Leveraging Stack Traces for Spectrum-based Fault Localization in the Absence of Failing Tests

Lorena Barreto Simedo Pacheco

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

July 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Lorena Barreto Simedo Pacheco**

Entitled: **Leveraging Stack Traces for Spectrum-based Fault Localization in the Absence of Failing Tests**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
Dr. Diego Elias Costa

______________________________ Examiner
Dr. Ching Yee Suen

______________________________ Examiner
Dr. Diego Elias Costa

______________________________ Supervisor
Dr. Tse-Hsun (Peter) Chen, Dr. Jinqiu Yang

Approved by ______________________________
Dr. Joey Paquet, Chair
Department of Computer Science and Software Engineering

__________ 2024 ______________________________
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Leveraging Stack Traces for Spectrum-based Fault Localization in the Absence of Failing Tests

Lorena Barreto Simedo Pacheco

Bug fixing is a crucial task in software maintenance to hold user trust. Although various automated fault localization techniques exist, they often require specific conditions to be effective. For example, Spectrum-Based Fault Localization (SBFL) techniques need at least one failing test to identify bugs, which may not always be available. Bug reports, particularly those with stack traces, provide detailed information on system execution failures and are invaluable for developers. This study focuses on utilizing stack traces from crash reports as fault-triggering tests for SBFL. Our findings indicate that only 3.33% of bugs have fault-triggering tests, limiting traditional SBFL efficiency. However, 98.3% of bugfix intentions align directly with exceptions in stack traces, and 78.3% of buggy methods are reachable within an average of 0.34 method calls, proving stack traces as a reliable source for locating bugs. We introduce a new approach, SBEST, that integrates stack trace data with test coverage to enhance fault localization. Our approach shows a significant improvement, increasing Mean Average Precision (MAP) by 32.22% and Mean Reciprocal Rank (MRR) by 17.43% over traditional stack trace ranking methods.

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Dr. Tse-Hsun (Peter) Chen and Dr. Jinqiu Yang, for their support and guidance throughout my project. Their expertise, valuable insights, and feedback were essential in shaping the direction of my research and enhancing the quality of my work. I also extend my appreciation to the SPEAR Lab, the CREATE SE4AI group and all the colleagues and professors I had the privilege of working with during the course of my Master's.

In addition, I want to express my utmost gratitude to my husband, Alexandre, who has been my biggest supporter throughout this journey. Without his love, hard work, and unwavering dedication to supporting my dream, none of this would have been possible. This achievement is as much his as it is mine.

Finally, I am beyond grateful to my parents, my sister, and my godmother, Helenice, for their unconditional love. They inspire and motivate me to be strong every day. I would not be where I am today without their sacrifices.

Contribution of Authors

The work presented in this thesis was submitted to the journal Empirical Software Engineering and is currently under revision.

In addition to that, a short paper titled "DVC in Open Source ML-development: The Action and the Reaction"[1] was published in CAIN 2024. It is not presented in this thesis because it has no direct correlation with the discussed topic.

[1]https://dl.acm.org/doi/10.1145/3644815.3644965

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 Motivation . . . 1

1.2 Overview of the Methodology . . . 2

1.3 Research Questions . . . 3

1.3.1 RQ1:Are the test failures related to the bug in crash reports? . . . 3

1.3.2 RQ2: What is the relationship between stack traces and buggy location? . . 3

1.3.3 RQ3: Can we utilize the stack traces to help detect the buggy locations? . . 4

1.4 Contributions . . . 4

1.5 Thesis Organization. . . . 4

2 Background and Motivation 6

3 Related Work 9

3.1 Fault Localization . . . 9

3.2 Bug Reports and Stack Traces . . . 10

4 Data Collection and Case Study Setup 12

4.1 Data Collection . . . 12

4.1.1 Collecting Bug Reports with Stack Traces . . . 12

4.1.2 Collecting Test Coverage 13
4.2 Evaluation Metrics 15

5 Results **17**
5.1 RQ1 - Are the test failures related to the bug in crash reports? 17
5.1.1 Motivation 17
5.1.2 Approach 17
5.1.3 Results 19
5.2 RQ2 - What is the relationship between stack traces and buggy location? 22
5.2.1 Motivation 22
5.2.2 Approach 23
5.2.3 Results 23
5.3 RQ3 - Can we utilize the stack traces to help detect the buggy locations? 28
5.3.1 Motivation 28
5.3.2 Approach 29
5.3.3 Results 31

6 Discussion **34**
6.1 On the use of stack traces in the absence of failing tests. 34
6.2 Leveraging stack trace rankings for enhanced fault localization. 34
6.3 Encouraging the integration of stack traces to complement execution information. . 35

7 Threats to Validity **36**
7.1 External Validity. 36
7.2 Internal validity. 37
7.3 Construct validity. 37

8 Future Work **38**

9 Conclusion **41**

Bibliography **43**

List of Figures

Figure 2.1 An example of a bug report (COMPRESS-181). The bug report addresses a problem in reading TAR files that contain symlinks, which cause an IOException to be thrown. The report contains the stack trace of the error (gray box), and an example of TAR file (simple-aix-native-tar.tar) that triggers the bug is attached. When this bug report was created, there were no failing tests 7

Figure 4.1 An example of the structure of stack trace entries in Java. 13

Figure 5.1 Extract from the bugfix patch for Cli-5, classified in the Exception Prevention category. 24

Figure 5.2 Extract from the bugfix patch for Closure-152, classified in the Exception Conversion category. 25

Figure 5.3 Extract from the bugfix patch for COMPRESS-12, classified in the Exception Wrapping category. 26

Figure 5.4 Call Graph for the bug Math-79, which has a 3-call distance between the stack trace and the buggy method. 27

Figure 5.5 Bugfix intention type distribution in the studied bugs. 27

List of Tables

Table 4.1 An overview of our studied systems from Defects4J v2.0.0. *#Total Bugs*, *#Bugs with Stack Traces*, *#Studied Bugs*, *LOC*, and *#Tests* show the total number of bugs in Defects4j, the total number of bugs with stack traces, the number of bugs studied, lines of code, and tests in each system, respectively. 14

Table 5.1 Description of the terms in the Ochiai formula. *Covered* indicates if the component (in our case, the method) was executed or not during the testing (i.e., is covered or not by the test). *Test* indicates if the test case failed or passed during its execution. 18

Table 5.2 Test execution results for the crash report bugs. *#Bugs*, *#Tests* and *#Bugs with Fault-triggering Tests* show the total number of studied bugs, the total amount of tests and the total number of bugs with fault-triggering tests; while *Failing Tests (Avg.)*, and *Fault-triggering Tests (Avg.)* represent the average number of failing tests and fault-triggering tests, respectively, calculated among the bugs. 20

Table 5.3 Fault-triggering tests results for the bugs without stack traces. *#Bugs*, *#Bugs with Fault-triggering Tests* and *#Bugs without fault-triggering tests* show the total number of studied bugs, the total number of bugs with fault-triggering tests and the total number of bugs without fault-triggering tests, respectively. 20

Table 5.4 Ochiai results for all the studied systems.*#Bugs* represents the total number of studied bugs, while *#Bugs with FT* is the number of bugs in which there is at least one Failing Test. 22

Table 5.5 Fault localization results for all the studied projects. *#Bugs* represents the total number of studied bugs in each project. 32
Table 8.1 LLM multi-agent implementation results compared to SBEST 40

Chapter 1

Introduction

1.1 Motivation

Locating and fixing software bugs is an essential yet frequently expensive and time-consuming task. Bug-fixing activities not only delay the project's progress but also increase the software's overall cost, making them a critical part of software maintenance. Considering the complexity and rapid evolution of modern software systems, the ability to efficiently and accurately locate bugs is more important than ever.

To address this challenge, previous research has proposed numerous different automated fault localization techniques (Abreu, Zoeteweij, & van Gemund, 2007; Abreu, Zoeteweij, & Van Gemund, 2009; Cui, Jia, Chen, Zheng, & Liu, 2020; Hong et al., 2017; Jones & Harrold, 2005; Naish, Lee, & Ramamohanarao, 2011; Papadakis & Le Traon, 2012, 2015; Wang et al., 2022; W. E. Wong, Debroy, Gao, & Li, 2013), among which Spectrum-Based Fault Localization (SBFL) approaches are prominent due to its high accuracy (de Souza, Chaim, & Kon, 2016). SBFL techniques leverage information derived from test executions — specifically, the spectrum of code elements executed during passing and failing tests — to calculate the probability of every code element being faulty (i.e. suspiciousness score).

The effectiveness of SBFL highly depends on the availability of failing test cases that uncover the bug (i.e., fault-triggering tests). However, studies indicate that in real-world scenarios, especially in continuous deployment and integration environments, failing tests may not always be present or identified when bugs are reported (Chen, Rafi, Wang, et al., 2023; Kabadi et al., 2023). This absence of fault-triggering tests significantly impacts the applicability of SBFL, as it depends on it to distinguish between faulty and non-faulty code.

In the absence of failing tests, alternative approaches must be considered. Exception stack traces, commonly included in crash reports, provide dynamic execution information that can be extremely valuable for debugging. Studies show that developers usually rely on them when investigating bugs (Schroter, Schröter, Bettenburg, & Premraj, 2010; Zimmermann et al., 2010). In addition, previous research (Chen, Chen, & Wang, 2021a) shows that stack traces provide valuable information on the cause of the bugs and their location in the source code. These stack traces, which outline the sequence of method calls leading up to an exception, offer implicit clues about where a bug may reside, providing a fallback localization method in environments where traditional test-based approaches falter.

1.2 Overview of the Methodology

In this thesis, we address the challenges of fault localization in scenarios where traditional fault-triggering test data is scarce. We conduct our study on the Defects4J 2.0 dataset (Just, Jalali, & Ernst, 2014), a widely recognized resource for research in fault localization and program repair. However, prior studies (Chen et al., 2023; Kabadi et al., 2023) have reported that some tests in Defects4J were added post hoc, following the reporting of bugs, which could bias the bug localization process. To avoid this issue and better simulate the state of the codebase at the time developers first address a bug, we utilize the commit data immediately preceding the bug report rather than relying directly on the buggy commits provided by Defects4J. Our initial analysis of the bugs reveals a critical limitation: only 3.33% of crash reports, which are bug reports that contain stack traces, possess fault-triggering tests. Even for bug reports without stack traces, only 10.19% of the bugs have fault-triggering tests. This low incidence of failing tests challenges the efficacy of Spectrum-Based Fault Localization (SBFL) methods, which rely heavily on test outcomes to identify bug locations. To overcome this limitation, we propose leveraging the stack traces—rich in contextual and execution information about the bug's root causes—as a substitute for the failing tests. We introduce a novel approach, Spectrum-Based Localization Enhanced by Stack Traces (SBEST), which integrates stack trace analysis within the SBFL framework. Our method significantly improves the accuracy of fault localization by utilizing the comprehensive code execution information allied with the exception stack traces. This integration enables a more detailed understanding of system behaviour at the moment of failure, thereby addressing the gaps left by the unavailability of failing tests.

