Noname manuscript No.
(will be inserted by the editor)

Improving Bug Management using Correlations in Crash Reports

Shaohua Wang · Foutse Khomh · Ying Zou

Received: date / Accepted: date

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 bug reports for the top crash types. Because a bug can trigger a crash in different usage scenarios, different crash types are sometimes related to the 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 bug reports, as a crash correlation group. In this paper, we propose five rules to identify correlated crash types automatically. We propose an algorithm to locate and rank buggy files using crash correlation groups. We also propose a method to identify duplicate and related bug reports. Through an empirical study on Firefox and Eclipse, we show that the first three rules can identify crash correlation groups using stack trace information, with a precision of 91% and a recall of 87% for Firefox and a precision of 76% and a recall of 61% 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. The proposed duplicate bug report identification method achieves a recall of 50% and a precision of 55% on Firefox, and a recall of 47% and a precision of 35%

Shaohua Wang
School of Computing, Queen's University, Kingston, Ontario, Canda
E-mail: shaohua@cs.queensu.ca

Foutse Khomh
SWAT Lab, DGIGL, Polytechnique Montréal, Montréal, QC, Canada
E-mail: foutse.khomh@polymtl.ca

Ying Zou
Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada
E-mail: ying.zou@queensu.ca

on Eclipse. Developers can combine the proposed crash correlation rules with the new bug localization algorithm to identify and fix correlated crash types all together. Triagers can use the duplicate bug report identification method to reduce their workload by filtering duplicate bug reports automatically.

Keywords Crashes · Crash Reports · Stack Traces · Bug Localization · Bug Duplication

1 Introduction

Nowadays, many big software organizations such as Microsoft[1] and Mozilla[2] 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 a large amount 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 bug reports 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. A crash type can be linked to multiple duplicate or correlated bug reports. A duplicate bug report describes a problem already filed. Two bug reports are considered to be correlated if the occurrence of one bug in one bug report causes the bug in the other report to occur. We refer to a group of crash types related to identical or correlated bug reports, 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 report with a crash type CT_2 and another bug report with a crash

[1] http://www.microsoft.com/en-ca/default.aspx
[2] http://www.mozilla.org/en-US/

type CT_3. CT_1 belongs to two crash correlation groups, *i.e.,* $\{CT_1, CT_2\}$ and $\{CT_1, CT_3\}$.

The identification of crash correlation groups can help developers identify correlated crash types and 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.

Many studies have been performed on the use of stack traces in crash reports to locate and fix bugs. Schröter *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 proposed a grouping approach for crash reports, based on a comparison of failing stack traces using the Levenshtein distance [7], and build sub-groups of crash reports of a crash type. Their sub-grouping strategy can improve the existing Mozilla crash reporting system and this improvement can help to reduce the bug fixing time by more than 5% based on their empirical study.

In our previous work published at the 10^{th} Working Conference on Mining Software Repositories [8], we propose three rules to identify correlated crash types automatically, using structural information about the crash types (*i.e.*, the crash signatures and stack traces).

In this paper, in addition to using structural information, we investigate the possibility to identify correlated crash types using temporal and semantic information. The temporal information is related to the co-occurrence time of crash types and the semantic information is related to the textual similarity between user comments provided for the crash types. Moreover, we also explore the possibility of using crash correlation groups to help development teams fix bugs and identify duplicate bug reports.

We conduct our study using Firefox crash reports and Eclipse bug reports. We address the following five research questions:

RQ1. Can we identify correlated crash types using crash type signature and stack traces?

We strive to propose simple rules for the identification of crash correlation groups (*i.e.* correlated crash types) using the structural information of crash types. First, we examine the signatures of crash types and generate a rule to automatically identify crash correlation groups. 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%. To improve on the results, 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 91% and an average recall of 87%. On Eclipse, the three rules achieve an average precision of 76% and an average recall of 61%. The average execution time of the three rules is in the order of 128 seconds. The scalability is preserved.

RQ2. Can we identify correlated crash types using the occurrence times of crash events?

A group of crash types reported by the same users frequently, within a short time period, can be correlated. We examine the co-occurrences of crash types and propose one additional rule to detect correlated crash types automatically. This rule can identify crash correlation groups in Firefox with an average precision of 52% and an average recall of 58%. The highest recall it can achieve is 84%. This rule is not applicable to Eclipse, since the time of user comments being posted in Eclipse bugzilla is not the actual time of the occurrence of exceptions.

RQ3. Can we identify correlated crash types using the textual similarity between users comments about the crash events?

The user comments describe the crashing scenarios of crash types. Correlated crash types could have similar user comments, therefore we examine the similarity between text mined from user comments of crash types and propose one additional rule to detect correlated crash types automatically. This rule identifies crash correlation groups in Firefox with an average precision of 54% and an average recall of 46%. On Eclipse, the rule achieves an average precision of 42% and an average recall of 30%.

RQ4. Can the 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.

RQ5. Can the correlated crash types help identify duplicate bug reports?

We investigate the possibility of using the correlated crash types to identify duplicate or related bug reports. Our proposed approach, using the relations among crash correlation groups for duplicate bug reports identification, can achieve a precision of 55% and a recall of 50% on Firefox, and a precision of 38% and a recall of 47% on Eclipse. This confirms that using correlations between crash types can help identify duplicate bug reports.

This paper is an extended version of our earlier work [8]. The original work:

– proposes one rule based on the comparison of crash type signatures of crash types and two rules based on stack traces to group correlated crash types;

– conducts an empirical study of the effectiveness of the three rules on stack traces from Firefox crash reports and Eclipse bug reports;
– proposes an approach, using the correlations between crash types within a crash correlation group, to help development teams locate buggy files.
– conducts an empirical study of the effectiveness of our approach for locating buggy files on Firefox and Eclipse.

We extend the earlier work in the following aspects:

1. We build two more additional rules: One rule is based on the co-occurrence time of crash types; the other one is based on the textual similarity between crash types.
2. We conduct an empirical study on Firefox and Eclipse to verify the effectiveness of the two rules identifying correlated crash types.
3. We propose an approach using the relations between crash correlation groups to identify duplicate and related bugs.
4. We conduct an empirical study of the effectiveness of our approach for identifying duplicate and related bug reports on Firefox and Eclipse.

The rest of this paper is organized as follows. Section 2 explains the process of crash reporting and introduces stack traces and crash types. Section 3 introduces the experimental setup. Section 4 presents the research questions of our study; for each research question, we present the motivation, introduce the analysis approach and discuss the results of our study. Section 5 discusses threats to the validity. Section 6 summarizes the related literature. Finally, Section 7 concludes the paper and outlines some avenues for future work.

2 Background

2.1 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 crashes and include failing stack traces in comments made on the crashes. The other users can also share their failing stack traces by making comments on the filed bug reports including the crashes. The failing stack traces in the comments of bug reports as well as other information in the bug reports can help developers to reproduce and fix the bugs.

However, not all users file bug reports 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

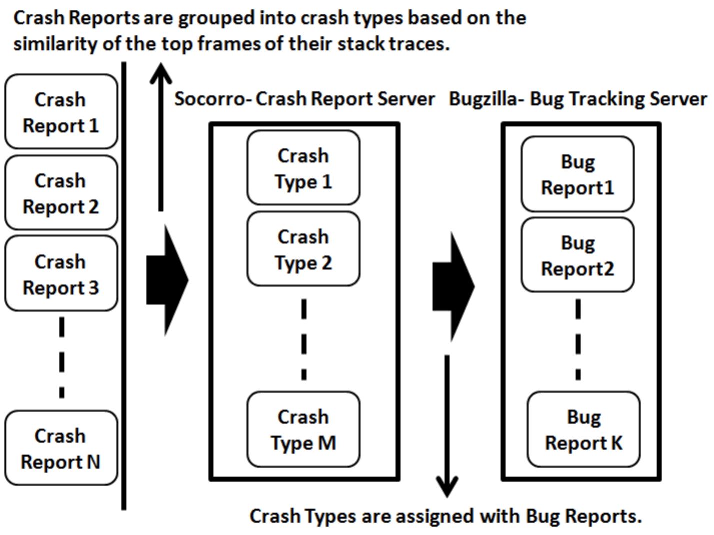
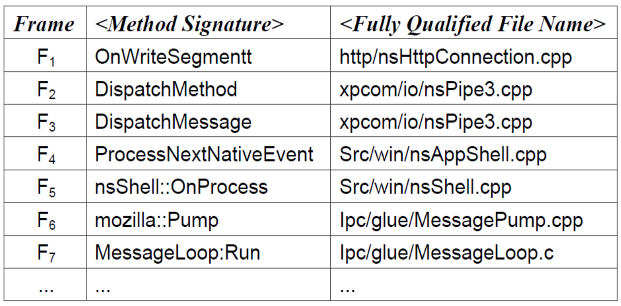

Crash Reports are grouped into crash types based on the similarity of the top frames of their stack traces.
Crash Report 1
Crash Report 2
Crash Report 3
Crash Report N
Socorro- Crash Report Server
Crash Type 1
Crash Type 2
Crash Type M
Bugzilla- Bug Tracking Server
Bug Report1
Bug Report2
Bug Report K
Crash Types are assigned with Bug Reports.

