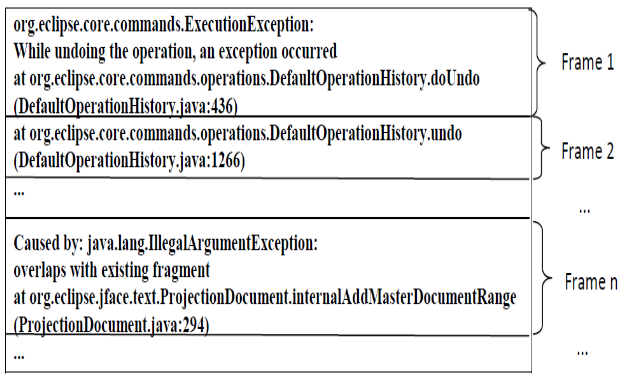

P1 = <file1><memloc1>
P2 = <file2><op2><meth2><param2>
radhslib.dll@0x3b6f | nsHttpConnection::OnWriteSegmentt(char const)

Fig. 2: Example of Crash Type Signature from the Mozilla Socorro server

The format in which Eclipse's stack traces are reported is different from the format of Firefox's stack traces. Figure 3 presents the example of a stack trace extracted from Eclipse's bug reports and Figure 4 describes the structure of each frame.



```
org.eclipse.core.commands.ExecutionException:
While undoing the operation, an exception occurred
at org.eclipse.core.commands.operations.DefaultOperationHistory.doUndo
(DefaultOperationHistory.java:436)
at org.eclipse.core.commands.operations.DefaultOperationHistory.undo
(DefaultOperationHistory.java:1266)
...
Caused by: java.lang.IllegalArgumentException:
overlaps with existing fragment
at org.eclipse.jface.text.ProjectionDocument.internalAddMasterDocumentRange
(ProjectionDocument.java:294)
...
```

Fig. 3: Example of Stack Trace from Eclipse

```
[Exception] ( [:] [Message])? ( [at] [qfilePath] [.] [Method] [(] [File] [:] [Line] [)] )*
```

Fig. 4: Structure of a Frame in an Eclipse's Stack Trace

On this Figure 4, *Exception* is the name of a Java exception (*e.g.*, org.eclipse.core.commands.ExecutionException), *Message* is the description of the exception (*e.g.*, While undoing the operation, an exception occurred), *qfilePath* is the path in the file directory structure, of the method *Method* in which the exception was raised (*e.g.*, org.eclipse.jface.text.projection.internalAdd), *File* is the name of the file that caused the exception (*e.g.*, ProjectionDocument.java), and *Line* is the exact location in *File* where the exception was triggered. A stack trace from Eclipse is mapped to the format of Firefox's stack traces as follows: $methSign = \langle Exception|Message|Method\rangle$ and $qfileName = \langle qfilePath|File\rangle$. If $Exception = NULL$, the $methSign = Method$.

We regroup Eclipse's stack traces with similar top frames into crash types using as signature the concatenation $\langle File|Method\rangle$ from their common top frame. This approach is similar to the grouping of Firefox's crash reports in the Mozilla Socorro server.

IV. Experimental Setup

This section discusses our data collection and processing.

A. Data Collection

We conduct our study using stack traces from two different software systems: Firefox (written mainly in C/C++) and Eclipse (written in Java). Firefox is an open-source Web browser developed by the Mozilla Corporation. It is currently the third most widely used browser, with approximately 24% usage share worldwide [8]. Eclipse is an open-source integrated development environment. It is a platform used both in the open-source community and the industry.

We analyze 7 beta versions of Firefox, *i.e.*, Firefox-4.0b1 to Firefox-4.0b7. For each beta version, we download the summaries of all related crash types stored in the Socorro server. We select the crash types for which at least one bug report is filed. For each selected crash type, we download the latest 100 crash reports ranked based on the crashing time. For crash types with less than 100 crash reports, we download all the crash reports. In total, we obtained 1,256 crash types. For all the bugs filed for our selected crash types, we retrieve their reports from Bugzilla. We download the Firefox change logs to extract a list of files changed to fix a bug.

To the best of our knowledge, only the Mozilla Foundation has opened the crash reports of its products to the public. To verify the replicability of our study on other systems, we downloaded the MSR Mining Challenge 2008[2] data set containing 213,000 Eclipse bug reports filed between October 2001 and December 2007.

B. Data Processing

Figure 5 shows an overview of our data processing approach. First, we process Firefox crash reports and Eclipse bug reports to extract failing stack traces and the IDs of bugs filed for the crashes. Then, we identify crash correlation groups (CCGs) defined by developers. Next, we parse Firefox and Eclipse change logs to identify bug fixes locations and, we map these bug fixes locations to the stack traces. The remainder of this section elaborates on each of these steps.

1) Extraction of Failing Stack Traces and Bug IDs: We now discuss in details the extraction of stack traces and bug IDs for Firefox and Eclipse.