1.3 Research Questions

In particular, we study and answer three research questions (RQs).

1.3.1 RQ1:Are the test failures related to the bug in crash reports?

We study the presence of fault-triggering tests in crash reports (i.e., bug reports that contain stack traces) and their impact on the SBFL results. We find that only 3.33% of the crash report bugs contain fault-triggering tests, which profoundly degrades the SBFL efficiency in locating these bugs.

The limited execution data underscores the importance of incorporating stack traces into the SBFL process, enhancing the potential to accurately identify bug locations in the absence of traditional test failures.

1.3.2 RQ2: What is the relationship between stack traces and buggy location?

Given that the stack traces are the only execution information available in most of the studied bug reports, we aim to study how they relate to the buggy location and how they can be used to locate these bugs. To do so, we inspect the type of modification performed in the bugfix commit and the distance between the stack trace and the buggy methods. We find that in 98.3% of the studied bugs, the bugfix intention is directly correlated with the exception in the stack trace (e.g., adding code to handle the exception). In addition, 78.3% of the buggy methods are directly reachable from the stack traces, with an average distance of 0.34 method calls. This indicates that leveraging the stack traces to reconstruct the execution path at the time of the exception can be an effective method for pinpointing the bugs.

1.3.3 RQ3: Can we utilize the stack traces to help detect the buggy locations?

Given the previous findings, we propose a new approach, SBEST, that integrates the stack trace information into Spectrum-Based Fault Localization. We compare our approach with two baselines: the Ochiai results and the position of the methods in the stack trace. Ochiai proves to be the least effective technique by a significant margin. On the other hand, we find that the stack traces alone can locate 34 out of the 60 bugs in the Top-5. This reinforces that the stack traces are a vital source of information about the bug location and should be prioritized in FL. In addition, SBEST achieves an improvement of 32.22% on MAP and 17.43% on MRR compared to the Stack Trace ranking. The enhancement of metrics suggests that the incorporation of coverage information effectively improved the ranking of stack traces, thereby providing the tool with a more comprehensive view of system execution at the time of failure.

1.4 Contributions

We summarize the contributions of this thesis as follows:

- Contribution 1. We present a comprehensive study that evaluates the effectiveness of combining stack traces with test coverage data.

- Contribution 2. We make available (*Leveraging Stack Traces for Spectrum-based Fault Localization in the Absence of Failing Tests*, 2024) the complete information (including stack traces, test results and detailed code coverage) on a set of 60 crash reports, which took us more than 100 hours to obtain.

- Contribution 3. We present a novel approach, SBEST, that incorporate stack traces into the SBFL, and significantly improve bug localization, thereby addressing one of the key challenges in automated debugging processes.

1.5 Thesis Organization.

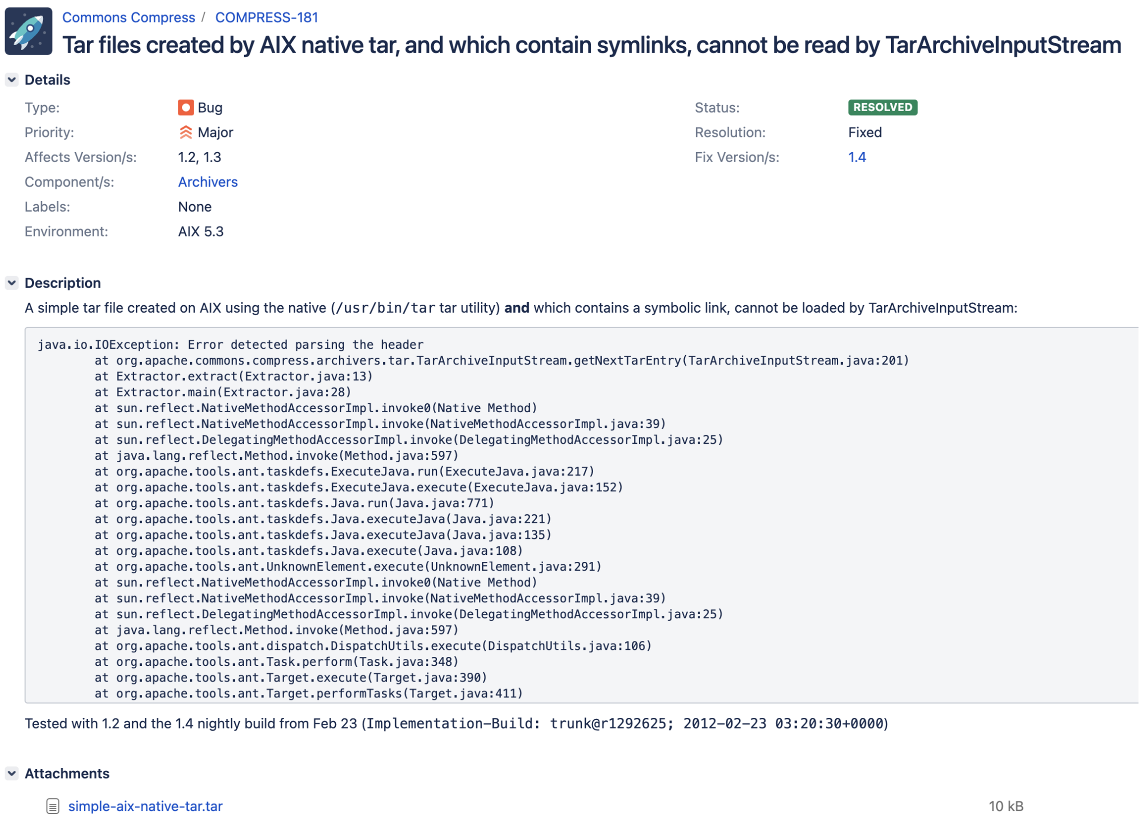
Chapter 2 details the background and the motivation of this thesis. Chapter 3 surveys the related work. Chapter 4 explains the experimental setup. Chapter 5 presents the results Chapter 6 discusses the implications of our findings. Chapter 7 presents the threats to validity. Chapter 8 discusses potential future work. Chapter 9 concludes the thesis.

Chapter 2

Background and Motivation

Bug Reports contain important information for developers to fix the issues that users or other developers encounter in the software (Chen et al., 2021a). A bug report typically includes a title, a description, attached files, and comments. For example, Figure 2.1 shows a bug report from the Defects4j 2.0 dataset. The title offers a concise one-line description of the problem. The description provides developers with the necessary details to investigate the bug, such as the stack traces illustrating the system execution details. When additional information is required, users and developers use the comments and attached files sections to further communicate on the bug. In Figure 2.1 bug COMPRESS-181, a problem occurs when the system reads TAR files with symlinks. In particular, the description provides developers with a stack trace that illustrates an IOException is being thrown. The description also provides the affected nightly builds, which show no test failures.

To assist developers in debugging such bugs, prior studies have proposed Spectrum-Based Fault Localization (SBFL) techniques (Baudry, Fleurey, & Le Traon, 2006; Chen, Chen, & Chen, 2022; Lou et al., 2021). SBFL relies on the information from failing tests to identify potentially faulty code. Hence, SBFL assumes that ***test failures that are relevant to the bug exist***. SBFL works by comparing the code execution of passing and failing tests to identify their differences. The potential problematic code is then identified based on these differences. SBFL is widely studied in prior studies for its simplicity and efficiency (Pearson et al., 2016; Santelices, Jones, Yu, & Harrold, 2009). However, when there is no failing test, SBFL can not distinguish buggy code elements. For instance, in the case of bug COMPRESS-181 (Figure 2.1), in which there are no failing tests, SBFL



Commons Compress / COMPRESS-181

Tar files created by AIX native tar, and which contain symlinks, cannot be read by TarArchiveInputStream

Details

Type:	Bug	Status:	RESOLVED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	1.2, 1.3	Fix Version/s:	1.4
Component/s:	Archivers		
Labels:	None		
Environment:	AIX 5.3		

Description

A simple tar file created on AIX using the native (`/usr/bin/tar` tar utility) **and** which contains a symbolic link, cannot be loaded by TarArchiveInputStream:

```
java.io.IOException: Error detected parsing the header
        at org.apache.commons.compress.archivers.tar.TarArchiveInputStream.getNextTarEntry(TarArchiveInputStream.java:201)
        at Extractor.extract(Extractor.java:13)
        at Extractor.main(Extractor.java:28)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.apache.tools.ant.taskdefs.ExecuteJava.run(ExecuteJava.java:217)
        at org.apache.tools.ant.taskdefs.ExecuteJava.execute(ExecuteJava.java:152)
        at org.apache.tools.ant.taskdefs.Java.run(Java.java:771)
        at org.apache.tools.ant.taskdefs.Java.executeJava(Java.java:221)
        at org.apache.tools.ant.taskdefs.Java.executeJava(Java.java:135)
        at org.apache.tools.ant.taskdefs.Java.execute(Java.java:108)
        at org.apache.tools.ant.UnknownElement.execute(UnknownElement.java:291)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.apache.tools.ant.dispatch.DispatchUtils.execute(DispatchUtils.java:106)
        at org.apache.tools.ant.Task.perform(Task.java:348)
        at org.apache.tools.ant.Target.execute(Target.java:390)
        at org.apache.tools.ant.Target.performTasks(Target.java:411)
```

Tested with 1.2 and the 1.4 nightly build from Feb 23 (`Implementation-Build: trunk@r1292625; 2012-02-23 03:20:30+0000`)

Attachments

simple-aix-native-tar.tar 10 kB

Figure 2.1: An example of a bug report (COMPRESS-181). The bug report addresses a problem in reading TAR files that contain symlinks, which cause an IOException to be thrown. The report contains the stack trace of the error (gray box), and an example of TAR file (simple-aix-native-tar.tar) that triggers the bug is attached. When this bug report was created, there were no failing tests

can not help prioritize the faulty locations. This is because, in the absence of failing tests, there is no coverage spectrum between passing and failing executions. Thus, it becomes impossible to prioritize any code element for investigation, as SBFL relies on failing executions as a guide to identify where faults might be present. In this case, all the code locations receive the exact same suspiciousness score of zero, and the bug cannot be located.

In this thesis, we investigate in detail the characteristics of Defects4j 2.0 crash reports (i.e., bug reports that contain stack traces). We dissect information such as the test failures, coverage, and stack traces by conducting a quantitative and qualitative analysis of the data and propose a new approach to address the limitations found in SBFL in such scenarios.

