EXTENDING THE REACH OF FAULT LOCALIZATION TO ASSIST IN AUTOMATED DEBUGGING

An Ran Chen

Under the supervision of Dr. Tse-Hsun (Peter) Chen

Doctoral Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

August 2023

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: An Ran Chen

Entitled: Extending the Reach of Fault Localization to Assist in Automated Debugging

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

Doctor Of Philosophy Software Engineering

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

Signed by the final examining committee:

Dr. Manar Amayri	Chair
Dr. David Lo	External Examiner
Dr. Nikolaos Tsantalis	Examiner
Dr. Juergen Rilling	Examiner
Dr. Yan Liu	Examiner
Dr. Tse-Hsun (Peter) Chen	Thesis Supervisor (s)

Approved by

Dr. Leila Kosseim Chair of Department or Graduate Program Director

August 7, 2023
Date of Defence

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

Abstract

Extending the reach of fault localization to assist in automated debugging

An Ran Chen, Ph.D.
Concordia University, 2023

Software debugging is one of the most time-consuming tasks in modern software maintenance. To assist developers with debugging, researchers have proposed fault localization techniques. These techniques aim to automate the process of locating faults in software, which can greatly reduce debugging time and assist developers in understanding the faults. Effective fault localization is also crucial for automated program repair techniques, as it helps identify potential faulty locations for patching.

Despite recent efforts to advance fault localization techniques, their effectiveness is still limited. With the increasing complexity of modern software, fault localization may not always provide direct identification of the root causes of faults. Further, there is a lack of studies on their application in modern software development. Most prior studies have evaluated these techniques in traditional software development settings, where only a single snapshot of the system is considered. However, modern software development often involves continuous and fine-grained changes to the system. This dissertation proposes a series of approaches to explore new automated debugging solutions that can enhance software quality assurance and reliability practices, with a specific focus on extending the reach of fault localization in modern software development.

The dissertation begins with an empirical study on user-reported logs in bug reports, revealing that re-constructed execution paths from these logs provide valuable debugging hints. To further assist developers in debugging, we propose using static analysis techniques for information-retrieval and path-guided fault localization. By leveraging execution paths from logs in bug reports, we can improve the effectiveness of fault localization techniques. Second, we investigate the characteristics of operational data in continuous integration that can help capture faults early in the testing phase. As there is currently no available continuous integration benchmark that incorporates continuous test execution and failure, we present T-Evos, a dataset that comprises various operational data in continuous integration settings. We propose automated fault localization techniques that integrate change information from continuous integration settings, and demonstrate that leveraging such fine-grained change information can significantly improve their effectiveness. Finally, the dissertation investigates the data cleanness in fault localization by examining developers' knowledge in fault-triggering tests. The study reveals a significant degradation in the performance of fault localization techniques when evaluated on faults without developer knowledge.

Through case studies and experiments, the proposed techniques in this dissertation significantly improve the effectiveness of fault localization and facilitate their adoption in modern software development. Additionally, this dissertation provides valuable insights into new debugging solutions for future research.

To my grandfather, who from what I've heard must have been amazing.

Acknowledgments

The ultimate outcome of this dissertation lies not in the publications I have authored or the research talks I have delivered, but in the growth I have experienced as a researcher. Hence, I would like to express my gratitude to all the mentors, researchers, and collaborators who have nurtured me throughout my graduate studies. Without their support, I would not have been able to reach this milestone.

First and foremost, I am immensely grateful to my supervisor, Dr. Tse-Hsun (Peter) Chen, who has been an exceptional mentor, guide, and source of inspiration throughout my academic journey. From our first meeting in his office, where I shared my conceivable yet ambitious research ideas, he patiently guided me in transforming those ideas into a well-defined research agenda. Dr. Chen not only taught me in becoming into a proficient researcher but also introduced me to the academic world. He supported the development of my research skills, writing, presentations, teaching, mentoring, and much more. I am truly grateful for the confidence he placed in me as a researcher. Thank you sincerely, Peter.

I would also like to extend my gratitude to my graduate committee members, Dr. Nikolaos Tsantalis, Dr. Juergen Rilling, and Dr. Yan Liu, for their valuable feedback, constructive critiques, and guidance. Their insights and comments have significantly improved the quality of this dissertation and inspired me to pursue further research. I am also deeply honored to have Dr. David Lo as my external examiner. His remarkable research work on software engineering and debugging has been a true inspiration for the development of my research idea.

Many professors, researchers, and mentors have played a crucial role to my development as a researcher. First, Dr. Shaowei Wang, who guided me and broaden my research horizons. I grew a lot as a researcher under his mentorship. Thank you to Dr. Xin Xia, as well as Dr. Kui Liu, who introduced me into new research communities and explored career opportunities. Also thanks to Dr. Junjie Chen, who warmly integrated me into Tianjin University and exposed me to the research expertise of his amazing research team. Thank you to Dr. Shaohua Wang, who guided me in research. Finally, a special thank you to Dr. Weiyi Shang who recommended me to Peter in early 2018.

I consider myself incredibly fortunate to have had the opportunity to work with remarkable labmates and colleagues. I am indebted to all members of our outstanding SPEAR group, and I extend a special thank you to the senior labmates and alumni, Dr. Zhenhao Li, Zishuo Ding, Zi Peng, Dr. Maxime Lamothe, and Arthur Vitui, for their companionship and support throughout my Ph.D. journey.

To my parents, who have always been a constant source of support and encouragement, I express my deepest gratitude. Whenever I travelled far, my luggage was filled with their unconditional love and support. Thank you. Lastly, I would like to express my appreciation to my fiancée Mengyue and our beloved cat Rum for being by my side during this long journey. A new journey awaits, and I am grateful to have you both with me.

Related Publications

In all chapters and related publications of this dissertation, I made contributions including formulating the initial research idea, conducting surveys of related work and research background, implementing the code, performing case studies, analyzing the results, and writing and finalizing the work. My co-authors assisted me by reviewing the initial idea, referencing related work, reviewing the writing, and providing feedback.

Earlier versions of the chapters in this dissertation were published and submitted as follows:

1. *Demystifying the Challenges and Benefits of Analyzing User-Reported Logs in Bug Reports* (Chapter 3)
   An Ran Chen, Tse-Hsun (Peter) Chen, Shaowei Wang, Empirical Software Engineering (EMSE), 2021. Pages 1-30.

2. *Pathidea: Improving Information Retrieval-Based Bug Localization by Re-Constructing Execution Paths Using Logs* (Chapter 4)
   An Ran Chen, Tse-Hsun (Peter) Chen, Shaowei Wang, IEEE Transactions on Software Engineering (TSE), 2021. Pages 2905-2919.

3. *T-Evos: A Large-Scale Longitudinal Study on CI Test Execution and Failure* (Chapter 5)
   An Ran Chen, Tse-Hsun (Peter) Chen, Shaowei Wang, IEEE Transactions on Software Engineering (TSE), 2022. Pages 2352-2365.

4. *How Useful is Code Change Information for Fault Localization in Continuous Integration?* (Chapter 6)
   An Ran Chen, Tse-Hsun (Peter) Chen, Junjie Chen, IEEE/ACM International Conference on Automated Software Engineering (ASE), 2022. Michigan, USA.

5. *Back to the Future! Studying Data Cleanness in Defects4J and its Impact on Fault Localization* (Chapter 7)
An Ran Chen, Md Nakhla Rafi, Tse-Hsun (Peter) Chen, Shaohua Wang, submitted to an international conference on software engineering, 2023.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software debugging is one of the most time-consuming tasks in software maintenance. On average, developers spend 50% of their time on debugging and bug fixing, rather than on implementing new features [32]. When an issue occurs, developers would create a bug report that documents the necessary information for others to reproduce, diagnose, and fix the bug. However, due to limited time and resources, many bugs remain unfixed for a long period of time. A prior study [46] finds that it often takes several months for developers to address a bug report, which further hinders the user-perceived quality of the system. With the increasing complexity of software, the consequences of software bugs can be severe, with a report [88] estimating the cost of dealing with poor software quality (e.g., failures and defects) to be USD$2.41 trillion in the United States alone in 2022. The substantial cost associated with software debugging has prompted researchers and practitioners to propose automated debugging solutions. For instance, fault localization techniques [14, 54, 71, 116, 125, 128, 177, 193, 206] have been proposed to assist in locating and understanding the root causes of faults.

Although the usefulness of fault localization techniques have been well studied [15, 86, 94, 144, 155], their application is still comprised in modern software development. For example, fault localization techniques still suffer from the lack of effectiveness [69, 177, 178, 185, 188] due to the increasing complexity of modern software. Another limitation is the lack of research on adapting fault localization to modern software development. Most prior studies [17, 78, 95, 150, 183] have evaluated these techniques in traditional software development settings, where only a single snapshot of the system is considered. In contract, in modern software development, and especially in continuous integration, developers make continuous and finer-grained changes to the system.

In recent years, the use of Continuous Integration (CI) has become a common practice to assist in software maintenance. This presents an opportunity to leverage fine-grained operational data for extending the scope of fault localization techniques. The goal of this dissertation is to explore new automated debugging solutions that can help developers improve their quality assurance and reliability practices, extending the reach of fault localization in modern software development. The continuous integration process provides new opportunities for debugging and opens doors for utilizing continuous integration data in fault localization. This, in turn, can reduce debugging time and minimize the impact of bugs on users.

1.1 Research Statement

While fault localization has proven to be a popular automated debugging solution, its effectiveness and adaptation to the modern software context are crucial research directions for it to be widely adopted in practice. We believe that by identifying operational data and developing techniques to leverage them, developers can perform debugging tasks more effectively, and we can gain a better understanding of the root causes of faults. Therefore, we propose the following:

> By mining software repositories and performing program analysis, we can improve the effectiveness of existing fault localization techniques and propose new techniques that are readily available for use in modern software development.

In this dissertation, we consider user-reported logs and fine-grained code change information as potential debugging information that can enhance the performance of fault localization. Moreover, we recognize them as valuable resources to help improve software quality assurance and reliability practices for future studies.

1.2 Research Overview

Overall, this dissertation presents several techniques aimed at broadening the applicability and ease-of-use of fault localization in modern software development. This dissertation is divided into three main parts, each providing a perspective on the potential of fault localization in modern software development: the utilization of user-reported logs, the application of fault localization in continuous integration, and the refinement of fault localization data. In particular, Chapter 3 and 4 propose utilizing user-reported logs as new debugging knowledge in fault localization. Chapter 5 and 6 discuss the extension of fault localization in continuous integration, with the use of fine-grained code and coverage changes information. Chapter 7 covers fault localization data cleanness. Together, these parts aim to extend the reach of fault localization. The rest of this introduction describes fault localization techniques, and the key research challenges addressed in each of the subsequent chapters of this dissertation.

1.3 Fault Localization

Fault localization techniques are typically carried out in two stages. The initial stage involves defining and reducing the search space. Since the entire code base is extensive and contains thousands of files, this step aims to identify the code locations that are more likely to be faulty within the program. The second stage focuses on ranking the code locations based on their likelihood of containing faults. Prioritization is given to the most suspicious code locations, allowing developers to examine them first. Typically, fault localization techniques provide developers with a ranked list of suspicious code locations, assisting them in locating the faults and comprehending the root causes. Code locations that rank higher on the list are considered more suspicious and required closer examination. A broad family of fault localization techniques [14, 54, 71, 116, 125, 128, 177, 193, 206] has been proposed and developed, each trying to attack the problem from a unique perspective in terms of the input data, root cause, granularity and accuracy of the localization. For instance, *Information Retrieval-based (IR-based) Fault Localization* [54, 128, 152, 157, 161, 171, 180, 206] utilizes information retrieval techniques to leverage the textual information from a bug report as a query, and generate a ranked list of source code files based on their textual similarity to the query. This ranked list helps prioritize the files that are likely to contain the fault. In Section 2.3, we further discuss the main steps of IR-based techniques. Another popular family of fault localization is *Spectrum-based Fault Localization (SBFL)* [15, 16, 17, 77, 133, 182]. The base intuition of SBFL is that code entities covered by a larger number of failing tests but fewer passing tests are more likely to be associated with faults. In Section 2.4, we present SBFL in detail.

1.4 Challenge: Lack of Effectiveness

Effective fault localization plays a crucial role in reducing debugging time and assisting developers in bug fixing. It is also essential for other research directions, such as automated program repair (APR) [63, 97, 99, 114, 146, 190]. APR techniques rely on the effectiveness of fault localization to identify potential faulty locations for patching. Researchers have explored various approaches to enhance effectiveness of fault localization by incorporating additional information, such as analyzing similar bug reports from the past [206], leveraging software development history [161, 175], and utilizing structured information present in bug reports [171].

However, in practice, fault localization techniques still suffer from limited effectiveness [69, 177, 178, 185, 188]. For example, as modern software becomes more complex, fault localization may not directly pinpoint the root causes of faults [177, 178, 185]. Moreover, when attempting to isolate faults at a finer granularity, such as the statement level, fault localization techniques can encounter the issue of tie [69, 188]. This occurs when statements within the same block are equally suspicious due to shared coverage, code history, or similar textual information.

A recent study [86] has revealed that automated fault localization techniques are greatly appreciated by developers in the industry. The study indicated that over 97% of developers consider research on fault localization to be either "Essential" or "Worthwhile". However, despite the progress made in fault localization techniques, practitioners have high expectations for their adoption in real-world scenarios. They expect these techniques to identify faults among the top 5 positions and achieve a minimum accuracy of 75%. Unfortunately, the effectiveness of fault localization is currently compromised in practice. Chapter 3 and 4 address this research challenge. Chapter 3 studies the challenges and benefits of analyzing user-reported logs in bug reports for assisting in debugging. Chapter 4 proposes an information retrieval-based fault localization by re-constructing execution paths using logs, which can help to improve the effectiveness of fault localization.

1.5 Challenge: Adaptation to Modern Software Development Practices

The lack of research on fault localization techniques in modern software development is another crucial factor hindering their wide adoption. Our research is motivated by two key observations. First, in modern software development, users typically submit bug reports to notify developers about bugs [28]. However, by the time the bugs are reported, they have already impacted the users, making it too late to resolve them in production. To address this, CI practices enable the daily execution of functional tests to detect faults before they reach the users [126, 158]. CI also offers valuable information in the form of code coverage and test results, which can aid in the debugging process. Second, most prior studies [17, 78, 95, 150, 183] have evaluated these techniques in traditional software development settings, where only a single snapshot of the system is considered. However, in the context of modern software development, especially in continuous integration, developers make continuous and finer-grained changes to the system. This means that when a new test failure occurs, the fine-grained information associated with these changes can provide valuable insights for locating the fault [25, 148]. Additionally, the atomic nature of code changes in modern development limits unintended consequences, making fault isolation feasible with more accessible and less costly diagnosis metrics. Therefore, Chapter 5, 6 investigate fault localization in continuous integration to assist in adapting fault localization techniques to modern software development practices. Particularly, Chapter 5 provides a large-scale longitudinal study on continuous test execution and failure, aiming to gain a better understanding of the operational information in modern software development setting (i.e., continuous integration). This chapter also introduces a new benchmark for fault localization in continuous integration. Chapter 6 proposes changed-based fault localization techniques specialized for continuous integration settings. Finally, Chapter 7 aims to bridge the gap between fault localization research and its practical application in modern software development by focusing on data cleanness.

1.6 Contributions

In this dissertation, we study how leveraging user-reported logs can enhance debugging information and contribute to improved fault localization techniques in modern software development. Additionally, we explore the utilization of fine-grained change information, such as code changes and coverage changes, to extend the scope of fault localization. These types of information are readily available and automatically executed in modern development pipelines. Our contributions are as follows:

1. We show that even without any advanced techniques, the user-reported logs may provide a good indication of the fixed classes. To assist future research in analyzing logs, we propose to leverage the execution paths that are re-constructed from these logs to provide additional debugging supports.

2. We propose Pathidea, an IRFL approach that leverages logs in bug reports to reconstruct execution paths. Pathidea integrates the execution paths information to further improve the performance of fault localization.

3. We present a dataset, T-Evos, which contains fine-grained CI test execution information collected on a commit-by-commit basis. We also conduct an empirical study on the collected test execution data. Finally, we provide some possible research directions that may be done using T-Evos.

4. We present finer-grained change information, code and coverage changes as a new direction to improve fault localization. Both change information are less costly to obtain and may be readily available for systems following CI practices.

5. We present a case study that examines the developer's knowledge in the fault-triggering tests, which contributes to a significant overestimation of the performance of fault localization techniques.