Fig. 1: Mozilla Crash Report System

report repository (*e.g.,* the Mozilla Socorro crash report server as illustrated in Fig. 1) 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. Some crash reports contain comments discussing the crashes in the reports from users. Crash reports are grouped into crash types and ranked based on their frequency of occurrence. We discuss the grouping of crash reports in Section 2.2. For the top crash types, bug reports are created in a bug tracking system and linked to their corresponding crash types. 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. Moreover, users can make comments on a bug in the filed bug reports and some comments also contain stack traces (*e.g.,* Eclipse's bug reports). Bug reports are triaged and assigned to developers for fixing.



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. 2: Example of Stack Trace from Firefox

2.2 Stack Traces, Crash Reports and Crash Types

A stack trace is an ordered set of frames $\langle$ F$_1$, F$_2$, ..., 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 2 presents an example of stack trace extracted from a crash report of Firefox.

Each crash report contains a failing crash stack trace. 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 crash time of a crash type is the time of its first crash report received by Socorro server. Usually, 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 *crash type signature* of the crash type, for example in Figure 2, the method signature *OnWriteSegmentt* of frame F$_1$ is used as a crash type signature. 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 can be represented in 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 or class name, an operator or a separator, a method, and a parameter.

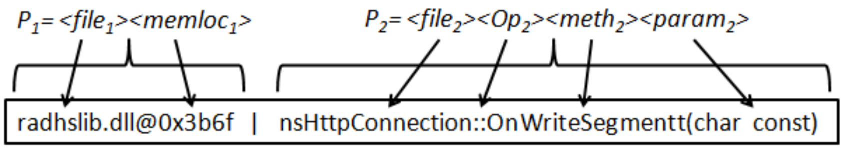
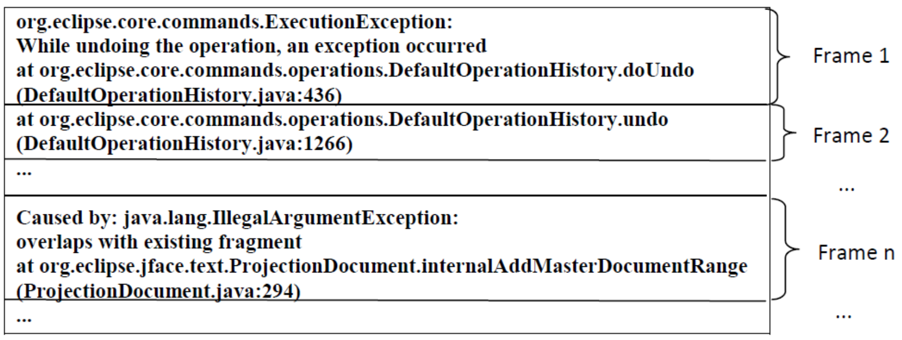

P1= <file1><memloc1>
P2= <file2><Op2><meth2><param2>
radhslib.dll@0x3b6f | nsHttpConnection::OnWriteSegmentt(char const)