Firefox: For each crash types selected for our study, we use a Perl script to extract the list of crash reports of the crash type and the failing stack traces contained in the crash reports. We

2. http://msr.uwaterloo.ca/msr2008/challenge/

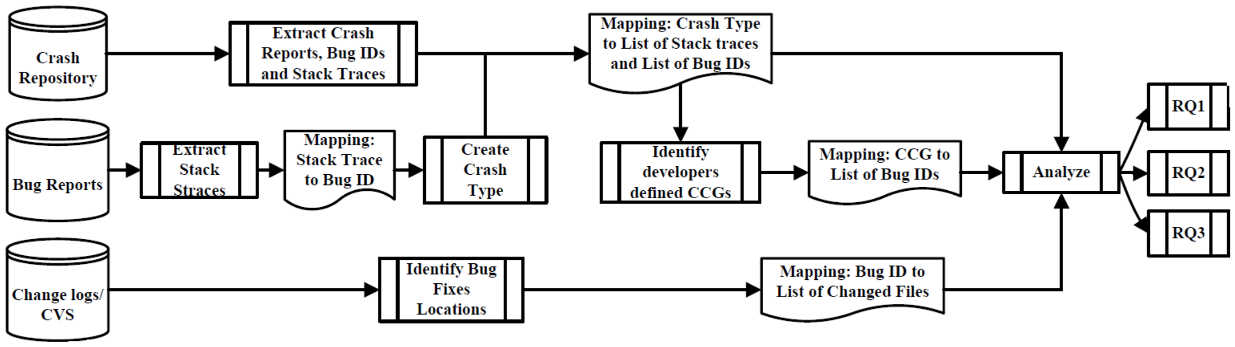

Crash Repository
Extract Crash Reports, Bug IDs and Stack Traces
Mapping: Crash Type to List of Stack traces and List of Bug IDs
Bug Reports
Extract Stack Straces
Mapping: Stack Trace to Bug ID
Create Crash Type
Identify developers defined CCGs
Mapping: CCG to List of Bug IDs
Analyze
RQ1
RQ2
RQ3
Change logs/ CVS
Identify Bug Fixes Locations
Mapping: Bug ID to List of Changed Files

Fig. 5: Overview of our approach to study correlations between crash types. CCG represents Crash Correlation Group.

also extract the IDs of all the bugs filed for the crash types. We obtain a mapping linking each crash type to the list of its crash reports and the list of bug IDs filed against the crash type.

Eclipse: Using our Perl script, we parse the 213,000 bug reports contained in the 2008 MSR Mining Challenge data set and extract all comments posted for each bug. We process the comments using regular expressions to extract the failing stack traces of the bugs in a similar way as Betttenburg *et al.* [9]. More specifically, we search for keywords "java", "Exception", and "org" in the comments. We obtain 22,379 bug reports containing at least one stack trace. Because some bug reports contain more than one stack trace, we obtain 29,874 stack traces that we link to their corresponding bug IDs. We verify all the stack traces manually to ensure that they are correct.

2) Identification of Developers Defined CCGs: We identify *developers defined CCGs* by grouping together crash types that are linked to the same bugs. We creates groups containing at least two crash types. The links between crash types and bugs are established by developers during the triaging and debugging of crash types. These links are updated during the bug fixing process, therefore we are confident that the crash types collectively linked together to a bug are correlated.

Overall we obtain 144 *developers defined CCGs* containing a total of 792 crash types from Firefox dataset and 1306 *developers defined CCGs* containing 2837 crash types from Eclipse dataset. In this study, we use *developers defined CCGs* as our gold standard to evaluate the performance of our crash type correlation identification rules. For each *developers defined CCG*, we maintain the list of bugs filed for the group.

3) Identification of Bug Fixes Locations: We parse Firefox and Eclipse change logs and apply the heuristics by Sliwersky *et al.* [10] to identify bug fixes locations. Precisely, we parse commit log messages using a Perl script and extract bug IDs and specific keywords, such as "fixed" or "bug" to identify bug fixing commits. For each bug fixing commit, we extract the list of files that were changed to fix the bug. In the following, we use the two lists of files obtained for Firefox and Eclipse as our gold standard to evaluate the performance of our bug localization algorithm and refers to them as Bug Fixing Location Mapping.

V. Experimental Results

This section presents and discusses the results of our research questions. For each research question, we present the motivation behind the question, the analysis approach and a discussion of our findings.

RQ1: Can an analysis of crash signature help identify correlated crash types?

Motivation: Schroter *et al.* [4] observed that when multiple failing stack traces are available, developers fix the bugs quickly. Therefore, the identification of crash correlation groups early in the debugging process will not only help developers fix groups of correlated crash types all together, but it will also help them fix the bugs faster. The identification of crash correlation groups can also help development teams to better manage their resources, for example, by assigning correlated bugs to experienced developers and increasing their priority. Crashes are reported continuously by users until they are fixed. Therefore, by fixing groups of correlated crash types early, development teams can reduce the amount of incoming crash reports. In this research question, we aim to provide developers with a simple rule that can be used to identify crash correlation groups automatically. We strive for building a rule requiring only an analysis of crash types signatures. In this way, development teams would be able to process large amount of crash types efficiently since no deep analysis of the content of crash reports will be required.

Approach: In the following we introduce a rule for the identification of crash correlation groups. This rule was derived from a manual analysis of 40 Firefox's crash types selected randomly. The rule identifies similarities between the signatures of correlated crash types.

We define a *contains* relation between crash signature elements as follows. Given a crash type signature $S = P_1|P_2|\ldots|P_n$, for two elements $P_i = \langle file_i\rangle\langle op_i\rangle\langle meth_i\rangle\langle param_i\rangle\langle memloc_i\rangle$ and $P_j = \langle file_j\rangle\langle op_j\rangle\langle meth_j\rangle\langle param_j\rangle\langle memloc_j\rangle$ of S, if $(file_i = file_j) \wedge \{op_i, meth_i, param_i\} \subseteq \{op_j, meth_j, param_j\}$ then P_j *contains* P_i.

We also define a binary relation $\subset$ on the set of all crash types signatures $\mathbb{S}$.