Chapter 3

Related Work

3.1 Fault Localization

Automated fault localization methods are a recurrent theme in the literature, being Spectrum-Based Fault Localization (SBFL) and Mutation Based Fault Localization (MBFL) two of the most famous representatives. SBFL is a very consolidated class of Fault Localization techniques. These approaches use the test information to localize the bugs, and they require at least one failing test to work. They work by analyzing which parts of the code are more frequently executed by failing tests compared to passing tests. Ochiai (Abreu et al., 2007) is one of the most famous SBFL techniques and is presented in more details in Section 5.1.2 Other representatives of this class are Tarantula (Jones & Harrold, 2005), Op2 (Naish et al., 2011), BARINEL (Abreu et al., 2009) and DStar (W. E. Wong et al., 2013): similar approaches based on different formulas to compute the suspiciousness score. Given that SBFL techniques are light weighted and fast, they are also applied to other kinds of techniques such as interactive fault-localization (Gong, Lo, Jiang, & Zhang, 2012) and program repair (Le Goues, Dewey-Vogt, Forrest, & Weimer, 2012; Ye, Martinez, & Monperrus, 2022).

MBFL techniques (Hong et al., 2017; Moon, Kim, Kim, & Yoo, 2014; Papadakis & Le Traon, 2012, 2015; Wang et al., 2022) are based on mutation analysis. For each part of the code covered by failed test cases, a set of mutants is created, and the test suite is executed. Based on the test results, the suspiciousness scores are calculated, and the most suspicious locations are ranked. Despite presenting promising results, this kind of technique is very costly and time-consuming since it requires multiple executions of the test suite. Strategies for reducing this overhead and improving the general accuracy have been studied, such as ways prioritizing the mutants (Liu, Li, Zhao, & Gong, 2018), prioritizing the tests executed for each mutant (de Oliveira, Camilo-Junior, de Andrade Freitas, & Vincenzi, 2018) and a mix of spectrum and mutant-based approaches (Cui et al., 2020). One common point between SBFL and MBFL approaches is that both types of techniques require failing tests to work. However, this might not always be the case in real life.

Most of the mentioned studies use Defects4j to evaluate the results. Defects4j is one of the most famous benchmarks for test-based fault localization research. Nonetheless, recent studies (Chen et al., 2023; Kabadi et al., 2023) show that often in Defects4j, tests that were created after the bug-report creation are added to the buggy patch and thus contain developer knowledge about the bug investigation process. These tests were artificially appended to the buggy commit in Defects4j to simulate the scenario of a bug with test failures. This fact indicates that many fault localization techniques that were evaluated in Defects4j might have very different results when applied to real-world scenarios.

3.2 Bug Reports and Stack Traces

The bug report quality is a common topic in previous research. Knowing which parts of the report are more important in the bug-fixing process is very important to guide new studies on fault localization. Our focus on the stack trace information is supported by some previous results.

In the work by (Bettenburg et al., 2008), the authors conduct a survey to understand what makes a good bug report from the point of view of the developers. They found that information about the steps to reproduce the bug, stack traces, and test cases are considered very helpful. In the work by (Schroter et al., 2010), the authors analyze the intersection between the methods in the stack trace and the methods modified in the bugfix commit. They found out that more than 47% of the stack traces extracted from bug reports contained at least one buggy method.

An empirical study from (Chen et al., 2021a) analyzes the logs' (stack traces and log snippets) importance in crash reports. By extensively analyzing 1,561 bug reports with logs, they concluded that logs often provide helpful information about where the bug is located. They found out that in 73% of the bug reports, there is an overlap between the classes in the logs and the fixed classes.

Chapter 4

Data Collection and Case Study Setup

In this chapter, we start by describing the data collection process and presenting our dataset. After this, we detail the evaluation metrics.

4.1 Data Collection

4.1.1 Collecting Bug Reports with Stack Traces

In this thesis, we study crash reports (i.e., bug reports that contain stack traces) in 15 projects from Defects4j version 2.0.0. Defects4j (Just, Jalali, & Ernst, 2014) is a benchmark and framework of real bugs used in many previous Software Engineering studies related to bug repair and fault localization (Chen, Chen, & Wang, 2021b; Li, Wang, & Nguyen, 2021; Lutellier et al., 2020; Ye et al., 2022). The bugs in Defects4j comprise a wide variety of systems with different characteristics, with sizes ranging from 4K to 90K lines of code (LOC) and the number of tests varying from 54 to 7,911. Although Defects4J provides some basic information about the bugs (e.g., bug report ID and the commit hash of the fixes), it does not contain the stack traces provided in the bug reports. Hence, we start by extracting the bug report URL for all the bugs in the Defects4J repository. To collect the bug reports' textual information (title, description, and comments) and their creation date, we implement a crawler to retrieve information from the project management tool of each system (Jira, GitHub Issues, or Source Forge).

In total, we collected 803 bugs for which the corresponding bug reports were available. For

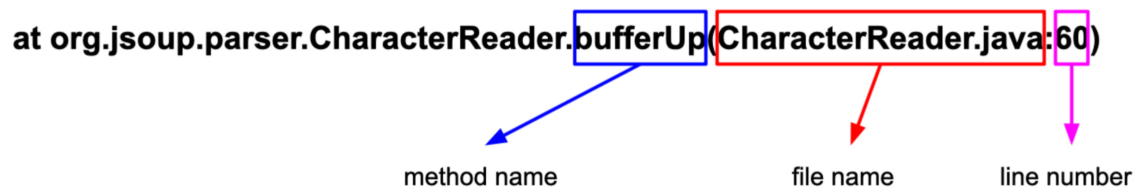


Figure 4.1: An example of the structure of stack trace entries in Java.

each bug, we then combine the bug report information with the information available in Defects4j, including the bugfix commit hash and the list of fault-triggering tests. ***Fault-triggering tests are defined as tests that fail in the buggy commit and pass in the bugfix commit***. They are tests that cover the buggy code and are essential for pinpointing the buggy location when using SBFL approaches. As shown in Figure 4.1, stack traces have a pre-defined pattern. Therefore, we implement a regex-based parser to identify stack traces in bug reports. Our parser identifies the stack traces and extracts the file name, method name, and line number from each stack trace entry. Among all the 803 bugs we collected from Defects4J, we identified 89 bugs that have stack traces in the bug report.

4.1.2 Collecting Test Coverage

To obtain the test information for fault localization, we need to collect the test execution results, detailed test coverage, and the bug-fix patch. However, there are some limitations in Defects4j. Recent studies (Chen et al., 2023; Kabadi et al., 2023) found that many bugs in Defects4J v1.0 contain tests from the "future" (i.e., added by developers after the bugs were fixed) in their designated buggy commits. This is problematic because the test coverage may contain developers' knowledge of the bug, which can cause noise and bias in the result of the downstream research. Because of this, we do not directly utilize Defects4J's data in our study. Instead, since we have the creation date of each bug report, we extract the commit right before the bug report creation from the project's repository. This commit represents the code that was available to developers when they started to address the bug report. We refer to this commit as a ***bug report commit*** to differentiate it from the buggy commit provided by Defects4J.

Table 4.1: An overview of our studied systems from Defects4J v2.0.0. *#Total Bugs*, *#Bugs with Stack Traces*, *#Studied Bugs*, *LOC*, and *#Tests* show the total number of bugs in Defects4j, the total number of bugs with stack traces, the number of bugs studied, lines of code, and tests in each system, respectively.

System	**#Total Bugs**	**#Bugs with Stack Traces**	**#Studied Bugs**	**LOC**	**#Tests**
Cli	39	3	2	4K	94
Closure	174	8	8	90K	7,911
Codec	18	1	1	7K	206
Collections	4	1	1	65K	1,286
Compress	47	11	11	9K	73
Csv	16	1	2	2K	54
Gson	18	3	3	14K	720
JacksonCore	26	4	4	22K	206
JacksonDatabind	112	27	10	4K	1,098
Jsoup	93	8	8	8K	139
JxPath	22	1	1	25K	308
Lang	64	4	4	22K	2,291
Math	106	6	3	85K	4,378
Mockito	38	8	2	11K	1,379
Time	26	2	2	28K	4,041
Total	803	89	60	380K	24,302

Due to the above-mentioned issues in Defects4J, our next step is to collect the test results, and detailed test coverage for the bug report commits. To do that, we utilized GZoltar (Campos et al., 2012). GZoltar is a debugging, fault localization, and coverage extraction tool for Java applications that was often applied to analyze Defects4J bugs (Küçük, Henderson, & Podgurski, 2019; Pearson et al., 2017; Silva et al., 2021; Zhang et al., 2021). For each crash report, we check out the bug report commit, compile the system, and execute the tests. We utilize the GZoltar CLI tool to implement a script to extract the test coverage and test execution results for the bugs from our study. Since there are many ancient commits, we encountered many challenges in compiling the system and running the tests. For example, since some commits in a project can use different versions of JVM, we had to implement a tool to automatically switch JVM versions when analyzing the commits. There are also many dependency-related issues, where a project uses different versions of a library across commits, but upgrading/downgrading the library version can cause dependency conflicts that lead to compilation errors. In total, we spent over 100 hours compiling and collecting the data. Despite our best efforts, we excluded some bugs where we were not able to execute and collect the test coverage information.

Table 4.1 provides detailed information on the bugs studied in this work, i.e., bugs from Defect4J 2.0 with valid bug reports. In total, we include 803 bugs from 15 open-source projects. In this work, we target user-reported bugs with stack traces reported, i.e., crash-reporting bugs. Among the 15 projects, there are 89 crash-reporting bugs (i.e. bugs with stack traces in their bug reports), which represents 11.08% of the total set. Of these 89 bugs, we were able to compile and collect the test execution information for 63 bugs. Finally, we excluded three bugs from our study whose root cause is not in a method, as this work focuses on method-level bug localization. The absence of buggy methods can happen in two different situations: (1) The bugfix involves altering lines outside the methods (for example, changing the value of a global variable); or (2) The bugfix includes the creation of new methods, but no existing method is updated. In total, we study 60 bugs. Despite the low percentage of crash reports found in Defects4j, they match with the percentage of bugs with stack traces in other projects (Chen et al., 2021a).

4.2 Evaluation Metrics

In this thesis, we conduct a series of experiments on locating bugs using stack traces. In order to evaluate and compare the results, we utilize a set of well-consolidated metrics in the fault localization field. Previous studies suggest that performing fault-localization at the file level lacks precision (Kochhar, Xia, Lo, & Li, 2016) while opting for the statement-level granularity can cause the approach to miss important code context (Parnin & Orso, 2011). Hence, in this thesis, we conduct our analysis at the *method level*.