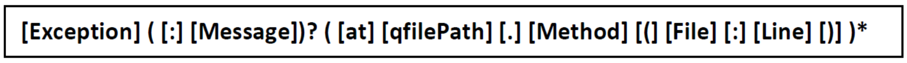
Fig. 3: Example Crash Type Signature from the Mozilla Socorro server



```
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. 4: Example of a Stack Trace from Eclipse

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 and the approach of composing a signature, *e.g.,* the Firefox Browser written in C++, Op is generally either the scope operator "::" or a separator "_". Figure 3 shows an example crash type signature from the Mozilla Socorro server. This signature is composed of two elements. The first element P_1 contains $File$ and $MemoryLocation$. The Op, $Method$, and $Parameter$ are $NULL$. In the second element P_2, the memory location is $NULL$.



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

Fig. 5: Structure of a Frame used in an Eclipse Stack Trace

The format used in Eclipse's stack traces is different from the format used in Firefox's stack traces. Figure 4 presents an example of a stack trace extracted from Eclipse's bug reports and Figure 5 shows the structure of a Frame in Eclipse stack traces.

In Figure 5, *Exception* is the name of a Java exception (*e.g.,* org.eclipse.core .commands.ExecutionException as shown in the Frame 1 in Figure 4), *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* in which the exception was raised (*e.g.,* org.eclipse.jface.text.projection .internalAdd as shown in Figure 4), *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 $Exception = NULL$, then $methSign = Method$.

We regroup Eclipse stack traces with similar top frames into crash types using 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.

3 Experimental Setup

This section discusses our data collection and processing.

3.1 Data Collection

We conduct our study on two 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 [9]. Eclipse is an open-source integrated development environment. It is a platform used both in the open-source community and the industry.

Table 1: Descriptive Statistics of Our Data Set on Firefox.

Version	4.0b1	4.0b2	4.0b3	4.0b4	4.0b5	4.0b6	4.0b7
# of Crash Reports	237,923	74,650	128,899	231,403	199,946	299,994	149,570
Total Number of Crash Reports studied:				1,322,385			

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 Socorro server. We select the crash types for which at least one bug report is filed. For each selected crash type, we download the Firefox crash reports, based on their crashing time from latest to earliest, from Socorro server. Table 1 reports the descriptive statistics of our dataset. In total, we obtained 1,256 crash types. For all the bug reports filed for our selected crash types, we retrieve the bug reports from Bugzilla. We download the Firefox change logs to extract a list of files changed to fix a bug.

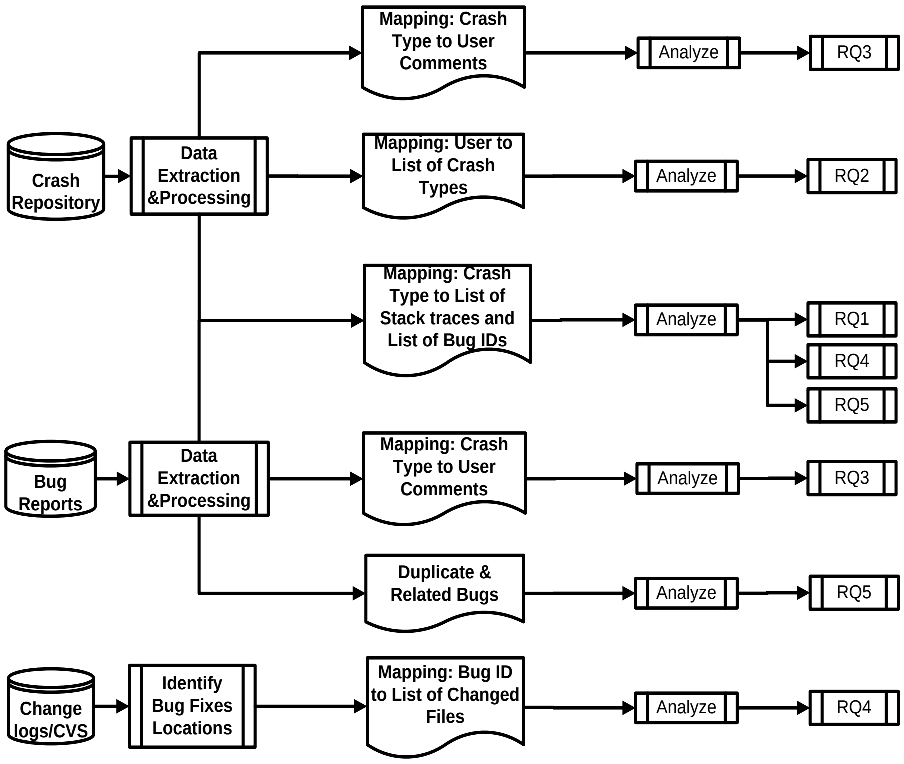

Crash Repository
Data Extraction &Processing
Mapping: Crash Type to User Comments
Analyze
RQ3
Mapping: User to List of Crash Types
Analyze
RQ2
Mapping: Crash Type to List of Stack traces and List of Bug IDs
Analyze
RQ1
RQ4
RQ5
Bug Reports
Data Extraction &Processing
Mapping: Crash Type to User Comments
Analyze
RQ3
Duplicate & Related Bugs
Analyze
RQ5
Change logs/CVS
Identify Bug Fixes Locations
Mapping: Bug ID to List of Changed Files
Analyze
RQ4

Fig. 6: Overview of our approach to study correlations between crash types.

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[3] data set containing 213,000 Eclipse bug reports filed between October 2001 and December 2007.

3.2 Data Processing

Figure 6 shows an overview of our data processing approach. First, we process Firefox crash reports to extract failing stack traces, user comments, user environment information, *e.g.,* crash time and operating system, and IDs of bugs filed for the crashes. Second, we parse Eclipse bug reports to extract failing stack traces and their descriptions from user comments, and the IDs of bugs filed for the crashes. Third, we identify crash correlation groups (CCGs) defined by developers for the validation of our approach. Fourth, we use user environment information to identify users who report crashes. Then, we conduct word normalization on the user comments of crashes. 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.

3.2.1 Data Extraction from Firefox and Eclipse

We now discuss in details the data extraction for Firefox and Eclipse.

Firefox. For each crash type selected in our study, we extract the list of crash reports of the crash type and the failing stack traces contained in the crash reports by parsing HTML pages. We extract the possible user comments in the crash reports of each selected crash type and maintain a mapping between a crash type and its user comments used as the textual description of the crash type. We further extract user environment information such as operating system and crash time and maintain a mapping between user environment information and each crash report. We 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. Furthermore, we download the bug reports using the extracted IDs of the bugs filed for the crash types, and mine groups of duplicate and related bug reports.

Eclipse. We parse the 213,000 bug reports contained in the 2008 MSR Mining Challenge data set and extract all of the comments posted by users (*e.g.*, developers) for each bug. Unlike Firefox, the Eclipse stack traces are embedded in the comments of Eclipse bug reports. We process the comments using regular expressions to extract the failing stack traces of the bug reports in a similar way as Betttenburg *et al.* [11]. We obtain 22,379 bug reports having

[3] http://msr.uwaterloo.ca/msr2008/challenge/

comments which contain at least one stack trace. We obtain 29,874 stack traces that we link to their corresponding bug report IDs. A bug report ID is linked with a set of stack traces. We cleanse and verify all of the extracted stack traces manually to ensure that there is no chaos (*e.g.*, English words describing the crash scenario) in the extracted stack traces. In addition, we mine duplicate bug report relations between the bug reports. After the stack traces extracted from the comments of a bug report, we keep the remaining words in the comments as textual description for the stack traces. We group the extracted stack traces into crash types using the approach in Section 2.2 and maintain a mapping between a crash type (*i.e.*, a set of stack traces) and its textual description.

3.2.2 Identification of Developers-defined Crash Correlation Groups

To validate our proposed rules for identifying correlated crash types, we build a gold standard by mining *Developer-defined Crash Correlation Groups* from our dataset. More specifically, we identify *Developer-defined Crash Correlation Groups (CCGs)* by grouping together crash types that are linked to the same bugs. We create 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 *Developer-defined CCGs* containing a total of 792 crash types from the Firefox dataset and 1306 *Developer-defined CCGs* containing 2837 crash types from the Eclipse dataset. In this study, we use *Developer-defined CCGs* as our gold standard to evaluate the performance of our crash type correlation identification rules. For each *Developer-defined CCG*, we maintain the list of bugs filed for the group.

3.2.3 Identification of Users

The Firefox crash reports do not contain personal information to identify unique users reporting the crashes due to privacy concerns. To identify users reporting crashes, we have to use heuristics and adopt the approach in [10]. When we process the Firefox crash reports, we extract the following available information on the crash events:

– the install age (in seconds) since the installation or the last update of the user's system;
– the date at which the crash was processed on the server;
– the crash time on the user's operating system when the crash occurred (this time can shift around with clock resets);
– the uptime (in seconds) since the user's operating system was launched;
– the last crash of the user;

– the other user's environment information: operating system name, operating system version, architecture (*e.g.,* x86) and CPU family model and stepping.

For each crash report, we use crash time to subtract the "install age" to obtain the installation time when the user, who reports the crash, installed Firefox. We use the installation time, other user's environment information and the last crash times to build a vector of unique profiles; each profile represents a user. We associate each unique profile with the list of crash types for which crash reports contain information corresponding to the profile. In this way, we obtain a mapping between each user and his corresponding crash reports. We sort the crash types from each user based on their crash times from newest to oldest. In total, we identify 1,048,576 users (*i.e.,* groups of crash types) from 1,322,385 crash reports.

3.2.4 Identification of Bug Fix Locations

We parse Firefox and Eclipse change logs and apply the heuristics by Sliwersky *et al.* [12] to identify bug fix 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 refer to them as Bug Fixing Location Mapping.