Lets S_A and S_B be two crash types signatures where, $S_A = P_1^A|P_2^A|\ldots|P_n^A$ and $S_B = P_1^B|P_2^B|\ldots|P_m^B$, with

$P_i^A = \langle file_i^A \rangle \langle op_i^A \rangle \langle meth_i^A \rangle \langle param_i^A \rangle \langle memloc_i^A \rangle$,
$P_j^B = \langle file_j^B \rangle \langle op_j^B \rangle \langle meth_j^B \rangle \langle param_j^B \rangle \langle memloc_j^B \rangle$,
$i \in \{1 \ldots n\}$, $j \in \{1 \ldots m\}$, and $m \geq n$.

$S_A \subset S_B$ if $\forall\ P_i^A$, $i \in \{1 \ldots n\}$, $\exists\ j \in \{1 \ldots m\}$ | P_j^B *contains* P_i^A. Table I presents some examples of comparison of crash types signatures using $\subset$.

TABLE I: Example of the Comparison of Crash Type Signatures

nsContentUtils::CanCallerAccess $\subset$ nsContentUtils::CanCallerAccess(nsPIDOMWindow*)
nsStyleContext::Release() $\subset$ nsStyleContext:: nsStyleContext
nvumdshim.dll@0x1845c $\subset$ nvumdshim.dll@0x1b115
nsDiskCacheStreamIO::FlushBufferToFile() $\subset$ strstr |nsDiskCacheStreamIO::FlushBufferToFile()

Rule 1: Crash Signature Comparison. Given two crash types CT_A and CT_B with signatures S_A and S_B respectively, CT_A and CT_B are correlated if $S_A \subset S_B$ or $S_B \subset S_A$.

Rule 1 compares the strings of the signatures of two crash types and uses the *contains* relation to decide if they are correlated.

To assess the performance of the proposed rule we proceed as follows: First, we filter out from our data set, all the 40 crash types that were used to discover the rule. Next, we rank the remaining Eclipse and Firefox's crash types based on their creation date. The creation date of a crash type from Firefox is the date on which the first crash report was received. For Eclipse crash types it is the date on which the oldest stack trace in the crash type was reported in a bug report. Finally, we apply the rule incrementally on the crash types to identify crash correlation groups. We compare the obtained crash correlation groups to *developers defined CCGs* and compute the precision and the recall of the rule using respectively Equation (1) and Equation (2). The precision value measures the fraction of retrieved crash correlation groups that are correct, while the recall value measures the fraction of correct crash correlation groups that are retrieved.

$$precision = \frac{|\{correct\ CCGs\} \bigcap \{retrieved\ CCGs\}|}{|\{retrieved\ CCGs\}|} \quad (1)$$

$$recall = \frac{|\{correct\ CCGs\} \bigcap \{retrieved\ CCGs\}|}{|\{correct\ CCGs\}|} \quad (2)$$

Findings: We obtain a precision of 100% and a recall of 68% for Firefox. All the crash correlation groups of Firefox retrieved using the proposed rule (*i.e.*, Rule 1) are correct. For Eclipse, the rule achieved a precision of 69% and a recall of 46%. We attribute the low recall observed for Eclipse to missing information in crash types signatures; indeed Eclipse crash type signatures contain neither parameters nor memory locations information. However, achieving a 69% precision with a simple rule like Rule 1 is already a good result. Moreover, Rule 1 identifies crash types correlation groups very efficiently. We were able to process 752 Firefox crash types in 4.53 seconds and 2797 Eclipse crash types in 22.32 seconds on a Lenovo Thinkpad laptop with an Intel Core i7-2620M CPU 2.7CHz processor and 8GB RAM.

RQ2: Can a detailed analysis of stack traces improve the identification of correlated crash types?

Motivation: In this research question, we investigate if a detailed analysis of stack traces can improve the identification of crash correlation groups. We aim to improve the 68% recall obtained for Firefox and the 46% recall obtained for Eclipse using **RQ1**. A higher recall will enable the discovery of more crash correlation groups, resulting in further improvements of the bug fixing process and the management of resources.

Approach: We manually analyzed 400 stack traces extracted from 400 Firefox's crash reports. The crash reports were selected randomly from 40 crash types which were also selected randomly. From this analysis, we derived the following two additional rules for the identification of crash correlation groups.

Rule 2: Top Frame Comparison. Given two crash types CT_A and CT_B with top frames $\mathrm{F}_1^A = methSign_1^A|qfileName_1^A$ and $\mathrm{F}_1^B = methSign_1^B|qfileName_1^B$, respectively. CT_A and CT_B are correlated if $qfileName_1^A = qfileName_1^B$. We remove file extensions when comparing *fully qualified file names* $qfileName_1^A$ and $qfileName_1^B$.

Rule 2 can be applied on the following example from Firefox 4.0b1. The top frames of the crash types *js_GetGCThingTraceKind* and *js_IsAboutToBeFinalized* are respectively $js\_GetGCThingTraceKind|js/src/jsgc.h$ and $js\_IsAboutToBeFinalized|js/src/jsgc.cpp$. These two crash types are correlated and linked to the bug 514819.

As illustrated by the above example, Rule 2 compares the *fully qualified file names* of the top frames of two crash types to verify if the crash types are correlated. When two crash types have the same *fully qualified file name* in their top frame, the two crash types are correlated.

We also analyze the other subsequent frames in the stack traces of a crash type to further improve the identification of crash types correlations. We introduce the concept of *closed ordered sub-sets of frames* for crash types.