Mean Average Precision (MAP). MAP is a metric that considers the rank of all the buggy methods in a given project. It is calculated by taking the mean of the Average Precision (AP) across all these bugs, where the following formula defines the AP, and m is the set of buggy files from a given bug report.

$$AP = \frac{\sum_{i=1}^{m} i/Pos(i)}{m}$$

Mean Reciprocal Rank (MRR). MRR considers the position where the first buggy file was ranked and takes the mean from all the bug reports. It is computed by the following formula, where K is the set of bug reports and $rank_i$ is the position of the first buggy method in the rank.

$$MRR = \frac{1}{K}\sum_{i=1}^{K}\frac{1}{rank_i}$$

Top K. The Top K metric represents the number of bugs in which at least one buggy method is located between the first K best-ranked methods in the approach. According to a previous study (Parnin & Orso, 2011), developers only check a limited amount of suspicious locations. Due to this, we use 1, 3, and 5 as the values for K.

Chapter 5

Results

In this chapter, we discuss the results of our research questions (RQs). For each RQ, we present the motivation, our approach and the results.

5.1 RQ1 - Are the test failures related to the bug in crash reports?

5.1.1 Motivation

Prior research (Abreu et al., 2007; Jones & Harrold, 2005; Naish et al., 2011) uses SBFL to assist in debugging, which relies on failing test coverage. The assumption behind SBFL is that the system has at least one failing test covering the bug (i.e. fault-triggering failing test). Intuitively, if fault-triggering tests fail on the bug report commits (the nearest commit when a bug report was reported), we can leverage SBFL techniques to pinpoint the buggy locations. However, it is not clear if fault-triggering tests are available and failing when a bug report is created (Haben, Habchi, Papadakis, Cordy, & Traon, 2023; Just, Jalali, Inozemtseva, et al., 2014). Therefore, in this RQ, we study whether there are fault-triggering tests (and if these tests fail) at the moment of the bug report creation and their impact on the SBFL's efficiency.

5.1.2 Approach

To understand whether the test failures are related to the bugs in crash reports, we analyze the number of crash reports that contain failing fault-triggering tests. As discussed in Section 4.1.2, for

Table 5.1: Description of the terms in the Ochiai formula. *Covered* indicates if the component (in our case, the method) was executed or not during the testing (i.e., is covered or not by the test). *Test* indicates if the test case failed or passed during its execution.

n	**Covered**	**Test**
n_{00}	no	passed
n_{10}	yes	passed
n_{01}	no	failed
n_{11}	yes	failed

each bug report, we perform our study on the bug report commits to avoid biases of "future" tests. In particular, we execute the fault-triggering tests provided by Defects4J and collect the test results with GZoltar. We also investigate whether fault-triggering tests are related to other types of bug reports (without stack traces) and compare the test results with crash reports. A prior study (Chen et al., 2023) identified a set of bugs without stack traces from Defects4J. Therefore, we used their dataset and identified a total of 157 bugs without stack tracks. This data will allow us to draw a comparison between crash report bugs and bugs without stack traces.

To study the impact of missing failing fault-triggering tests on SBFL techniques, we select Ochiai (Abreu et al., 2007) as the baseline since it is widely used and has been shown to perform well on real faults (Abreu, Zoeteweij, & Van Gemund, 2006; Le, Thung, & Lo, 2013; Pearson et al., 2017). We use the detailed coverage and the test results obtained via the Gzoltar execution to apply the Ochiai formula at the method level. The formula calculates the suspiciousness of each code statement, which allows us to rank the methods based on how probable they contain the bug. The SBFL suspiciousness formulas vary for each technique, but all are based on the idea that the code that is more covered by failing tests and less covered by passing tests is more likely to be buggy. The Ochiai formula assigns a suspiciousness score between 0 (not suspect) and 1 (highly suspect) for each piece of code element (e.g., class, method, or statement). If applied at a method level, we calculate the suspicious score of a given method *j* as:

$$s_o = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) * (n_{11}(j) + n_{10}(j))}} \tag{1}$$

in which the terms are as defined in Table 5.1. Each term *n* in the formula (e.g., n_{00}) corresponds to the number of tests that obey specific criteria for the method *j* for which the suspiciousness score is being calculated. The first criterion determines the method coverage, whether the method is covered by a test (n_{1x}) or not (n_{0x}). The second criterion indicates the execution status, passing (n_{x0}) or failing (n_{x1}). So, if a test covers the method *j* and its execution fails, for instance, this test will be computed under n_{11}. As a calculation example, suppose that we have the following test case results for a given method M1:

- 6 failing test cases that cover M1 (n_{11})
- 2 failing test cases that do not cover M1 (n_{10})
- 10 passing test cases that cover M1 (n_{01})

By applying Equation 1, we get that the Ochiai score for method M1 is approximately 0.530.

5.1.3 Results

Only 3.33% of the crash report bugs contain fault-triggering tests (failing test cases that uncover the bug). Table 5.2 shows the test execution and fault-triggering test results for the crash report bugs. When comparing with the bugs without stack traces (Table 5.3), we obtain the following numbers:

- **Crash Reports:** 2 out of 60 bugs have fault-triggering tests **(3.33%)**
- **Bugs without Stack Traces:** 16 out of 157 bugs have fault-triggering tests **(10.19%)**

Based on the findings, we observed that the majority of the crash report bugs (96.67%) either lack fault-triggering tests or do not trigger them (i.e., fault-triggering tests did not fail). In addition, we can see that the percentage of fault-triggering tests in crash report bugs is significantly less when compared to the set of bugs without stack traces (3.33% versus 10.19%). The reason may be that such exception-related issues are more often triggered during production and not during testing. Another reason may be that developers are more likely to handle exceptions for debugging purposes rather than fault prevention (Shah, Görg, & Harrold, 2008). This shift in the use of exception handling as debugging aids increases the likelihood of exception-related issues occurring in production. In such cases, applying traditional SBFL techniques would not be ideal, given the incapacity of the

Table 5.2: Test execution results for the crash report bugs. *#Bugs*, *#Tests* and *#Bugs with Fault-triggering Tests* show the total number of studied bugs, the total amount of tests and the total number of bugs with fault-triggering tests; while *Failing Tests (Avg.)*, and *Fault-triggering Tests (Avg.)* represent the average number of failing tests and fault-triggering tests, respectively, calculated among the bugs.

System (#Bugs)	#Tests	Failing Tests (Avg.)	Fault-triggering Tests (Avg.)	#Bugs with Fault-Triggering Tests
Cli (2)	94	2	0	0
Closure (8)	7,911	2.125	0	0
Codec (1)	206	0	0	0
Compress (11)	73	1	0	0
Csv (1)	54	0	0	0
Gson (3)	720	0	0	0
JacksonCore (4)	206	0.5	0.25	1
JacksonDatabind (10)	1,098	26.9	0.1	1
Jsoup (8)	139	0.125	0	0
JxPath (1)	308	0	0	0
Lang (4)	2,291	8.5	0	0
Math (3)	4,378	0.667	0	0
Mockito (2)	1,379	21.5	0	0
Time (2)	4,041	20.5	0	0
Total (60)	24,302	5.987	0.025	2

Table 5.3: Fault-triggering tests results for the bugs without stack traces. *#Bugs*, *#Bugs with Fault-triggering Tests* and *#Bugs without fault-triggering tests* show the total number of studied bugs, the total number of bugs with fault-triggering tests and the total number of bugs without fault-triggering tests, respectively.

System	#Bugs	#Bugs with fault-triggering tests	#Bugs without fault-triggering tests
Cli	11	4	7
Closure	47	2	45
Codec	4	0	4
Compress	5	1	4
Csv	1	1	0
Gson	1	1	0
JacksonCore	2	0	2
JacksonDatabind	2	2	0
Lang	26	4	22
Math	35	0	35
Mockito	20	1	19
Time	3	0	3
Total	157	16	141

tool to differentiate buggy statements when there are no test failures covering them. Due to the absence of failing fault-triggering tests, the runtime information included in the bug report (i.e., stack traces) serves as the final resource for assistance. In short, studying how to leverage stack traces for fault localization is an important supplement to traditional SBFL techniques that use failing fault-triggering tests.

> SBFL techniques may not be effective in addressing crash reports due to the limited presence of fault-triggering failing tests (3.33% of all bug reports).

To study how the absence of fault-triggering tests impacts SBFL, we apply Ochiai to our studied bugs with stack traces. To evaluate the approach, we utilize the metrics presented in Section 4.2, specifically Top-1, Top-3, Top-5, MAP and MRR. We exclude from the Ochiai results the bugs without failing tests, given that the presence of failing tests is necessary for the computation of the suspicioness score. Out of 60 bugs, 23 did not have any failing tests. Note that even if there is a failing test, the test may not be related to the bug that we are interested in (i.e., it is not a fault-triggering test).

On the bug report commit, SBFL performed poorly in all the projects, only being able to locate 2/60 bugs among the Top-5 methods. Table 5.4 shows the SBFL results applied to the bug report commit for the crash report bugs that contain failing tests. Ochiai could not locate any bug in Top-1 and located only 1 in Top-3 and 2 in Top-5. The highest MAP and MRR values across projects are 0.0693 and 0.0836, respectively, both in the JacksonCore project. Considering that the values of MAP and MRR range between 0 and 1, the obtained values are extremely low. In contrast, a prior study by (Chen et al., 2022) that evaluates Ochiai on the original Defects4J benchmark achieved an average MAP of 0.30 and an average MRR of 0.42, which is significantly higher than our results obtained when there is no failing fault-triggering tests.

The MAP and MRR values assess the effectiveness of fault localization techniques in returning relevant results in the top ranking. These metrics are important indicators that demonstrate the usefulness of fault localization techniques. According to a prior survey by (Kochhar et al., 2016), 80% of developers consider a fault localization technique successful if it can localize bugs in the

Table 5.4: Ochiai results for all the studied systems.*#Bugs* represents the total number of studied bugs, while *#Bugs with FT* is the number of bugs in which there is at least one Failing Test.

System (#Bugs | #Bugs with FT)	Top-1	Top-3	Top-5	MAP	MRR
Cli (2 | 2)	0	0	0	0.0045	0.0045
Closure (8 | 8)	0	0	0	0.0007	0.0012
Codec (1 | 0)	-	-	-	-	-
Collections (0 | 0)	-	-	-	-	-
Compress (11 | 6)	0	0	1	0.0079	0.023
Csv (1 | 0)	-	-	-	-	-
Gson (3 | 0)	-	-	-	-	-
JacksonCore (4 | 2)	0	1	1	0.0693	0.0836
JacksonDatabind (10 | 8)	0	0	0	0.0005	0.0005
Jsoup (8 | 1)	0	0	0	0	0
JxPath (1 | 0)	-	-	-	-	-
Lang (4 | 4)	0	0	0	0.0008	0.0008
Math (3 | 2)	0	0	0	0.0002	0.0001
Mockito (2 | 2)	0	0	0	0.0060	0.0281
Time (2 | 2)	0	0	0	0.0003	0.0003

top 5 positions. Hence, our finding shows the inefficacy and limitation of SBFL techniques due to the lack of failing tests related to the bug reports.