4 Research Questions

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

RQ1. Can we identify correlated crash types using crash type signature and stack traces?

Motivation. Schröter *et al.* [4] observed that when multiple failing stack traces are available, developers fix the bugs quickly. Therefore, the identification of crash correlation groups (*i.e.* correlated crash types) 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 simple rules that can be used to identify crash correlation groups automatically. First, we strive for building a rule requiring only an analysis of crash type signatures. In this way, development teams would be able to process large amounts of crash types efficiently since no deep analysis of the content of crash reports will be required. Second, we investigate if a detailed analysis of stack traces can improve the identification of crash correlation groups.

A higher recall will enable the discovery of more crash correlation groups, resulting in further improvement of the bug fixing process and the management of resources.

Analysis Approach. To answer **RQ1**, we introduce the following three rules for the identification of crash correlation groups.

These rules were derived from a manual analysis of 40 of Firefox crash types selected randomly.

We define a *contains* relation between crash type 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 define a binary relation $\subset$ on the set of all crash type signatures $\mathbb{S}$.

Let S_A and S_B be two crash type 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 $m \geq n$.

Table 2: Example of the Comparison of Crash Type Signatures

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

$S_A \subset S_B$ if $\forall\ P_i^A$, $i \in \{1\ldots n\}$, $\exists\ j \in \{1\ldots m\} \mid P_j^B$ *contains* P_i^A. Table 2 presents some examples of comparison of crash type signatures using $\subset$.

Rule 1: Crash type 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 identifies similarities between the signatures of correlated crash types. More specifically, it compares the strings of the signatures of two crash types and uses the *contains* relation to decide if they are correlated.

To investigate if a detailed analysis of stack traces can improve the identification of crash correlation groups, we manually analyzed 400 stack traces extracted from 400 of Firefox crash reports. The crash reports were selected randomly from our 40 randomly selected crash types. 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* $F_1^A = methSign_1^A|qfileName_1^A$ *and* $F_1^B = methSign_1^B|qfileName_1^B$*, 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 2 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 $1 < k \leq l \leq n_i \mid (\mathrm{G}_1 = qfileName_k^i) \wedge \ldots \wedge (\mathrm{G}_m = qfileName_l^i)$, 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 $\exists i \in \{1, \ldots, p\} \mid SubF$ 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 BI-Directional Extension based frequent closed sequence mining (BIDE) pattern mining algorithm proposed by Wang and Han [13]. 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.*

Table 3: 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

Rule 3 examines the FCSFs of two crash types. If two crash types have a common FCSF, they are correlated. For example, there are two crash types from Firefox 4.0b7: *RtlIntegerToUnicodeString* and *_SEH_prolog*. The Rule 3 mines the stack traces of both crash types to identify whether these two crash types share common closed ordered sub-sets of frames. The closed ordered sub-set of frames is identified as illustrated in Table 3. 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 report whose id is 591599.

To assess the performance of Rule1, Rule 2 and Rule 3, we proceed as follows: First, we filter out from our data set, all the 40 crash types that were used to discover the rules. Second, we rank the remaining Eclipse and Firefox crash types based on their creation date to mimic the current practice. 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. Next, we apply successively Rule 1, Rule 2 and Rule 3 to the crash types one by one to identify crash correlation groups. Older crash types are processed first. Every crash type is tested against all the other crash types to verify its membership of crash correlation groups. When three rules are combined together, two crash types are in a crash correlation group as long as they satisfy one of three rules.

We compare the obtained crash correlation groups to *Developer-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.

$$precision = \frac{|\{correct\, CCGs\} \bigcap \{retrieved\, CCGs\}|}{|\{retrieved\, CCGs\}|} \tag{1}$$

$$recall = \frac{|\{correct\, CCGs\} \bigcap \{retrieved\, CCGs\}|}{|\{correct\, CCGs\}|} \tag{2}$$

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 and 30 first crash reports in each crash type. 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 and the threshold 0.5.

Table 4: Precision and Recall of using Rule 1, Rule 2 and Rule 3 together for Different Thresholds.

Threshold	Rule 1 + Rule 2 + Rule 3			
	Firefox		Eclipse	
	Precision(%)	Recall(%)	precision(%)	Recall(%)
0.1	78	84	70	58
0.2	83	84	70	58
0.3	85	85	75	63
0.4	92	87	79	65
0.5	94	90	79	65
0.6	90	85	79	65
0.7	88	84	77	62
0.8	84	83	77	62
0.9	75	83	75	58
1	70	83	75	58

Findings. We obtain a precision of 100% and a recall of 68% for Firefox using Rule 1. All the crash correlation groups of Firefox retrieved using Rule 1 are correct. For Eclipse, Rule 1 achieved a precision of 69% and a recall of 46%. We attribute the low recall observed for Eclipse to missing information in crash type signatures; indeed Eclipse crash type signatures contain neither parameters nor memory location information. However, achieving a 69% precision with a simple rule like Rule 1 is already a good result. Moreover, Rule 1 identifies crash type 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 2.7GHz processor and 8GB RAM.

We obtain a precision of 45% and a recall of 48% for Firefox using Rule 2, and a precision of 40% and a recall of 52% for Eclipse. When we apply Rule 1 and Rule 2 together, we obtain a precision of 89% and a recall of 83% on Firefox, and a precision of 75% and a recall of 58% on Eclipse. The results indicate that Rule 2 increase the recall obtained with Rule 1 by 15% on Firefox and 12% on Eclipse.

Table 5: Precision and Recall of using Rule 1, Rule 2 and Rule 3 together for Different Number of Crash Reports. NCR stands for Number of Crash Reports.

NCR	Rule 1 + Rule 2 + Rule 3			
	Firefox		Eclipse	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
10	89	85	75	58
20	91	88	77	62
30	94	90	79	65
40	94	90	77	62
50	90	87	74	59
100	88	83	72	55

Table 4 shows that when the threshold of relative support used to identify frequent closed ordered sub-sets of frames is ≥ 0.5, Rule 2 and Rule 3 increase 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 5 shows that our three rules do not require the analysis of a large number of crash reports. High precision and recall (*i.e.,* ≥ 0.65) are achieved with as little as 10 stack traces per crash types on both Firefox and Eclipse stack traces. 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.

Table 6: Summarized Results of Using Rule 1, Rule 2 and Rule 3. The value in parentheses shows the percent difference in results caused by using one more rule on correlation group identification.

Rules	Firefox		Eclipse	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
Rule 1	100	68	69	46
Rule 1+ Rule 2	89 (-11)	83 (+15)	75 (+6)	58 (+12)
Rule 1+ Rule 2+ Rule 3	94 (+5)	90 (+7)	79 (+4)	65 (+7)

Table 6 summarizes the results obtained by using different sets of rules. Rule 2 improves the recall of Rule 1 on Firefox and Eclipse. However Rule 2 decreases the precision of Rule 1 on Firefox by 11% and increases the precision of Rule 1 on Eclipse by 6%. Based on the results in Table 4 and Table 5, when the threshold values of relative support and number of crash reports are set to be 0.5 and 30 respectively for Rule 3, Rule 3 improves the precision and recall obtained by using Rule 1 and Rule 2 together.

RQ2. Can we identify correlated crash types using the occurrence times of crash events?

Motivation. Two crash types that co-occur frequently within a short time period and from the same user's machine are likely to be correlated. Besides studying the structural information (*i.e.*, crash signatures and stack traces) of crash types in **RQ1**, in this research question, we investigate the possibility of using the occurrence time of crash events (*i.e.*, temporal information of crash types) to identify crash correlation groups. Specifically, we search for any set of crash types that frequently co-occur together within a time period (*e.g.,* one day, three days, one week and two weeks) and that originate from the same users' machines. The shorter the time period is, the sooner the crash types can be linked and processed together for bug fixing.

Analysis Approach. To achieve the goal set in this research question, we introduce a rule to identify crash correlation groups using frequent patterns of co-occurrences of crash types on users' machines.

