Improving Bug Localization by Mining Crash Reports: An Industrial Study

Marcos Medeiros*, Uirá Kulesza*, Rodrigo Bonifacio†, Eiji Adachi*, Roberta Coelho*
**Federal University of Rio Grande do Norte* (Natal, Brazil)
Email: marcosamm@gmail.com, uira@dimap.ufrn.br, eijiadachi@imd.ufrn.br, roberta@dimap.ufrn.br
†*University of Brasília* (Brasília, Brazil)
Email: rbonifacio@unb.br

***Abstract*—The information available in crash reports has been used to understand the root cause of bugs and improve the overall quality of systems. Nonetheless, crash reports often lead to a huge amount of information, being necessary to consolidate the crash report data into groups, according to a set of well-defined criteria. Recent research work have proposed different criteria and techniques to group crash report data, making more effective the process of finding the root causes of a bug and showing the performance of the approaches in the context of open source applications (such as IDEs and web browsers). In spite of that, it is still not clear how these approaches perform in other application domains, such as enterprise systems. In this paper, we present an industrial study in this field. We tailor existing approaches to find and group correlated crash reports, and identify buggy files in the domain of web-based systems. We then evaluate the performance of the resulting criteria and technique in industrial settings – identifying and ranking the classes that are more likely to contribute to a crash and thus might need a fix. We also check if the methods changed by the developers to fix a bug are present in the stack traces of the crash report groups used to identify the buggy classes. Our study provides new pieces of evidence of the potential use of crash report groups to indicate buggy classes and methods using stack traces information. For instance, we successfully identify buggy classes with recall varying from 61.4% to 77.3%, considering the top 1, top 3, top 5, and top 10 suspicious buggy files identified and ranked by our approach. We also found that 80% of changed methods from the closed bug fix issues appeared in related stack traces of the crash report groups. Finally, the approach also received positive response from the project leaders of the evaluated projects to help their bug resolution processes.**

***Index Terms*—Software crash, Bug Correlation, Bug localization, Crash reports, Stack traces**

I. Introduction

Software developers can leverage crash reports as a mechanism to build a general comprehension about possible root causes of bugs and then to improve software quality by fixing them [1]–[3]. Indeed, developers might collect such an information using built-in automatic crash reporting tools or log-based tools that monitor the execution of software systems. Each crash report usually maintains a set of runtime information, such as the user requested functionality, execution date/time, and an associated stack trace. A stack trace is an ordered set of frames where each frame refers to a method signature. Developers often use the information available in crash reports to identify and correct existing bugs [4], [5].

Despite the benefits of using crash reports to automatically identify bugs and help with bug fixing tasks, crash reporting and logging tools usually need to deal with a large volume of crash reports [1], [6]. To mitigate this problem, crash reports can be grouped based on the similarity of their associated stack traces. The stack traces of grouped crash reports can then be used by developers to facilitate bug identification and fixing. Accordingly, over the last years, many research works have explored the use of stack traces to aggregate crash reports [7]–[12]; as well as to locate and correct bugs [5], [13]–[20].

For instance, Dhaliwal et al. [21] proposed a new way of automatically grouping crash reports of the Firefox browser—based on the Levenshtein's distance similarity algorithm of the stack traces. Their approach leads to a reduction of 5% in the time that is necessary to correct faults. Wang et al. [11], [12] proposed five rules to automatically group correlated crash reports. They conducted an empirical study with data from Firefox and Eclipse. Their approach identified correlated crash reports with a precision of 91% (Firefox) and 76% (Eclipse). The authors of the mentioned work also developed a methodology to identify buggy files, based on the dimensions collected for each file present in the stack traces. Similarly, Wu et al. [19] also studied the identification of faulty code using groups of crash reports. They proposed a method called CrashLocator to locate defective functions related to groups of crash reports—previously merged using stack trace information. They have developed approaches to expand the stack traces using static analysis, and discriminative factors to rank suspicious functions. They located 50.6%, 63.7%, and 67.5% of failures by examining the top 1, top 5, and top 10 functions recommended by CrashLocator, respectively.

In this paper, we report our experience on tailoring and applying an existing approach for grouping crash reports and finding buggy code in a different application domain: large scale proprietary web-based systems. We first tailored, improved, and implemented existing techniques to locate and rank buggy files using groups of crash reports [12], [19]. We then conducted an industrial study to answer two research questions: *(RQ1) What is the performance of our stack-trace approach to identify buggy code files in Java web-based systems?* and *(RQ2) To what extent do methods from the closed bug fix issues appear in the stack traces of the associated crash report group?*

Regarding the first research question—involving a coarse-grained (files) perspective, after analyzing the top 1, top 3, top 5, and top 10 suspicious files suggested by our approach, and applying one previously proposed stack traces grouping rule (Crash Type Signature), we obtained, respectively, average recall of 61.4%, 77.3%, 77.3%, and 77.3%, while the mean average precision were 41.4%, 55.4%, 55.5%, and 55.5%. With respect to the second research question—involving a fine-grained (methods) perspective, we found that 80% of changed methods from the closed bug fix issues appeared in related stack traces of the crash report groups.

Our findings support and generalize previous results to a new application domain. We give evidence about the efficiency of using groups of crash reports to correctly indicate buggy files and methods present in stack traces. As part of this work, we have presented the results of this industrial study to the project leaders of the investigated systems, and the decision is to use our approach in their software development process. Currently, the development teams of the partner company are using our list of suspicious buggy files and methods to correct bugs related to the stack trace groupings.

II. Approach

We customized and implemented an approach based on previous work [12], [19]: (a) to group crash reports related to the same software bug; and (b) to rank files suspected of causing crashes. The main goal is to facilitate the resolution of bugs. The crash reports are clustered by stack traces according to specific rules (see details in Section II-A). Also, a list of suspicious buggy files is generated based on the stack traces of the grouped crash reports (see details in Section II-B).

A. Crash Report Grouping

In the first step of our approach, we group crash reports using information available in the stack traces. The primary purpose is to aggregate strongly correlated crash reports and to reduce the time necessary for data processing. We process and group the crash reports in five ordered and cumulative levels, considering the characteristics of their associated stack traces, as shown in Fig. 1.