Lets ST be a set of stack traces $\{T_1, T_2, \ldots, T_p\}$, where p is the number of stack traces in the set, $T_i = \langle\ \mathrm{F}_1^i, \mathrm{F}_2^i, \ldots, \mathrm{F}_{n_i}^i\ \rangle$, $\mathrm{F}_j^i = methSign_j^i|qfileName_j^i$, $j \in \{1, \ldots, n_i\}$, n_i is the number of frames in T_i, and $i \in \{1, \ldots, p\}$. Figure 1 shows an example of stack trace. Each frame in the stack trace has a *method signature* (*e.g.*, *OnWriteSegment* for F_1) and a *fully qualified file name* (*e.g.*, $http/nsHttpConnection.cpp$ for F_1).

Given an ordered set of frames $SubF = \langle\ \mathrm{G}_1, \ldots, \mathrm{G}_m\ \rangle$, For each T_i, $i \in \{1, \ldots, p\}$, if $\exists k, l$, with $1 < k \leq l \leq n_i$ | (G_1=$qfileName_k^i$)$\wedge \ldots \wedge$(G_m=$qfileName_l^i$), then $SubF$ is an ordered sub-set of frames of T_i. The value of each frame in $SubF$ is a *Fully Qualified File Name*.

Whenever $\exists i \in \{1, \ldots, p\}$ | $SubF$ is an ordered sub-set of frames of T_i, we denote $SubF$ as an ordered sub-set of frames of ST. $SubF$ is a *closed* ordered sub-set of frames of ST if there is no other ordered sub-set of frames of ST containing $SubF$.

The *absolute support* of $SubF$ is the number of $i \in \{1, \ldots, p\} \mid SubF$ is an ordered sub-set of frames of T_i. The *relative support* of $SubF$ is the $absolute\ support/p$. This *relative support* is the frequency of $SubF$ in ST. We consider an ordered sub-set of frames as *frequent* if its $relative\ support\ > 0.5$.

We mine all the stack traces of each crash type and extract frequent closed ordered sub-sets of frames (FCSF), using the BIDE pattern mining algorithm proposed by Wang and Han [11]. We chose the BIDE algorithm because it scales very well in the number of frequent closed patterns. In fact, BIDE does not require the maintenance of a set of candidate closed patterns. BIDE performs a strict depth first search and can output frequent closed patterns on the fly.

Rule 3: Frequent Closed Ordered Sub-Set Comparison. Given two crash types CT_A and CT_B with stack traces $ST_A = \{T_1^A, T_2^A, \ldots, T_p^A\}$ and $ST_B = \{T_1^B, T_2^B, \ldots, T_q^B\}$, respectively. If S_A^{Sub} (respectively S_B^{Sub}) is the set of frequent closed ordered sub-sets of frames of ST_A (respectively ST_B), $S_A^{Sub} \bigcap S_B^{Sub} \neq \emptyset \Rightarrow CT_A$ and CT_B are correlated.

Rule 3 examines the FCSFs of two crash types. If two crash types have a common FCSF, they are correlated. Rule 3 can be applied to the following example from Firefox 4.0b7. The stack traces of crash types *RtlIntegerToUnicodeString* and *_SEH_prolog* have in common the closed ordered sub-set of frames presented in Table II. The frequency of this sub-set of frames is 0.96 in *RtlIntegerToUnicodeString* and 0.90 in *_SEH_prolog*. Both *RtlIntegerToUnicodeString* and *_SEH_prolog* are correlated and linked to the bug 591599.

TABLE II: A Frequent Closed Ordered Sub-Sets of Frames Common to *RtlIntegerToUnicodeString* and *_SEH_prolog*

gfx/src/thebes/nsThebesDeviceContext.cpp
gfx/src/thebes/nsThebesGfxFactory.cpp
obj-firefox/xpcom/build/GenericFactory.cpp
xpcom/components/nsComponentManager.cpp
obj-firefox/xpcom/build/nsComponentManagerUtils.cpp

To assess the performance of Rule 2 and Rule 3, we proceed in a similar way as in RQ1: First, we filter out from our data set, all the 40 crash types that were used to discover the rules. Next, we rank the remaining Eclipse and Firefox crash types based on their creation date and apply successively Rule 2 and Rule 3 to the crash types one by one. Older crash types are processed first. The obtained crash correlation groups are compared to *developers defined CCGs* and precision and recall are computed using Equation (1) and Equation (2).

Rule 3 is dependent on the threshold 0.5 that is used during the identification of frequent closed ordered sub-sets of frames. Therefore we perform a sensitivity analysis to measure the impact of threshold selection on the results. Precisely, we repeat the evaluation of Rule 3 using thresholds 0.1 to 1 by step 0.1. Rule 3 is also dependent on the number of stack traces that are processed for each crash type. We repeat the evaluation of Rule 3 using 10, 20, 30, 40, 50, and 100 first crash reports in each crash type.

Findings: Table III shows that when the threshold used to identify frequent closed ordered sub-sets of frames is ≥ 0.5, Rule 2 and Rule 3 increase significantly the recall obtained with Rule 1 without decreasing the precision. For both Firefox and Eclipse, the best precision and recall are obtained with a threshold value of 0.5.

Table IV shows that our three rules does not required the analysis of a large number of crash reports. High precisions and recalls (*i.e.*, $\geq$ 0.68) are achieved with as little as 10 stack traces per crash types on both Firefox and Eclipse stack traces. No significant improvement is observed when analyzing more crash reports per crash types. This result is particularly important since software organizations receive millions of incoming crash reports every day. Using our rules, they can identify crash correlation groups efficiently by analyzing only the first 10 incoming crash reports of every crash types.

RQ3: Can an analysis of correlated crash types help identify buggy files?