Let U be a set of users $\{U_1, U_2, \ldots, U_n\}$, where n is the number of users who reported a crash. For each user U_i, the group of crash types reported by U_i is $\langle$ C_1^i, C_2^i, …, $\mathrm{C}_{n_i}^i$ $\rangle$, where n_i is the number of crash types reported by the user U_i, C_j^i is the jth crash type reported by the user U_i, $j \in \{1, \ldots, n_i\}$, $i \in \{1, \ldots, n\}$.

Given a set of crash types $SubC = \langle$ C_1,…, C_m $\rangle$, where $m \geq 2$, for each user U_i, $i \in \{1, \ldots, n\}$, if $\exists k, l$, with $1 < k \leq l \leq n_i \mid (\mathrm{C}_1{=}C_k^i) \wedge \ldots \wedge (\mathrm{C}_m{=}C_l^i)$, then $SubC$ is a sub-set of crash types of U_i.

The *absolute support* of $SubC$ is the number of $i \in \{1, \ldots, n\} \mid SubC$ is a sub-set of crash types of U_i. We mine all the groups of crash types of users and extract frequent sub-sets of crash types, using AprioriTID [14], an algorithm for discovering frequent item-sets (groups of crash types appearing frequently) among users. To be able to capture more sub-sets of crash types, we set the absolute support threshold value of the algorithm to 2, *i.e.,* as long as a sub-set appears twice among users and it contains at least two crash types, we consider it as frequent.

Once the sets of frequent sub-sets of crash types are identified, we use the crash times of crash types to validate these frequent sub-sets. Given a time window (*e.g.,* one day or one week), if all crash types of a sub-set occur within the time window, we keep this sub-set as valid.

Rule 4. Time-based co-occurrence of crash types comparison
Given two crash types CT_A and CT_B, if they are in a frequent sub-set of crash types SubC and co-occurred within a given time window, they are correlated.

To assess the performance of Rule 4, similar to RQ1, we rank the Firefox crash types based on their creation date and apply Rule 4 to the crash types one by one to mimic the current practice. Older crash types are processed first. We test a crash type against all the other crash types to identify its crash correlation group. The obtained crash correlation groups are compared to *Developer-defined CCGs* and precision and recall are computed using Equation (1) and Equation (2). Since Eclipse's stack traces are mined from comments contained in bug reports and because the occurrence time of these stack traces (*i.e.*, the crash events) is not the same as the time at which the stack traces were posted on bug reports, we cannot apply Rule 4 on Eclipse data. Rule 4 requires the exact occurrence time of crash events. We also apply successively Rule 1, Rule 2, Rule 3 and Rule 4 on crash types one by one to identify crash correlation groups. Based on the results in **RQ1**, the threshold values of relative support and the number of crash reports are set to be 0.5 and 30 respectively for Rule 3. When four rules are combined together, two crash types are in a crash correlation group as long as they satisfy one of four rules.

Also, Rule 4 is dependent on the threshold value of the time window used during the validation of frequent sub-sets of crash types. To measure the impact of threshold selection on our results, we perform a sensitivity analysis. Precisely, we repeat the evaluation of Rule 4 using time windows of one day, three days, one week and two weeks.

Table 7: Precision and Recall of Rule 4 for Different Length of Time Windows on Firefox.

Length of Time Window	Precision (%)	Recall (%)
One Day	52	58
Three Days	45	62
One week	42	80
Two weeks	40	84

Table 8: Precision and Recall of using Rule 1, Rule 2, Rule 3 and Rule 4 together for Different Length of Time Windows on Firefox.

Length of Time Window	Rule 1 + Rule 2 + Rule 3+ Rule 4	
	Precision(%)	Recall(%)
one day	88	87
Three Days	84	87
One Week	82	87
Two weeks	79	92

Findings. Table 7 shows that the length of the time window affects the precision and recall of Rule 4. With the decrease of the length of the time window, the precision increases. However, the recall decreases as more false positives are also introduced. This result was expected since a wider time window retains more frequent sub-sets of crash reports. Although the result indicates that the precision improves as the time window gets smaller, we could not test Rule 4 with a time window smaller than 1 day, such as 1 hour or 10 hours, due to the size of our dataset. When the time window is set to be smaller than 1 day, very few sub-sets of crash reports are returned. Table 8 shows that the Rule 4 improves the recall obtained with Rule 1, Rule 2 and Rule 3, but it decreases the precision as more false positives are introduced.

RQ3. Can we identify correlated crash types using the textual similarity between users comments about the crash events?

Motivation. Comments posted for a crash report or a bug report from users provide valuable information about the description of crashes and the varied scenarios in which they occurred. In this research question, we investigate the possibility of using textual similarity between user comments of crash types to identify crash correlation groups.

Analysis Approach. To answer this research question, we introduce a rule to identify crash correlation groups using the textual similarity between comments provided by users about the crash types.

Each crash type has its textual description which is a set of user comments. We merge the set of user comments into a single document. Each document has a set of terms $\{TM_1^i, TM_2^i, \ldots, TM_m^i\}$; m is the total number of terms in the textual description. We have a mapping between a crash type and a set of terms.

We use vector space model [15], a widely used technique in traditional information retrieval, to calculate the textual similarity between crash types. In the vector space model, each document (*i.e.,* a crash type in our case) is represented as an N-dimensional vector, where N is the number of unique terms appearing in all the documents and W_i, where $1 < i \leq N$, is the weight of the i^{th} term in the vector $\langle$ W$_1$, ..., W$_N$ $\rangle$ and defined by Equation 3.

$$W_i = TF_i \times IDF_i \tag{3}$$

In Equation 3, TF is the Term Frequency value and IDF is the Inverse Document Frequency value. The *Term Frequency* is the frequency of a term appearing in a document. The *Inverse Document Frequency* diminishes the weight of terms that occur very frequently in the whole corpus and increases the weight of terms that occur rarely. We calculate the TF_i as shown in Equation (4) and the IDF_i as shown in Equation (5) for each term.

$$TF_i = \frac{|\{occurrences\ of\ i_{th}\ term\ in\ the\ document\}|}{|\{total\ terms\ in\ the\ document\}|} \tag{4}$$

$$IDF_i = \log\left(\frac{|\{total\ documents\ in\ the\ corpus\}|}{|\{documents\ having\ the\ i_{th}\ term\}|}\right) \tag{5}$$

After the vectors are created for each document (*i.e.*, a crash type in our case), we can calculate the similarity of a pair of documents through a formula defining the similarity of two vectors. Typically, for two vectors $V_1 = \langle$ W$_1$1, W$_1$2 ..., W$_1N$ $\rangle$ and $V_2 = \langle$ W$_2$1, W$_2$2 ..., W$_2N$ $\rangle$, the similarity of V_1 and V_2 equals the value of the Cosine similarity [16], defined in Equation 6, of V_1 and V_2.

$$Sim = \frac{\sum_{i=1}^{n} W_1 i \times W_2 i}{\sqrt{\sum_{i=1}^{n} (W_1 i)^2} \times \sqrt{\sum_{i=1}^{n} (W_2 i)^2}} \tag{6}$$

Rule 5. Textual similarity of crash types comparison
*Given two crash types CT_A and CT_B, if the similarity value of their textual description is greater than a threshold value (*e.g., *0.75), they are correlated.*

To assess the performance of Rule 5, we process all the Firefox crash reports and Eclipse bug reports for each crash type and rule out any crash types without user comments. Second, we construct a user comment document CM of a crash type by merging user comments from each crash report. Third, we turn user comment documents into vectors and compute the similarity value for every pair of crash types.

To reduce the effect of word inflection on the textual similarity calculation of user comments of crash types, we conduct word normalization on the user comments from Firefox crash reports and Eclipse bug reports. More specifically, we conduct word tokenization to parse user comments into word tokens by splitting them using delimiters like space, punctuation mark, etc. Second, we remove non-English words using Wordnet[4], a large lexical database for English. Finally, we remove stop words and use the Morpha Stemmer[5] to stem the words to their root form.

Finally, we test each crash type against all the other crash types to identify crash correlation groups. We also apply successively Rule 1, Rule 2, Rule 3, Rule 4 and Rule 5 on Firefox crash types, and Rule 1, Rule 2, Rule 3 and Rule 5 on Eclipse crash types, one by one to identify crash correlation groups. When the rules are combined together, two crash types are in a crash correlation group as long as they satisfy one of the rules. We compare the obtained crash correlation groups to *Developer-defined CCGs* and compute the precision and the recall of the rule using respectively Equation (1) and Equation (2).