Identical Stack Trace. Initially, we group crash reports whose stack traces are identical (STA = STB). The signature that represents each group is the stack trace itself. In addition to the signature, we save the ids of each crash report belonging to the group and the dates of the oldest and most recent log occurrence.

Equivalent Signature. Nonetheless, after grouping identical stack traces, we realized distinct groups whose signatures were almost identical. They differ only in the identification number of the proxy, automatically generated accessor method, among others. Consider two stack traces STA and STB, where

- *at ...GeneratedMethodAccessor10184.invoke()* $\in$ STA
- *at ...GeneratedMethodAccessor10272.invoke()* $\in$ STB

That is, these lines of the stack traces only differ with respect to the numbers *10184* and *10272*. For these

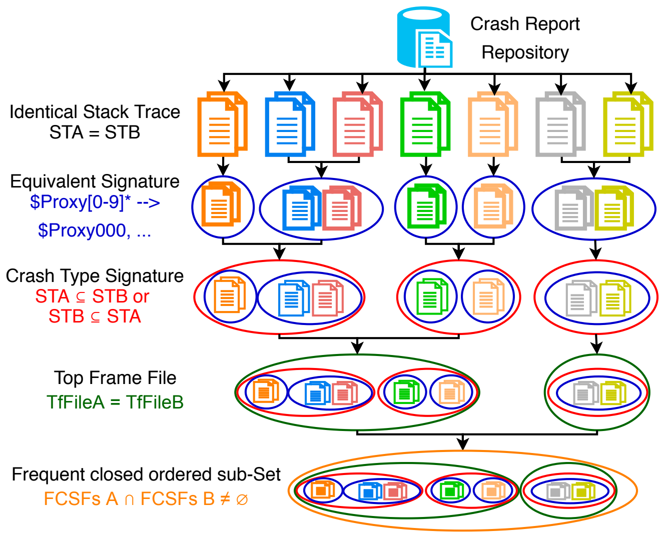


Fig. 1. Crash report grouping levels by stack traces

small differences, we also consider two stack traces (e.g., STA and STB) equivalent, and we group their crash reports. We use regular expressions (such as $Proxy[0-9]*, GeneratedMethodAccessor[0-9]*, $$EnhancerByCGLIB$$[0-9,az,A-Z]*, and $$FastClassByCGLIB$$[0-9,a-z,A-Z]*) to replace these numbers with 000 (e.g., leading to *at ...GeneratedMethodAccessor000.invoke()* in the examples above). We also aggregate groups with equivalent signatures into a "super-group" whose signature represents the two stack traces.

After that, we created the other three levels of grouping based on rules proposed by Wang et al. [12], always joining previously created groups. We have only used three rules out of 5 rules detailed in the previous work [12]) because, in the original work, the authors concluded that using only these first three specific rules together outperforms other configurations using the remaining rules. We detail the rules we use in our approach in what follows.

Rule 1 (*Crash Type Signature*) identifies similarities among correlated fault types when comparing strings of two crash types, and groups two stack traces STA and STB when one contains the other (STA $\subseteq$ STB or STB $\subseteq$ STA). To apply this rule, we only consider the rows of a stack trace with packages, classes, and methods (e.g., at java.lang.reflect.Method.invoke (Method.java:606)), since the rest might differ because of the idiom, for example. We also ignore the line number present in the stack trace, because simple file formatting, blank line insertion, or comment inclusion can change the line number without modifying any code statement. In other words, we did not make a simple comparison of strings to check if one stack trace contains the other.

Rule 2 (*Top Frame File*) correlates two fault types when they have the same qualified file name in the most inner frame of the call stack (exception signaler), that is, at the top of the stack. Consider two stacktraces STA and STB, where:

- *at s.p.ClassMBean.methodA(ClassMBean.java:280)* is the signaler of STA

- *at s.p.ClassMBean.methodB(ClassMBean.java:251)* is the signaler of STB

In both cases, the qualified file name is the same (*s.p.ClassMBean*). Rule 2 groups this kind of similar stack traces.

Rule 3 (*Frequent Closed Ordered Sub-Set*) takes into account whether two groups of crash reports have at least one common frequent closed ordered sub-sets of frames (FCSF), which is a type of frequent closed sequence [22], [23]. The idea of this rule is to correlate fault types whose majority of error stack traces have at least a subset of frequently called frames in the same order. For the resulting groups after the application of rule 2, we extracted the frequent closed ordered subsets of frames (FCSFs) using the Sparesort [24] implementation of the BIDE algorithm [22]. BIDE (BI-Directional Extension) is a general purpose and efficient algorithm for mining frequent closed sequences (in our case, sequences of stacktrace frames). We configure the BIDE parameters and thresholds using the same settings of a previous work [22].

B. Ranking suspicious files and methods

After grouping the crash reports using the procedures detailed in Section II-A, we mine information of the files present in the stack traces of the crash report groups. We use this information to rank the files that are more likely to produce the crashes in a given crash report group. To this end, our approach leverages three discriminative factors proposed by Wu et al. [19], though adapted to a coarse-grained level (files, instead of methods): Inverse Average Distance to Crash Point, Inverse Bucket Frequency, and File Frequency. In what follows, we detail these factors.

Inverse Average Distance to Crash Point (IAD): If a file appears closer to the crash point, it is more likely to cause the crash. The IAD factor measures how close a file is to the crash point:

$$IAD(f,B) = \frac{1}{1 + \sum_{j=1}^{n} DCP_J(f)/n} \tag{1}$$

where n is the number of crash reports of group B and $DCP_j(f)$ is the shortest distance between the file f and the crash point in the stack trace of the j^{th} crash report of group B.

Inverse Bucket Frequency (IBF): If a file appears in stack traces caused by many different faults, it is less likely to be the cause of a specific fault. The IBF factor measures the discriminative power of a file concerning all groups of crash reports:

$$IBF(f) = log(\frac{\#B}{\#B_f} + 1) \tag{2}$$

where $\#B$ is the total number of groups, and $\#B_f$ is the number of groups whose stack traces contain the file f.

File Frequency (FF): If a file often appears in stack traces caused by a particular fault, then it is likely to be the cause of this fault. The FF factor measures the frequency that a file appears in stack traces of a group of crash reports:

$$FF(f,B) = \frac{N_{f,B}}{N_B} \tag{3}$$

where $N_{f,B}$ is the number of stack traces of group B that the file f appears. N_B is the total number of stack traces in group B.