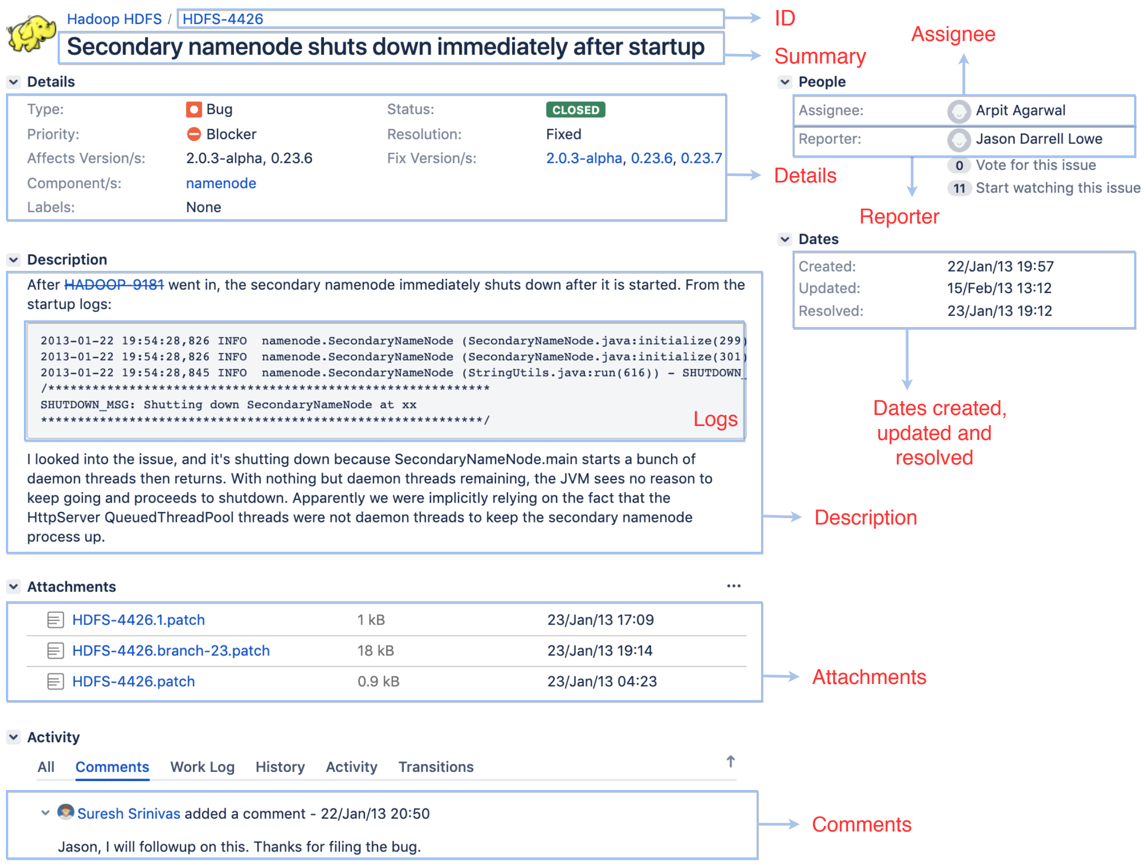
Chapter 2

Background and Related Works

In this section, we discuss the various debugging information related to the work of this dissertation. Then, we provide additional background on popular families of fault localization, *Information Retrieval-based Fault Localization (IRFL)* and *Spectrum-based Fault Localization (SBFL)*. We also discuss the commonly used metrics for evaluating the effectiveness of fault localization (i.e., Top-N, Mean Average Precision, and Mean Reciprocal Rank).

2.1 Bug Reports

Bug reports contain information to help developers diagnose reported bugs. A prior study [27] points out that from the developers' perspective, a good bug report should have a clear description and other important debugging information, such as logs. On bug tracking systems such as Jira, bug reports typically contain the following fields: Summary, Status, Details (including Type, Status, Priority, Resolution, Affects Versions, and Fix Versions), Assignee, Reporter, Description, Attachments, and Comments. Figure 1 shows an example of a bug report from the Hadoop Common system. The Summary section gives an overview of the bug. The Description section provides an explanation to the bug, and may contain the logs for debugging hints and some user-specific runtime information (e.g, describe the specific use case or hardware environment). The Status field provides the status of the bug report in the workflow. The Resolution field indicates the final resolution assigned to the reported bug (e.g., FIXED, DUPLICATE, WON'T FIX). The Affects Versions field is usually provided



Hadoop HDFS / HDFS-4426

Secondary namenode shuts down immediately after startup

Details

Type:	Bug	Status:	CLOSED
Priority:	Blocker	Resolution:	Fixed
Affects Version/s:	2.0.3-alpha, 0.23.6	Fix Version/s:	2.0.3-alpha, 0.23.6, 0.23.7
Component/s:	namenode		
Labels:	None		

People

Assignee:	Arpit Agarwal
Reporter:	Jason Darrell Lowe

0 Vote for this issue
11 Start watching this issue

Dates

Created:	22/Jan/13 19:57
Updated:	15/Feb/13 13:12
Resolved:	23/Jan/13 19:12

Description

After ~~HADOOP-9181~~ went in, the secondary namenode immediately shuts down after it is started. From the startup logs:

```
2013-01-22 19:54:28,826 INFO  namenode.SecondaryNameNode (SecondaryNameNode.java:initialize(299)
2013-01-22 19:54:28,826 INFO  namenode.SecondaryNameNode (SecondaryNameNode.java:initialize(301)
2013-01-22 19:54:28,845 INFO  namenode.SecondaryNameNode (StringUtils.java:run(616)) - SHUTDOWN_
/************************************************************
SHUTDOWN_MSG: Shutting down SecondaryNameNode at xx
************************************************************/
```

I looked into the issue, and it's shutting down because SecondaryNameNode.main starts a bunch of daemon threads then returns. With nothing but daemon threads remaining, the JVM sees no reason to keep going and proceeds to shutdown. Apparently we were implicitly relying on the fact that the HttpServer QueuedThreadPool threads were not daemon threads to keep the secondary namenode process up.

Attachments

HDFS-4426.1.patch	1 kB	23/Jan/13 17:09
HDFS-4426.branch-23.patch	18 kB	23/Jan/13 19:14
HDFS-4426.patch	0.9 kB	23/Jan/13 04:23

Activity

All Comments Work Log History Activity Transitions

Suresh Srinivas added a comment - 22/Jan/13 20:50

Jason, I will followup on this. Thanks for filing the bug.

Figure 1: An example bug report **(HADOOP-4426)** on Jira.

```
2009-02-12 08:35:36,417 INFO org.apache.hadoop.mapred.TaskTracker: Task
    attempt_200902120746_0297_r_000033_0 is in COMMIT_PENDING
2009-02-12 08:35:36,417 INFO org.apache.hadoop.mapred.TaskTracker:
    attempt_200902120746_0297_r_000033_0 0.33333334% reduce > sort
```

Figure 2: An example of log snippets. (**HADOOP-5233**)

```
17/07/14 13:31:58 INFO hdfs.DFSClient: Exception in createBlockOutputStream java.
io.EOFException:
    at org.apache.hadoop.hdfs.protocolPB.PBHelper.vintPrefixed(PBHelper.java:2280)
    at org.apache.hadoop.hdfs.DFSOutputStream$DataStreamer.createBlockOutputStream
    (DFSOutputStream.java:1318)
  ...
```

Figure 3: An example of exception logs. (**HDFS-8475**)

by the reporter, whereas the Fix Versions field is added by the assignee after bug fixes. Sometimes, either the reporter or the assignee might attach patches or tests in the Attachments section. In the Comments section, developers may further discuss the bug, provide opinions, and potentially ask for additional technical details. To note that the same idea applies for bug reports on Bugzilla, although they do not contain the Fix Versions field.

2.2 User-reported Logs in Bug Reports

To assist developers to diagnose and fix bugs, reporters may attach logs in their bug reports. Typically, there are two types of logs in bug reports: **log snippets**, which record software system execution at run time; and **exception logs**, which record the stack traces when an exception happens. Figure 2 and Figure 3 show an example of log snippets and exception logs, respectively. A log snippet is an ordered set of log messages generated by logging statements during runtime. Each log is often composed of the timestamp, verbosity level (e.g., debug, info, error, or fatal), class name, and detailed log message. An exception log contains information on multiple sets of stack frames (i.e., stack trace) when an exception happens. Exception logs are recorded together with log snippets to provide a more detailed view of the system execution when an exception happens [60]. Exception logs often contain the timestamps, thrown exceptions (e.g., NullPointerException), and the fully-qualified file names, method signatures, and line numbers for the method calls on the stack frames.

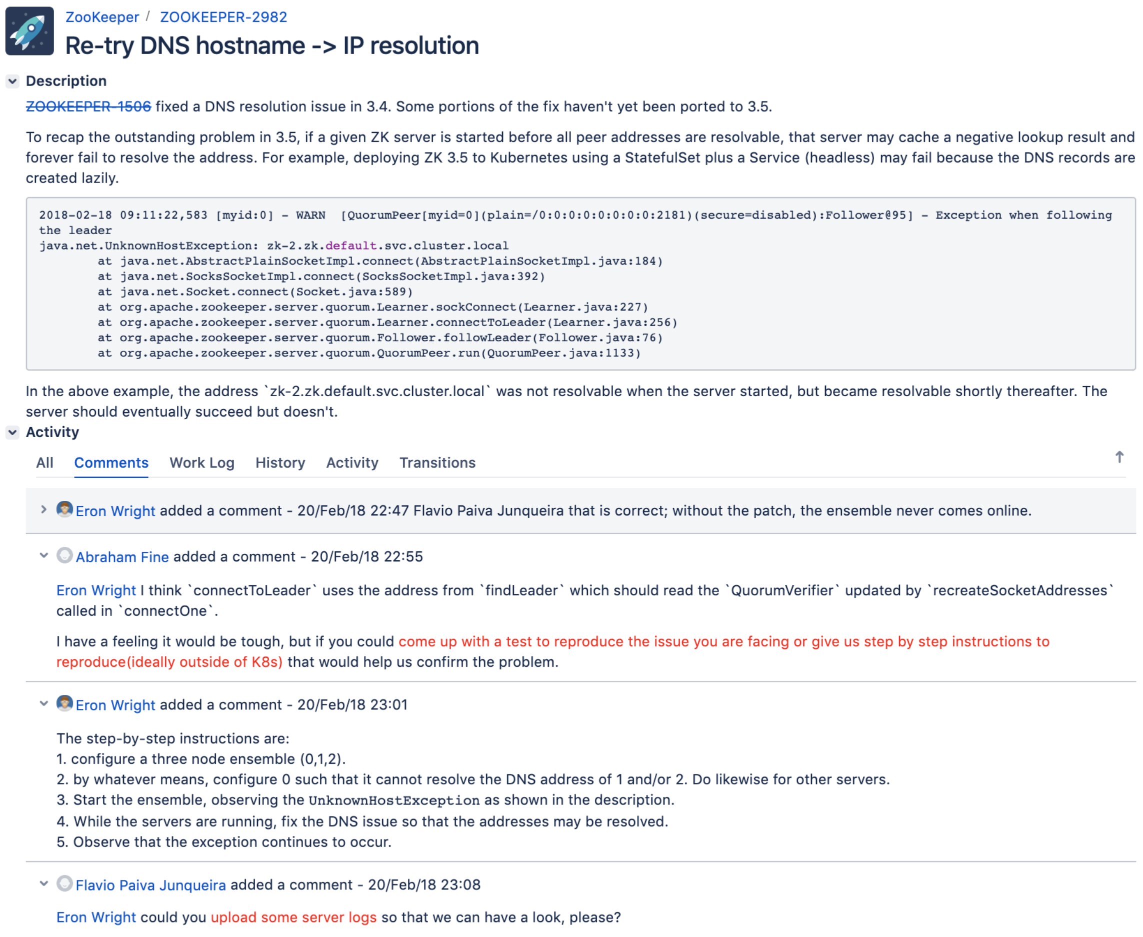
In bug reports, logs may be attached in the Description and Comments sections. Reporters often open a bug report and report the failing stack traces or log snippets in the Description to assist developers in bug fixing. Occasionally, developers may discuss the bug report and request more logs from the reporter in the Comments section. Figure 4 shows such an example, where the developer first asked for a step-by-step instruction to reproduce the bug, then demanded server logs.

2.3 Information Retrieval-based Fault Localization (IRFL)

Information Retrieval-based Fault Localization, or IRFL, is a localization technique that leverage bug reports to rank the source code files based on their likelihood to contain faults. This technique assists developers in identifying the faulty files, and provides a better understanding of the root causes of the fault. In IRFL techniques, locating bugs is transformed into a search problem, where each bug report represent a search query, with the source code files as the document collection to search from. Prior studies [53, 54, 117, 119, 128, 147, 152, 157, 161, 171, 180, 184, 206] present a number of variations of IRFL techniques. For context of the implementation details, we discuss how IRFL techniques typically locates faults. The main steps include: (1) how bug reports are preprocessed, (2) how source code files are pre-processed, (3) how suspicious files are prioritized, and (4) how the suspicious files are evaluated.

2.3.1 Preprocessing Bug Reports

The preprocessing step involves extracting the textual contents from the bug report (e.g., description and summary) and standardizing words in bug reports with that of source code files. This extracted information is then utilized to narrow down the search space of source code files that are directly associated with the given bug report. The preprocessing typically include: tokenization, text normalization, stopword removal, and stemming. We first split the sentences from bug reports into words (tokens). Then, we normalize special identifiers into constituent words such as breaking the camel cased word `getName` into tokens `get` and `name`. We remove English stopwords, programming language specific keywords, special symbols and punctuations



ZooKeeper / ZOOKEEPER-2982

Re-try DNS hostname -> IP resolution

Description

~~ZOOKEEPER-1506~~ fixed a DNS resolution issue in 3.4. Some portions of the fix haven't yet been ported to 3.5.

To recap the outstanding problem in 3.5, if a given ZK server is started before all peer addresses are resolvable, that server may cache a negative lookup result and forever fail to resolve the address. For example, deploying ZK 3.5 to Kubernetes using a StatefulSet plus a Service (headless) may fail because the DNS records are created lazily.

```
2018-02-18 09:11:22,583 [myid:0] - WARN  [QuorumPeer[myid=0](plain=/0:0:0:0:0:0:0:0:2181)(secure=disabled):Follower@95] - Exception when following
the leader
java.net.UnknownHostException: zk-2.zk.default.svc.cluster.local
        at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:184)
        at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
        at java.net.Socket.connect(Socket.java:589)
        at org.apache.zookeeper.server.quorum.Learner.sockConnect(Learner.java:227)
        at org.apache.zookeeper.server.quorum.Learner.connectToLeader(Learner.java:256)
        at org.apache.zookeeper.server.quorum.Follower.followLeader(Follower.java:76)
        at org.apache.zookeeper.server.quorum.QuorumPeer.run(QuorumPeer.java:1133)
```

In the above example, the address `zk-2.zk.default.svc.cluster.local` was not resolvable when the server started, but became resolvable shortly thereafter. The server should eventually succeed but doesn't.

Activity

All Comments Work Log History Activity Transitions

Eron Wright added a comment - 20/Feb/18 22:47 Flavio Paiva Junqueira that is correct; without the patch, the ensemble never comes online.

Abraham Fine added a comment - 20/Feb/18 22:55

Eron Wright I think `connectToLeader` uses the address from `findLeader` which should read the `QuorumVerifier` updated by `recreateSocketAddresses` called in `connectOne`.

I have a feeling it would be tough, but if you could come up with a test to reproduce the issue you are facing or give us step by step instructions to reproduce(ideally outside of K8s) that would help us confirm the problem.

Eron Wright added a comment - 20/Feb/18 23:01

The step-by-step instructions are:
1. configure a three node ensemble (0,1,2).
2. by whatever means, configure 0 such that it cannot resolve the DNS address of 1 and/or 2. Do likewise for other servers.
3. Start the ensemble, observing the `UnknownHostException` as shown in the description.
4. While the servers are running, fix the DNS issue so that the addresses may be resolved.
5. Observe that the exception continues to occur.

Flavio Paiva Junqueira added a comment - 20/Feb/18 23:08

Eron Wright could you upload some server logs so that we can have a look, please?

Figure 4: An example of a bug report **(ZOOKEEPER-2982)** highlighting the discussions between the reporter and developer (assignee). The bug report addresses a server problem when resolving for the host address on Zookeeper clusters. In the Comments section, the developer asked the reporter to provide some server logs to help the bug fix (higlighted in red).

to reduce the noise of tokens that are irrelevant to the faults. Finally, we perform stemming to reduce each word to its root form. For instance, the tokens recorded and recording are simplified to record. This increases the probability of matching shared words between the bug report and source code files.

In addition to analyzing the textual contents, several IRFL techniques [95, 152, 157, 171, 173, 180, 184, 206] have been proposed to incorporate additional information from bug reports to enhance the effectiveness of fault localization. Zhou et al. [206] suggested considering similar bug reports that have been previously resolved to identify the files that require bug fixes. Saha et al. [152] considered the structure of bug reports and employed structured retrieval techniques to model the source code structure. Le et al. [95] developed an approach that analyzes the textual descriptions in bug reports to identify suspicious words, which are then used to localize faults. Wang and Lo [171] proposed a comprehensive approach that combines version history, similar bug reports, and structural information to improve fault localization. Around the same time, Moreno et al. [128], Wong et al. [180] and Wu et al. [184] focused on analyzing stack traces provided in bug reports to pinpoint the root causes of faults.