Motivation: With the growing complexity of software systems, the demand for efficient techniques to identify suspicious source code fragments that may contain bugs have increased. However, locating bugs in software systems is not an easily automatable process. Although many bug localization techniques have been proposed in the literature, there is no particular technique that is suitable for every software systems [12]. Moreover, most techniques require both failing and successful test cases to be effective. Consequently, when only failing stack traces are available, developers usually apply only intuitive techniques, such as the inspection of the top 10 frames of failing stack traces. Previous work [4] have shown that buggy files are often in the top 10 frames of failing stack traces.

In this research question we aim to propose a technique to automatically locate buggy files that need to be corrected to fix bugs. We intend to build a technique that can rank suspicious buggy files effectively, reducing the effort required to examine the files. The proposed technique should also leverage knowledge of crash correlation groups in order to help debugging teams fix correlated crash types all together.

Approach: Similar to RQ1 and RQ2, we randomly sampled 40 Firefox crash types with a resolved fix. For each Firefox crash type we randomly selected 10 crash reports and extracted the contained stack traces. In total, we obtained 400 stack traces. We manually examined these stack traces and derived the bug localization method Buggy Files Finder (BFFinder) presented below. BFFinder analyzes correlations between crash types and builds a Bayesian Belief Networks (BBN) [13] to compute the probability that a file appearing in a failing stack trace is buggy. We apply BFFinder on Firefox and Eclipse separately. Figure 6 depicts the steps of BFFinder. In the following, we elaborate more on these steps.

Step 1: Extraction of frequent closed ordered sub-sets of frames. The BIDE pattern mining algorithm is

TABLE III: Precision and Recall of Rule 2 and Rule 3 for Different Thresholds. P stands for Precision, and R stands for Recall

Threshold	Rule 1				Rule 1 + Rule 2				Rule 1 + Rule 2 + Rule 3			
	Firefox		Eclipse		Firefox		Eclipse		Firefox		Eclipse	
	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)
0.1	100	68	69	46	100	83	75	58	86	84	70	58
0.2	100	68	69	46	100	83	75	58	86	84	70	58
0.3	100	68	69	46	100	83	75	58	86	85	75	63
0.4	100	68	69	46	100	83	75	58	94	90	79	65
0.5	100	68	69	46	100	83	75	58	100	90	79	65
0.6	100	68	69	46	100	83	75	58	100	89	79	65
0.7	100	68	69	46	100	83	75	58	100	87	77	62
0.8	100	68	69	46	100	83	75	58	100	84	77	62
0.9	100	68	69	46	100	83	75	58	100	83	75	58
1	100	68	69	46	100	83	75	58	100	83	75	58

TABLE IV: Precision and Recall of Rule 2 and Rule 3 for Different Number of Crash Reports. NCR stands for Number of Crash Reports, P stands for Precision, and R stands for Recall

NCR	Rule 1				Rule 1 + Rule 2				Rule 1 + Rule 2 + Rule 3			
	Firefox		Eclipse		Firefox		Eclipse		Firefox		Eclipse	
	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)
10	100	68	69	46	100	83	75	58	100	90	79	65
20	100	68	69	46	100	83	75	58	100	91	80	67
30	100	68	69	46	100	83	75	58	100	92	80	70
40	100	68	69	46	100	83	75	58	100	92	80	65
50	100	68	69	46	100	83	75	58	100	89	80	65
100	100	68	69	46	100	83	75	58	100	86	75	58

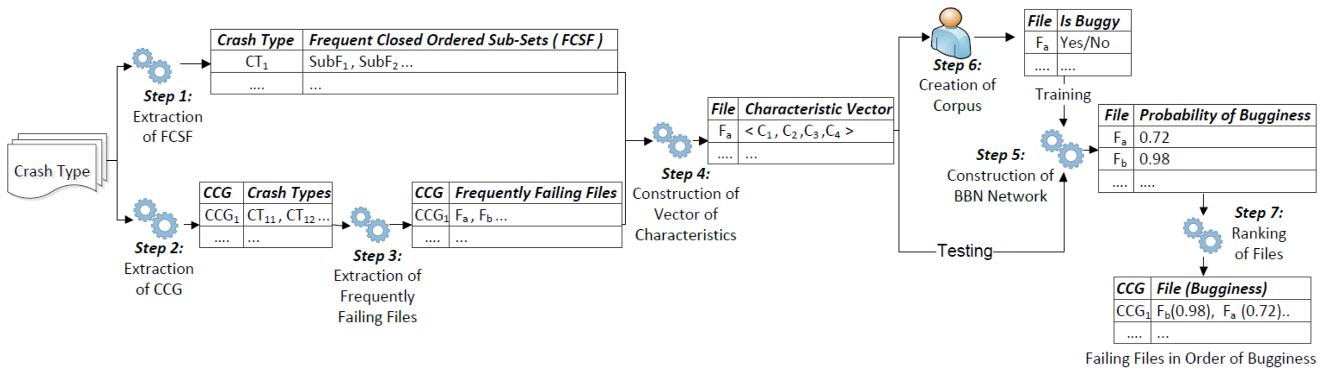

Crash Type
Step 1: Extraction of FCSF
Crash Type | Frequent Closed Ordered Sub-Sets (FCSF)
CT1 | SubF1, SubF2 ...
Step 2: Extraction of CCG
CCG | Crash Types
CCG1 | CT11, CT12 ...
Step 3: Extraction of Frequently Failing Files
CCG | Frequently Failing Files
CCG1 | Fa, Fb ...
Step 4: Construction of Vector of Characteristics
File | Characteristic Vector
Fa | < C1, C2, C3, C4 >
Step 6: Creation of Corpus
File | Is Buggy
Fa | Yes/No
Training
Testing
Step 5: Construction of BBN Network
File | Probability of Bugginess
Fa | 0.72
Fb | 0.98
Step 7: Ranking of Files
CCG | File (Bugginess)
CCG1 | Fb(0.98), Fa (0.72)..
Failing Files in Order of Bugginess

Fig. 6: Overview of the steps of BFFinder; CCG stands for Crash Correlation Group

applied on each crash type to extract its set of frequent closed ordered sub-sets of frames.