The ranking of the suspicious files is accomplished using a combination of the three factors, generating a score for each file f present in the stack traces of group B:

$$Score(f) = IAD(f,B) * IBF(f) * FF(f,B) \tag{4}$$

This method assigns higher scores to files that appear more often in stack traces in a group, less frequently in stack traces in other groups, and closer to the crash point. The rank of suspected files containing crashes that triggered the crash report group is generated by calculating the score of each file in the stack traces and sorting them in descending order.

Suggesting suspicious methods. Although we use an algorithm to rank the files, regarding methods we follow a different strategy. For each of the ranked files, we identified the methods that appeared in the stack traces of the respective group of crash reports and suggest them to the developers, showing the most frequent ones first. This decision was motivated by the lack of access to the source code repository during the initial stage of the study. Nonetheless, we show that, even without access to source code repositories, its is possible to identify both suspicious files and methods using stack traces only.

III. Study Settings

The goal of this study is to investigate whether and to what extent information contained in crash reports can help developers locating bugs in enterprise web-based systems. Next, we present the details about the design of our industrial study. We first detail the research questions related to our investigation. Then, we present the characteristics of the analyzed projects/systems. Finally, we discuss our data collection and analysis procedures.

A. Research Questions

The following research questions are addressed in our study:

(RQ1) What is the performance of our stack-trace approach to identify buggy code files in Java web-based systems? - This research question investigates if our crash report grouping based approach can contribute to identify code/class files responsible for system crashes more efficiently. In particular, our study focuses on large scale industrial web-based systems to analyze whether previous results are also maintained for this domain.

(RQ2) To what extent do methods from the closed bug fix issues appear in the stack traces of the associated crash report group? - In this research question, we investigate if the methods that were modified to fix bugs during the maintenance of the systems also appear in the list of methods suggested by our approach. The suggested methods of our approach are the

most frequent in the stack traces of the grouped crash reports (Section II-B). This information might be useful to indicate if the possible faulty methods suggested by our approach are strongly related to the methods presented in the corrected bug issues of the systems.

B. Target Systems

We selected three large scale web-based Java systems as the target systems of our study. These systems were chosen for this study for two main reasons. First, regarding relevance, these are non-trivial systems in use for several years and their development teams keep track of crash reports in a database. They have also reported the difficulty to use the large amount of crash reports during the tasks of bug fixing. Second, regarding convenience, we have access to the crash reports and issue tracker tools of the target systems.

These web-based systems are used and customized by more than 50 government and academic institutions — supporting many and different business processes. They are implemented using mainstream Java enterprise technologies and frameworks, such as Java Server Faces, Spring, and Hibernate. Table I characterizes the target systems in terms of their size (i.e., number of classes and lines of code) and the average number of user requests (coming from a browser or a mobile app) per day in the main institution that use them. The company responsible for the development of the systems has about 120 developers and uses modern agile methodologies (scrum, kanban) and practices.

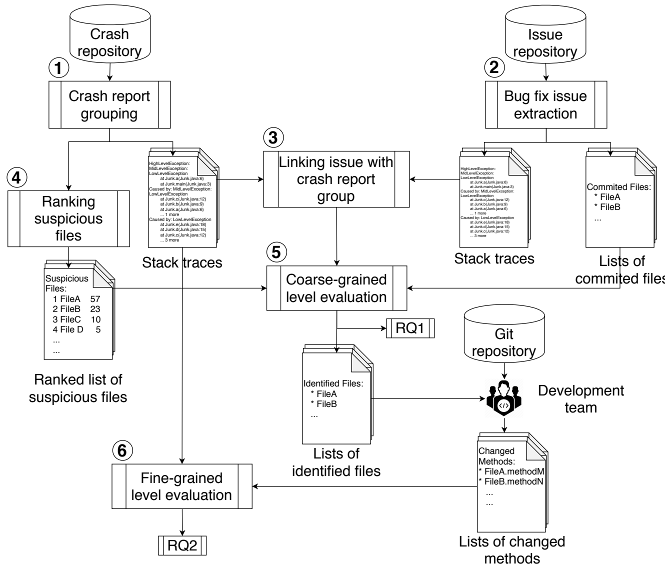
TABLE I
TARGET SYSTEMS CHARACTERIZATION

System	Classes	Lines of code	Avg. No. of Daily Requests
SYS1	7,305	1,155,790	1,193,825
SYS2	6,842	1,313,942	121,934
SYS3	3,827	667,810	79,264
Total	17,974	3,137,542	1,395,023

C. Study Procedures

In order to build a general understanding about the feasibility of using the stack trace for bug localization in the context of large scale web-based systems, we group crash reports, extract bug fix issues with stack traces and fixed files, link crash report groups with bug fix issues by stack traces, and, finally, we rank suspicious files. To answer RQ1, we evaluate the performance of the approach comparing the modified classes of resolved bug fix issues that have associated stack traces against a list of suspected buggy files that are present in the stack traces of the correlated crash reports. To answer RQ2, we obtained with the development team the fixed methods for each of the files identified by the approach, and we checked if they appear on the stack traces of the respective crash report group. Figure 2 shows an overview of the study procedure. Next we detail each step in the figure.

Crash report grouping. (Fig 2, Step 1) This step aims to group crash reports related to the same bug. In our study, we collect crash reports for all the studied systems from



Crash repository
1
Crash report grouping
Issue repository
2
Bug fix issue extraction
3
Linking issue with crash report group
4
Ranking suspicious files
Stack traces
Stack traces
Commited Files: * FileA * FileB ...
Lists of commited files
5
Coarse-grained level evaluation
Suspicious Files: 1 FileA 57 2 FileB 23 3 FileC 10 4 File D 5
Ranked list of suspicious files
RQ1
Git repository
Identified Files: * FileA * FileB ...
Development team
Lists of identified files
6
Fine-grained level evaluation
Changed Methods: * FileA.methodM * FileB.methodN
Lists of changed methods
RQ2

Fig. 2. Study overview

a ElasticSearch instance that stores all the web requests with associated crash reports. Each crash report has a stack trace and an associated URL. We use the ElasticSearch Rest API to query the crash reports created between April and December/2019. We then cluster the crash reports according to the procedures detailed in Section II-A.