2.3.2 Preprocessing Source Code Files

The preprocessing steps for source code involve tokenization, text normalization, stop-word removal, and stemming, similar to the preprocessing steps used for bug reports. These steps help standardize the tokens in the source code files. In terms of pre-processing source code, previous studies [90, 180] have also proposed segmenting the source code files to reduce noise in larger files, as only a small portion of a large file may be relevant to the bug report. To enhance the localization capability, other studies [117, 161] have demonstrated the usefulness of version histories in evaluating the likelihoods of source code files containing faults. Sisman and Kak [161] prioritized locations in the source code by modeling past faulty modification histories, while Loyola et al. [117] explored fault-introducing changes to improve localization performance.

2.3.3 Prioritizing Suspicious Files

A popular information retrieval technique used to prioritize the more suspicious files is the *Vector Space Model* (VSM). VSM is an algebraic model used to represent queries and documents as vectors of index terms. Documents and queries are represented by weighted vectors of n-dimensional space, where n denotes the number of unique terms in the document collection (i.e., corpus). In the context of IRFL, the bug report serves as the search query, while the entire code base of source code files is treated as the document collection. The objective is to identify the source code file that contains the highest textual similarity (e.g., measured by cosine similarity) with the given bug report.

In VSM, we first calculate the vector of weights for the query and documents. The weight of each term is often represented with the *Term Frequency-Inverse Document Frequency* (TF-IDF) of each corresponding term. *Term Frequency* (TF) refers to the frequency a specific term appears within a document. On the other hand, *Inverse Document Frequency* (IDF) focuses on the importance of a term in a document collection, which is determined by the number of documents in the corpus that contain the term. Terms with higher values of TF-IDF are considered more important. Then, we calculate the cosine similarity between query vector and document vectors. Formula 1 calculates the cosine similarity between query q and document d based on their vector representations:

$$Sim(q,d) = \frac{q \cdot d}{\|q\|\|d\|} = \frac{\sum_{i=1}^{n} q_i \times d_i}{\sqrt{\sum_{i=1}^{n}(q_i)^2} \times \sqrt{\sum_{i=1}^{n}(d_i)^2}} \tag{1}$$

where q_i represents the weighted vector of term i that appears in the query q. Similarly, d_i represents the the weighted vector of term i in document d. After calculating the similarity score for each document, the next step involves sorting the documents based on their respective similarity scores. This sorting process generates a ranked list, starting from the most suspicious document (i.e., source code file) to the least suspicious, which provides developers with a prioritized list to examine.

2.4 Spectrum-based Fault Localization (SBFL)

Spectrum-based fault localization, or SBFL, is a fault localization technique that identifies the faulty locations based on passing and failing test executions. SBFL sorts all program elements, such as statements or methods, based on a suspiciousness score. A program element with a higher suspiciousness score is considered more likely to contain faults. The goal of SBFL is to rank the faulty program elements at the top of the list, with high suspiciousness scores assigned to them. With this ranked list, developers would inspect the program elements with the highest suspiciousness scores first, saving both debugging time and effort. The computation of the suspiciousness score is based on the test execution profiles. During the SBFL process, tests are executed on the target program, and test execution profiles are collected. These profiles contain information about the execution of each program element, indicating whether the tests covering the program element (e.g., statement) pass or fail. Based on this information, the likelihood of a statement containing faults can be computed to calculate the suspiciousness score. Previous studies [15, 16, 17, 77, 133, 182] have proposed various SBFL formulas for calculating the suspiciousness score. The common intuition is that code locations covered by more failing tests are more likely to contain faults. These formulas typically use statistics such as *e_f*, *e_p*, *e_f*, and *e_p*, where *e_f*, *e_p* represent the number of failing and passing tests that execute the program element *e*, while *e_f* and *e_p* represent the numbers of failing and passing tests that do not execute the program element *e*. One of the most widely used and well-studied SBFL techniques is Ochiai [15, 16], which calculates the suspiciousness score using the formula:

$$Ochiai(e) = \frac{e_f}{\sqrt{e_f + n_f} \cdot \sqrt{e_f + e_p}} \tag{2}$$

2.5 Evaluation Metrics for Fault Localization

For evaluating the effectiveness of fault loacalization techniques, the following three metrics are widely used: Top-N, mean average precision, and mean reciprocal rank.

Top-N: Top-N tracks the number of bugs where at least one faulty source code file is discovered within the top N ranked results. N is commonly set at (1, 5, 10, and 15). This metric focus on detecting one of the faulty locations early in the ranked list for more bugs (early precision).

Mean Average Precision (MAP): MAP takes into account the ranks of all faulty files. It is calculated by computing the mean of the average precision across all bug reports. It assesses scenarios where developers need to explore the ranked list extensively to find as many relevant results as possible. The average precision (AP) is calculated as:

$$AP = \frac{\sum_{i=1}^{m} i/Pos(i)}{m} \tag{3}$$

Mean Reciprocal Rank (MRR): The reciprocal rank is computed as the reciprocal of the position at which the first faulty file is discovered. The Mean Reciprocal Rank (MRR) is then determined by taking the average reciprocal rank across all bug reports. Formula 15 presents the calculation for the mean reciprocal rank, where K represents a set of bug reports, and $rank_i$ represents the rank of the first buggy file in the i-th bug report.

$$MRR = \frac{1}{K} \sum_{i=1}^{K} \frac{1}{rank_i} \tag{4}$$

Part I

Utilizing User-reported Logs in Fault Localization

Chapter 3

An Empirical Study on the User-Reported Logs in Bug Reports

Bug reports often contain logs that offer crucial debugging information for developers. When debugging, developers rely on these logs to analyze system executions and understand the root causes of the faults. Intuitively, user-provided logs in bug reports illustrate the problems that users encounter and may help developers with the debugging process [27]. There are a number of techniques that analyze crash reports or system logs to help developers in debugging [30, 65, 75, 102, 164, 184, 195, 196]. However, these prior techniques often assume that developers have access to the entire system-generated logs or instrumented system runtime data. In practice, such information may not always be available due to privacy or technical concerns [34, 156, 160]. For instance, users usually only attach a portion of the logs in their bug reports, since the size of the entire log file is often several gigabytes or even larger [48, 160].

This chapter presents an empirical study that examines the challenges and benefits of utilizing users-reported logs in bug reports for software debugging. We begin by providing the motivation for our study, followed by the description of our case study setup. Our findings show that logs often provide a good indication of where a fault is located. However, we also find a significant number of bug reports where there was no overlap between the logged classes and the faulty classes. Upon manual investigation, we find that many logs lack system execution information, only showing the point of failure (e.g., exception) without providing direct hint on the actual root cause. Furthermore, our call graph analysis reveals potential approaches that can aid developers in debugging by performing static analysis on the system execution and establishing connections between the logs. These approaches may provide additional debugging information that are not readily available from the logs. In summary, our findings shed light on potential future research directions to improve the attachment and analysis of logs in bug reports, aiming to better assist developers in their debugging.

In the next chapter, we extend the utilization of user-reported logs to develop an information retrieval-based fault localization technique that achieves high effectiveness.

An earlier version of this chapter has been published at Empirical Software Engineering (EMSE), 2021. Pages 1-30. Chen et al. [38]

3.1 Motivation

Bug reports provide important information for developers to fix the problems that users encounter [21, 27]. Typically, when reporters create a bug report, they need to provide a title, the severity (e.g., major or minor), the description of the problem, and system-generated logs (e.g., log messages or stack traces) which illustrate the system execution paths when the problem occurs. In particular, such logs may contain valuable debugging information for developers [27, 157, 207]. Based on the user-provided information, developers then diagnose the problem and resolve the issue. In general, developers first look at the description of the bug report and manually examine the attached logs. Then, developers investigate where the logs were generated in the source code to find out where the bug might be. Finally, developers manually examine the source code and the corresponding logs, trying to understand how the system was executed when the bug happened and resolve the bug.

Intuitively, user-provided logs in bug reports illustrate the problems that users encounter and may help developers with the debugging process [27]. A number of prior studies aim to debug or reproduce bugs using system execution information [75, 164, 184, 195, 196]. However, these prior approaches often assume that developers have access to the entire system-generated logs or instrumented system runtime data. Such debugging data may not always be available to developers. In many cases, developers need to rely on data in bug reports for debugging, which may be incomplete or inaccurate [27]. Insufficient debugging information remains a challenge for developers in gaining a full understanding of the problem, resulting in frequent delays in resolving the bug report [27]. Therefore, the objective of this chapter is to investigate the potential of user-provided logs in bug reports as valuable hints for debugging. Our findings offer initial insights on leveraging readily-available information in bug reports to assist developers with debugging. Additionally, we aim to gain a deeper understanding of the challenges developers may face when analyzing user-provided logs and explore potential solutions to overcome these challenges.

Table 1: An overview of the studied systems.

System	LOC	Type	Code maturity	Selected bug history range
ActiveMQ	480k	Messaging server	> 10 years	2008-01-01 to 2019-01-19
AspectJ	447k	Aspect-oriented extension	> 10 years	2008-01-01 to 2019-01-19
Hadoop Common	364K	Common utilities	> 10 years	2008-01-01 to 2019-01-19
HDFS	560K	Distributed storage	> 10 years	2008-01-01 to 2019-01-19
MapReduce	291K	Distributed processing system	> 10 years	2008-01-01 to 2019-01-19
YARN	313K	Resource manager	> 5 years	2012-07-18 to 2019-01-19
Hive	1.7M	Data warehouse	> 5 years	2008-10-15 to 2019-01-19
PDE	369k	Tools for plug-ins development	> 10 years	2008-01-01 to 2019-01-19
Storm	346k	Distributed processing system	> 5 years	2013-12-11 to 2019-01-19
Zookeeper	144k	Configuration service	> 10 years	2008-06-10 to 2019-01-19
Total	5.0M	-	-	-

3.2 Case Study Setup

3.2.1 Studied Systems

Table 1 shows an overview of the studied systems. We conduct our case study on 10 Java-based open source systems: ActiveMQ, AspectJ, Hadoop Common, HDFS, MapReduce, YARN, Hive, PDE, Storm and ZooKeeper. The size of the studied systems ranges from 144K to 1.7M lines of code. These studied systems are widely used in prior log-related studies and have high-quality logs [42, 108, 199]. The studied systems also cover different domains, varying from virtual machine deployment systems to data warehousing solutions. Most of the systems have more than 10 years of code development. We choose these systems because they are large in scale, actively maintained, well-documented, and have many bug reports that contain logs [42, 108].

3.2.2 Collecting and Filtering Bug Reports

We collect all the bug report data that is available on the Jira repository [22] of each studied system from 2008 (or the earliest bug creation date) to January 2019, and compute the lines of code (LOC) on the master branch (data collected in January 2019). To collect the bug reports, we built a web crawler that sends REST API calls to the Jira repositories. We select the bug reports based on the criteria that are used in prior bug report studies [42, 46, 199]. Namely, we select bug reports of the type *"Bug"*, whose status are *"Closed"* or *"Resolved"*, with the resolution *"Fixed"* and priority marked as *"Major"* or above. Additionally, we only select the bug reports that have corresponding code changes in the code repository (i.e., having commit messages that contain the bug report ID), so we can verify that the bugs are indeed fixed. At the end of this process, we collected a total of 8,848 bug reports.

3.2.3 Identifying Bug Reports that Contain Logs

In this study, **we consider two types of logs: *log snippets* and *stack traces***. We refer *log snippets* as the system-generated logs and refer *stack traces* as the reported messages in stack frames (i.e., in the case of exception). These two types of logs are often the only information that is available for debugging production problems [60, 195, 197]. A log snippet is composed of consecutive log messages generated at runtime. Log messages often contain a static message (e.g., in Java, in the follow code, `Logger.info("static_message" + method())`, *static_message* is an example of a static message), values for dynamic variables, and the log verbosity level (e.g., info, warning, or error). An example log message is: *"2018-08-29 15:37:47.891 Utils [INFO] Interrupted while waiting for fencing command: cd"*, where it shows the timestamp of when the event happened, the executed class (i.e., *Utils*), the log level (i.e., *INFO*), and the log message (i.e., *Interrupted while waiting for fencing command: cd*). Note that such log messages usually contain system execution information and may not always be an indication of an error [48, 195]. The second type of logs is the system generated exception message and stack trace. Stack traces show the stack frame of the system when exceptions occur. Typically, reporters attach logs in the bug description or as comments.

Since the studied systems use specific logging conventions on the structure of the log snippets (e.g., ordered as timestamps, verbosity level, class name, and message), we use regular expressions to capture them in the Description and Comments sections of bug reports [42]. Specifically, we look for log snippets by extracting lines that contain timestamps and log-related keywords (e.g., *info*, *debug*, and *error*). We look for stack traces in a similar fashion by using both keywords (e.g., a line beginning with *"at..."*) and line formats (e.g., followed by method invocation, class name, and line number) that are specific to stack traces.

3.2.4 Collected Bug Reports

In general, we find that there is a non-negligible percentage (an average of 21.5% across all systems) of bug reports that contain logs (i.e., either log snippets, stack traces, or both). Table 2 shows the number of bug reports in the studied systems. We use *BRWL* to refer to *bug reports with logs*, and *BRNL* to represent *bug reports without logs*. In total, 1,561 (18%) bug reports contain logs and 7,287 (82%) bug reports do not contain any logs. We also observe that 6% to 47% (an average of 21.5%) of the bug reports contain at least one type of logs, which indicates that logs are often attached by reporters to help describe problems. In addition, reporters are more likely to include stack traces in a bug report compared to log snippets. Specifically, 10% (161/1,561) of BRWL have only log snippets compared to 66% (1,029/1,561) of BRWL that have only stack traces. One possible reason is that

Table 2: Bug reports in the studied systems. *BR* represents bug reports; *BRNL* represents bug reports with no logs and *BRWL* represents bug reports with logs (i.e., either contain log snippets, stack traces, or both).

System	BR with only log snippets	BR with only stack traces	BR with both	Total BRWL	Total BRNL	Total BR
ActiveMQ	10	55	27	92 (15%)	502 (85%)	594
AspectJ	0	42	3	45 (24%)	140 (76%)	185
Hadoop Common	23	71	58	152 (21%)	573 (79%)	725
HDFS	29	99	74	202 (17%)	964 (83%)	1,166
MapReduce	27	100	66	193 (34%)	382 (66%)	575
YARN	29	147	96	272 (47%)	304 (53%)	576
Hive	4	109	16	129 (6%)	2,102 (94%)	2,231
PDE	23	342	0	365 (17%)	1,763 (83%)	2,128
Storm	7	44	13	64 (17%)	316 (83%)	380
Zookeeper	9	20	18	47 (16%)	241 (84%)	288
Total	161	1,029	371	1,561 (18%)	7,287 (82%)	8,848

stack traces are more straightforward to interpret (e.g., with clear exception messages and stack traces); whereas the information in the log snippets may vary depending on how reporters attach the logs and how developers write the logging statements in the source code [108, 195, 196]. However, many bug reports still contain both log snippets and stack traces, which shows that both types of logs are commonly provided in bug reports to help debugging.

3.3 Case Study Results

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

3.3.1 RQ1: Are Bug Reports With Logs Resolved Faster Than Bug Reports Without Logs?

Motivation: Prior studies [27, 207] found that log snippets and stack traces are useful debugging information in bug reports. Presumably, and as found in prior research [27, 198, 207], bug reports that contain logs may take a shorter amount of time to resolve compared to bug reports that do not have logs. However, prior research only studies bug reports with either log snippets or stack traces but did not study the combination of both types of logs. In addition, as also shown in Section 2.2, developers may ask for more logs and may thus delay the bug resolution time. Therefore, in this RQ, we revisit whether bug reports with logs are resolved faster than bug reports without logs, and if bug reports with logs in the Comments section take more time to resolve.

Approach: We analyze the bug resolution time for the bug reports that we collected in Section 3.2. In particular, we study the bug reports that have a corresponding code change in the code repository. For each analyzed bug report, we calculate the bug resolution time (in days) by taking the difference between the bug resolution date and bug report creation date [46]. We statistically compare the bug resolution time of the bug reports with logs (BRWL) and the bug reports without logs (BRNL). We use Wilcoxon rank-sum test to study if there exists a statistically significant difference between the resolution time of BRWL and BRNL. We select Wilcoxon rank-sum test because it is a non-parametric test that does not have an assumption on the distribution of the data [127]. To further show the magnitude of the difference, we compute the effect size. We use Cliff's Delta, which is also a non-parametric test, as the effect size measurement to quantify the amount of difference between BRWL and BRNL [50]. We assess the magnitude by using the thresholds provided by Romano et al. [149]:

$$\text{effect size} \begin{cases} \text{negligible}, & \text{if } |d| < 0.147 \\ \text{small}, & \text{if } 0.147 \le |d| < 0.33 \\ \text{medium}, & \text{if } 0.33 \le |d| < 0.474 \\ \text{large}, & \text{if } 0.474 \le |d| \end{cases} \tag{5}$$

Results: ***In general, BRWL takes more time to resolve compared to***

Table 3: A comparison of the bug resolution time (in days) between the bug reports with logs (**BRWL**) and the bug reports without logs (**BRNL**) across the studied systems.