> Due to a lack of failing fault-triggering tests, traditional SBFL techniques have inferior localization results.

5.2 RQ2 - What is the relationship between stack traces and buggy location?

5.2.1 Motivation

In the previous RQ, we observed that most crash report bugs do not have fault-triggering tests. Because of that, the stack traces stand out as the best source of execution information. In this RQ, we want to study how stack traces are related to the buggy location and how they can be used to locate these bugs. To do so, we look at two things: (i) what is the type of modification performed in the bug fix to resolve the exception, and (ii) how far away the buggy methods (i.e., methods updated in the bug fix) are from the methods in the stack traces.

5.2.2 Approach

To study the type of modification performed in the bugfix to handle the exception, we manually examine the bug-fix patch of all the studied bugs. Based on the modification performed, we classify the bug-fix intention type into four categories:

(1) **Exception Prevention:** This category includes code modifications to prevent a specific exception's recurrence. For example, the bugfix patch from *Cli-5* introduced a new conditional structure to avoid the occurrence of the reported *NullPointerException* (Figure 5.1).

(2) **Exception Conversion:** In this type of bugfix, the exception is converted into a warning or error message. For instance, to fix the bug *Closure-152* (Figure 5.2), the developers handled the *ClassCastException* being thrown and created a warning detailed message.

(3) **Exception Wrapping:** This category represents the cases in which the developers fix the bug by wrapping the exception into another exception type. One example is the bug *COMPRESS-12*, in which the exception *IllegalArgumentException* was wrapped into the exception *IOException* (Figure 5.3)

(4) **Keep Throwing:** Finally, this category includes the bugs in which no exception handling-related code is found in the bugfix.

To study the distance between the stack trace and the buggy methods, we utilize the stack trace content and the code in the bug report commit to create a call graph for each bug. We then measure the minimum distance between a method in the stack trace and one of the buggy methods, if reachable. For example, the bug Math-79 in Figure 5.4 has a 3-call distance between the stack trace and the buggy method. The distance is zero for a bug if the buggy method is recorded in the stack trace.

5.2.3 Results

In 83% of the cases, the bugfix intention is to prevent the exception from happening again. Figure 5.5 shows the bugfix intention type distribution for studied bugs. In the majority of the fixes

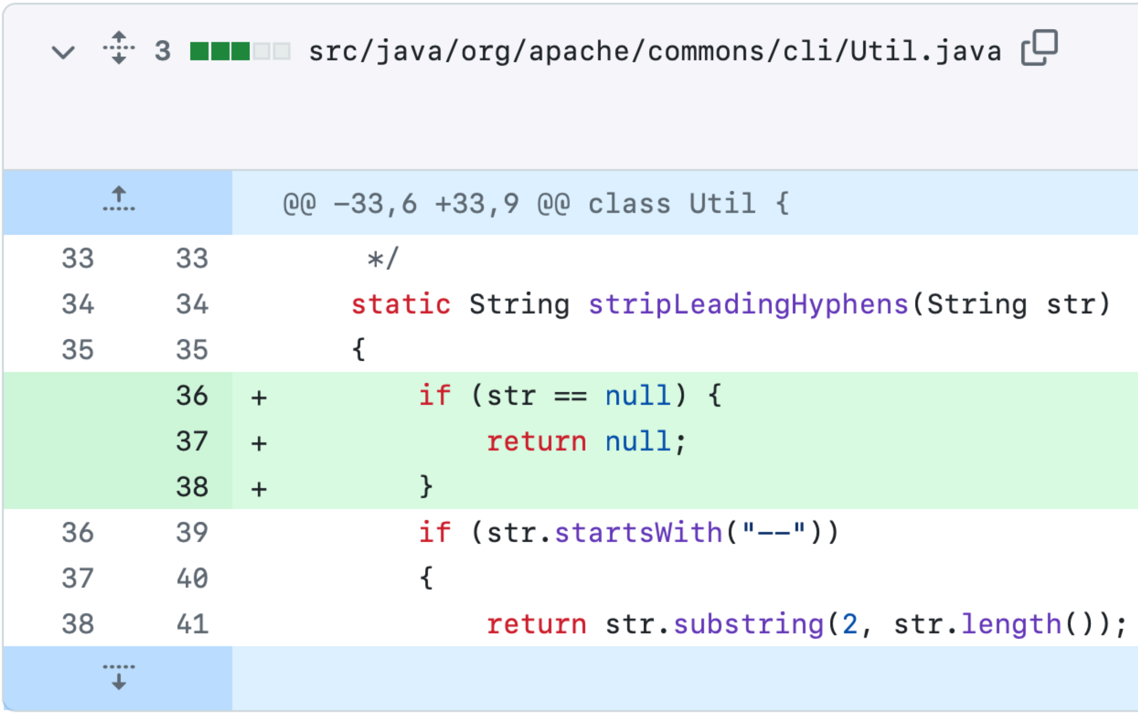


Figure 5.1: Extract from the bugfix patch for Cli-5, classified in the Exception Prevention category.

(83.3%), the intention behind the fix is **Exception Prevention** (i.e., to prevent the exception from happening again). This means that the stack traces are directly related to the root cause of the bug, which suggests that stack trace is an essential source of information to locate the bug. The following two predominant categories are **Exception Wrapping** (10.0%) and **Exception Conversion** (5.0%), which also possess a correlation between the fix and the stack traces. In only one bug, *COMPRESS-31*, the exception-related code was not touched for the fix (**Keep Throwing** category). After a more detailed investigation, we noticed that, despite this bug being in the Defect4j database, its resolution field was marked as “Not A Problem” on Jira, which explains why nothing was done about the exception.

In 78.3% of the bugs, the buggy methods are directly reachable from the stack trace, with an average distance of 0.34 method calls. Upon examining the proximity between stack traces and buggy methods, our analysis reveals that 66.67% of the bugs feature at least one buggy method directly present in the stack trace, indicating a zero distance. Furthermore, a total of 78.3% of the bugs contain buggy methods that are reachable from the stack trace (i.e., distance is zero or

```diff
src/com/google/javascript/jscomp/FunctionTypeBuilder.java
@@ -18,6 +18,7 @@

 import static com.google.javascript.jscomp.TypeCheck.BAD_IMPLEMENTED_TYPE;
 import static com.google.javascript.rhino.jstype.JSTypeNative.FUNCTION_FUNCTION_TYPE;
+import static com.google.javascript.rhino.jstype.JSTypeNative.OBJECT_TYPE;
 import static com.google.javascript.rhino.jstype.JSTypeNative.UNKNOWN_TYPE;
 import static com.google.javascript.rhino.jstype.JSTypeNative.VOID_TYPE;

@@ -125,6 +126,12 @@ final class FunctionTypeBuilder {
       "JSC_TEMPLATE_TYPE_EXPECTED",
       "The template type must be a parameter type");

+  static final DiagnosticType THIS_TYPE_NON_OBJECT =
+      DiagnosticType.warning(
+          "JSC_THIS_TYPE_NON_OBJECT",
+          "@this type of a function must be an object\n" +
+          "Actual type: {0}");
```

Figure 5.2: Extract from the bugfix patch for Closure-152, classified in the Exception Conversion category.

```diff
8
src/main/java/org/apache/commons/compress/archivers/tar/TarArchiveInputS…
@@ -195,7 +195,13 @@ public TarArchiveEntry getNextTarEntry() throws IOException {
                 return null;
             }

-            currEntry = new TarArchiveEntry(headerBuf);
+            try {
+                currEntry = new TarArchiveEntry(headerBuf);
+            } catch (IllegalArgumentException e) {
+                IOException ioe = new IOException("Error detected parsing the header");
+                ioe.initCause(e);
+                throw ioe;
+            }
             entryOffset = 0;
             entrySize = currEntry.getSize();
```

Figure 5.3: Extract from the bugfix patch for COMPRESS-12, classified in the Exception Wrapping category.

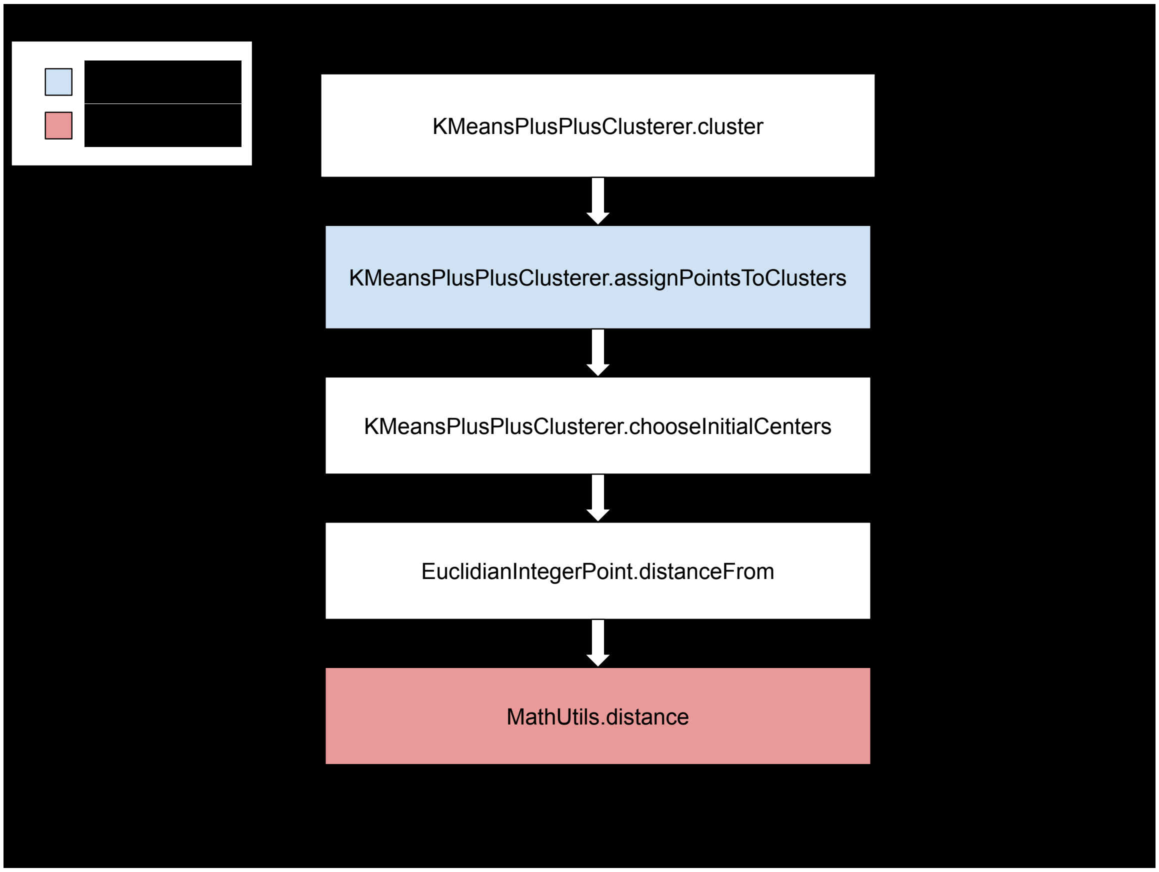


Figure 5.4: Call Graph for the bug Math-79, which has a 3-call distance between the stack trace and the buggy method.