Bug Fix Issue Extraction. (Fig 2, Step 2) The purpose of this step is to identify the bug fix issues, modified code files to fix the bug, and stack traces that are used to link the issues with crash report groups. In our study, we used a specific RedMine plugin that keeps informations of the files that a bug fix has changed. We used the Redmine/Chiliproject Java API[1] project to query the Redmine Rest API and find the issues created and closed between April and December/2019, which are labeled as "Errors" or "Known defect" and with the status "Closed". These two labels are used to distinguish issues that refer to bugs from other kind of issues (new features, improvements, documentation, among others). In summary, we looked for resolved bug fix issues for each of the target systems.

We queried these issues, collecting those that had some stack trace reported, and Git merge request information. We performed these steps by removing HTML tags and using regular expressions to identify and extract the stack traces, Git revision numbers, operation types, and committed files. Typically, small stack traces are very generic and would be linked to multiple groups, confusing the results. Besides, they seemed incomplete when compared to the others, so we only collect stack traces with at least five frames. We also ignore files added to the source code during a commit, since they could not have appeared in the stack traces of the crash report groups once they did not exist. In short, our study focused only on Java files that were modified.

[1]https://github.com/taskadapter/redmine-java-api

Linking Issues with CrashReport Groups. (Fig 2, Step 3). We link an issue to a crash report group whenever a stack trace STR in an issue is similar to at least one stack trace STG of a crash report group. We used three criteria to identify similarities. The first checks whether there is any group of crash reports whose signature STG is precisely equal to the stack trace STR reported in the issue (*STR = STG*). The second verifies if some group has the same signature to the stack equivalent to STR (*Equivalent(STR) = STG*). Finally, we match the stack traces and crash report group signature by ignoring specific detailed messages of the system (*ST_WithoutMessages(Equivalent(STR) = ST_WithoutMessages(STG)*), since might differ because of the idiom.

Ranking suspicious files. (Fig 2, Step 4). In this step, for each crash report group linked to a bug fix issue, we ranked the suspicious files, following the procedures we detailed in Section II-B. We generate lists with the Top 1, 3, 5 and 10 suspicious files. Top N ranked lists only contain code files from each analyzed system, that is, we do not rank files from third-party libraries. We select the system files filtering them by package name.

Coarse-grained level evaluation. (Fig 2, Step 5) We evaluate the performance of our algorithm for ranking suspicious buggy files by comparing the ranked list of suspicious files with the list of committed files. We use the following metrics:

Recall@N: The percentage of pairs *(Issue, CrashReportGroup)* where at least one of the files changed to fix a failure was discovered by examining the Top N (N = 1, 3, 5, 10) of the returned suspicious files. Thus, the higher the metric values, the better the fault localization performance.

Mean Average Precision [25], [26]: Mean of the average precision scores for a set of queries that quantifies the performance of information retrieval:

$$MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q} \tag{5}$$

where Q is the number of queries.

$$AveP = \frac{\sum_{k=1}^{n}(P(k) * rel(k))}{number\ of\ relevant\ documents} \tag{6}$$

where $P(k)$ is the precision at cut-off k in the list, and $rel(k)$ is an indicator function equaling 1 if the item at rank k is a relevant document, and zero otherwise.

In our case, Q is the number of pairs *(Issue, CrashReportGroup)*. The *relevant documents* are the files changed to solve the Issue, and the queries are the Top N (N = 1, 3, 5, 10) of the returned suspicious files for the CrashReportGroup. The higher the MAP value, the better the fault localization performance.

Fine-grained level evaluation. (Fig 2, Step 6). The purpose of this step is to verify whether the modified methods to fix the bug are among those suggested by our approach, that is, if they appear in the stack traces of the respective crash report group. For each one of the identified files, that is, the committed files present in the top N suspicious files, we request the development team to mine the Git code repository to identify the methods that have been changed in the bug-fixing commits. We then processed the stack traces of the groups that revealed buggy files to count the number of times a given method appears in the stack trace. We did this counting based on the full qualified name of methods, which includes the class and package names. We also compared the changed methods for each committed file with the methods present in the stack traces, to verify if they were the most frequent ones. Finally, we count how many methods from each of the identified files have appeared in the group's stack traces, even if it has not been changed in the commits.

IV. Results and Discussion

Here we present the main findings of our research. In Section IV-A we detail the assessment of our approach at the coarse-grained level (files), answering our first research question. In Section IV-B we present the results of our second assessment, at the fine grained-level (methods)—and thus answering our second research question.

A. First Assessment: Coarse-grained level

To answer the first research question, we aggregate crash reports into groups, extract bug fix issues, and link them according to the steps described in Section III-C. For each group linked to an issue, we rank suspicious files based on stack traces. Finally, we compared the list of committed files to fix the bug with the ranked list of suspected buggy files from the crash report group linked to the issue.

The total number of crash reports processed for SYS1, SYS2, and SYS3 were, respectively, 74 331, 6547, and 4144. Table II shows the total of crash reports and the number of groups at each level, using identical stack trace criteria, equivalent signature, rule 1, rule 2, and rule 3. Note that the configuration *rule 1+2+3* formed a small number of groups, reducing the number of crash report groups from 575 (after applying rule 1+2) to 539 (after applying rule 1+2+3). It is important to observe that the rules are applied cumulatively. That is, we always use the Identical Stack Trace and Equivalent Signature rules before applying rule 1.

TABLE II
CRASH REPORTS

System	CR[a]	Identical ST[b]	Equivalent ST	Rule 1	Rule 1+2	Rule 1+2+3
SYS1	74331	28374	7671	4409	334	308
SYS2	6547	3486	1009	745	123	121
SYS3	4144	1961	1540	918	118	110
Total	85022	33821	10220	6072	575	539

[a]Crash Report; [b]Stack Trace

We found 281(from a total of 3238 issues analyzed) bug fix issues with reported stack traces. In addition, 144 have both stack trace and associated code commits to fix the bug. In some cases, the stack traces are too small, they have less than five frames. Therefore we ignored them in the study. We also ignored non Java source code files linked to the bug-fixing commits. At the end, we selected 113 issues with stack traces with more than five frames, and whose bug fixes changed Java implementation files. Table III shows the number of (bug) fixed issues of each system, issues with reported stack traces by users or developers, issues with reported stack traces and changed files, and selected issues.