Project	**BRWL median resolution**	**BRNL median resolution**	**p-values**	**Cliff's Delta**
ActiveMQ	21.5	1.0	<0.001	**0.73 (large)**
AspectJ	14.0	25.0	0.89	0.01 (negligible)
Hadoop Common	7.0	1.0	<0.001	**0.58 (large)**
HDFS	27.5	4.0	<0.05	**0.46 (medium)**
MapReduce	23.5	1.0	0.19	0.69 (large)
YARN	10.0	2.0	0.47	0.53 (large)
Hive	7.0	3.0	0.56	0.25 (small)
PDE	3.0	6.0	<0.05	0.11 (negligible)
Storm	4.0	3.0	<0.001	**0.28 (small)**
Zookeeper	91.0	1.0	0.15	0.88 (large)
Average	20.9	4.7	-	-

BRNL. Table 3 shows the median resolution time of bug reports with logs (BRWL) and without logs (BRNL). We find that the median resolution time ranges from 3 to 91 days for BRWL, and ranges from 1 to 25 days for BRNL. Our results show that such differences are statistically significant in four out of 10 studied systems (ActiveMQ, Hadoop Common, HDFS, and Storm), where the effects range from small to large. Figure 5 further shows the beanplots that compare the density of the resolution time distribution between BRWL and BRNL. We limit the Y-axis to 15 days to better visualize the difference between the resolution time of BRWL and BRNL (most BRNL are resolved within 15 days). As illustrated in Figure 5, the distribution of the resolution time for BRNL generally has a long tail. In other words, most BRNL are resolved in a very short amount of time (within two to three days), and almost all BRNL are resolved within 15 days. BRWL, on the other hand, have more uniform distributions in the studied systems. To better illustrate this finding, Figure 6 shows the boxplots that compare the median resolution time between BRWL and BRNL in range of 15 days. BRNL are generally resolved in a shorter amount of time than that of BRWL.

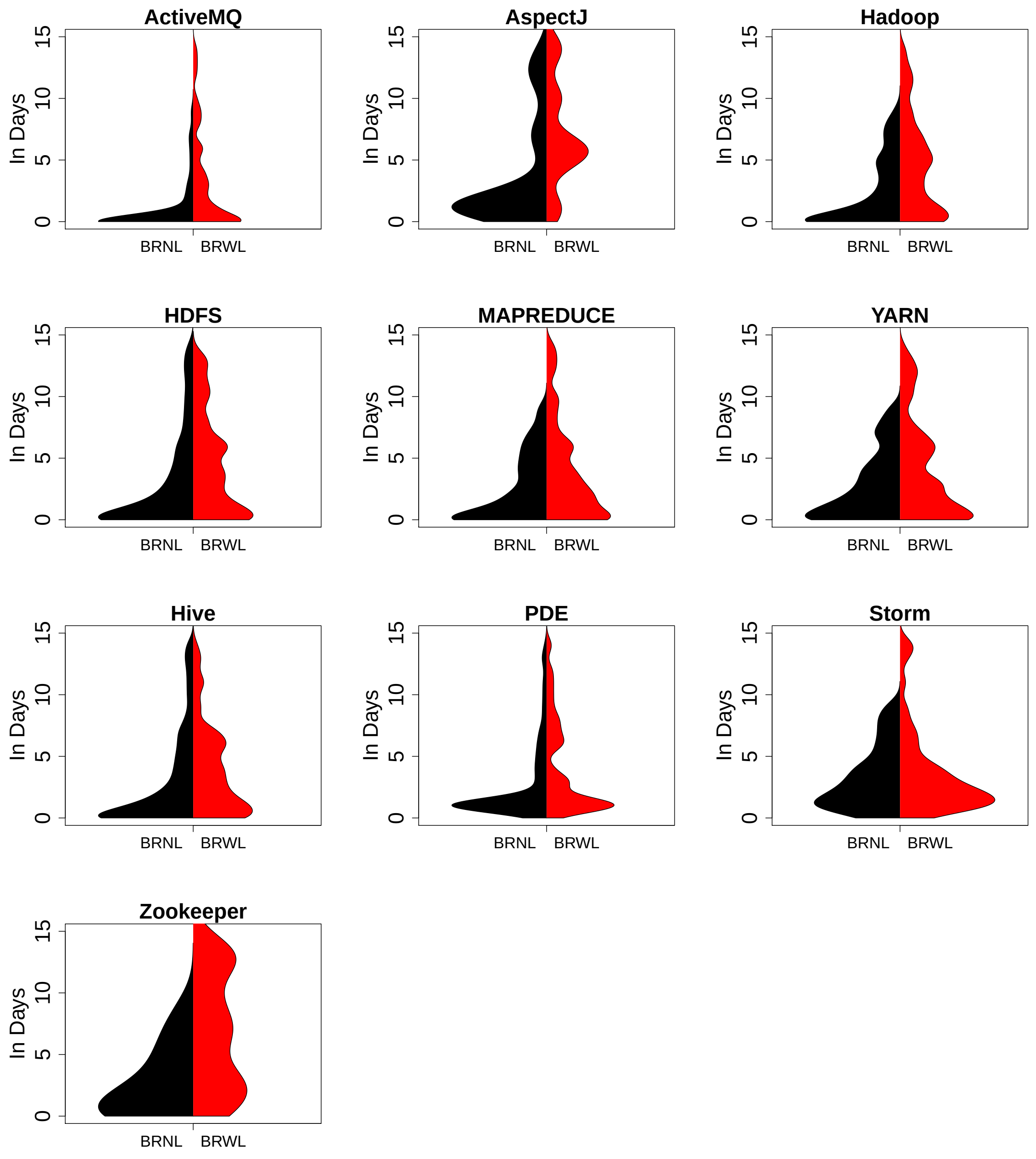


Figure 5: Beanplots to illustrate the densities of resolution time (in days) distribution for BRWL and BRNL in range of 15 days.

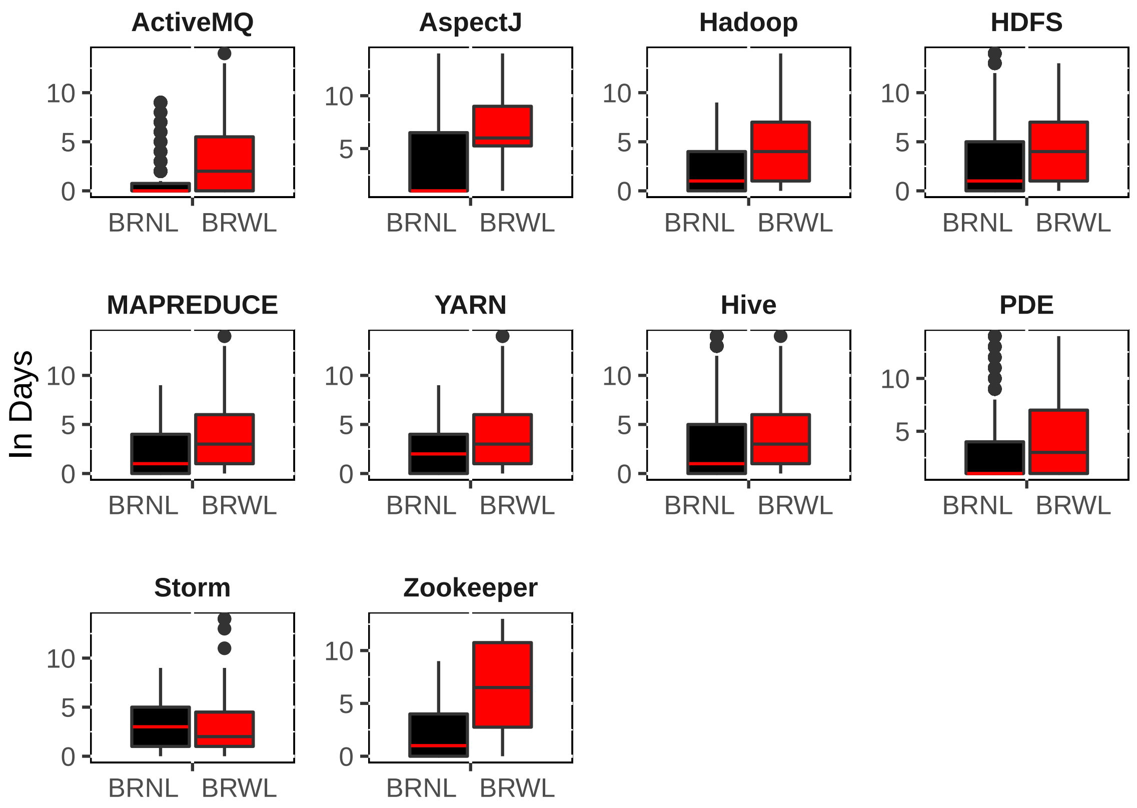


Figure 6: Boxplots to illustrate the median resolution time (in days) for BRWL and BRNL in range of 15 days.

Prior studies [27, 207] found that log snippets and stack traces are important debugging information in bug reports. However, even though such information is useful for debugging, we find that BRWL take more time to resolve compared to BRNL. Hence, we further investigate the possible factors that may increase the resolution time for BRWL. We first study where the logs are attached in bug reports. As discussed in Section 2.2, developers may request more logs in the Comments section of a bug report, which may take time for the reporter to provide and delay the bug fixing. Table 4 shows the percentage of BRWL with *logs only in the Description section* (i.e., BRWL-D) and BRWL with *logs in the Comments section* (i.e., BRWL-C, both BRWL with logs only in the Comments and BRWL with logs in both the Description and Comments), along with their respective median number of log lines and median resolution time. We find that the BRWL-C covers from 17% to 68% (an average of 43%) of BRWL. In addition, the median number of log lines in the Comments section is comparable to that of the Description section. For the median resolution time, however, BRWL-C require much more time to resolve (i.e., medians are 1.1 to 36.8 times slower) compared to that of BRWL-D. The Wilcoxon rank-sum test shows that the resolution time from BRWL-D is statistically significantly different from the BRWL-C ($p < 0.001$). We use Cliff's Delta to assess the magnitude of this difference, which results to a small effect size (i.e., $|d|$ is 0.31). We further examine the Spearman rank correlation between the number of log lines in the Comments section and the resolution time. Although the correlation is not strong, we find that there are some correlations between the bug resolution time and the number of log lines in the Comments section (0.20 across all studied systems). Our finding shows that it is common for developers to ask for more logs to diagnose a bug, and having more logs in the Comments section may increase bug resolution time. In other words, the initial-attached logs may be insufficient for debugging. Figure 4 illustrates an example of such cases. The bug report ZOOKEEPER-2982 highlights an Internet Protocol address (IP) resolution bug in the ZooKeeper server. Although the reporter initially added some stack traces in the bug description illustrating the root cause, he was later asked by the developer to provide the steps to reproduce the bug and some server logs to help the bug fix.

Different from other studied systems, our finding shows that, in Eclipse PDE and AspectJ, the bug reports with logs are resolved faster than the ones without. The

Table 4: A comparison of the number of log lines and median resolution time between BRWL that have logs only in Description (BRWL-D) and BRWL that have logs in Comments (i.e., BRWL-C, BRWL with logs only in Comments and BRWL with logs in both Description and Comments).

Project	**# of BRWL-C**	**Median # of log lines**	**Median resolution time (dates)**	**# of BRWL-D**	**Median # of log lines**	**Median resolution time (dates)**
ActiveMQ	40 (43%)	28	221	52 (57%)	21	6
AspectJ	15 (33%)	6	84	30 (67%)	6	11
Hadoop Common	72 (47%)	12	8	80 (53%)	13	7
HDFS	116 (57%)	22	37	86 (43%)	13	21
MapReduce	111 (58%)	18	36	82 (42%)	17	19
YARN	132 (49%)	18	13	140 (51%)	14	6
Hive	41 (32%)	22	18	88 (68%)	29	5
PDE	83 (23%)	9	27	282 (77%)	11	1
Storm	11 (17%)	27	14	53 (83%)	17	4
Zookeeper	32 (68%)	16	116	15 (32%)	25	46
	total: 653 (42%)	**avg:** 18	**avg:** 57	**total:** 908 (58%)	**avg:** 17	**avg:** 13

median resolution time for BRWL and BRNL are 3 and 6 days for PDE, respectively, and the difference is statistically significant (p-value < 0.05) with a negligible effect size. The median resolution time for BRWL and BRNL are 14 and 25 days for AspectJ, respectively, and the difference is not statistically significant (p-value $=$ 0.89). After some investigation, we find that, compared to other studied systems, Eclipse PDE and AspectJ have the least percentage of BRWL-C. As shown in Table 4, BRWL-C take more time to resolve. For PDE and AspectJ, there are only 23% and 33% of the bug reports that have logs in the Comments section, respectively.

Another factor that associates with the bug resolution time is the complexity of bug fixes. We further compare the complexity of the bug fixes between BRWL and BRNL. For each bug report, we compute the number of changed lines of code (i.e., the total number of additions and deletions). In general, we find that the median number of changed lines of code is 51 for BRWL and 30 for BRNL. We also calculate the non-parametric Wilcoxon rank-sum test to compare the number of changed lines between BRWL and BRNL. The Wilcoxon rank-sum test shows that BRWL is statistically significant different from BRNL in terms of changed lines ($p < 0.001$). To assess the magnitude of this difference, we use Cliff's Delta. The difference between the number of changed lines of code for BRWL and BRNL is negligible (i.e., $|d|$ is 0.12). In short, we find that the bug fixes for BRWL are larger than BRNL, which may be positively correlated with the longer fixing time of BRWL.

We find that BRWL takes more time to resolve (median ranges from 3 to 91 days) compared to BRNL (median ranges from 1 to 25 days). Our further investigation shows that the initially-attached logs may not be sufficient for debugging (i.e., developers often ask for more logs in the Comments section of a bug report), and the bug fixing size of BRWL is, in general, larger than BRNL (median is 51 vs 30 lines of code).

3.3.2 RQ2: Are There Overlaps Between Logged Classes and Fixed Classes?

Motivation. Logs illustrate important system run-time information. When debugging user-reported bugs, logs (i.e., either log snippets, stack traces, or both) are usually the only source of information that is available to developers [60, 195, 197]. Developers need to manually analyze the logs to diagnose the problem. Hence, if the attached logs are unclear or insufficient, debugging can become even more time consuming and challenging [93, 195, 198]. Even though prior studies have leveraged logs to assist bug localization [128, 171, 180], it is still not clear about the direct effects of the logs and their possible limitations. In this RQ, we study the overlap between the logged classes (i.e., classes that generated the logs) and the fixed classes (i.e., classes where developers applied bug fixes). Our findings provide the empirical evidence on the importance and usefulness of providing additional tools and information to help developers in analyzing user-provided logs in bug reports.

Approach. Our goal is to study if there exist overlaps between the logged classes and the fixed classes (i.e., whether or not at least one of the fixed classes is the same as the classes that generated the user-reported logs). Our first step is to extract the logged classes from bug reports. As mentioned in Section 3.2.3, we capture the logs using regular expression. Specifically, we look for log snippets by extracting log lines that contain timestamps (e.g., `17/07/14 13:31:58`, verbosity level (e.g., `INFO`), and fully-qualified class name (e.g., `org.apache.hadoop.mapred.TaskTracker`). We highlight stack traces in a similar fashion by using the `at` keyword, followed by a fully-qualified class name, method invocation, and line number. At the end of the first step, we get

Table 5: An overview of the bug reports with fixed classes overlapping with the logged classes. The average numbers are computed based on each bug report. The percentage of fixed classes located in logs is the ratio of the fixed classes in logs to the total fixed classes in bug report (# fixed classes in logs / # total fixed classes).

Project	**# of BR with fixed classes located in logs**	**Avg. # of fixed classes per BR**	**Avg. # of logged classes per BR**	**% of fixed classes located in logs**
ActiveMQ	25 (58%)	2.3	15.3	41.6%
AspectJ	23 (65%)	2.0	5.5	33.7%
Hadoop Common	87 (65%)	2.4	6.8	50.0%
HDFS	119 (71%)	2.8	16.1	48.2%
MapReduce	108 (70%)	2.2	8.6	49.7%
YARN	192 (79%)	3.3	11.2	51.0%
Hive	91 (75%)	2.9	12.6	51.6%
PDE	291 (81%)	4.5	14.4	24.5%
Storm	30 (55%)	2.0	7.6	38.7%
Zookeeper	29 (63%)	2.2	6.7	46.2%
	total: 995 (73%)	**avg:** 2.7	**avg:** 10.5	**avg:** 43.5%

a list of fully-qualified class names covered in logs.