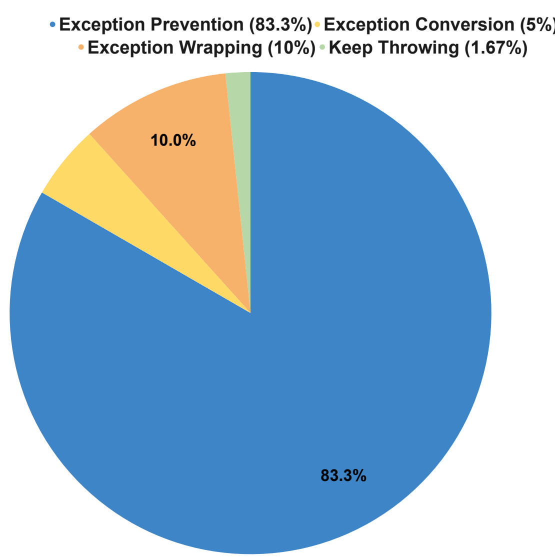


Figure 5.5: Bugfix intention type distribution in the studied bugs.

more), averaging a very short distance from the stack trace methods of 0.34 method calls. In such scenarios, leveraging stack traces to reconstruct the execution path at the time of the exception can be an effective method for pinpointing the bug. We conduct a manual analysis of the remaining 21.7% of the bugs in which the bugs methods are unreachable from the stack trace. We find that, in these cases, the stack traces only contain external entries (i.e., entries referring to external libraries) or the provided stack traces were incomplete. In short, our findings highlight the potential of using stack traces for fault localization.

> In 98.3% of the studied bugs, the bugfix intention is directly correlated with the exception in the stack trace (Exception Prevention, Exception Conversion or Exception Wrapping). In addition, 78.3% of the buggy methods are reachable from the stack traces, having an average distance of 0.34 method calls. This shows that the stack traces are a valuable source of information about the bug.

5.3 RQ3 - Can we utilize the stack traces to help detect the buggy locations?

5.3.1 Motivation

In RQ1, we found that the existing failing tests rarely matched the actual fault-triggering tests. Without fault-triggering tests, the performance of SBFL techniques is greatly affected. In addition, in RQ2, we found that the stack traces are a valuable source of information about the bug. The intention behind the bugfix is usually related to the exception, and the buggy methods are often a short distance away from the stack traces. Prior studies (Chen et al., 2021a, 2021b) show that developers usually rely on stack traces when investigating for bugs, as they provide essential information about the buggy location. Stack traces, similar to fault-triggering tests, carry contextual and execution information associated with the root causes of bugs. In a way, stack traces can be used to represent the coverage of a failing fault-triggering test. Therefore, in this RQ, we aim to investigate how stack traces can be leveraged to complement test cases in locating bugs.

5.3.2 Approach

To understand how stack trace information can contribute to locating buggy methods, we proposed to evaluate an approach called SBEST (Spectrum-Based localization Enhanced by Stack Traces). SBEST is a fault localization approach based on SBFL that incorporates stack trace information with the test coverage data. This technique applies SBFL principles, but instead of using the failing fault-triggering test coverage in the Ochiai formula, it considers the methods that appear on the stack trace entries as the source of the failure. We define a method that is causing a test failure if it appears on any of the frames in a stack trace. Prior studies (Chen et al., 2021b; C.-P. Wong et al., 2014a) have also shown that the position of methods in the stack trace can be useful in fault localization. Therefore, we incorporate a Stack Trace (ST) score into our approach. We compare our approach with two baselines. We use Ochiai as the first one because it is shown to be one of the best SBFL techniques and performs very well on real faults (Pearson et al., 2017). In addition, we leverage the position of the stack trace as the second baseline. For this, we rank the top entries in the stack trace as more suspicious. Below, we describe our approach in detail.

Our overall approach, SBEST, consists of the sum of two scores: the SB score and the ST score.

Spectrum Based (SB) Score. The first one, the SB score, is based on an approach designed to use the information from the stack traces in the fault localization process. It is based on the Ochiai formula, and it utilizes existing test coverage information. However, instead of using the failing tests as the hint of the bug location (since there may not be any test failure or the failure is not related to the reported bug, as found in RQ1), we use the tests covering the methods in the stack trace as the fault-triggering tests. We base this change on the fact that the stack trace represents the system execution at the moment of the failure in the same way that the fault-triggering tests do. More specifically, we utilize the methods present on the first five internal entries from the stack traces to select the failing tests that will be applied to Ochiai's formula. We base this design on the fact that the higher the stack trace entry, the more probable it is to be related to the bug (C.-P. Wong et al., 2014b).

The selection of failing tests is determined by counting the number of lines each existing test covers in the Stack Trace methods. We first compute the number of covered lines from these methods for each test by applying the following formula:

$$ST\_Covered\_lines\_number_t = \sum_{m=1}^{5} CL_m \tag{2}$$

in which *m* represents each of the top 5 methods in the Stack Trace, and CL_m is the number of lines in the correspondent method that is covered by the test. Then, we select the set of the *X* highest results by applying the following formula, in which T represents the set of tests and X is a threshold for the number of selected tests:

$$\begin{aligned} T_{\text{failing}} = \{t \in T \mid t \text{ is one of the } X \text{ tests with the highest} \\ ST\_Covered\_lines\_number_t\} \end{aligned} \tag{3}$$

We noticed that selecting a value for *X* that was too low made the approach often miss buggy locations because it was too focused on just a few methods. On the other hand, a high threshold also generates inaccurate results since the approach is unable to differentiate between the methods in the stack trace. Upon experimentation, we noticed that setting it to 15 gave the best results. Finally, we apply the Ochiai formula (1) using these 15 tests as failing tests and all the remaining as passing tests.

Stack Trace (ST) Score. The SB score described above merges the information in the stack traces with the test coverage to compute the suspiciousness score for a given method. In addition to that, we make use of the ST score used in a previous study (Chen et al., 2021b) in order to boost the stack trace's impact into the calculated score. To do this, after having computed the SB_score following the approach described before, we sum to it a ST_score calculated in the following manner:

$$ST\_score = \begin{cases} \frac{1}{ST\_rank} & \text{if } ST\_rank \leq 10 \\ 0.1 & \text{if } ST\_rank > 10 \\ 0 & \text{if method not found} \end{cases} \tag{4}$$

In which *ST_rank* is the position in which the given method appears in the stack trace after the external entries are removed. For example, if the method appears in the second position, the ST_score would be 1/2=0.5.

Suspiciouness Score. After having both the STCB Ochiai score and the ST Score, we compute the suspiciousness score of a giving method by applying the formula:

$$Suspiciouness\_score = SB\_score + ST\_score \tag{5}$$

Suspiciouness Rank. After having the final suspiciousness score, we generate the final suspiciousness rankings. The higher the suspiciousness score, the higher the ranking. For instance, if the highest calculated suspiciousness score is 1.0 for the method *M8*, *M8* is going to be the first method in the suspiciousness rank, as it has the highest probability of containing the bug.

5.3.3 Results

The Stack Traces ranking alone locates 34 out of the 60 bugs in the Top-5. Table 5.5 presents the metrics results for the 2 baselines as well as for the SB_score alone and the SBEST approach for all the systems. We can see that the worst-performing technique is Ochiai, which, as discussed before, locates only 2 bugs in Top-5. The best-performing approach is SBEST, but the Stack Traces ranking is a close second. The Stack Trace ranking locates 16 bugs in Top-1, 27 in Top-3 and 34 in Top-5. Surprisingly, the Stack Traces alone provide very good results, locating more than half of the bugs (56.67%) within the Top 5. This shows that the stack traces are a very important source of information about the bug location and should be prioritized when it comes to studying FL in crash reports. The SB_score is able to locate some bugs that the Stack Trace ranking does not locate, but in general, it does not perform very well, identifying only 18.33% of the buggy methods in the Top 5. We can see, however, that the SB_score is substantially improved by the addition of the ST_score. The resultant technique SBEST performs slightly better than the Stack Traces in the Top 1 and Top 3, locating 17 and 32 bugs, respectively, and locates only one bug less in the Top 5, totalling 33 bugs spotted within the Top 5 rank. This technique and the Stack Traces ranking take turns as the best localization approach for each project. For example, the Stack Trace ranking is better overall for the JacksonCore project, while SBEST has the top performance on the Lang bugs. In total, SBEST locates 17 bugs in Top-1, 32 in Top-3 and 33 in Top-5. The average MAP and MRR for SBEST are 0.42846 and 0.49647, respectively, representing a 32.22% improvement on the MAP

Table 5.5: Fault localization results for all the studied projects. *#Bugs* represents the total number of studied bugs in each project.