Step 2: Identification of crash correlation groups. Rule 1, Rule 2, and Rule 3 are applied successively on the signatures of the crash types and their stack traces to identify crash correlation groups.

Step 3: Extraction of frequently failing files. For each crash correlation group, the list of files appearing in all the failing stack traces of the crash correlation group is created. In case of crash types not involved in any correlation group, the list of files appearing in all the failing stack traces of the crash type is created instead. We refer to this list as the list of *frequently failing files*.

Step 4: Construction of vectors of characteristics for files. Each file appearing in a failing stack trace is mapped into a feature vector of four dimensions.

- The first dimension captures the event of the file appearing in a frequent closed ordered sub-sets of frames, *i.e.*, it counts the number of times that the file appeared in a FCSF.
- The second dimension captures the event of the file appearing in a closed ordered sub-sets of frames common to all the stack traces of crash types in a crash correlation group. *i.e.*, it counts the number of times that the file appeared in a FCSF common to all the stack traces in a crash correlation group. If a file is not involved in a crash correlation group, this dimension captures the appearance of the file in a FCSF that is common to all the stack traces of its crash type.
- The third dimension captures the failure frequency of the file, *i.e.*, the number of appearance of the file in a list of *frequently failing files*.
- The fourth dimension captures the number of times that the file appeared in the top ten frames of a stack trace.

Step 5: Construction of a Bayesian Belief Networks to rank files. The vector of characteristics obtained in **Step 4** are used to structure a BBN. The input nodes of this BBN correspond to the four dimensions of a vector of characteristics, while the output node is the probability of a file being buggy.

Step 6: Creation of a corpus to train the BBN. The vectors of characteristics of Firefox files extracted from the 400 Firefox stack traces examined manually are used to calibrate the BBN; we have knowledge of buggy files for these stack traces. Given the vector of characteristics of any other file, the trained BBN is executed to compute the probability that the file is buggy.

Step 7: Ranking of files based on the probability of containing a bug. For each crash correlation group, the files extracted from all the stack traces are ranked based on the probability that they contain a bug. High rankings are assigned to files with high probabilities. Files appearing on the stack traces of crash types that are not involved in any crash correlation group are ranked using the same criteria.

The construction of BFFinder is guided by the following observations made during the manual examination of Firefox sample of 40 crash types with 400 stack traces:

- *Observation 1:* 75% of Firefox files changed to fix bugs related to a crash type (respectively a crash correlation group) appear in all the stack traces of the crash type (respectively the crash correlation group), *i.e.,* they are frequently failing files.
- *Observation 2:* Whenever there are FCSFs for a crash type, 80% of files changed to fix bugs related to this crash type appear among the frames of a FCSF.
- *Observation 3:* As reported by previous studies (*e.g.*, [4]) on Eclipse stack traces, we found that approximately 65% of bugs in our Firefox sample were located in the files from the top 10 frames of the failing stack traces.

To assess the performance of BFFinder, we proceed as follows: First, we filter out from our data set, all the 40 Firefox crash types that were used to derive BFFinder. We also remove crash types that are associated to unfixed bugs. Then, we randomly selected 40 Eclipse crash types to train BFFinder for Eclipse stack traces. Next, we execute **Step [1–4]** of BFFinder to build the vector of characteristics of all the files that appeared in a stack trace of the remaining crash types. For each obtained vector, we run the BBN of BFFinder to compute the probability that the corresponding file is buggy. We apply **Step 7** to rank Eclipse and Firefox files in our data set. Using the two lists of buggy files (from Eclipse and Firefox) extracted from change logs as our gold standard (*i.e.*, see Section IV-B3), we compute the k-precision and the k-recall of BFFinder following Equation (3) and Equation (4).

$$k - precision = \frac{\#\, of\, buggy\, files\, in\, top\, k\, results}{k} \tag{3}$$

$$k - recall = \frac{\#\, of\, buggy\, files\, in\, top\, k\, results}{|\{buggy\, files\}|} \tag{4}$$

Because the performance of machine learners, such as BBNs, is generally impacted by the quality of the training corpus, we perform a further evaluation to measure the impact of the size of our training corpus on the performance of BFFinder. Precisely, for each system (*i.e.*, Eclipse and Firefox), we create different training corpus containing respectively 50%, 60%, 70% and 80% of all crash types from the systems and compute different k-precisions and k-recalls. We use our Bug Fixing Location Mapping (see Section IV-B3) to identify buggy files in the different training corpus and to evaluate the results of BFFinder.

Findings: On average, BFFinder achieves a recall of 72% for Firefox and 84% for Eclipse on the top 10 files reported as buggy. These high recalls suggest that BFFinder can be used efficiently with a short history of past bug locations, since the BBN was trained using only 40 Firefox crash types for Firefox and 40 Eclipse crash types for Eclipse. When the training corpus is increased to 80% of all crash types for each system, BFFinder achieves a recall of 92% for Firefox and a recall of 90% for Eclipse on average, on the top 10 files reported as buggy. These top 10 files represent only 5.5% of Firefox files and 3.8% of Eclipse files contained in the failing stack traces. Therefore, using BFFinder, debugging teams can recover respectively 92% and 90% of Firefox and Eclipse buggy files by examining only 5.5% of potential buggy candidates in Firefox and 3.8% of potential buggy candidates in Eclipse.