The next step is to extract the list of fixed classes for each bug report. We follow prior studies [85, 209] by linking the bug reports to the associated bug fixing commits using bug IDs. In the studied systems, developers are required to record the bug IDs in commit messages. Therefore, we use the `git log | grep BUG_ID[^\d]` command to find the corresponding bug fixing commits of a bug. Once we get these commits, we find the list of fixed Java files and compute for their fully-qualified class name from the `package` declaration statement (e.g., `package org.apache.hadoop.mapred.TaskTracker`). Finally, we compared the fixed classes that overlap with the logged classes. To note that both the logged classes and fixed classes are collected at outer class-level. To further refine our analysis, we exclude 191 bug reports that did not modify any existing Java classes. We then conduct a manual study on these bug fixes to examine the reason.

Results. ***Classes covered in user-reported logs provide a good indication of where the bug may be located.*** Table 5 shows the overview of the bug reports where the fixed classes have an overlap with the logged classes. We find that 88%

Table 6: A comparison of the median resolution time for the bug reports with fixed classes located in logs and the ones without.

Project	**# of BR with fixed classes located in logs**	**Median resolution time (days)**	**# of BR with no fixed classes located in logs**	**Median resolution time (days)**
ActiveMQ	25 (58%)	**14**	18 (42%)	**57**
AspectJ	23 (65%)	16	22 (35%)	13
Hadoop Common	87 (65%)	7	47 (35%)	6
HDFS	119 (71%)	**18**	48 (29%)	**26**
MapReduce	108 (70%)	**11**	46 (30%)	**26**
YARN	192 (79%)	10	52 (21%)	7
Hive	91 (75%)	7	30 (25%)	6
PDE	291 (81%)	**3**	70 (19%)	**6**
Storm	30 (55%)	**4**	25 (45%)	**25**
Zookeeper	29 (63%)	**60**	17 (37%)	**117**
	total: 995 (73%)	**avg:** 15	**total:** 375 (27%)	**avg:** 29

(1,370/1,561) bug reports modified existing Java classes when fixing bugs. We further study the remaining 191 bug reports that did not modify any existing Java class later in this RQ. There are 73% (995/1,370) bug reports that have an overlap between the fixed classes and the logged classes. In other words, to a large extent, logs provide direct information for developers to diagnose and fix a bug. In addition, Table 5 shows the number of classes covered in user-reported logs. We find that the user-reported logs often cover 5.5 to 16.1 unique classes and these logged classes have an overlap with 24.5% to 51.6% of the fixed classes. Given the fact that, on average, fixing a bug report requires only modifying 2 to 4.5 classes in the studied systems. Our finding shows that even without any advanced techniques, the user-reported logs may provide a good indication of the fixed classes. Furthermore, on some systems, the median resolution time is drastically reduced for bug reports that have class overlap. Table 6 shows the median resolution time for bug reports with class overlap and the ones without. For bug reports with class overlap, the resolution time can be reduced up to 6.3 times. However, as we also find, not all fixed classes are found in logged classes. Further improvement can be done to better assist developers. For instance,

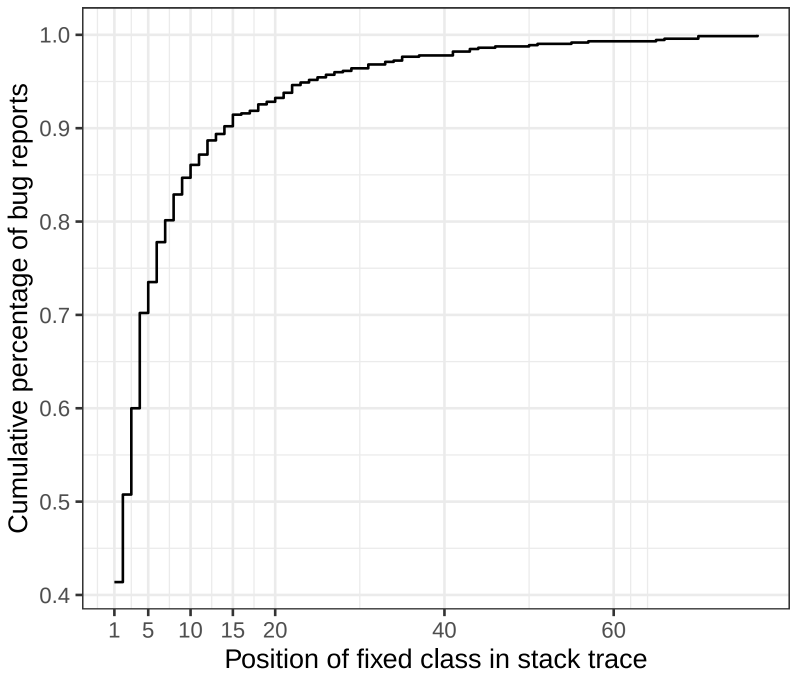


Figure 7: Cumulative percentage of bug reports for the position of fixed class in stack trace.

future research can develop tools to reconstruct the execution path based on the user-reported logs to assist developers with bug fixing as we observe cases where the fixed classes are located on the execution path.

Similar to the prior study conducted by Schroter et al. [157], we further analyze the bug reports with fixed classes in stack traces (725/995) to study the position of the fixed class in the stack frames. Figure 7 shows an overview between the position of the fixed class in stack trace and the cumulative percentage of bug reports. We observe that 40% of the bug reports have the fixed class located at the first stack frame, 70% have the fixed class located within the top-5 stack frames, and more than 90% have the fixed class located within the top-15 stack frames. However, when we further analyze the relationship between the position of the fixed class and the resolution time of the bug report, the Spearman correlation is nearly zero (0.08). One potential reason is that bug reports are only marked as resolved or fixed after they have been tested, code-reviewed, and integrated into the production environment. There are many factors that can influence the resolution time (e.g., time of bug triage and replication). As the position of the fixed class is only relevant to the debugging process, its effect becomes less significant to the overall resolution time. Therefore, our finding shows that there is no clear correlation between the position of the fixed class in the stack frame and the bug resolution time.

Table 7 shows examples where there is an overlap between the logged classes and fixed classes. HADOOP-5233 (i.e., first row in Table 7) reports a bug where the reducer transits from COMMIT_PENDING to RUNNING state while it should wait for the commit response. The user-provided logs show the unexpected transition from COMMIT_PENDING state, generated by the *TaskTracer* class. The bug fix to HADOOP-5233 adds a conditional logic to ignore the progress update in the *Task-Tracker* class whenever the state changes from COMMIT_PENDING to RUNNING. Thus, the logged class *TaskTracer* overlaps with the fixed class. Other changes (i.e., the changes that occur in *JobInProgress*, *Task*, *TaskInProgress* and *TaskStatus*) make sure that the COMMIT_PENDING task entry is properly removed from the tracker. HDFS-10512 (i.e., second row in Table 7) describes a bug that triggers an unexpected NullPointerException in the *VolumeScanner* class (i.e., a volume scanner is responsible to scan block data to detect data corruptions) while reading for a volume variable through the *DataNode.reportBadBlocks* method call. The bug fix essentially added a conditional operator to verify whether the volume variable is null in the *DataNode* class. The changes to *FsDatasetImpl* and *VolumeScanner* are to adopt existing codes to the changes. In addition, a new test case is added to the *TestFsDatasetImpl* class to test the *DataNode.reportBadBlocks* method when the volume is null. The logged classes overlaps with the fixed classes *DataNode* and *VolumeScanner*.

We further manually examine the bug reports in which no existing Java classes were modified in the bug fix. We manually study a statistically representative random sample of 162 bug reports out of the 191 bug reports (with a confidence level of 95% and a confidence interval of 3%). We classify these bug reports into four categories: *non-Java code changes*, *configuration file changes*, *only added new Java classes*, and *incorrect commit*. *Non-Java code changes* (85/162) are bug fixes performed on programming source code files other than .java. Such source code files are usually system-specific. For example, in HIVE, a big majority of these bug reports changed test query files (*.q*) and test query result files (*.q.out*). *Configuration file changes* (65/162) are bug fixes that only modified configuration files, such as managing dependencies in *.xml* file for Maven projects. *Only added new Java classes* (8/162)

Table 7: Examples of direct mapping between logged classes and fixed classes. Note that we simplify these examples by only showing the class name instead of the fully-qualitified class name.

Bug Report	**Logs**	**Logged classes**	**Fixed classes**	**Overlaps**

HADOOP-5233	`...`
	`2009-02-12 08:35:36,417 INFO org.apache.hadoop.map-red.TaskTracker: Task attempt_200902120746_0297_r_000033_0 is in COMMIT_PENDING`
	`2009-02-12 08:35:36,417 INFO org.apache.hadoop.map-red.TaskTracker: attempt_200902120746_0297_r_000033_0 0.33% reduce > sort`
		TaskTracker	JobInProgress
			Task
			TaskInProgress
			TaskStatus
			TaskTracker
				TaskTracker

HDFS-10512	`...`
	`2016-04-07 20:30:53,831 ERROR org.apache.hadoop.hdfs.server.datanode.VolumeScanner: VolumeScanner (/dfs/dn, DS-89b72832-2a8c-48f3-8235-48e6c5eb5ab3) exiting because of exception java.lang.NullPointer-Exception`
	`  at org.apache.hadoop.hdfs.server.datanode.DataNo-de.reportBadBlocks(DataNode.java:1018)`
	`  at org.apache.hadoop.hdfs.server.datanode.Volume-Scanner$ScanResultHandler.handle(VolumeScanner.java:287)`
	`  at org.apache.hadoop.hdfs.server.datanode.Volume-Scanner.scanBlock(VolumeScanner.java:443)`
	`  at org.apache.hadoop.hdfs.server.datanode.Volume-Scanner.runLoop(VolumeScanner.java:547)`
	`  at org.apache.hadoop.hdfs.server.datanode.Volume-Scanner.run(VolumeScanner.java:621)`
	`...`
		DataNode
		VolumeScanner
			DataNode
			FsDatasetImpl
			TestFsDatasetImpl
			VolumeScanner
				Datanode
				VolumeScanner

are bug reports where only new Java classes were added to the studied system, and no existing Java class was modified. For such bugs, it is impossible for the logs to be mapped to a new fixed class that is yet to exist in the system. We also find that this type of bug fixes is uncommon and developers often modify other configuration files to adopt the newly added Java classes. Finally, *incorrect commit* (4/162) consists of bug reports where bug fixes were committed with the incorrect bug ID. In short, our findings show that it is common for developers to modify files that are written in different programming languages, and some bugs can actually be fixed by modifying configuration files. Future studies should consider the polyglot nature of modern software systems and the importance of configuration files in fixing bugs.

The fixes to 88% (1,370/1,561) of the BRWL included modifications to existing Java classes. We find that 73% (995/1,370) of the bug reports have overlaps between the logged classes and fixed classes. Depending on the quality of the logs, the logged classes can locate up to 51.6% (44% on average) of the fixed classes. Although the user-provided logs provide a good indication on the bug fixing locations in some situations, there is still an average of 56% of the fixed classes that have no overlap with the logged classes.

3.3.3 RQ3: Why do some fixed classes have no overlap with the logged classes?

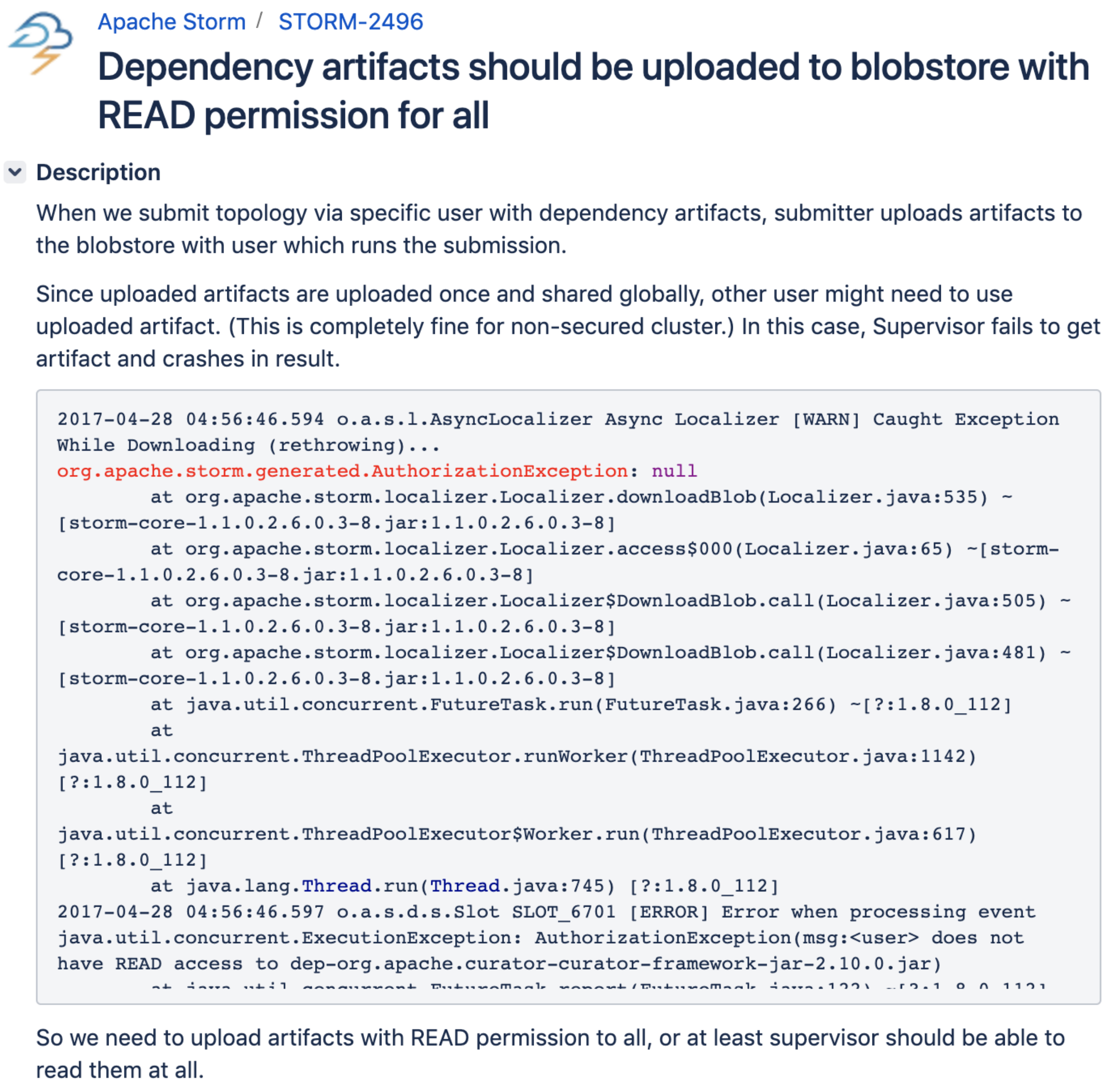
Motivation. Unlike bugs that are uncovered during development phases, many user-reported bugs are difficult to reproduce and often lack test cases [168, 197, 199]. In such cases, developers rely on logs during the debugging process [195, 196, 197]. However, as we found in RQ2, even though there is an overlap between logged classes and fixed classes, there are some bugs where the user-provided logs cannot help identify fixed classes (27%, 375/1,370) after excluding the bug reports that had no modified Java class in bug fixes. Therefore, in this RQ, we manually investigate the reasons why certain user-provided logs fail to find the fixed classes (i.e., cannot help identify any fixed classes). Our findings may provide insights on helping researchers and practitioners improve the current logging practice.

Approach. We manually study the bug reports in which the logs could not help identify fixed classes at all. From RQ2, we find that 27% (375/1,370) of bug reports have no overlaps between the logged classes and fixed classes. Hence, we then manually study 278 out of 375 such bug reports to achieve a confidence level of 95% and a confidence interval of 3% [127]. We manually studied the bug reports. In particular, we examined the bug reports, the attached logs, the bug fixes, source code classes, and the development history (e.g., prior commits) to understand the reason. We took notes while studying each bug report. At the end of the process, we uncovered a list of categories for which there was no direct mapping between logged classes and fixed classes. We then revisited and assigned each bug report to the uncovered categories. We verified the assigned categories and any discrepancy (e.g., on which category the bug report belongs to) is discussed until there is a consensus.

Results. In total, we uncovered two categories of reasons for which there was no direct mapping between the logged classes and fixed classes. Below, we discuss each category in detail.