Similar to Rule 3 and Rule 4, Rule 5 is dependent on a threshold value. Therefore we perform a sensitivity analysis to measure the impact of threshold selection on the results. Precisely, we repeat the evaluation of Rule 5 using similarity threshold values of: 0.7, 0.75, 0.8, 0.85, 0.9, 0.95.

Rule 5 is also dependent on the number of crash reports of each crash type that are processed to extract users comments. A higher number of crash

[4] http://wordnet.princeton.edu/
[5] http://mvnrepository.com/artifact/edu.washington.cs.knowitall/morpha-stemmer

reports is likely to produce more users comments, which in turn will probably produce more meaningful terms. Therefore, we perform another sensitivity analysis to measure the impact of the number of processed crash reports on the results of Rule 5. Precisely, we repeat the evaluation of Rule 5 using 30, 50, 100 and all crash reports respectively.

Table 9: Precision and Recall of Rule 5 for Different similarity values when all the crash reports of a crash type are processed (to extract user comments). SV stands for similarity value.

SV	Firefox		Eclipse	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
0.7	42	57	38	34
0.75	52	55	40	32
0.8	54	48	40	32
0.85	55	45	45	28
0.9	60	40	45	28
0.95	62	32	46	26

Table 10: Precision and Recall of Rule 5 for Different Number of Crash Reports when the similarity threshold is 0.75. NCR stands for Number of Crash Reports. All means all the crash reports of a crash type in our corpus.

NCR	Firefox		Eclipse	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
30	32	38	24	18
50	36	42	30	22
100	46	50	36	30
All	52	55	40	32

Table 11: Precision and Recall of using Rule 1, Rule 2, Rule 3, Rule 4 and Rule 5 together for Different similarity values when all the crash reports of a crash type are processed (to extract user comments). SV stands for similarity value.

SV	First Three Rules+Rule 4+Rule 5 Firefox		First Three Rules+Rule 5 Eclipse	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
0.7	65	92	57	65
0.75	67	92	60	65
0.8	70	92	60	65
0.85	72	92	62	65
0.9	75	92	62	65
0.95	77	92	64	65

Findings. Table 9 shows that precision increases and recall decreases when the threshold similarity value is increased. This is an expected result frequently observed in Information Retrieval (IR) studies. A high similarity threshold value generally reduces the rate of false positives but increases the number of false negative.

Table 10 shows that the number of crash reports, that are processed to extract users comments, affects the performance of Rule 5. The more a crash type is commented, the higher will be the number of terms in the user comment document of this crash type, which in turn increases the odds of identifying a similar crash type using Rule 5.

Table 11 shows that using Rule 5 cannot improve the results obtained by using others rules, because all the correct Crash Correlation Groups identified by Rule 5 can also be identified by using other rules together (*i.e.*, Rule 1 + Rule 2 + Rule 3 on Eclipse). However, when we combine Rule 1 and Rule 5 together, we obtain a precision of 80% and a recall of 74% on Firefox, and a precision of 79% and a recall of 68% on Eclipse, when the similarity value is 0.95 and 50 crash reports of each crash type used for mining user comments. Rule 5 can improve the results obtained with Rule 1.

Based on the results in RQ1, RQ2 and RQ3, the combination of Rule 1, Rule 2 and Rule 3 is the best combination. When the threshold values of Rule 3 are set to be 0.5 for relative support and 30 for the number of crash reports, the combination achieves a precision of 94% and a recall of 90%.

RQ4. Can the 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 has 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 system [18]. 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] has shown that buggy files are often in the top 10 frames of failing stack traces.

In this research question, we explore the possibility of using correlated crash types for localizing buggy files. 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.

Analysis Approach. To answer this question, 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

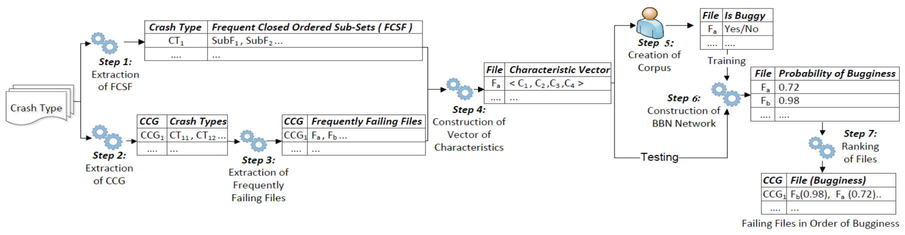

Crash Type
Step 1: Extraction of FCSF
Crash Type | Frequent Closed Ordered Sub-Sets (FCSF)
CT₁ | SubF₁ , SubF₂ ...
Step 2: Extraction of CCG
CCG | Crash Types
CCG₁ | CT₁₁ , CT₁₂ ...
Step 3: Extraction of Frequently Failing Files
CCG | Frequently Failing Files
CCG₁ | Fa , Fb ...
Step 4: Construction of Vector of Characteristics
File | Characteristic Vector
Fa | < C₁ , C₂ ,C₃ ,C₄ >
Step 5: Creation of Corpus
File | Is Buggy
Fa | Yes/No
Training
Testing
Step 6: 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. 7: Overview of the steps of BFFinder; CCG stands for Crash Correlation Group

and derived the bug localization method Buggy Files Finder (BFFinder) presented below. BFFinder analyzes correlations between crash types and builds a Bayesian Belief Network (BBN) [22] to compute the probability that a file appearing in a failing stack trace is buggy. We apply BFFinder on Firefox and Eclipse separately. Figure 7 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 applied on each crash type to extract its set of frequent closed ordered sub-sets of frames.

Step 2. Identification of crash correlation groups. In this paper, we propose five rules to identify crash correlation groups. More specifically, Rule 1, Rule 2, and Rule 3 are applied on the signatures of the crash types and their stack traces to identify crash correlation groups, Rule 4 is applied on the co-occurrences of crash types and Rule 5 is applied on the user comments of crash types. In this step, we apply Rule 1, Rule 2 and Rule 3 together to identify crash correlation groups, due to the promising results of using them together.

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. 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.

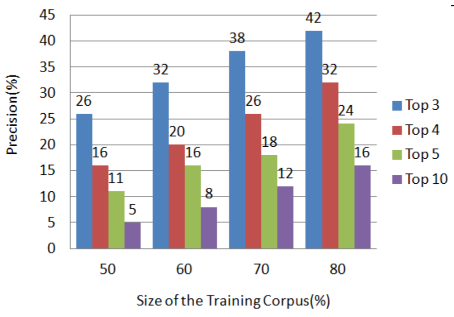
Step 6. Construction of a Bayesian Belief Network to rank files. The vector of characteristics obtained in **Step 4** are used to structure a BBN. The input nodes of this BBN correspond to the four dimensions of a vector of characteristics, while the output node is the probability of a file being buggy. This probability is used to rank the files.

Step 7. Ranking of file 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.

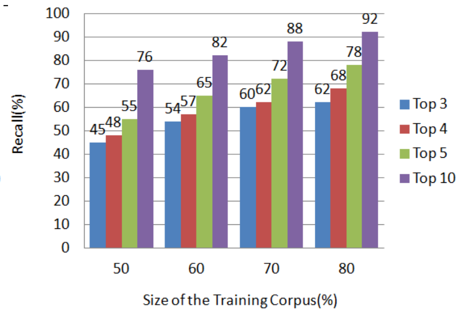
The construction of BFFinder is guided by the following observations made during the manual examination of the Firefox sample of 40 crash types with 400 stack traces:

– *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.,* on Eclipse stack traces [4]) , 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 with 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; In **Step 2**, we apply Rule 1, Rule 2 and Rule 3 together to identify crash correlation groups, due to the promising results of using them together. 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

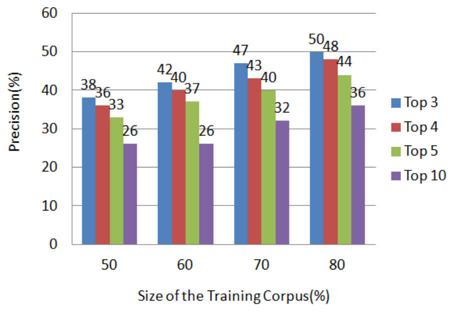


(a) Precision on Firefox

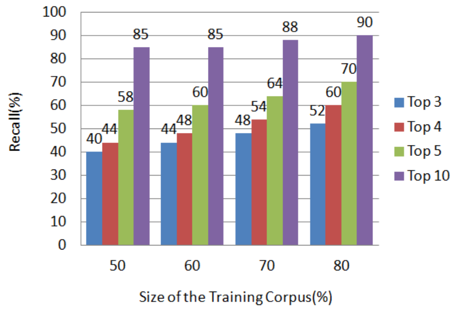


(b) Recall on Firefox

Fig. 8: Precision and Recall of Top 3, Top 4, Top 5 and Top10 Frames Candidate Reported by BFFinder for Different Training Corpora on Firefox.



(a) Precision on Eclipse



(b) Recall on Eclipse

Fig. 9: Precision and Recall of Top 3, Top 4, Top 5 and Top10 Frames Candidate Reported by BFFinder for Different Training Corpora on Eclipse.

buggy files (from Eclipse and Firefox) extracted from change logs as our gold standard (*i.e.*, see Section 3.2.4), we compute the k-precision and the k-recall of BFFinder following Equation (7) and Equation (8).

$$k - precision = \frac{\#\, of\ buggy\ files\ in\ top\ k\ results}{k} \tag{7}$$

$$k - recall = \frac{\#\, of\ buggy\ files\ in\ top\ k\ results}{|\{buggy\ files\}|} \tag{8}$$

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 3.2.4) 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 recall values 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.

Fig. 8 and Fig. 9 shows results of precision and recall for top 3, top 4, top 5 and top 10 frames respectively, using different training corpora. These results show that precision and recall increases with the size of the training corpus, meaning that when more information about the location of past bugs is available, the precision and 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.

RQ5. Can the correlated crash types help identify duplicate bug reports?

Motivation. Due to the large number of existing bug reports, it is challenging for triaging teams to examine all of the existing bug reports to detect duplications of bug reports or related bug reports (*i.e.,* bug reports having "blocks" or "depend on" relationships[6] among them). An efficient approach of detecting duplicate or related bug reports can reduce both the workload of triagers and the possibility of passing duplicate bug reports onto bug fixers. In this research question, we explore the possibility of using the correlations between crash types (*i.e.,* crash correlation groups) to help identify duplicate bug reports.