Table V shows results of precision and recall for top 3, top 4 and top 5 frames respectively, using different training corpus. These results show that precisions and recalls increase with the size of the training corpus. Meaning that when more information about the location of past bugs is available, the precision and the recall of BFFinder can be improved. When looking at precision and recall on the top 3 files, we observe that BFFinder can achieve a recall of 62% for Firefox and 52% for Eclipse. Hence, by only looking at 3 files reported by BFFinder as buggy, debugging teams can recover 62% of Firefox bugs and 52% of Eclipse bugs. Moreover, BFFinder allows them to fix correlated bugs all together.

VI. Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [14].

Construct validity threats concern the relation between theory and observation. In this work, the construct validity threats are mainly due to measurement errors. We extract stack traces by parsing the HTML Firefox crash reports and analyzing the comments section of Eclipse bug reports. To identify bug fix locations, we mine Mercurial logs and CVS logs, and apply the heuristics by Sliwersky *et al.* [10]. We map bug fix locations to stack traces using string matching. Although this technique may not be a hundred percent accurate, it has been used satisfactorily in many previous studies, *e.g.*, [4], [6], [10].

TABLE V: Precision and Recall of Top 3, Top 4, and Top 5 Frames Candidate Reported by BFFinder for Different Training Corpus. STC stands for Size of the Training Corpus, P stands for Precision, and R stands for Recall

STC(%)	Top 3				Top 4				Top 5			
	Firefox		Eclipse		Firefox		Eclipse		Firefox		Eclipse	
	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)
50	26	45	38	40	16	48	36	44	11	55	33	58
60	32	54	42	44	20	57	40	48	16	65	37	60
70	38	60	47	48	26	62	43	54	18	72	40	64
80	42	62	50	52	32	68	48	60	24	78	44	70

Threats to internal validity concern our selection of subject systems, tools, and analysis method. We use the stack traces posted by users in Eclipse bug reports and form Eclipse crash signatures following the same approach as the Mozilla Firefox team. The stack traces may not be complete and the relationship between Eclipse crash types may not be complete.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. The Mercurial repository of Firefox is publicly available to obtain commit logs. The Socorro crash server is also available publicly [15], to obtain the same data for the same releases. Eclipse bug reports from the 2008 MSR Mining Challenge are also publicly available.

VII. RELATED WORK

In this section, we summarize the related work on field crash reports, bug correlation, and analysis of stack traces.

A. Analysis of Field Crash Reports

Many techniques have been proposed to prioritize groups of similar crash reports during debugging activities. Podgurski niet al. [16] introduced a failure clustering approach to group similar crash reports together in order to fix the larger groups. Kim et al. [17] introduced a machine learning technique to predict crash reports that will become top crashers and which they claim should be fixed in priority. Khomh et al [18] analyzed the entropy of field crashes and proposed an entropy based approach for the triaging of field crash reports. The approach assigns high priorities to crashes with high entropies and high frequencies, *i.e.*, crashes affecting a large number of users frequently. The bug localization method presented in this paper (*i.e.*, BFFinder) can be combined with the aforementioned techniques to help development teams to correct high priority bugs efficiently.

B. Bug Correlation and Localization

Bug correlation and bug localization have been researched extensively. Lee and Soffa [19] proposed a bug correlation algorithm to identify causal relationships among bugs in a software system. Libit et al. [20] studied predicate patterns in correct and incorrect executions traces and proposed an algorithm to identify the predictors of a bugs. Ball et al. [21] developed a localization technique for error traces from a model checker. This technique identifies transitions that only appear in failing traces (but not in correct traces). Jones et al. [22], [23] proposed a visualization based technique named Tarantula to aid developers to, locate errors and bugs in software systems, by diagnosing the execution traces of successful and fail test cases. Nessa et al [24] developed a fault localization algorithm based on N-gram analysis, to rank the executable statements of a software system by their level of suspicion. The above techniques emphasize the importance of crashing threads for bug localization. However, none of them can be used to analyze crashing threads from crash reports. These techniques rely highly on instrumentation, predicates, and coverage reports, or successful traces, which limits their applicability.

C. Analysis of Stack Traces

The use of stack traces by developers during bug fixing activities has been investigated to a great extent. Schroter et al. [4] examined bug fixing activities in Eclipse and observed that when failing stack traces are available, developers fix the bugs faster. Moreover, the bugs are fixed in files from the top 10 frames of the failing stack traces. Dhaliwal et al. [6] analyzed the use of stack traces by Firefox developers and outline some limitations in the crash grouping process of Mozilla. They proposed a crash reports grouping approach based on failing stack traces comparisons using the Levenshtein distance [7] within a crash type. Brodie et al. [5] proposed an approach to identify similar bugs using stack trace comparisons and historical data of previous bugs. Some visualization techniques have also been proposed by Chan et al. [25] and Kim et al. [26] to assist development teams in the identification of relations between crashes. Although many of these approaches have investigated similarities between stack traces, none has attempted to identify crash correlation groups for crash types. In this paper we propose three rules to identify crash correlation groups using an analysis of failing stack traces.

VIII. CONCLUSION AND FUTURE WORK

The analysis of crash reports for bug fixing is a very challenging task that require a large amount of manual work from developers. In this study, we propose three rules to identify correlated crash types automatically. We also propose a bug localization method called Buggy Files Finder (BFFinder) to locate and rank buggy files from the stack traces in crash reports. BFFinder uses our three rules: Crash Signature Comparison (*i.e.*, Rule 1), Top Frame Comparison (*i.e.*, Rule 2) and Frequent Closed Ordered Sub-Set Comparison (*i.e.*, Rule 3) to identify correlated crash types. Using a Bayesian Belief Networks, BFFinder computes and ranks files from stack traces based on their probability to be buggy.