Logs that show the failure but not the fault (i.e., the root cause) (266/278). We find that reporters in most of the 278 studied bug reports attached related logs to the bug, but the logged classes do not have an overlap with the fixed classes. In all the cases that we manually studied, the logs are reported to illustrate an unexpected behavior (i.e., the failure [68]). The majority of the cases (i.e., 202 bug reports) are related to stack traces. As stack traces are used to provide debugging information at the point of failure, the faulty classes (i.e., the cause of the bug) do not fall into the stack frames of the stack traces. Figure 8 shows an example. In STORM-2496, a reporter attached stack traces to show the failure *AuthorizationException* when users upload dependency artifacts. In this stack trace, we see the list of stack frames leading to the exception and the state of the user's access permission being `null`. However, *DependencyUploader*, the essential class that manages permissions and where the bug fix was applied, is not shown in the logs. The reporter also did not attach the logs that happened before the exception, which may show the execution path that led to the exception and help locate the root cause.

Similar to the prior study by Moreno et al. [128], we further analyze the shortest path in the call graph between the fixed classes and classes found in the logs at class level. Although some user-provided logs cannot help to identify any fixed classes, we want to investigate how far away the logged classes are from the fixed classes in the system. Thus, we further analyze the distance between the fixed classes and the



Apache Storm / STORM-2496

Dependency artifacts should be uploaded to blobstore with READ permission for all

Description

When we submit topology via specific user with dependency artifacts, submitter uploads artifacts to the blobstore with user which runs the submission.

Since uploaded artifacts are uploaded once and shared globally, other user might need to use uploaded artifact. (This is completely fine for non-secured cluster.) In this case, Supervisor fails to get artifact and crashes in result.

```
2017-04-28 04:56:46.594 o.a.s.l.AsyncLocalizer Async Localizer [WARN] Caught Exception
While Downloading (rethrowing)...
org.apache.storm.generated.AuthorizationException: null
        at org.apache.storm.localizer.Localizer.downloadBlob(Localizer.java:535) ~
[storm-core-1.1.0.2.6.0.3-8.jar:1.1.0.2.6.0.3-8]
        at org.apache.storm.localizer.Localizer.access$000(Localizer.java:65) ~[storm-
core-1.1.0.2.6.0.3-8.jar:1.1.0.2.6.0.3-8]
        at org.apache.storm.localizer.Localizer$DownloadBlob.call(Localizer.java:505) ~
[storm-core-1.1.0.2.6.0.3-8.jar:1.1.0.2.6.0.3-8]
        at org.apache.storm.localizer.Localizer$DownloadBlob.call(Localizer.java:481) ~
[storm-core-1.1.0.2.6.0.3-8.jar:1.1.0.2.6.0.3-8]
        at java.util.concurrent.FutureTask.run(FutureTask.java:266) ~[?:1.8.0_112]
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
[?:1.8.0_112]
        at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
[?:1.8.0_112]
        at java.lang.Thread.run(Thread.java:745) [?:1.8.0_112]
2017-04-28 04:56:46.597 o.a.s.d.s.Slot SLOT_6701 [ERROR] Error when processing event
java.util.concurrent.ExecutionException: AuthorizationException(msg:<user> does not
have READ access to dep-org.apache.curator-curator-framework-jar-2.10.0.jar)
        at java.util.concurrent.FutureTask.report(FutureTask.java:122) ~[?:1.8.0_112]
```

So we need to upload artifacts with READ permission to all, or at least supervisor should be able to read them at all.

Figure 8: An example bug report (**STORM-2496**) that shows the reporter attached logs to illustrate unexpected behaviors (i.e., failure). The bug fix was applied in a related class (i.e., *DependencyUploader*), but the class is not shown in the stack trace.

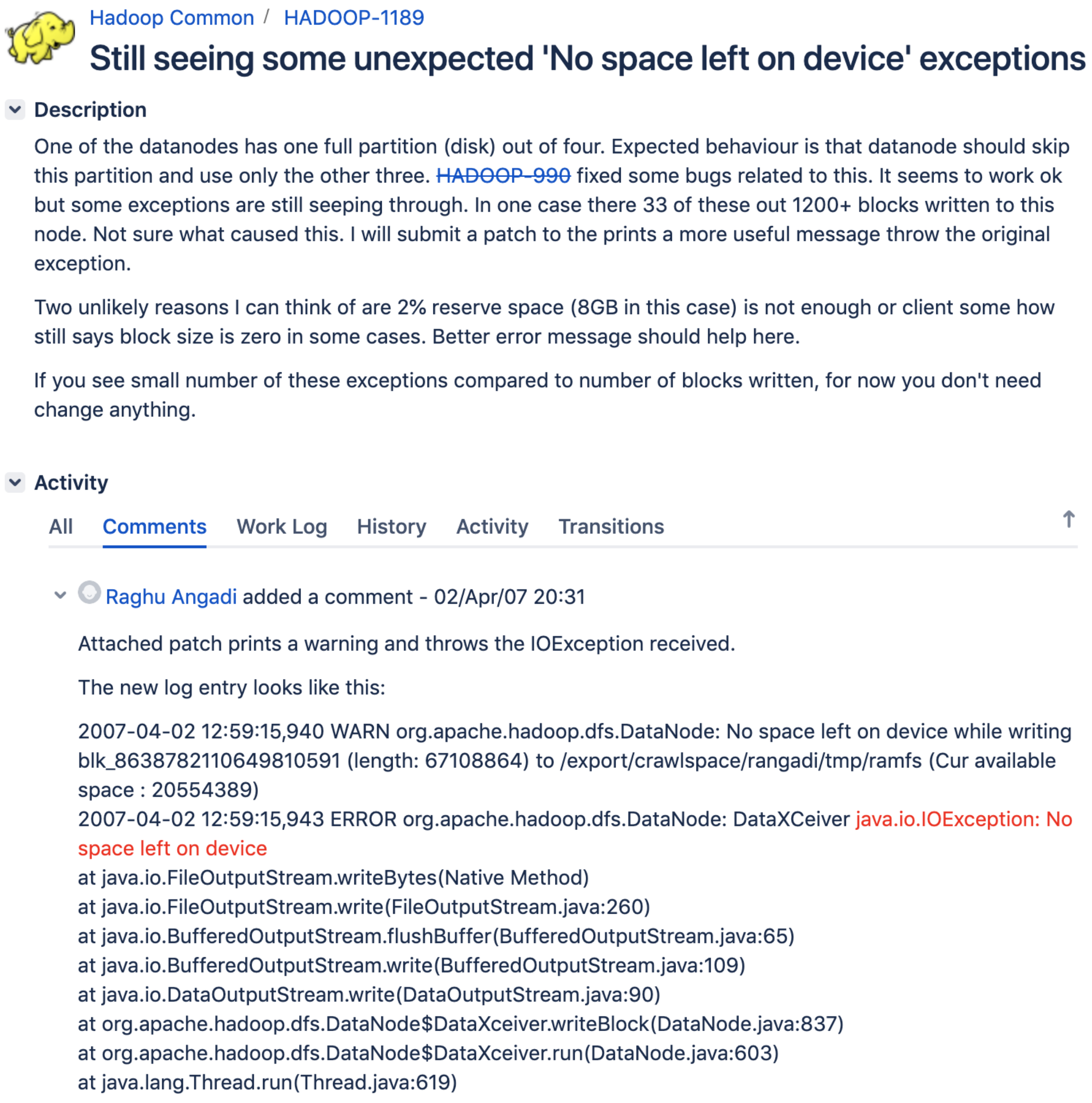
Table 8: Percentage of bug reports and fixed classes located at different distances, where the distance is calculated as the shortest path between the logs and fixed classes in terms of class invocations. When there is no path between the logs and fixed classes, the fixed classes are marked as *unreachable*.

# BR				**# fixed classes**			
total	**dist=1**	**dist>1**	**unreachable**	**total**	**dist=1**	**dist>1**	**unreachable**
266	61 (23%)	13 (5%)	192 (72%)	564	83 (15%)	19 (3%)	462 (82%)

logged classes. First, we select the commit prior to the bug fixing commit as our affected version (i.e., the bug is still unresolved). Then, we derive the system call graph on the affected version using JavaParser [1]. JavaParser is a static analysis tool that transforms the source code to Abstract Syntax Tree (AST) for Java applications. We traverse the method calls in the ASTs to uncover all the paths in the call graph. Once the paths are generated, we calculate the distance for the shortest path, if it exists, between the fixed classes and the logged classes by applying depth-first search.

Table 8 shows the percentage of bug reports and fixed classes located at different distances. For the 266 bug reports that belong to this category, 61 (23%) bug reports have fixed classes that are one distance away from the classes shown in the logs, 13 (5%) are two distance or further, and 192 (72%) bug reports have fixed classes that are unreachable from the classes in the logs. The result implies that 28% of the studied bug reports have the fixed classes that are reachable (i.e., one distance away or further in the call graph) to the classes in logs. Besides, in terms of the number of fixed classes in stack traces, our finding shows that up to 18% of the fixed classes (15% that are one distance away from the logged classes, and 3% are two distance or further) can be located in the call graph. The result shows that even for some of the bug reports which have no overlap between the logged classes and fixed classes, the execution path re-constructed from the logged classes may be used to suggest the potential bug fixing locations.

There are a few other reasons where the fixed classes cannot be located using the attached logs. Figure 9 shows an example of such bug reports. In this example, *DataNode* throws *IOException* when one of the partitions does not have enough



Still seeing some unexpected 'No space left on device' exceptions

Description

One of the datanodes has one full partition (disk) out of four. Expected behaviour is that datanode should skip this partition and use only the other three. ~~HADOOP-990~~ fixed some bugs related to this. It seems to work ok but some exceptions are still seeping through. In one case there 33 of these out 1200+ blocks written to this node. Not sure what caused this. I will submit a patch to the prints a more useful message throw the original exception.

Two unlikely reasons I can think of are 2% reserve space (8GB in this case) is not enough or client some how still says block size is zero in some cases. Better error message should help here.

If you see small number of these exceptions compared to number of blocks written, for now you don't need change anything.

Raghu Angadi added a comment - 02/Apr/07 20:31

Attached patch prints a warning and throws the IOException received.

The new log entry looks like this:

```
2007-04-02 12:59:15,940 WARN org.apache.hadoop.dfs.DataNode: No space left on device while writing
blk_8638782110649810591 (length: 67108864) to /export/crawlspace/rangadi/tmp/ramfs (Cur available
space : 20554389)
2007-04-02 12:59:15,943 ERROR org.apache.hadoop.dfs.DataNode: DataXCeiver java.io.IOException: No
space left on device
at java.io.FileOutputStream.writeBytes(Native Method)
at java.io.FileOutputStream.write(FileOutputStream.java:260)
at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)
at java.io.BufferedOutputStream.write(BufferedOutputStream.java:109)
at java.io.DataOutputStream.write(DataOutputStream.java:90)
at org.apache.hadoop.dfs.DataNode$DataXceiver.writeBlock(DataNode.java:837)
at org.apache.hadoop.dfs.DataNode$DataXceiver.run(DataNode.java:603)
at java.lang.Thread.run(Thread.java:619)
```

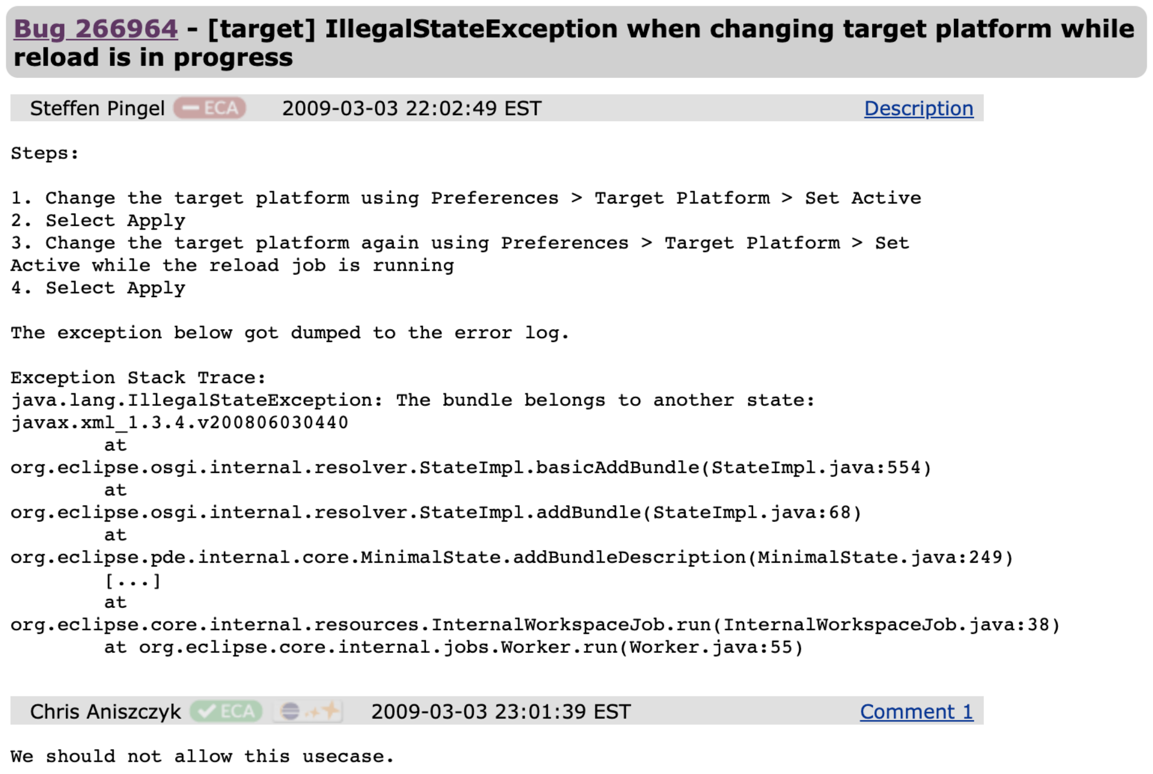
Figure 9: An example bug report (**HADOOP-1189**) that highlights the insufficient disk space left in one of the partitions. The bug fix updated the *FSDataset* class which is not shown in the logs (but based on our manual study, the *FSDataset* class is invoked between the first log and the second log).

remaining disk space. The logs show the execution of blocks (i.e., in a distributed storage system, blocks are essentially chunks of files that are stored across DataNodes) when writing to *DataNode*. The bug occurs inside the *getAvailable* method, from the *FSDataset* class, that incorrectly calculates the available space. The *getAvailable* method was executed as part of the execution between the first log snippet carrying the message of "*No space left on device while writing ...*" and the stack trace throwing the *IOException*. However, the class (i.e., *FSDataset*) is not recorded in the stack trace since calls to the class have returned before throwing exceptions, so are no longer available in the stack. Since logs are expensive to maintain and monitor [108, 196], developers may need to prioritize on logging the essential code snippets. Hence, some code snippets may be ignored and not logged. As shown in the previous example, an important code snippet was hidden between two logging statements. One potential direction for future research is to focus on reconstructing the execution path among logs and uncover the hidden paths between logs to further assist debugging.

Our finding indicates that reporters often only attach debugging information for the point of failure (e.g., stack traces). Although such information is helpful, there is a missing link between the failure and the root cause of the problem in the source code. Reporters may consider attaching additional logs (e.g., log snippets) that show the execution of the system in addition to stack traces. Additional research is required to help reporters provide missing logs in bug reports that complete the execution information and help developers with debugging the problem.

To better illustrate the cases where the fixed classes are unreachable through the call graph, Figure 10 shows such example. The bug report Eclipse PDE 266964 shows an *IllegalStateException* when modifying the preferred platform. This error is due to the user job that keeps running while the user switches the target platform. The stack trace shows that the *Worker* class continues to process the user job which leads to the *IllegalStateException*. The developers discussed in the comments that such use cases should not be allowed. The fixed classes were TargetPlatformPreferencePage2, TargetEditor and LoadTargetDefinitionJob. The fix ensured that any existing jobs are cancelled before the target platform switches. In such cases, the bug fix occurs in a small workflow change of the system, and it is almost impossible for developers to show such details in logs.