System (#Bugs)	Technique	Top-1	Top-3	Top-5	MAP	MRR
Cli (2)	Ochiai	0	0	0	0.0045	0.0045
	Stack Trace	1	1	2	0.6	0.6
	SB_score	0	0	1	0.1333	0.1333
	SBEST	1	1	2	0.6	0.6
Closure (8)	Ochiai	0	0	0	0.0007	0.0012
	Stack Trace	1	2	2	0.1710	0.2170
	SB_score	0	0	0	0.0003	0.005
	SBEST	1	2	2	0.1462	0.1671
Codec (1)	Ochiai	0	0	0	-	-
	Stack Trace	1	1	1	0.3333	1
	SB_score	1	1	1	0.3893	1
	SBEST	1	1	1	0.5853	1
Compress (11)	Ochiai	0	0	1	0.0079	0.023
	Stack Trace	3	6	8	0.4122	0.4425
	SB_score	0	3	3	0.1404	0.1587
	SBEST	3	9	9	0.4620	0.4904
Csv (1)	Ochiai	0	0	0	-	-
	Stack Trace	0	1	1	0.3333	0.3333
	SB_score	1	1	1	1	1
	SBEST	1	1	1	1	1
Gson (3)	Ochiai	0	0	0	-	-
	Stack Trace	1	2	2	0.5196	0.5196
	SB_score	1	1	1	0.3338	0.3338
	SBEST	1	2	2	0.5101	0.5101
JacksonCore (4)	Ochiai	0	1	1	0.0693	0.0836
	Stack Trace	1	1	1	0.25	0.25
	SB_score	0	0	0	0.0409	0.0457
	SBEST	0	1	1	0.1609	0.1634
JacksonDatabind (10)	Ochiai	0	0	0	0.0005	0.0005
	Stack Trace	1	2	3	0.1686	0.2446
	SB_score	0	1	1	0.0125	0.0212
	SBEST	1	3	3	0.0359	0.0777
Jsoup (8)	Ochiai	0	0	0	0	0
	Stack Trace	3	3	5	0.4410	0.4444
	SB_score	0	0	0	0.0214	0.0318
	SBEST	3	4	4	0.4070	0.4279
JxPath (1)	Ochiai	0	0	0	-	-
	Stack Trace	1	1	1	0.3333	0.3333
	SB_score	0	0	0	0.0625	0.0625
	SBEST	1	1	1	1	1
Lang (4)	Ochiai	0	0	0	0.0008	0.0008
	Stack Trace	1	3	3	0.4375	0.4375
	SB_score	0	1	1	0.1411	0.1411
	SBEST	2	3	3	0.5835	0.5835
Math (3)	Ochiai	0	0	0	0.0002	0.0001
	Stack Trace	2	2	2	0.2215	0.4583
	SB_score	0	0	0	0.0428	0.0292
	SBEST	2	2	2	0.1372	0.3472
Mockito (2)	Ochiai	0	0	0	0.0060	0.0281
	Stack Trace	0	1	1	0.0238	0.1667
	SB_score	0	1	1	0.0371	0.25
	SBEST	0	1	1	0.0371	0.25
Time (2)	Ochiai	0	0	0	0.0003	0.0003
	Stack Trace	0	1	2	0.2917	0.2917
	SB_score	1	1	1	0.5001	0.5001
	SBEST	0	1	1	0.3333	0.3333
Total (60)	Ochiai	0	1	2	0.00902	0.01421
	Stack Trace	16	27	34	0.32406	0.40992
	SB_score	4	10	11	0.20396	0.26517
	SBEST	17	32	33	0.42846	0.49647

and a 17.43% in the MRR when compared to the Stack Traces. These results show that the stack traces can complement existing passing test information to help locate the bug root causes.

> Overall, out of 60 studied bugs, SBEST successfully located 17 bugs in Top-1, 32 in Top-3 and 33 in Top-5. In addition, it achieves an improvement of 32.22% on MAP and 17.43% on MRR when compared to the Stack Trace ranking.

Chapter 6

Discussion

In this chapter, we will explore the implications of the findings presented and examine their potential impact on future research.

6.1 On the use of stack traces in the absence of failing tests.

From the results obtained in Chapter 5, we observe that the majority of the crash report bugs do not contain fault-triggering tests. In such cases, the test failure information is not available for the fault localization process, causing techniques such as SBFL to fail. The stack traces, on the other hand, demonstrate to be deeply correlated to the bug cause. We found that, in 66% of the bugs, at least one buggy method is directly listed in the stack trace. Even when this is not the case, the buggy methods are usually just a few calls away from it. In addition, the most common bugfix intention is to prevent the exception in the stack trace from happening again, which highlights the big association between them.

6.2 Leveraging stack trace rankings for enhanced fault localization.

In the real world, test failures are not always present. Especially when it comes to production-phase bugs, it is important to look for alternative sources of information for fault localization. Making use of the data available in logs and stack traces shows to be a good path to take in such scenarios. Even though previous studies have used stack traces for Fault Localization, many are based on information retrieval (IRFL) approaches (Lam, Nguyen, Nguyen, & Nguyen, 2017; Saha, Lease, Khurshid, & Perry, 2013; Zhou, Zhang, & Lo, 2012). IRFL techniques treat all the content in the bug report, including the stack traces, as textual information, therefore missing important context such as the ranking of the stack traces. Our results show that the stack traces ranking alone was able to locate more than half of the bugs within Top-5 and, thus, should be better utilized.

6.3 Encouraging the integration of stack traces to complement execution information.

SBEST, our approach combining the stack trace ranking with the code coverage, shows promising results. The improvement in the metrics implies that the coverage information was able to enhance the stack trace ranking, aiding in obtaining a better overview of the system execution in the moment of failure. Our main goal when studying this technique was to gain a deeper understanding of the scenario of these bugs and how each piece of information available can help with fault localization. We believe that this is just the first step and that future research can benefit from the aforementioned findings to build more intricate techniques. In addition, we make all the data from this study available at Zenodo[1], including the detailed code coverage and test results of the bug report commits that took us 100 hours to obtain. We think that this data will be useful for future research on subjects such as fault localization, automatic bug fixing, test generation, etc. Our data does not contain post-bug-fixing development knowledge, which will allow researchers to tailor tools more aligned with the real-world scenario.

[1] https://zenodo.org/records/11062413

Chapter 7

Threats to Validity

In this chapter, we discuss the key threats to the validity of our findings and their implications for interpreting the results.

7.1 External Validity.

Threats to external validity relate to how generalizable our findings are. To minimize this threat, we conduct our studies in 15 systems from Defects4j, a famous framework of real bugs used in multiple other studies. These systems are widely used and vary in size and number of tests. Even though every system is written in Java, our method is not restricted to Java-based systems. In addition, despite the low percentage of bugs with stack traces in Defects4j, this aligns with the percentage reported in other systems (Chen et al., 2021a). Moreover, we opted not to use the buggy version made available in Defects4j but the code available at the moment of the bug report creation in order to better emulate the scenario in which the developers investigate the bug. Finally, even though we opted for applying only Ochiai when analyzing SBFL's efficiency when applied to crash report bugs, it is shown to be one of the best SBFL techniques and performs very well on real faults (Pearson et al., 2017).

7.2 Internal validity.

Threats to internal validity relate to the extent to which the results of the study can be attributed solely to the experimental treatments and not to flaws in the experimental design. In this study, we only analyze Defects4j bugs. Although Defects4j is widely used (Chen et al., 2021b; Li et al., 2021; Lutellier et al., 2020; Ye et al., 2022), the findings might be different in other systems, specially non-Java systems. Another threat to internal validity is the way in which we selected the bug reports with stack traces. We collect stack traces in the description and comments, which is the typical place in which they are added. However, in some cases, developers may also add stack traces via attachments. Nonetheless, this is very rare, representing less than 1% of the cases based on a previous study from (Chen et al., 2021a).

7.3 Construct validity.

Construct validity refers to how well the study's procedures and metrics accurately capture the concepts they intend to investigate. We use three evaluation metrics in our study: Top K, MAP, and MRR. These metrics are commonly used in FL and have been used in many previous studies (Chen et al., 2021b; Kochhar et al., 2016; Wen et al., 2021; Xia, Bao, Lo, & Li, 2016).

Chapter 8

Future Work

In this study, we obtained promising results with SBEST by incorporating stack trace information into Spectrum-Based Fault Localization (SBFL). Our analysis indicates that stack trace rankings are crucial for accurately identifying the bug location, and test coverage can also be beneficial even in the absence of failing tests. However, we believe that the results can be further improved by applying more sophisticated techniques to the same bugs.

Recently, Large Language Model (LLM) Based Multi-Agent Systems (He, Treude, & Lo, 2024) have been increasingly utilized for a variety of tasks, particularly in the software engineering domain. These systems leverage multiple LLMs that communicate and collaborate to perform complex tasks more efficiently than individual models. Examples of their application include code generation (Qian et al., 2023), where the system can automatically write code based on given requirements, and program repair (Bouzenia, Devanbu, & Pradel, 2024), where the system identifies and fixes bugs in existing code. We propose that applying LLM agents to the context of this study is a promising direction to enhance bug localization results. By utilizing the collaborative capabilities of LLM-based multi-agent systems, we can achieve more accurate and efficient identification of bugs, leading to improved software quality and reduced development time.

As a proof of concept, we developed a simplified LLM-based multi-agent bug localization tool[1] using ChatGpt 3.5, Python and Langchain. This tool consists of two agents, a tester and a debugger, who have access to seven tools to obtain details about the bug:

[1] `https://github.com/SPEAR-SE/llm-bug-localization`

- **Get bug report textual information:** Returns the textual information contained in the bug report, including the title and the description. Comments were not included to avoid the presence of the developer's knowledge about the bug-fixing process.

- **Get stack trace:** Returns the stack trace(s) from the bug.

- **Get test IDs:** Returns a list with all the test IDs from the bug.

- **Get tests that better cover the stack trace:** Returns a list with the test IDs from the tests that better cover the top 5 stack trace entries, limited to a maximum of 15 tests. Those are the same tests used in SBEST implementation.

- **Get test body by ID:** Takes the test ID as a parameter and returns the corresponding test body.

- **Get methods covered by a test:** Returns a list with all the methods covered by a given test.

- **Get method body and signature by ID:** Takes the method ID as a parameter and returns the method body and signature.

- **Get method body by signature:** Takes the method signature as a parameter and returns the method body.

With these tools, the agents can obtain comprehensive details about the system under failure, including the stack trace and test coverage information, similarly to SBEST. The agents were instructed to gather relevant information, analyze it, and engage in discussions until they reached a consensus on the top five most suspicious methods. Once an agreement was reached, the agents were requested to return a ranked list of these methods.

To evaluate the effectiveness of this approach, the tool was executed for all 60 bugs included in this study. Each bug scenario was analyzed in detail, and the agents collaboratively determined the most likely candidates for containing bugs. The ranking produced by the agents was then compared with the actual list of buggy methods identified in the system. This comparison allowed us to assess the accuracy and reliability of the LLM-based multi-agent system in identifying potential bugs.

The results demonstrated the potential of using LLM-based multi-agent systems for bug localization, highlighting their ability to analyze complex software systems, collaborate effectively, and produce accurate rankings of suspicious methods. Specifically, the LLM multi-agent implementation was able to locate 23 bugs in the Top-1 metric, 30 in the Top-3, and 34 in the Top-5, thereby outperforming SBEST, as shown in Table 8.1.

Table 8.1: LLM multi-agent implementation results compared to SBEST

Technique	**Top-1**	**Top-3**	**Top-5**
SBEST	17	32	33
LLM multi-agent	23	30	34

Future research directions could explore the integration of additional AI techniques such as reinforcement learning and advanced natural language processing (NLP) methods to further enhance the performance of LLM-based multi-agent systems in bug localization. Additionally, expanding the dataset and applying the system to a broader range of software projects could provide deeper insights into its scalability and generalizability.