TABLE III
ISSUES

System	Issues with ST[a]	Issues with ST[a] & CF[b]	Selected Issues
SYS1	158	98	78
SYS2	96	31	21
SYS3	27	15	14
Total	281	144	113

[a]Reported Stack Trace; [b]Changed File

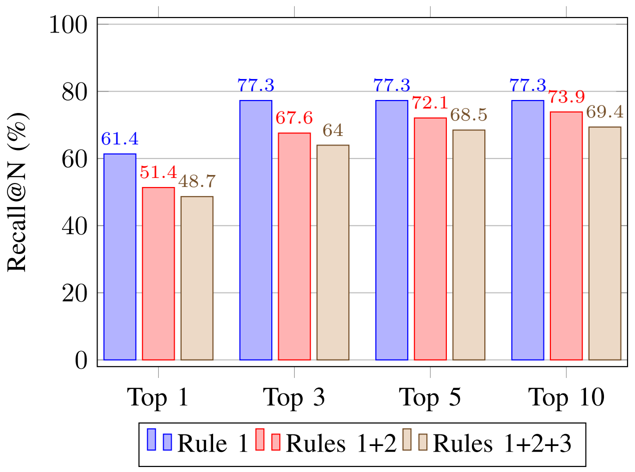
The last step in the development of our dataset was to link the issues with the groups. We conducted the study with the application of rule 1, rules 1 and 2, and the three rules together. Table IV shows the number of pairs formed among issues and crash report groups, the number of distinct issues and groups for each system, taking into account the groups formed up to the application of rule 1, rule 1+2 and rule 1+2+3. The formed links involved 6752 (8 % of the total) crash reports distributed in groups, and 113 (40 % of the total of issues with stack traces) selected bug issues. Using rule 1 only, some issues link to more than one group, so the number of groups (131) is greater than the number of issues (101), which is the same in all combinations, and more pairs between issues and crash report groups (132) are formed. Applying rule 2, some groups formed by rule 1 were aggregated into one, decreasing the number of groups (89) and pairs (111). It is important to mention that the number of issues linked to the same group has increased. Similar behavior occurred when we added the rule 3. However, the number of pairs has not changed since more than one issue has linked to the same group.

The average number of modified files per issue was 2.79 in our dataset, the median was 1 file, and the standard deviation of 5.22 files. In 62.37 % of the cases, the issues had a single file modified, 2 files were changed in 18.81 % of the cases, 3 files in 3.9 % of the cases, 4 files in 3.9 % of the cases, and more than 4 files in 10.89 % of cases.

For each crash report group in our dataset, we rank the suspicious files and we evaluate the performance of the ranking approach comparing the Top N suspicious files with files effectively changed to fix the bug (committed files).

We measure the overall performance of our approach by calculating the recall and MAP metrics considering all pairs (Issue, CrashReportGroup) for the three systems. Fig. 3 shows the values found for the recall and MAP metrics, considering the ranked list of Top with 1, 3, 5, and 10 suspicious files. It also exhibits the values for these metrics when applying rules 1 alone (blue bars), rules 1 and 2 together (red bars), and the three rules together (brown bars). It is important to notice that rule 1 alone obtained superior values for the recall and precision metrics, and that it was possible to get the best results for both metrics by analyzing only three suspicious files. The recall and MAP values decrease when we consider rules 1+2 and 1+2+3 for the Top 10 suspicious file configuration. Nonetheless, applying rules 1 and 2 together decreased the number of crash report groups from 6072 to 575. There is a clear trade-off here: although the approach for grouping crash reports reduces significantly the amount of information developers need to analyse, it leads to a negative impact on both metrics we use in our research (Recall@N and MAP).

We can also observe similar values for the recall and precision metrics when applying rule 1 for the Top 3, 5 and 10 suspicious files in Fig. 3. The reason for that is because of the use of Rule 1 (Crash Type Signature) groups very similar stack traces that keep stable the number of suspicious class files. With a smaller number of suspicious files to rank, during the comparison with the changed files from bug fix issues, they usually appear in the Top 3, and as a consequence, also in the Top 5 and 10, when applying rule 1.



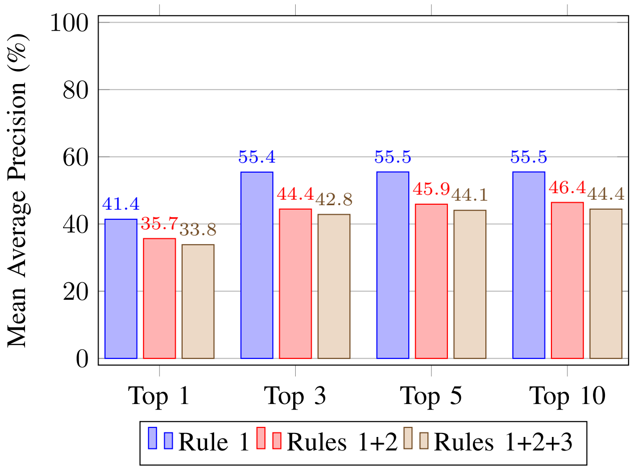


Fig. 3. Mean of Recall and Precision

We also measured the number of distinct issues for which it was possible to identify at least one changed file. We did this to investigate if the number of distinct issues also varies depending on the rules. The higher the number of found distinct issues, the better the fault localization performance.