Analysis Approach. To answer this question, we introduce the following two relations on crash correlation groups.

Same group relation. If a set of bug reports is assigned to a crash correlation group, we consider these bug reports are duplicate or related.

Contain relation. Given two crash correlation groups $CCG_1 = \{CT_1^1, CT_2^1, \ldots, CT_m^1\}$ and $CCG_2 = \{CT_1^2, CT_2^2, \ldots, CT_n^2\}$, where m is the number of crash types in CCG_1 and n is the number of crash types

[6] http://eigen.tuxfamily.org/index.php?title=Bugzilla

in CCG_2, if $\exists k, l$, with $1 < k \leq n$, $1 < l \leq m$, and (CT_1^1=CT_k^2 or CT_1^1 $\subset CT_k^2$ or $CT_k^2 \subset CT_1^1$)$\wedge \ldots \wedge$(CT_m^1=CT_l^2 or $CT_m^1 \subset CT_l^2$ or $CT_l^2 \subset CT_m^1$), we consider CCG_2 *contains* CCG_1 and the bug reports associated with them are duplicated or related, where $\subset$ (*i.e. contains* relation between crash type signatures) is defined in ***Rule 1***.

For example, we have two crash correlation groups:
CCG_1= { nsQueryInterface::operator(), nsContentUtils::CanCallerAccess, nsContentUtils::CanCallerAccess(nsPIDOMWindow*) }, and its bug report id 612383.
CCG_2= {nsQueryInterface::operator(), nsContentUtils::CanCallerAccess, nsDOMConstructor::Create(unsigned short const* nsDOMClassInfoData const* nsGlobalNameStruct const* nsPIDOMWindow* nsDOMConstructor**) }, and its bug report id 606421.

Since nsContentUtils::CanCallerAccess (from CCG_2) $\subset$ nsContentUtils:: CanCallerAccess(nsPIDOMWindow*)(from CCG_1), so the above two groups have a **Contain Relation** (*i.e.* CCG_2 *contains* CCG_1), and their assigned bug reports are duplicated.

To assess the performance of using these two relations to identify duplicate bug reports and related bug reports, we perform two experienments:

Experienment 1: We identify these two relations from *Developer-defined Crash Correlation Groups (CCGs)* and use these relations to predict pairs of duplicate bug reports and pairs of related bug reports.

Experienment 2: We use Rule 1, Rule 2 and Rule 3 together to identify crash correlation groups, because the combination of Rule 1, Rule 2 and Rule 3 can identify more correct crash correlation groups than other combinations of rules do. Based on the results in RQ1, the combination of three rules can identify 90% of *Developer-defined Crash Correlation Groups.* We then identify the two relations between crash correlation groups to predict pairs of duplicate bug reports and pairs of related bug reports.

The obtained pairs of duplicate bug reports and related bug reports are compared with the ones mined from Firefox crash reports and Eclipse bug reports separately. The precision and recall are computed using Equation (9) and Equation (10).

$$precision = \frac{|\{correct\, pairs\} \bigcap \{retrieved\, pairs\}|}{|\{retrieved\, pairs\}|} \tag{9}$$

$$recall = \frac{|\{correct\, pairs\} \bigcap \{retrieved\, pairs\}|}{|\{correct\, pairs\}|} \tag{10}$$

Findings. Table 12 shows the results of identifying bug report duplication and related bug reports using *Developer-defined crash groups* (*i.e.* our gold standard for validation our approach). It confirms that using crash correlation groups can help identify bug report duplication and related bug reports. Our method for bug report duplication identification has a better precision and recall on Firefox than Eclipse.

Table 12: Precision and Recall of duplicate bugs and related bugs identification using *Developer-defined crash correlation groups.*

	Firefox		Eclipse	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
Duplicate bugs identification	55	50	38	47
Related bugs identification	32	55	52	48

Table 13: Precision and Recall of duplicate and related bug report identification using crash correlation groups generated by our rules: Rule 1+ Rule 2 + Rule 3.

	Firefox		Eclipse	
	Precision(%)	Recall(%)	Precision(%)	Recall(%)
Duplicate bugs identification	51	45	30	34
Related bugs identification	26	50	45	37

Table 13 presents the results of identifying bug duplication and related bugs using the crash correlation groups generated by our proposed rules Rule 1, Rule 2 and Rule 3 together. The results are lower than ones in Table 12, because the identified crash correlation groups using Rule 1, Rule 2 and Rule 3 together contain false groups compared with *Developer-defined crash groups*, which confirms that the number of crash correlation groups affect the results of our approach for identifying duplicate and related bugs.

5 Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [23].

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.* [12]. 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, 12]. We use a heuristic [10] based on “install age”, “crash times”, configuration and architecture of crashing systems to identify the unique users of our studied versions of Firefox. The rule 4 of our study critically relies on the identification of users reporting the crash types. In [10], the heuristic has been validated, but more validations are needed to strengthen the findings.

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 [24], to obtain the same data for the same releases. Eclipse bug reports from the 2008 MSR Mining Challenge are also publicly available.

6 Related Work

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

6.1 Analysis of Field Crash Reports

Many techniques have been proposed to prioritize groups of similar crash reports during debugging activities. Podgurski et al. [25] introduced a failure clustering approach to group similar crash reports together in order to fix the larger groups. Kim et al. [26] 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 [10] 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. All of the above approaches focus on grouping field crash reports and prioritize the groups of crash reports for bug fixing. However, our approach of this paper is to identify relations among crash types (*i.e.*, a group of simialr crash reports is considered as a crash type) for bug fixing and bug report duplication identification. Furthermore, 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.

6.2 Bug Correlation, Duplication and Localization

Bug correlation and bug localization have been researched extensively. Lee and Soffa [27] proposed a bug correlation algorithm to identify causal relationships among bugs in a software system. Liblit et al. [28] studied predicate patterns in correct and incorrect execution traces and proposed an algorithm to identify the predictors of a bug. They claim that their proposed algorithm can be used to detect a variety of both anticipated and unanticipated causes of failures. Ball et al. [29] 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. [30,31] 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 failing test cases. Nessa et al [32] 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, these techniques rely highly on source code instrumentation, predicates, and coverage reports, or successful traces, which limits their applicability of only analyzing crashing threads from crash reports for bug localization. In this paper, our apporach only analyzes crashing threads and do not require source code analysis.

Bug report duplication has been researched extensively. Wang et al. [19] used both information retrieval techniques and execution traces to detect duplicate reports. however due to the difficulty of obtaining execution traces for existing reports, C. Sun et. al [20] proposed to use a discriminative approach comparing textual similarity descriptions of the bug reports. C. Sun et. [21] used not only text but also other features that are available in BugZilla, e.g., version of the product or the priority of the report, to identify duplicate bug reports, and they extended one of the latest textual similarity measures in information retrieval for retrieving structured documents namely BM25F. Similar to our study, they all applied text mining techniques to measure text similarities, however, in our study, we explore the possibility of using correlations between crash types to help identify duplicate bug reports and related bug reports.

6.3 Analysis of Stack Traces

The use of stack traces by developers during bug fixing activities has been investigated to a great extent. Schröter 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 report 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. Glerum et. al. [33] introduced the Windows Error Reporting (WER) system which groups detailed crash reports using a bucketing algorithm. The bucketing algorithm uses multiple heuristics specific to the application supported by WER and updated by developers manually. Dang et al. [34] propose ReBucket which is a method for clustering crash reports based on call stack similarities to improve the accuracy of bucketing. Some visualization techniques have also been proposed by Chan et al. [35] and Kim et al. [36] 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 five rules to identify crash groups using an analysis of failing stack traces.