We conducted a case study using Firefox and Eclipse stack traces and found that when applied together, the three rules achieve a precision of 100% and a recall of 90% for Firefox, and a precision of 79% and a recall of 65% for Eclipse. We obtain a precision of 100% and a recall of 68% for Firefox, and a precision of 69% and a recall of 46% for Eclipse when using only Rule 1. Our three rules do not require the analysis of a large number of crash reports. High precisions and recalls (*i.e.*, $\geq 68\%$) are achieved with as little as 10 crash reports per crash types.

Our case study also shows that with a training corpus containing only 40 Firefox crash types, BFFinder achieves a recall of 72% on the top 10 files reported as buggy. When trained on 80% of the corpus, the recalls of BFFinder are 92% for Firefox and 90% for Eclipse, on the top 10 files reported as buggy. These results suggest that BFFinder can be used efficiently with little information about the location of past bugs. When more information on the location of past bugs is available, the precision and recall of BFFinder is improved. Using BFFinder, debugging teams can recover 92% of buggy files by examining only 5.5% of all the files contained in Firefox's stack traces and 90% of buggy files by examining only 3.8% of all the files contained in Eclipse's stack traces. BFFinder allows debugging teams to locate and fix correlated bugs all together. In future work, we plan to implement our proposed rules and our bug localisation method BFFinder into a tool to assist development teams during the triaging of crash reports and the fixing of bugs.

Acknowledgment

The authors would like to thank Tejinder Dhaliwal and Feng Zhang, of Queen's University, for his help during data collection and for his many useful comments on this work.

References

[1] Connecting with customers. *http://www.microsoft.com/mscorp/execmail-/2002/10-02customers.mspx*, last accessed on March 27,2012.

[2] Firefox Stability Improvement. *http://blog.mozilla.com/metrics/2010/04-/08/dramatic-stability-improvements-in-firefox/*, last accessed on March 22, 2012.

[3] Socorro: Mozilla's Crash Reporting Server. *http://blog.mozilla.com/webdev/2010/05/19/socorro-mozilla-crash-reports/*, last accessed on March 22, 2012.

[4] Adrian Schroter, Nicolas Bettenburg, Rahul Premraj.*Do stack Traces Help Developers Fix Bugs?*. MSR 2010: 7th IEEE Working Conference on Mining Software Repositories, pp. 118-121,2010.

[5] 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.

[6] Tejinder Dhaliwal, Foutse Khomh, Ying Zou. *Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox*, Proc. the 27th IEEE international Conference on Software Maintenance(ICSM), September 25-30, 2011, Williamsburg, VA, USA. IEEE.

[7] J.B.Kruskal. *An Overview of Sequence Comparison:Time Warps, String Edits, and Macromolecules*, SIAM Review. Vol. 25, No. 2. pp.201-237. April 1983.

[8] Web browsers (Global marketshare), Roxr Software Ltd. Retrieved on January 12, 2012, http://bit.ly/81klgi.

[9] N. Betttenburg, R. Premraj, T.Zimmermann, and S. Kim, *Extracting structual information from bug reports*, in Proceedings of the international working conference on Mining software repositories 2008, May 10-11, 2008, Leipzig, Germany.

[10] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1-5, May 2005.

[11] Jianyong Wang and Jiawei Han. *BIDE:Efficient Mining of Frequent Closed Sequences*, In ICDE '04: Proceedings of the 20th International Conference on Data Engineering (2004), pp. 79-90.

[12] W. Eric Wong and Vidroha Debroy, *A Survey of Software Fault Localization*, Technical Report UTDCS-45-09, Department of Computer Science, The University of Texas at Dallas, November 2009.

[13] D.Michie, D.J.Spiegelhalter, and C.C. Taylor, *Machine Learning, Neural and Statistical Classification*. Prentice Hall, 1994.

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

[15] Mozilla Crash Reporting Server. *http://crash-stats.mozilla.com/products/Firefox*, last accessed on March 22, 2012.

[16] A. Podgurski, D.Leon, P.A. Francis, W.Masri, M.Minch, J.Sun, and B. Wang. *Automated Support for Classifying Software Failure Reports*, Proc. 25th International Conference on Software Engineering, pp.465-475, 2003.

[17] Dongsun Kim,Xinming Wang, Sunghun Kim, Andreas Zeller, S.C. Cheung and Sooyong Park, *Which Crashes Should I Fix First?:Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts*, IEEE Transactions on Software Engineering. VOL.37. NO.3. June 2011.

[18] Foutse Khomh, Brian Chan, Ying Zou, Ahmed E. Hassan. *An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox*, Proc. the 18th Working Conference on Reverse Engineering(WCRE), October 17-20, 2011, Lero,Limerick, Ireland. IEEE Computer Society.

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

[20] 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.

[21] T.Ball, NM. 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.

[22] 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.

[23] 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.

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

[25] B.Chan,Ying Zou, A.E.Hassan and A.Sinha. *Visualizing the Results of Field Testing*, International Symposium on Software Reliability Engineering, Mysuru, India, November 2009.

[26] Sunghun Kim, Thomas Zimmermann, Nachiappan Nagappan. *Crash Graphs: An aggregated view of multiple crashes to improve crash triage*, Dependable Systems and Networks (DSN),, pp. 486-493, 2011.