TABLE IV
PAIRS FORMED WITH ISSUES AND CRASH REPORT GROUPS

System	rule 1			rule 1+2			rule 1+2+3		
	Pairs[a]	Distinct Issues	Distinct Groups	Pairs[a]	Distinct Issues	Distinct Groups	Pairs	Distinct Issues	Distinct Groups
SYS1	98	72	97	81	72	62	81	72	53
SYS2	20	19	20	19	19	17	19	19	17
SYS3	14	10	14	11	10	10	11	10	09
Total	132	101	131	111	101	89	111	101	79

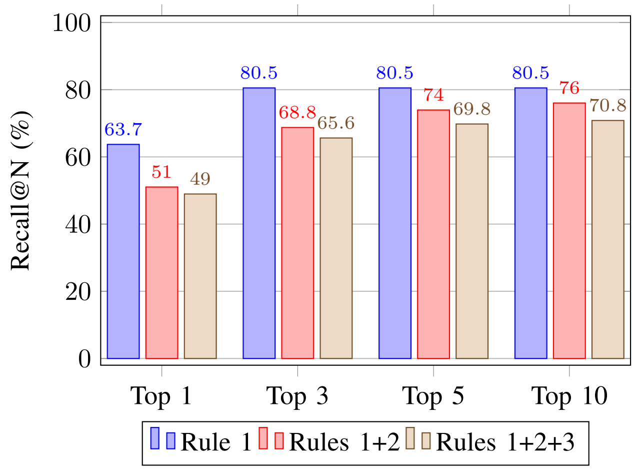
[a]*(Issue, CrashReportGroup)*

Table V shows the total number of issues per system, and the number of distinct issues for which it was possible to identify at least one file that introduced a bug looking at the first 1, 3, 5 and 10 suspicious files for each combination of rules analyzed. We can see that the number of issues decreases or remains the same in most cases when we add rules 2 and 3. For example, analyzing the Top 10 ranked list for SYS1, we found 57 issues using only rule 1. This value decreased to 53 issues when we added rule 2 and again decreased to 48 when rule 3 was added.

We also calculated recall and precision, considering only issues that had a maximum of three files modified to verify if the values changed significantly when the number of changed files per issue is smaller. In this case, the average of files changed per issue was 1.31, the median was 1, and the standard deviation was 0.56. Fig. 4 shows that the values of recall and precision metrics have improved in this scenario. Comparing with Fig. 3, we can see that the recall values varied between 0.31% and 3.26%, and the most significant differences obtained (80.5% to 77.3%) are related to the Top 3, 5, and 10 when applying only rule 1. The values of MAP varied more (between 4.35% and 8.06%) than the recall values, and the biggest differences also occurred applying only rule 1 - MAP values from 55.4% to 63.5%.

Answer to RQ1: Our study gives evidences that it is possible to identify faulty files from Java web-based systems—after grouping crash reports exclusively based on stack traces. When considering the top 3, top 5, and top 10 files, it was possible to identify one of the buggy files (that is, the file developers changed to fix an issue) in 77% of the cases (when considering the application of rule 1).

Given the obtained results, we investigated the pairs (Issue, CrashReportGroup) for which no files modified to fix the error appeared in the Top 10 suspicious files. This fact occurred for 30 pairs (22.72%) when we applied only rule 1 and in all cases no modified files appeared in the stack traces of the groups linked to the issue. Applying the rules 1 and 2 together, we noticed that the same situation occurred for 29 pairs (26.12%), where for 5 of them, the modified file appeared above the Top 10. Applying the three rules together, for 11 of 34 pairs (30.63%), at least 1 modified files appeared above Top 10. We noticed that with the application of rules 2 and 3, more stack traces were grouped, increasing the number of suspicious files. As a consequence, there was a change in the rank of suspicious



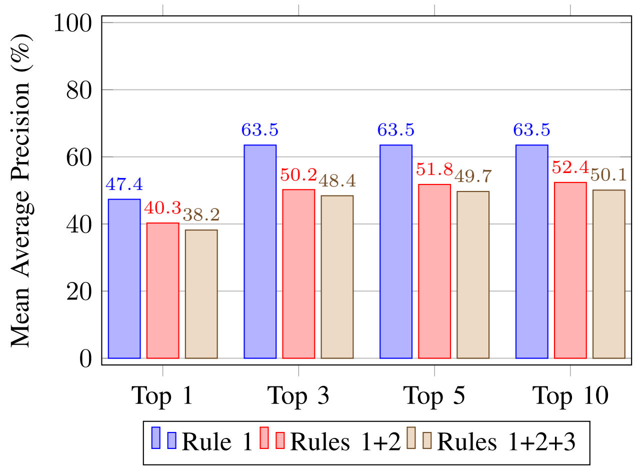


Fig. 4. Mean of Recall and Precision, considering issues that had a maximum of three modified files

files, removing from the Top 10 some modified files that fixed the bug. This happens because unrelated groups share the same top frame file and consequently are aggregated by rule 2.

Finally, we investigated the reasons that lead rule 1+2+3 to merge only a small group of crash reports. We found that only 4.9% of are FCSFs common to at least two groups had source code files from one of the studied systems. We observe that most of the clusters built by rule 3 occurred because of generic FCSFs, consisting of files other than the source code of the system. A large number of method calls present in the stack traces are from third-party library classes such as Apache, Java Server Faces, Hibernate or libraries developed by the company

TABLE V
DISTINCT ISSUES FOUND

System	Issues	Distinct issues found											
		Rule 1				Rules 1+2				Rules 1+2+3			
		Top 1	Top 3	Top 5	Top 10	Top 1	Top 3	Top 5	Top 10	Top 1	Top 3	Top 5	Top 10
SYS1	72	43	57	57	57	36	49	51	53	34	46	48	48
SYS2	19	13	14	14	14	11	15	15	15	10	14	14	15
SYS3	10	5	7	7	7	5	5	7	7	5	5	7	7
Total	101	61	78	78	78	52	69	73	75	49	65	69	70

itself and shared by various applications. This generic FCSFs may have caused the union of unrelated crash report groups.

B. *Second Assessment: Fine-grained level*

In collaboration with the development teams of the systems, we extracted the name of the methods that have been changed in the bug-fixing commits by mining these information from the issue tracker system and Git code repository. After that, we checked whether those methods appear on the list of suggested methods of our approach. The list represents the methods that appeared most frequently in the stack traces of the respective crash report group. Finally, we checked how many methods from each of the identified files were present on the stack traces of the respective crash report group.

Fig. 5 shows the frequency of changed methods from bug fix issues in stack traces of the crash report groups. It is possible to see that about 80% of the changed methods (the methods developers changed to fix a bug) appeared in the stack traces of the crash report groups with some variation depending on the considered rules. In particular, we found 79.53% of changed methods when using rule 1, 82.5% using rule 1+2, and 82.1% using rule 1+2+3.