7 Conclusion and Future Work

The analysis of crash reports for bug fixing is a very challenging task that requires a large amount of manual work from developers. In this study, we investigate three crash type properties: stack traces, time and text to derive rules to identify correlated crash types automatically. We propose five rules: Crash Type Signature Comparison (*i.e.*, Rule 1), Top Frame Comparison (*i.e.*, Rule 2), Frequent Closed Ordered Sub-Set Comparison (*i.e.*, Rule 3), Time-based Co-occurrence of Crash Types Comparison (*i.e.*, Rule 4) and Textual Similarity of Crash Types Comparison (*i.e.*, Rule 5).

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 rules to identify correlated crash types. Using a Bayesian Belief Network, BFFinder computes and ranks files from stack traces based on their probability to be buggy. Furthermore, we apply the relations between crash correlation groups to identify duplicate bugs and related bugs.

We conducted a case study using Firefox and Eclipse to verify our proposed rules and methods for localizing bugs and identifying duplicate bugs. We found that when applied together, the first three rules achieve a precision of 91% and a recall of 87% for Firefox, and a precision of 76% and a recall of 61% for Eclipse. The first three rules do not require the analysis of a large number of crash reports. High precision and recall is achieved with as little as 10 crash reports per crash type. The fourth rule, identifying frequent sub-sets of crash types reported by users, can achieve a high recall (*i.e.*, 84%) when the crash times of these crash types are within a two week time window. The fifth rule investigates the possibility of using textual similarity of crash types to group them. The highest precision it can obtain is 62% when the threshold value of the clustering algorithm is set to 0.95.

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 recall 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. Moreover, our method for identifying duplicate bugs can achieve a precision of 55% and a recall of 50% on Firefox and a precision of 35% and a recall of 47% on Eclipse.

In future work, we plan to implement our proposed rules and our bug localization method BFFinder into a tool to assist development teams during the triaging of crash reports and the fixing of bugs.

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

References

1. Connecting with customers, http://www.microsoft.com/mscorp/execmail/2002/1002customers.mspx, last accessed on March 27,2012.
2. Firefox Stability Improvement, http://blog.mozilla.com/metrics/2010/04/08/dramaticstabilityimprovementsinfirefox/, 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. A. Schröter, N. Bettenburg, R. Premraj (2010), Do stack Traces Help Developers Fix Bugs?. MSR 2010: 7th IEEE Working Conference on Mining Software Repositories, pages 118-121, 2010.
5. M.Brodie, S.Ma, L.Rachevsky, J. Champlin (2005), Automatic Problem Determination Using Call-Stack Matching, Journal of Network and System Management, Volume 13, No 2, June 2005.
6. T. Dhaliwal, F. Khomh, Y. Zou (2011), Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox, Proc. the 27th IEEE international Conference on Software Maintenance, Williamsburg, VA, USA. September 25-30, 2011.
7. J.B.Kruskal (1983), An Overview of Sequence Comparison:Time Warps, String Edits, and Macromolecules, SIAM Review. Volume. 25, No. 2, pages 201-237, April 1983.
8. S. Wang, F. Khomh, Y. Zou (2013), Improving bug localization using correlations in crash reports, Proceedings of the 10th IEEE Working Conference on Mining Software Repositories, pages 247-256, San Francisco, CA, USA, 18-19 May 2013.
9. Web browsers (Global marketshare), Roxr Software Ltd., http://bit.ly/81klgi ,Retrieved on January 12, 2012.
10. F. Khomh, B. Chan, Y. Zou, A. E. Hassan (2011), An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox, Proceedings of the 18th Working Conference on Reverse Engineering, Lero, Limerick, Ireland, October 17-20, 2011.
11. N. Betttenburg, R. Premraj, T.Zimmermann, S. Kim (2008), Extracting structual information from bug reports, Proceedings of the 5th International Working Conference on Mining Software Repositories, Leipzig, Germany, May 10-11, 2008.
12. J. Śliwerski, T. Zimmermann, and A. Zeller (2005), When do changes induce fixes?, ACM SIGSOFT Software Engineering Notes, Volume 30, Issue 4, pages 1-5, July 2005.
13. J. Wang, J. Han (2004). BIDE:Efficient Mining of Frequent Closed Sequences, Proceedings of the 20th International Conference on Data Engineering, pages 79-90, 2004.
14. R. Agrawal, R. Srikant (1994), Fast Algorithm for Mining Association Rules in Large Databases, Proceedings of the 20th International Conference on Very Large Databases, pages 487-499, San Francisco, CA, USA, 1994.
15. V. Raghavan, M. Wong, A critical analysis of vector space model for information retrieval. Journal of the American Society for Information Science, Volume 37, Issue 5, pages 279-287, September 1986.

16. Cosine Similarity, http://en.wikipedia.org/wiki/Cosine_similarity, last access on October 27th, 2013.
17. L. Heyer, S. Kruglyak, S. Yooseph. Exploring Expression Data: Identification and Analysis of Coexpressed Genes, Genome Research, Volume 9, No. 11, pages 1106-1115, Cold Spring Harbor Laboratory Press, November 1999.
18. W. Eric Wong, V. Debroy (2009), A Survey of Software Fault Localization, Technical Report UTDCS-45-09, Department of Computer Science, The University of Texas at Dallas, November 2009.
19. X. Wang, L. Zhang, T. Xie, J. Anvik, J. Sun (2008), An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information, Proceedings of the 30th International Conference on Software Engineering, pages 461-470, Leipzig, Germany, May 10-18 2008.
20. C. Sun, D. Lo, X. Wang, J. Jiang, S. Khoo (2010), A discriminative model approach for accurate duplicate bug report retrieval, Proceedings of the 32th International Conference on Software Engineering, pages 45-54, Cape Town, South Africa, 2010.
21. C. Sun, D. Lo, S. Khoo, J. Jiang (2011), Toward more accurate retrieval of duplicate bug reports, Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, pages 253-22, Lawrence, Kansas, U.S.
22. D.Michie, D.J.Spiegelhalter, C.C. Taylor (1994), Machine Learning, Neural and Statistical Classification. Prentice Hall, 1994.
23. R.K.Yin (2002), Case Study Research:Design and Methods-Third Edition, 3rd edition, SAGE Publications, 2002.
24. Mozilla Crash Reporting Server. http://crash-stats.mozilla.com/products/Firefox, last accessed on March 22, 2012.
25. A. Podgurski, D.Leon, P.A. Francis, W.Masri, M.Minch, J.Sun, B. Wang (2003), Automated Support for Classifying Software Failure Reports, Proceedings of the 25th International Conference on Software Engineering, pages 465-475, 2003.
26. D. Kim,X. Wang, S. Kim, A. Zeller, S.C. Cheung, S. Park (2011), Which Crashes Should I Fix First?:Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts, IEEE Transactions on Software Engineering, Volume 37, No.3, June 2011.
27. W. Le, M. L. Soffa (2010), Path-Based Fault Correlation, Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering, Santa Fe, New Mexico, USA, November 2010.
28. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. I. Jordan (2005), Scalable Statistical Bug Isolation, Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 15-26, Chicago, Illinois, USA, June 2005.
29. T.Ball, NM. Naik, S.K.Rajamani (2003), From symptom to cause: Localizing Errors in Counterexample Traces, Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 97-105, 2003.
30. J. A. Jones, M.J.Harrold (2005), Empirical Evaluation of the Tarantula Automatic Fault-localization Technique, proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering, pages 273-282, December, 2005.
31. J. Jones, M.J.Harrold, J.Stasko (2002), Visualization of test information to assist fault localization, Proceedings of the 24th International Conference on Software Engineering, pages 467-477, Orlando, Florida, May 2002.
32. S.Nessa, M. Abedin, W. Eric Wong, L. Khan, Y. Qi (2008), Software Fault Localization Using N-gram Analysis, Proceedings of the 3rd International Conference on Wireless Algorithms, Systems, and Applications, LNCS, pages 548-559, 2008.
33. K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, G. Hunt (2009), Debugging in the (very) large: ten years of implementation and experience, Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 103-116Big Sky, Montana, USA, ACM, 2009.

34. Y. Dang, R. Wu, H. Zhang, D. Zhang, P. Nobel (2012), ReBucket: a method for clustering duplicate crash reports based on call stack similarity, Proceedings of the 2012 International Conference on Software Engineering, pages 1084-1093, Zurich, Switzerland.
35. B.Chan, Y. Zou, A. E. Hassan, A. Sinha (2009), Visualizing the Results of Field Testing, Proceedings of the 18th International Conference on Program Comprehension, pages 114-123, Minho, India, November 2009.
36. S. Kim, T. Zimmermann, N. Nagappan (2011), Crash Graphs: An aggregated view of multiple crashes to improve crash triage, Proceedings of the 2011 IEEE/IFIP 41th International Conference on Dependable Systems and Networks, pp. 486-493, 2011.