Code evolution (12/278). We find that sometimes the source code that generates



Bug 266964 - [target] IllegalStateException when changing target platform while reload is in progress

Steffen Pingel ECA 2009-03-03 22:02:49 EST Description

```
Steps:

1. Change the target platform using Preferences > Target Platform > Set Active
2. Select Apply
3. Change the target platform again using Preferences > Target Platform > Set
Active while the reload job is running
4. Select Apply

The exception below got dumped to the error log.

Exception Stack Trace:
java.lang.IllegalStateException: The bundle belongs to another state:
javax.xml_1.3.4.v200806030440
        at
org.eclipse.osgi.internal.resolver.StateImpl.basicAddBundle(StateImpl.java:554)
        at
org.eclipse.osgi.internal.resolver.StateImpl.addBundle(StateImpl.java:68)
        at
org.eclipse.pde.internal.core.MinimalState.addBundleDescription(MinimalState.java:249)
        [...]
        at
org.eclipse.core.internal.resources.InternalWorkspaceJob.run(InternalWorkspaceJob.java:38)
        at org.eclipse.core.internal.jobs.Worker.run(Worker.java:55)
```

Chris Aniszczyk ECA 2009-03-03 23:01:39 EST Comment 1

```
We should not allow this usecase.
```

Figure 10: An example bug report (**Eclipse PDE Bug 266964**) where the fixed classes are unreachable through the call graph.

the logs no longer exists. In other words, the logs that the reporters provide are from an older version of the system. The logging statements or the source code class may have been removed during evolution. In such cases, developers may have additional challenges in understanding and fixing the bug. In addition, we find that 28.1% (323/1,151) of the studied bug reports do not have values for the *Affects Version* field (i.e., entered by the reporter or developers to indicate which versions they observed the bug). Even if the bug reports have *Affects Version*, only 32.4% (268/828) of the bug reports have the same *Fix Version* as the *Affects Version*. Note that we exclude PDE and AspectJ bug reports from this analysis since the *Fix Version* field is not available on Bugzilla. Namely, developers often debug and perform the fix on a different version of the code and not on the reported *Affects Version*. Our finding highlights that version information is essential for a high-quality bug report. Therefore, reporters are strongly suggested to include version information of the buggy system when submitting a bug report. Future studies should also be conducted to help developers analyze such bug reports by taking the past development history (e.g., prior source code changes) into consideration, since the fixes may need to be applied to newer versions of the system.

Our manual study finds that some user-provided logs only show the unexpected behavior (i.e., failure), but do not show the root cause of a bug nor the execution that led to the failure. Reporters should consider attaching additional logs to assist in debugging. In addition, some attached logs are from prior versions of the systems and can no longer be found in the source code. Future research is required to utilize prior source code changes as an important debugging hint for developers when analyzing bug reports.

3.4 Discussion and Implication of Our Findings

In this section, we summarize our findings and provide some discussion and implications.

More research and supports are needed for logging code evolution. In our manual study in RQ3, we find that some user-provided logs (i.e., either stack traces, log snippets, or both) can no longer be found in the version that developers are working on. Different from a prior study [198], we found that it is not uncommon for logging statements or methods in stack traces to be removed from the source code. If developers are not familiar with the system, such logging statement changes can cause additional challenges during debugging. Future studies should consider analyzing software development history and help developers locate the user-provided logs, for which the corresponding logging statements/methods were deleted or moved. In addition, for reporters, it is essential to provide the version information of the system when reporting a bug.

Reporters need additional assistance on providing logs in bug reports. Although logs provide important debugging information for developers, reporters may not be able to provide accurate logs that can illustrate the problem. For example, we find that reporters may attach incomplete logs or logs that only illustrate the exception. Hence, future studies should also consider helping reporters provide more accurate logs that can better assist debugging. One potential direction is to study the part of system execution that is not illustrated in the reported logs to find the missing link between the failure and the root cause of the problem.

Future studies could consider using execution paths that are re-constructed from readily-available runtime data to provide additional debugging supports. We find that, even though the quality of user-provided logs may not be perfect, these logs still provide a good indication of the fixed classes. Our finding highlights a potential direction that may further assist developers with debugging. For example, future studies may leverage logs to re-construct the execution paths between each log message or stack frame. For instance, as shown in Figure 9, although the fixed class is invoked on the execution path leading to the bug, but it does not directly appear in the reported stack trace. Therefore, to further assist developers in debugging, additional research is needed to leverage user-provided logs in re-constructing the execution paths leading to failures.

3.5 Threats to Validity

In this section, we discuss the threats to validity related to this study.

3.5.1 External Validity.

Threats to external validity are related to the generalizability of our findings. To increase the generalizability of our study, we conduct our case study on 10 large-scale open source systems that vary in size and infrastructures (e.g., data warehouse, real-time computation system, distributed file system). These systems are actively maintained and widely used. Although all the systems are Java-based, our approach is not limited to Java systems. We present our approach in a generic way that can easily be adapted to fit systems in other programming languages (e.g., by changing the regular expression). To reduce the external threat to validity, we include systems from different domains, ranging from databases to software development tools. We found that the results are similar across the studied systems. However, other system types, such as mobile applications, may use logs differently (e.g., for in-house debugging [202]) and our findings may not hold. Future studies are encouraged to conduct the analysis on systems in more diverse domains to improve the generalizability of our findings. For RQ3, we mitigate the sampling bias by ensuring the sample falls into a confidence level of 95% with a confidence interval of 3%. When sampling for our manual data set, we carefully respect the sample size of each studied system and sampled proportionally according to the number of bug reports per system.

3.5.2 Internal Validity.

Threats to internal validity are related to experimenter errors and bias. Our study shows that the results of direct mapping between logged classes and fixed classes highly depend on the quality of user-provided logs. Thus, the extracted logs are an internal threat to the validity of our study. To mitigate this threat, we choose 10 systems that vary in software maturity, to better observe the difference in log quality of each studied system.

Another threat to internal validity is that we use bug IDs in commit messages to identify bug fixing commits. Although the developers in the studied systems are required to provide bug IDs in commit messages as part of the development guideline, there may still be some mistakes. For example, in our manual study in RQ3, we found a few cases where developers made a typo when providing bug IDs in the message. Nevertheless, we find such cases to be rare, and based on our manual study on a statistically representative sample, the heuristic has a very high precision (99%).

Another threat to internal validity is the way in which we collected the bug reports with logs. Typically, reporters attach logs in the bug description or as comments. Sometimes, when the logs are too long, reporters may upload them as attachments. Therefore, bug reports with logs might also include those that have log files in attachments. We further investigate this possibility, and find only a small number of the reporters upload logs as attachment (i.e., in 51 out of 8,849 bug reports, log files were added as attachment), which limits the impact of this threat.

In our study, we selected bug reports with priority "Major" or higher because bug reports with a lower priority may have less of an impact on the overall quality of the system. Moreover, these bug reports are less likely to be fixed. For example, we find that only 14% of the bug reports with logs marked as "Minor" or less were fixed in Hive, 13% in Hadoop Common and Storm, and 12% in MapReduce. Therefore, we follow prior studies [42, 46, 199] and focus our analysis on the bug reports with priority "Major" or higher.

3.5.3 Construct Validity.

In this chapter, we have two manual studies. One investigates the reasons why some bug reports had no modification on existing Java files. The other one studies the reasons why some bug reports have no overlaps between the logged classes and fixed classes. Human biases may be introduced. To reduce the bias of our analysis, we have a second author to verify the assigned categories and any discrepancies are discussed until consensus is reached.

Chapter 4

Pathidea: Improving Information Retrieval-Based Fault Localization by Re-Constructing Execution Paths Using Logs

In the last chapter, we conducted an empirical study on the usefulness of user-reported logs in debugging. Although these logs often offer valuable insights into the locations of faults, many of them may lack comprehensive system execution information. They often only indicate the point of failure without directly hinting at the actual root cause. In our empirical study, we presented logs as valuable information that can be leveraged to reconstruct system execution paths when an issue occurs, thereby assisting developers with debugging. A natural question that arises is how we can effectively utilize the reconstructed execution paths in a fault localization technique. This chapter covers an information retrieval-based fault localization (IRFL) approach, Pathidea, which addresses this aspect.

To help developers speed up the debugging process, researchers have proposed various IRFL approaches [29, 54, 90, 112, 117, 151, 161, 171, 206]. As introduced in Section 2.3, IRFL approaches leverage information retrieval to help developers locate potentially faulty files. Given a bug report, IRFL approaches use its textual information as queries to generate a ranked list of source code files, based on their textual similarity, that are potentially faulty. To improve the performance of IRFL, researchers have proposed approaches that utilize various information in software repositories, such as similar bug reports from the past [206], software development history [161, 175], and structured information in bug reports [171].

In addition to the textual description of a bug, developers often provide the system execution information when a bug happens. Developers may attach log snippets (e.g., `2015-07-01 19:24:12,806 INFO org.apache.ZooKeeper.ClientCnxn: Client session timed out`) or stack traces (e.g., `java.lang.NullPointerException`) that show a snapshot of the system execution. Prior studies [38, 108, 195] show that logs (i.e., log snippets or stack traces) can be mapped to source code to re-construct execution paths and assist developers with debugging. Even though a number of IRFL approaches [180, 206] try to leverage stack traces in bug reports, they only analyze the file names that appear directly in stack traces to boost the fault localization performance. Yet, the embedded system execution information, which can be re-constructed by linking the logs to their corresponding location in the source code, may further help improve the performance of IRFL approaches.

In this chapter, we proposed Pathidea, a new IRFL approach that uses static analysis to re-construct execution paths from logs in bug reports to help locate the potential faulty files. To the best of our knowledge, Pathidea is the first approach that incorporates the re-constructed execution paths into the IRFL approach. We conducted a case study to evaluate Pathidea on eight open source systems. The results demonstrate that Pathidea can identify faulty files with high precision and recall values, and outperforms existing state-of-the-art IRFL approaches. Our results show that the re-constructed execution paths can complement existing IRFL approaches by improving fault localization performance. Our evaluation presents a parameter sensitivity analysis and provides recommendations on setting the parameter values when applying Pathidea. In summary, our approach sheds light on further improving IRFL approaches by combining information in bug reports with the source code. Future studies may consider leveraging such execution paths information when designing IRFL approaches.

An earlier version of this chapter has been published at IEEE Transactions on Software Engineering (TSE), 2021. Pages 2905-2919. Chen et al. [39]

Bug ID	YARN-8209
Summary	NPE in DeletionService

Stack Trace

```
2018-04-25 23:38:41,039 WARN concurrent.ExecutorHelper
    ↪ (ExecutorHelper.java:logThrowableFromAfterExecute(63)): java.lang.NullPointerException
   at DockerClient.writeCommandToTempFile (DockerClient.java:109)
   at DockerCommandExecutor.executeDockerCommand (DockerCommandExecutor.java:3)
   at DockerCommandExecutor.executeStatusCommand (DockerCommandExecutor.java:192)
   at DockerCommandExecutor.getContainerStatus (DockerCommandExecutor.java:128)
   at LinuxContainerExecutor.removeDockerContainer (LinuxContainerExecutor.java:935)
   at DockerContainerDeletionTask.run (DockerContainerDeletionTask.java:61)
   at java.lang.Thread.run (Thread.java:748)
```

Figure 11: Stack traces extracted from the bug report YARN-8209.

4.1 Motivation

The textual information in bug reports often provide hints on where the faults may be located. To help developers reduce the needed time for locating the faults, researchers have proposed a series of IRFL approaches [29, 54, 90, 112, 117, 151, 161, 171, 206]. Although IRFL approaches have shown promising results, most of the prior studies only treat the information in bug reports as pure text. However, in addition to the textual description in bug reports, developers also heavily rely on the logs that the reporters provide to understand and debug issues [27]. As we discussed in the previous chapter, logs, either log snippets or stack traces, show the partial system execution when a problem occurs. Prior studies [108, 195] show that logs can be mapped to source code and assist developers with understanding the system execution during debugging and maintenance. Such valuable information may further help improve the performance of IRFL approaches.

Figure 11 depicts the stack trace extracted from the Description section of a bug report from YARN. Based on the textual information in the stack trace, IRFL approaches may identify files such as DockerClient, DockerCommandExecutor, Linux-ContainerExecutor, and DockerContainerDeletionTask (i.e., the name of the files shown in the stack trace) as potentially faulty files. However, the information is limited as we may overlook what happens *between each stack frame*. To resolve this bug, the developers provided a fix to the PrivilegedOperation file (shown in Figure 12), which

```
1   public class DockerCommandExecutor {
2
3     public static String executeDockerCommand (DockerCommand dockerCommand, ...)
        ↪ throws ContainerExecutionException {
4
5       PrivilegedOperation dockerOp = dockerCommand.preparePrivilegedOperation
          ↪ (dockerCommand, ...);
6
7       if (disableFailureLogging) {
8         dockerOp.disableFailureLogging();
9       }
...
31    }
...
251 }
```

Figure 12: Simplified source code from DockerCommandExecutor.executeDockerCommand. The fix of YARN-8209 was applied in PrivilegedOperation.

is called during the execution shown in the second last stack frame (at DockerCommandExecutor.executeDockerCommand (DockerCommandExecutor.java:3)), but not included in the stack trace (i.e., the call to PrivilegedOperation is already popped from the stack). Therefore, such *hidden execution* is invisible to IRFL approaches. Thus, in this chapter, we aim to utilize the execution paths information that can be reconstructed from logs to provide more information and improve fault localization performance.

4.2 Related Work

Some prior studies on IRFL leverage the system execution information (e.g., stack traces or test case execution) in bug reports to help locate faulty files. Dao et al. [54] leveraged the coverage, slicing, and spectrum information in failed test cases to improve IRFL. Wong et al. [180] propose BRTracer, a bug-report-oriented fault localization tool. The tool is built on top of BugLocator [206] and uses the file names that appear in stack traces to further rank the suspicious files. They studied 3,459 bug reports across three systems in which only 17% of the bug reports include stack traces. In addition to stack traces, Youm et al. [193] propose a new bug localization approach, BLIA (Bug Localization using Integrated Analysis) that integrates analyzed data by utilizing structured information in bug reports and source code files, code change history, similarity analysis of existing bug reports, and stack traces.

Prior studies only leverage the "visible" information in bug reports (e.g., stack traces). However, as shown in Figure 11 and 12, sometimes the hidden execution paths that can be re-constructed in bug reports can provide additional information for debugging and fault localization. In the previous chapter, we study if the logs in bug reports can be used to locate faulty files. We find that logs in bug reports provide a good indication on where the faulty files, although the provided logs may be outdated or can no longer be found in the source code. In this chapter, we leverage the stack traces and log snippets in bug reports to re-construct the system execution paths to complement IRFL approaches. We propose an IRFL approach called Pathidea and we find that it achieves better performance compared to state-of-the-arts such as BRTracer. Moreover, we find that the hidden execution information re-constructed from our path analysis can also help improve the performance of existing IRFL approaches (see details in Sections 4.5.1 and 4.5.2).

4.3 Pathidea: Path-guided Fault Localization

In this section, we first present an overview of Pathidea. Then, we describe each step of our approach in detail.

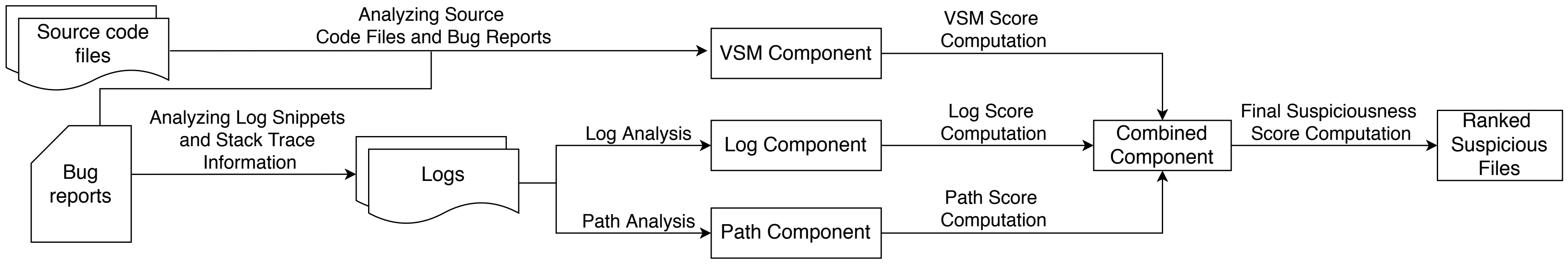


Figure 13: An overview of Pathidea.

4.3.1 An Overview of Pathidea.

Figure 13 shows an overview of our approach, which contains four major steps: (1) In the source code analysis step, we build a Vector Space Model (VSM) and compute an initial similarity score between the bug report and the source code files. We denote this initial score as *VSM score*. (2) In the log analysis step, we highlight the related files that appear directly in the logs using regular expression. Depending on the type of log, we apply different strategies to derive a boost score (i.e., to adjust the weight of the files), denoted as *log score*. (3) In the path analysis step, we re-construct the file-level execution paths from the logs to find the files which were called during the execution time. We assign a new boost score, which we denote as the *path score*, to these files. (4) Finally, we add the log and path scores into our initial similarity score to calculate the final suspiciousness score of a file. We rank the files based on the suspiciousness score and derive a list of ranked files for investigation. In the following subsections, we discuss the aforementioned steps in detail.