Fig. 5 also highlights if the changed methods in the bug fix issues have been found as the most frequent, as frequent as other methods, or the least frequent in the stack traces groups. Our research reveals that the method changed to fix a bug is the most frequently found on stack traces of the crash report groups (from 55.79% to 57.48%) when compared to other methods of the same buggy file that also appears in stack traces. In addition, the changed methods that appeared the same number of times as other unchanged methods were present between 17 and 28 times in the stack traces groups. In only 8 cases, which occurred when applying rules 1 and 2, or rules 1, 2 and 3 together, the modified methods were not the most frequent, that is, some other unchanged method of the same file appeared more often.

Fig. 6 shows how many methods from each of the identified files were present on the stack traces of the respective crash report group considering the different rules. Regarding the groups formed only by rule 1, a single method of the changed file appeared in the stack traces in 62.20% of the cases. In 29.92%, two methods were found, and three or more appeared in 7.87%. We obtained very similar results using rules 1 and 2 in conjunction or the three rules together, a single method of the file appeared in stack traces in about 48% of cases. Two methods were found in about 25% of the stack traces, and we

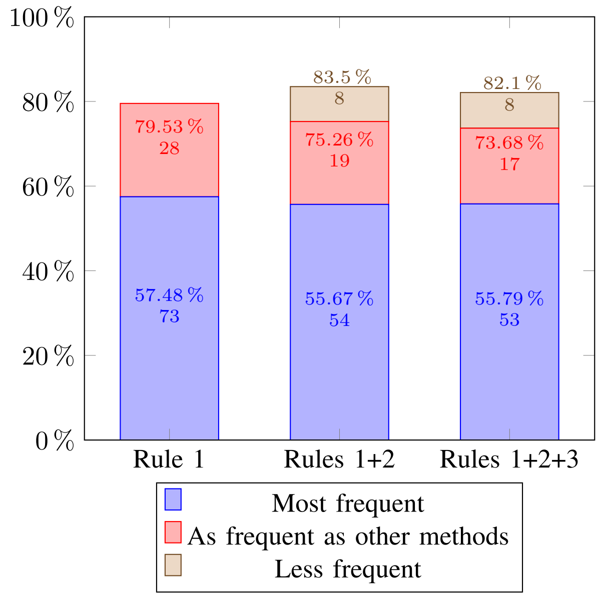


Fig. 5. Frequency of changed methods in crash report groups

also found three or more methods in about 25% of the cases. These results suggest that in most cases (between 74% and 92%) one or two methods from an changed file appear in the stack traces of the crash report groups.

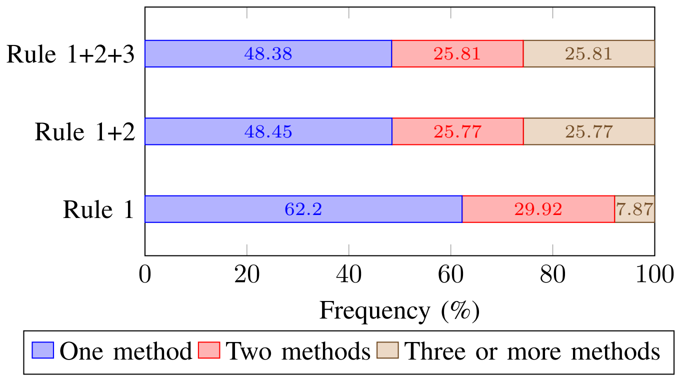


Fig. 6. Percentage of methods of the identified files present in crash report groups

Answer to RQ2: Our study suggests that from 79.53% to 83.51% of the changed methods appear in stack traces of the associated crash report group, depending on the considered set of rules. This finding supports previous research work, highlighting that crash reports are an effective asset for locating the root cause of bugs in both coarse and fine-grained scenarios.

We believe that the cases (between 16.49% and 20.47%) where changed files (see Fig. 5) were identified and no modified methods appeared in the stack traces from crash report groups are related to the fact that the methods that contain crash faults do not always reside in the crash stack traces, as described in [1], [19]. Additionally, the failure may not be located in a method. It may reside, for example, in annotations related to object-relational mapping or to control transient object instances.

V. Threats to validity

Construct validity threats. We may have made measurement errors since we extracted stack traces and changed files to fix bugs by parsing Redmine HTML issues reports. We also obtained data from stack traces to group them using regular expressions, and we map issues to crash report groups using string matching. In order to mitigate such threats we have carefully codified and tested our implementation. Bug fixing commits can also have changes associated to code refactoring. Because they are associated with bug fixing issues, it is expected that a high number of the methods associated with them are really fixing bugs. In our study, 72% of the bug fixing commits have only one method modified, 17.5% have two modified methods, and 10.5% have three or more modified methods, thus showing a high probability that most of the methods are associated with bugs.

Internal Validity Threats. We have extracted information from the stack traces and changed files to fix bugs posted by users, developers, and webhooks[2] in Redmine. These pieces of information may not be complete, and the quality of data could affect fault localization performance.

External Validity Threats. These threats concern the possibility to generalize our results. We examined issues and crash reports for three systems developed by a single company that used the same technology to implement them. These systems might represent the characteristics of a specific group of software such as Java web-based information systems. Additional studies need to be accomplished with other web-based systems in order to generalize our findings.

VI. Related Work

Dhaliwal et al. [21] verified that stack trace analysis of bug reports reported automatically by Firefox can help to identify between 57% and 80% of bugs. They also proposed a new approach to group crash reports based on the similarity of stack traces, creating the Trace Diversity metric, inspired by the Levenshtein distance. They established a threshold to consider that two elements belong to the same subgroup. The method used by the Firefox team groups the crash reports using the top method signature in the failing stack trace. The increment proposed by Dhaliwal et al. [21] uses the similarity of the stack traces to create subgroups, facilitating the location and correction of the error by the developers. The study reports a reduction of 5% in the error correction time.