Chapter 9

Conclusion

Spectrum-based fault Localization (SBFL) approaches are highly recognized due to their accuracy, efficiency, and simplicity. However, their effectiveness depends on the availability of fault-triggering failing tests. In the lack of such test failures, alternative approaches should be considered. In this paper, we research using stack traces for SBFL in the absence of failing tests. We study 60 crash reports from the Defects4J benchmark. We find that most of the studied crash reports do not contain fault-triggering failing tests. This results in very low efficiency when using traditional SBFL. On the other hand, when it comes to the stack traces, we find that in 98.3% of the studied bugs, the bugfix intention is directly correlated with the exception in the stack trace. In addition, in 78.3% of the bugs, the buggy method is directly reachable from the stack trace, with an average distance of only 0.34 method calls. This shows that the buggy methods are usually very close to the stack traces when it comes to the execution call graph. On top of that, our results show that even without any advanced technique, the stack traces alone provide a good indication of the buggy locations, being capable of locating more than half of the bugs on Top-5. Finally, we develop a simplified SBFL method called SBEST that uses the stack trace information in place of the failing tests and thus integrates it into the coverage information to perform the SBFL. SBEST was able to locate 17 bugs in Top-1, 32 in Top-3 and 33 in Top-5, which represents an improvement of 32.22% on MAP and 17.43% on MRR when compared to the Stack Trace ranking. The enhancement of the metrics suggests that the coverage information improved the stack trace ranking and helped the technique gain a clearer picture of the system's execution at the time of failure. We believe that future studies can benefit from the findings of this research to develop more intricate techniques. In addition, all the data that we have made available can help studies in areas such as automatic bug fixing, test generation, and, of course, fault localization.

References

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2006). An evaluation of similarity coefficients for software fault localization. In *2006 12th pacific rim international symposium on dependable computing (prdc'06)* (p. 39-46). doi: 10.1109/PRDC.2006.18

Abreu, R., Zoeteweij, P., & van Gemund, A. J. (2007). On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-mutation (taicpart-mutation 2007)* (p. 89-98). IEEE.

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2009). Spectrum-based multiple fault localization. In *2009 ieee/acm international conference on automated software engineering* (pp. 88–99).

Baudry, B., Fleurey, F., & Le Traon, Y. (2006). Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on software engineering* (pp. 82–91).

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., & Zimmermann, T. (2008). What makes a good bug report? In *Proceedings of the 16th acm sigsoft international symposium on foundations of software engineering* (pp. 308–318).

Bouzenia, I., Devanbu, P., & Pradel, M. (2024). Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*.

Campos, J., et al. (2012). Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th ieee/acm international conference on automated software engineering* (p. 378-381).

Chen, A. R., Chen, T.-H., & Chen, J. (2022). How useful is code change information for fault localization in continuous integration? In *37th ieee/acm international conference on automated software engineering* (p. 1-12).

Chen, A. R., Chen, T.-H., & Wang, S. (2021a). Demystifying the challenges and benefits of analyzing user-reported logs in bug reports. *Empirical Software Engineering*, *26*, 1-30.

Chen, A. R., Chen, T.-H., & Wang, S. (2021b). Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, *48*(8), 2905-2919.

Chen, A. R., Rafi, M. N., Wang, S., et al. (2023). Back to the future! studying data cleanness in defects4j and its impact on fault localization. *arXiv preprint arXiv:2310.19139*.

Cui, Z., Jia, M., Chen, X., Zheng, L., & Liu, X. (2020). Improving software fault localization by combining spectrum and mutation. *IEEE Access*, *8*, 172296-172307. doi: 10.1109/ACCESS.2020.3025460

de Oliveira, A. A. L., Camilo-Junior, C. G., de Andrade Freitas, E. N., & Vincenzi, A. M. R. (2018). Ftmes: A failed-test-oriented mutant execution strategy for mutation-based fault localization. In *2018 ieee 29th international symposium on software reliability engineering (issre)* (pp. 155–165).

de Souza, H. A., Chaim, M. L., & Kon, F. (2016). Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*.

Gong, L., Lo, D., Jiang, L., & Zhang, H. (2012). Interactive fault localization leveraging simple user feedback. In *2012 28th ieee international conference on software maintenance (icsm)* (pp. 67–76).

Haben, G., Habchi, S., Papadakis, M., Cordy, M., & Traon, Y. L. (2023). The importance of discerning flaky from fault-triggering test failures: A case study on the chromium ci. *arXiv preprint arXiv:2302.10594*.

He, J., Treude, C., & Lo, D. (2024). Llm-based multi-agent systems for software engineering: Vision and the road ahead. *arXiv preprint arXiv:2404.04834*.

Hong, S., Kwak, T., Lee, B., Jeon, Y., Ko, B., Kim, Y., & Kim, M. (2017). Museum: Debugging real-world multilingual programs using mutation analysis. *Information and Software Technology*, *82*, 80–95.

Jones, J. A., & Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th ieee/acm international conference on automated software engineering* (p. 273-282).

Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis* (p. 437-440).

Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., & Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 654–665).

Kabadi, V., Kong, D., Xie, S., Bao, L., Prana, G. A. A., Le, T.-D. B., ... Lo, D. (2023). The future can't help fix the past: Assessing program repair in the wild. In *2023 ieee international conference on software maintenance and evolution (icsme)* (pp. 50–61).

Kochhar, P. S., Xia, X., Lo, D., & Li, S. (2016). Practitioners' expectations on automated fault localization. In *Proceedings of the 25th international symposium on software testing and analysis* (pp. 165–176).

Küçük, Y., Henderson, T. A. D., & Podgurski, A. (2019). The impact of rare failures on statistical fault localization: The case of the defects4j suite. In *2019 ieee international conference on software maintenance and evolution (icsme)* (p. 24-28). doi: 10.1109/ICSME.2019.00012

Lam, A. N., Nguyen, A. T., Nguyen, H. A., & Nguyen, T. N. (2017). Bug localization with combination of deep learning and information retrieval. In *2017 ieee/acm 25th international conference on program comprehension (icpc)* (pp. 218–229).

Le, T.-D. B., Thung, F., & Lo, D. (2013). Theory and practice, do they match? a case with spectrum-based fault localization. In *2013 ieee international conference on software maintenance* (p. 380-383). doi: 10.1109/ICSM.2013.52

Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th international conference on software engineering (icse)* (pp. 3–13).

Leveraging stack traces for spectrum-based fault localization in the absence of failing tests. (2024). https://zenodo.org/records/11062413.

Li, Y., Wang, S., & Nguyen, T. N. (2021). Fault localization with code coverage representation learning. in 2021 ieee/acm 43rd international conference on software engineering (icse). *IEEE, 661ś673*.

Liu, Y., Li, Z., Zhao, R., & Gong, P. (2018). An optimal mutation execution strategy for cost reduction of mutation-based fault localization. *Information Sciences*, *422*, 572–596.

Lou, Y., Zhu, Q., Dong, J., Li, X., Sun, Z., Hao, D., … Zhang, L. (2021). Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 664–676).

Lutellier, T., et al. (2020). Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th acm sigsoft international symposium on software testing and analysis* (p. 101-114).

Moon, S., Kim, Y., Kim, M., & Yoo, S. (2014). Ask the mutants: Mutating faulty programs for fault localization. In *2014 ieee seventh international conference on software testing, verification and validation* (pp. 153–162).

Naish, L., Lee, H. J., & Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, *20*(3), 1–32.

Papadakis, M., & Le Traon, Y. (2012). Using mutants to locate" unknown" faults. In *2012 ieee fifth international conference on software testing, verification and validation* (pp. 691–700).

Papadakis, M., & Le Traon, Y. (2015). Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, *25*(5-7), 605–628.

Parnin, C., & Orso, A. (2011). Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis* (pp. 199–209).

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., … Keller, B. (2016). Evaluating & improving fault localization techniques. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03*, 27.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., … Keller, B. (2017). Evaluating and improving fault localization. In *2017 ieee/acm 39th international conference on software engineering (icse)* (p. 609-620). doi: 10.1109/ICSE.2017.62

Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., … Sun, M. (2023). Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Saha, R. K., Lease, M., Khurshid, S., & Perry, D. E. (2013). Improving bug localization using structured information retrieval. In *2013 28th ieee/acm international conference on automated software engineering (ase)* (pp. 345–355).

Santelices, R., Jones, J. A., Yu, Y., & Harrold, M. J. (2009). Lightweight fault-localization using multiple coverage types. In *2009 ieee 31st international conference on software engineering* (p. 56-66). doi: 10.1109/ICSE.2009.5070508

Schroter, A., Schröter, A., Bettenburg, N., & Premraj, R. (2010). Do stack traces help developers fix bugs? In *2010 7th ieee working conference on mining software repositories (msr 2010)* (pp. 118–121).

Shah, H., Görg, C., & Harrold, M. J. (2008). Why do developers neglect exception handling? In *Proceedings of the 4th international workshop on exception handling* (pp. 62–68).

Silva, A., et al. (2021). Flacoco: Fault localization for java based on industry-grade coverage. *arXiv preprint arXiv:2111.12513*.

Wang, H., Li, Z., Liu, Y., Chen, X., Paul, D., Cai, Y., & Fan, L. (2022). Can higher-order mutants improve the performance of mutation-based fault localization? *IEEE Transactions on Reliability*, *71*(2), 1157-1173. doi: 10.1109/TR.2022.3162039

Wen, M., Chen, J., Tian, Y., Wu, R., Hao, D., Han, S., & Cheung, S.-C. (2021). Historical spectrum based fault localization. *IEEE Transactions on Software Engineering*, *47*(11), 2348-2368. doi: 10.1109/TSE.2019.2948158

Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., & Mei, H. (2014a). 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).

Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., & Mei, H. (2014b). Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 ieee international conference on software maintenance and evolution* (p. 181-190). doi: 10.1109/ICSME.2014.40

Wong, W. E., Debroy, V., Gao, R., & Li, Y. (2013). The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, *63*(1), 290–308.

Xia, X., Bao, L., Lo, D., & Li, S. (2016). "automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *2016 ieee international conference on software maintenance and evolution (icsme)* (pp. 267–278).

Ye, H., Martinez, M., & Monperrus, M. (2022). Neural program repair with execution-based backpropagation. In *Proceedings of the 44th international conference on software engineering* (pp. 1506–1518).

Zhang, J., Liu, K., Kim, D., Li, L., Liu, Z., Klein, J., & Bissyandé, T. F. (2021). Revisiting test cases to boost generate-and-validate program repair. In *2021 ieee international conference on software maintenance and evolution (icsme)* (p. 35-46). doi: 10.1109/ICSME52107.2021.00010

Zhou, J., Zhang, H., & Lo, D. (2012). Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th international conference on software engineering (icse)* (pp. 14–24).

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., & Weiss, C. (2010). What makes a good bug report? *IEEE Transactions on Software Engineering*, *36*(5), 618-643. doi: 10.1109/TSE.2010.63