Wang et al. [12] evolved the work previously published [11], adding two more rules to the three that they had previously proposed. They obtained significant results in the empirical study with data from Firefox and Eclipse projects. They identified fault correlation groups using stack trace information with precision of 91% and recall of 87% for Firefox, and precision of 76% and recall of 61% for Eclipse. Our work uses the first three rules proposed by them to group crash reports using stack traces [11]. The main difference is that they started from the crash types generated by Socorro - the Mozilla's Crash Reporting System [3], and applied the rules to correlate these crash types. We build our groups from the beginning, so we made some adaptations like creating groups using rules 1 and 2. After that, we extract the FCSFs for each of those groups and then regroup using rule 3. We did not measure the performance of our grouping method to compare with the studies of Wang et al. since we found only two occurrences of issues marked as duplicate or related in our dataset. The two other rules proposed by them use occurrence times of crash events and textual similarity among user comments about the crash events to identify correlations among types of failures. We do not discuss either aspect of our work. The algorithm to locate and rank buggy files proposed in their study achieved 42% of precision and 62% of recall for Firefox, along with 50% of precision and 52% of recall for Eclipse, analyzing the Top 3 suggested files. The recall value increased when they analyzed the Top 10 candidates, getting 92% for Firefox and 90% for Eclipse. In our study, we obtained better results for Top 3 using the rule 1, being able to find at least one of the modified files to correct the bug in 77.3% of the cases and with a mean average precision of 55.4%. Analyzing the Top 10 suspicious files, the best value of recall was 77.3%, with MAP of 55.5%, also using the rule 1.

Wu et al. [19] proposed a method called CrashLocator to locate faulty functions by using an approach to expand crash stack information and generate approximate crash traces by discovering defective functions that are not present in the stack trace. Four factors are used to locate faults: function frequency, inverse bucket frequency, inverse average distance to crash point, and function's lines of code. They used Mozilla Foundation's runtime failure data and found 50.6%, 63.7% and 67.5% of failures while examining the Top 1, 5 and 10 functions recommended by the CrashLocator. They also improved Recall@10 metric values ranging from 23.2% to 45.8% when compared to conventional methods that only analyze the stack trace. In our work, we take inspiration from three of the four discriminative factors proposed by them, since no source code analysis was done. The granularity level studied by us (files) was greater than the researched by them (methods). We also did not expand the crash stack, that is, we only grouped crash reports based on the files found in the stack traces. We obtained the best mean recall results of 61.4%, 77.3%, and 77.3%, and MAP of 41.4%, 55.5%, and

[2]https://developer.github.com/webhooks/

55.5%, examining top 1, 5 and 10 suggested files, respectively. We already expect better results, since our approach looked for coarse-grained artifacts (files) while they looked for fine-grained ones (functions/methods). Although our goal was not to find buggy methods, we found that combining our approach to identify defective files with the analysis of the most frequent methods in the stack traces of the group has the potential to indicate the most likely to be faulty.

Even using different metrics and analyzed a smaller volume of issues and crash reports, we believe that our results were consistent with those reported by Wang et al. [12] and Wu et al. [19].

VII. Conclusions and Future Work

In this paper we explored the landscape of using stack traces available in crash reports to locate coarse-grained (classes or files) and fine-grained (functions or class methods) components that are likely to have introduced a failure. To this end, we implemented a crash report grouping approach based on previous research contributions. Our approach aims to indicate buggy components and thus reduce the cognitive effort to identify the root causes of a system crash.

We discussed an assessment of our approach using three non-trivial Java web systems, and our results bring new evidence that the use of crash reports is an effective approach for locating the origin of defects. For instance, using the configuration of our approach with best performance, we were able to correctly locate the file responsible for a crash in almost 78% of the cases. This file was within the top 10 most suspicious files, according to such a configuration. In addition, in more than 70% of cases, the method changed to fix a bug appears in stack traces of crash report groups at least the same number of times than the other methods of the buggy file. We have also learned in our study that our crash report grouping approach can benefit from the discarding of small stack traces or stack traces that do not refer to specific classes of the system.

We have presented the results of this study to the project leaders of the investigated systems, and the company has decided to use our approach in their software development process. Currently, the development teams of the company are using our list of suspicious buggy files and methods to correct bugs related to the stack trace groupings. We will continue to collect feedback of the usefulness, usability, and performance of the approach in order to improve it.

Acknowledgment

This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 425211/2018-5, and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] L. An and F. Khomh. Challenges and issues of mining crash reports. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, pp. 5–8, March 2015.

[2] K. Kovash. Dramatic stability improvements in firefox, 2010.

[3] L. Thomson. Socorro: Mozilla's crash reporting system, 2012.

[4] N. Bettenburg et al. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 308–318, 2008.

[5] A. Schroter et al. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 118–121. IEEE, 2010.

[6] K. Kinshumann et al. Debugging in the (very) large: ten years of implementation and experience. *Communications of the ACM*, 54(7):111–116, 2011.

[7] A. Podgurski et al. Automated support for classifying software failure reports. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 465–475. IEEE, 2003.

[8] F. Khomh et al. An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox. In *2011 18th Working Conference on Reverse Engineering*, pp. 261–270. IEEE, 2011.

[9] D. Kim et al. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering*, 37(3):430–447, 2011.

[10] Y. Dang et al. Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1084–1093. IEEE, 2012.

[11] S. Wang et al. Improving bug localization using correlations in crash reports. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 247–256. IEEE, may 2013.

[12] S. Wang et al. Improving bug management using correlations in crash reports. *Empirical Software Engineering*, 21(2):337–367, apr 2016.

[13] T. Ball et al. From symptom to cause: localizing errors in counterexample traces. In *ACM SIGPLAN Notices*, volume 38, pp. 97–105. ACM, 2003.

[14] J. A. Jones et al. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 467–477. IEEE, 2002.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273–282. ACM, 2005.

[16] S. Nessa et al. Software fault localization using n-gram analysis. In *International Conference on Wireless Algorithms, Systems, and Applications*, pp. 548–559. Springer, 2008.

[17] C.-P. Wong et al. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 181–190. IEEE, 2014.

[18] Y. Gu et al. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software*, 148:88–104, 2019.

[19] R. Wu et al. CrashLocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pp. 204–214, New York, New York, USA, 2014. ACM Press.

[20] R. Wu et al. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*, 23(5):2866–2900, 2018.

[21] T. Dhaliwal et al. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, number November 2009, pp. 333–342. IEEE, sep 2011.

[22] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings. 20th international conference on data engineering*, pp. 79–90. IEEE, 2004.

[23] X. Yan et al. Clospan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM international conference on data mining*, pp. 166–177. SIAM, 2003.

[24] H. Kazato et al. Extracting and visualizing implementation structure of features. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pp. 476–484. IEEE, 2013.

[25] C. D. Manning et al. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[26] A. Ang et al. Revisiting the practical use of automated software fault localization techniques. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 175–182. IEEE, 2017.