4.3.2 Analyzing Source Code Files and Bug Reports.

To analyze the source code files and bug reports, we follow common source code pre-processing steps [47]. We first tokenize the source code file into a series of lexical tokens and remove programming language specific keywords [139] (e.g., `for` and `while` for Java). Next, we split concatenated words based on camel case (e.g., getAverage) and underscore (e.g., get_average) and remove stopwords (e.g., *the* and *and*). We use the list of stopwords from the Natural Language Toolkit (NLTK) library in *Python* [138]. Finally, we perform Porter stemming to remove morphological affixes from words and derive their common base form (e.g., *running* becomes *run*). As mentioned in Section 2.3, the output of this process is a collection of corpus, where each document represents a source code file. Given a bug report, we extract the lexical tokens from the *summary* and *description* fields. To represent each bug report as a search query, we follow the same pre-processing steps described above.

Since larger files contain more tokens, by nature, large files are more likely to be favored in fault localization [180]. Thus, to treat all files equally regardless of its size, we follow a prior study [180] by using a segmentation approach when creating the corpus. The segmentation approach divides each file into multiple segments of code snippets of the same size. Namely, each document in the corpus represents a segment of code snippets from a source code file. Then, given a bug report, the corresponding file of the segment that has the highest suspiciousness score is marked as the most suspicious file for investigation. Similar to the study by Wong et al. [180], we set the segment size to 800 tokens.

More specifically, Formula 6 below calculates the VSM score between a file f and a bug report br, where $suspiciousness_{\max}(seg, br)$ is the maximum cosine similarity score between all the segments seg in f and br.

$$\text{VSM Score}(f, br) = suspiciousness_{\max}(seg, br). \tag{6}$$

4.3.3 Analyzing Log Snippets and Stack Trace Information

Logs provide an important source of information to developers. Prior studies [43, 48, 93, 195] have shown that developers often leverage logs to understand how the system was executed for debugging and testing purposes. Thus, our approach aims to utilize both types of logs (i.e., log snippets and stack traces) to further assist fault localization. We compute additional suspiciousness scores for the files that generate the logs. We denote the additional suspiciousness score computed from the logs as the log score.

To analyze the logs, we first capture them from a bug report using regular expressions. In particular, for stack traces, we check for the `at` keyword followed by a file name that ends with `.java`. For log snippets, we look for a timestamp followed by a verbosity level and a fully qualified class name. For instance, given the following log line "`2015-07-01 19:24:12,806 INFO org.apache.ZooKeeper.ClientCnxn: Client session timed out`", our regular expression captures "`2015-07-01 19:24:12,806`" as the timestamp, "`INFO`" as the verbosity level, and "`org.apache.ZooKeeper.ClientCnxn`" as the fully qualified class name. We use the fully qualified class name to derive its corresponding file name. Next, we verify in the source code repository whether the file exists. This helps us remove the files that are part of the external libraries. Finally, we calculate the log score differently for files extracted from stack traces and from log snippets. For stack trace, we use the rank of the file in the call stack to assess its suspiciousness score by following a prior study [180]. If a file appears on the top of the stack trace, it is ranked the first and receives a higher suspiciousness score. Given a rank position i, if i is within the top 10 ranks, the suspiciousness score is inversely proportional to the rank (e.g., the second ranked file receives a suspiciousness score of 0.5). For any rank position i beyond top 10, the file receives a constant suspiciousness score of *0.1*. Formula 7 below calculates the log score for files in a stack trace.

$$LogScore(f) = \begin{cases} \frac{1}{rank} & \text{if } rank \leq 10 \\ 0.1 & \text{if } rank > 10 \\ 0 & \text{if file not found} \end{cases} \tag{7}$$

For log snippets, we assign a constant value of *0.1* to every mapped file. We denote this constant value as α. We use α as a parameter to attribute a suspiciousness score to each file mapped from log snippets. In RQ3, we further investigate the sensitivity of the value for α.

When multiple stack traces are attached in a bug report, we regard them as equally valuable. Therefore, to further refine our approach, we reset the rank back to 1 when a new stack trace begins. In log snippets, when the same file appears multiple times, it is only computed once in the log score. Figure 14 shows an example of the log score computation. The logs start with a log snippet containing two log lines: "`task_r_1 done copying task_m_0`" and "`task_r_1 Copying task_m_1`". As both lines are generated by the same file, that is `ReduceTask.java`, the log score is only computed once with a constant value of 0.1. Two stack traces follow the log snippet. The first stack trace throws a `java.lang.OutOfMemoryError`, where the file `SequenceFile.java` appears in the first stack frame. Therefore, it is ranked as the first place, and its log score is 1.00. Similarly, the file in the second stack frame (i.e., `SequenceFile.java`) receives a log score of 0.50. The second stack trace throws a `java.lang.NullPointerException`, in which `InMemoryFileSystem.java`, `FileSystem.java` and `ReduceTask.java` receive their respective log score based on their order in the stack frames (i.e., 1.00, 0.50 and 0.33, respectively).

Logs

```
2008–01–07 21:02:13 INFO org.apache.hadoop.mapred.ReduceTask: task_r_1 done copying task_m_0
2008–01–07 21:02:13 INFO org.apache.hadoop.mapred.ReduceTask: task_r_1 Copying task_m_1
...
2008–01–07 21:02:13 WARN org.apache.hadoop.mapred.ReduceTask: java.lang.OutOfMemoryError: Java heap space
    at org.apache.hadoop.io.SequenceFile$Reader.init (SequenceFile.java:1345)
    at org.apache.hadoop.mapred.ReduceTask.run (ReduceTask.java:1311)
2008–01–07 21:02:31 ERROR org.apache.hadoop.mapred.ReduceTask: java.lang.NullPointerException: Map output copy
    ↪ failure
    at org.apache.hadoop.fs.InMemoryFileSystem.close (InMemoryFileSystem.java:378)
    at org.apache.hadoop.fs.FileSystem.getLength (FileSystem.java:449)
    at org.apache.hadoop.mapred.ReduceTask.run (ReduceTask.java:665)
```

Log Snippet #1

Rank	File	Log Score
1	ReduceTask.java	0.10

Stack Trace #1

Rank	File	Log Score
1	SequenceFile.java	1.00
2	ReduceTask.java	0.50

Stack Trace #2

Rank	File	Log Score
1	InMemoryFileSystem.java	1.00
2	FileSystem.java	0.50
3	ReduceTask.java	0.33

Figure 14: An example of the log score computation when there are log snippet and multiple stack traces.

4.3.4 Analyzing and Re-constructing Execution Paths.

As discussed in Section 2.3 and 4.1, most prior studies only consider the textual information that is available in the bug report. Since logs can be further mapped to the source code, there may be valuable information in the source code that can help developers to better understand the system execution. To this end, we analyze the logs and re-construct the potential execution paths. We describe the steps as follow.

We first extract the related methods and files that appear in the logs. We verify that such methods and files exist in the source code repository. Then, for each related method call, we derive the method-level Abstract Syntax Tree (AST) using Javaparser [137]. Javaparser is a static analysis tool that supports many Java versions and is actively maintained. The AST tree allows us to traverse the AST nodes and find the method calls inside each method declaration. We statically construct the execution path from the AST tree of the method by linking each method call into its method declaration. If a related method call appears in the execution path, we mark it as *visited*. The execution path continues to expand until the last method call in the log is *visited*. Once the execution path is re-constructed, we analyze the execution order of the related method calls and uncover the potential execution paths. Algorithm 1 shows the pseudo code of our implementation. The algorithm takes extracted logs from bug reports as input, and outputs the potential execution paths. First, we initiate a global variable (line 2) *executionPaths* to store the re-constructed execution path. When we iterate through the logs, we assign the current log at position *i* and the next log at position *i+1* (line 4-5). Then, the execution paths are derived from the logs (line 6). The *findPathBetween* function essentially implements the Breadth-First-Search (BFS) algorithm to traverse the call graph. In this process, we record every possible path that connects the current log to the next log. If two consecutive logs are identical (i.e., have the same log template), we remove one from the logs (e.g., the logs may be generated in a loop). Once the execution path is re-constructed, we store it in the local variable *paths* (line 6), which is then added to the global variable *executionPaths* (line 7). Lastly, we return the global variable *executionPaths* (line 9).

Note that a path is constructed for each sequential set of logs (e.g., logs belong to the same thread). Thus, after we have obtained the re-constructed execution paths, there may be some duplicated paths due to the looping of some logs generated at

Algorithm 1 Execution Paths Re-Construction Algorithm

Input: Extracted Logs

Output: Execution Paths

```
1: procedure findExecutionPath($logs$)
2:     initialise executionPaths
3:     for $i$=1; $i$<logs.length do
4:         currentLog = logs.atPosition($i$)
5:         nextLog = logs.atPosition($i$+1)
6:         paths = findPathsBetween(currentLog, nextLog)
7:         executionPaths.add(paths)
8:     end for
9:     return executionPaths
10: end procedure
```

runtime by different threads. Therefore, we compare the sequence of method calls inside each generated path and remove the duplicated paths.

In our experiment, we use a virtual machine, with a four-core Intel Xeon (Skylake, IBRS) CPU (2.10 GHz) and 5 GB of RAM. On average, the call graph analysis and path construction take 22 minutes for the entire system, where the size of our studied systems varies from 79k to 1.2 million source lines of code. Note that the call graph analysis only needs to be done once, since, in practice, we can incrementally update the call graph based on the code changes in each commit. Therefore, we believe that the additional call graph analysis and path construction time is reasonably acceptable and would not affect the usability of the approach.

Once the execution paths are re-constructed, we compute the path score for every file on the execution paths. We compute the path score as follows:

$$\text{PathScore}(f, br) = \beta \times N(VSMScore(f, br)) \tag{8}$$

Given a bug report br, $VSMScore(f, br)$ is the cosine similarity score between file f and the bug report, where N is the normalization function that normalizes $VSMScore(f, br)$ to a value in the range of between 0 and 1, and β denotes the weight of $VSMScore(f, br)$, that is between 0 and 1.

When a file appears on the path, the parameter β boosts the suspiciousness score to favor the files that were on the execution path. The faulty files may be one of the files that are on the execution path [195]. By introducing the path score, we are able to better distinguish the relevant files on the execution path from the less relevant files. In our study, we set β to 0.2 (we evaluate the effect of β in RQ3).

We explain the aforementioned path score computation in detail with Figure 15 that serves as our running example. We derive the running example from a real bug report. We simplify the code for the ease of explanation and limit the call graph to a depth of one. In this example, our goal is to derive the execution path from the logs and compute the path score. This process of re-constructing the execution path is analogous to the sailing boat traveling back to the shore. The idea is that the running execution (the sailing boat) navigates through the logging statements (the beacons) in order to reach the potentially faulty classes (the shore). First, the logging statements are an analogy of "beacons". The running execution finds and connects each of the logging statements in order to reconstruct the execution path. Starting from the reported logs, each log line is mapped to its corresponding logging statement by matching the static part of the logging statement. In our running example, by comparing each log line to the static part of the logging statements, we find that the log line "2019-01-07 21:02:13 INFO ReduceTask: task_r_1 initialized" is generated by the logging statement at line 3, and the log line "2019-01-07 21:02:13 INFO ReduceTask: runNewReducer called" is produced by the logging statement at line 12. Based on this information, we re-construct the execution path by connecting these two logging statements, and find the call path [2–6, 10, 11–16] that contains both of the logging statements at line 3 and 12. Since the logging statement at line 18 is not executed nor recorded, the analysis excludes *runOldReducer()* in the execution path. Then, the classes that appear on the execution path are treated as the potentially faulty classes. The list of classes collected in our running example are: JobConfiguration, Reducer, and Context. Note that only the classes that are relevant to the project are collected. As these classes might have data dependencies with the bug (or even contain the bug), we assign a PathScore to the files that contain these classes to boost their suspiciousness score based on Formula 8.

Logs

```
① 2019-01-07 21:02:13 INFO ReduceTask: task_r_1 initialized
② 2019-01-07 21:02:13 INFO ReduceTask: runNewReducer called
③ 2019-01-07 21:02:13 INFO ReduceTask: task_r_1 done
```

	Code Snippets	Executed	Class boosted by path
1	`class ReduceTask {`	●	-
2	`void run_task(String task_id, JobConfiguration job){`	●	-
3	`log.info(task_id + ' initialized'); //①`	●	-
4	`boolean useNewApi = job.getUseNewReducer();`	●	JobConfiguration
5	`if (useNewApi) {`	●	-
6	`runNewReducer();`	●	-
7	`} else {`	○	-
8	`runOldReducer();`	○	-
9	`}`	○	-
10	`log.info(id + 'done'); //③`	●	-
11	`void runNewReducer(){`	●	-
12	`log.info('runNewReducer called'); //②`	●	-
13	`Reducer reducer = createReducer(job);`	●	Reducer
14	`Context context = createReduceContext();`	●	Context
15	`reducer.run(context);`	●	-
16	`}`	●	-
17	`void runOldReducer(){`	○	-
18	`log.info('runOldReducer called');`	○	-
19	`ReduceValuesIter values = new ReduceValuesIter();`	○	-
20	`values.informReduceProcess();`	○	-
21 22	`}` `}`	○	-

Figure 15: An example of the execution path analysis for path score computation.

4.3.5 Calculating the Final Suspiciousness Score.

To incorporates the vector space model, log analysis and path analysis into one combined component, we calculate the final suspisiciousness score by summing up the normalized VSM score, log score, and path score (as shown in Formula 9).

$$FinalScore(f, br) = N(VSMScore(f, br)) + LogScore(f, br) + PathScore(f, br) \quad (9)$$

4.4 Case Study Setup

4.4.1 Data Collection

We evaluate the performance of our proposed fault localization approach on the bug reports, which contain logs (i.e., either log snippets, stack traces, or both), collected from eight open source systems. These systems are large in size, actively maintained, well-documented, and cover various domains ranging from big data processing to message brokers. For each system, we collect the bug reports as follows. The bug reports for all eight studied systems are available on the JIRA bug tracking repository [22]. Therefore, we implement a web crawler to collect bug reports from JIRA. We first select the bug reports that have the *resolution* status labeled as "Resolved" or "Fixed", the *priority* field is marked as "Major", "Critical", or "Blocker", and the *creation date* is 2010 or later. We download these bug reports in JSON format using the JIRA API [76]. Then, we examine the source code repository to further select the bug reports that have corresponding bug fixes by following prior studies [85, 209]. All the studied systems follow the convention to include the bug report identifier (e.g., HADOOP-1234) at the start of the commit messages (e.g., HADOOP-1234. Fix a typo in FileA.java) [18] and host the source code on Github. Therefore, we run the git command, `git log | grep bug_report_identifier[^\d]`, to check if a bug report identifier exists in any commit message. If a bug report identifier appears in a commit message, then there is a bug fix for the bug and we identify the commit as the bug fixing commit. Finally, to reduce noise, we exclude the bug reports in which no Java files were modified in the bug fixes. At the end of this process, we collected a total of 6,535 bug reports in the eight studied systems. After collecting the bug reports that have corresponding bug fixes, we further categorize the bug reports into *with logs* and

Table 9: An overview of our studied systems, where BRWL denotes bug reports with logs, and BRNL denotes bug reports without logs.

System	LOC	# Bug reports	# BRWL	# BRNL
ActiveMQ	338k	594	86 (14%)	508 (86%)
Hadoop Common	190k	725	257 (35%)	468 (65%)
HDFS	285k	1,166	229 (20%)	937 (80%)
MapReduce	198k	575	166 (29%)	409 (71%)
YARN	548k	576	241 (42%)	335 (58%)
Hive	1.2M	2,231	195 (9%)	2,036 (91%)
Storm	275k	380	61 (16%)	319 (84%)
ZooKeeper	79k	288	38 (13%)	250 (87%)
Total	3.1M	6,535	1,273	5,262

without logs.

Table 9 shows an overview of the bug reports that we collected. We denote bug reports with logs as *BRWL* and bug reports without logs as *BRNL*. In total, there are 1,273 bug reports with logs and 5,262 bug reports without logs. Although there are fewer bug reports with logs, the number is still non-negligible (around 20% of all bug reports). Thus, if we can leverage the embedded information in logs, we may better help improve fault localization.

4.4.2 Metrics for Evaluating Pathidea

To evaluate the effectiveness of our approach, we consider several commonly-used evaluation metrics for IR-based fault localization approaches [101, 171, 175]. First, we calculate the precision, recall, and F1-measure for the Top@N results (i.e., when examining the highest ranked N files). Then, we calculate the mean average precision and mean reciprocal rank (as described in Section 2.5). Below, we briefly describe the precision, recall, and F1-measure for Top@N results.

Precision@N. Given Top@N, the precision metric calculates the percentage of faulty files that are *correctly located* in the highest ranked N files. Precision@N is calculated as:

$$\text{Precision@}N = \frac{\#\, faulty\, files\, in\, top\, N}{N} \tag{10}$$

Recall@N. Given Top@N, the recall metric calculates the percentage of faulty files (out of all faulty files) that were located in the highest ranked N files. Recall@N is calculated as:

$$\text{Recall@}N = \frac{\#\, faulty\, files\, in\, top\, N}{\#\, total\, faulty\, files} \tag{11}$$

F1@N. F1-measure, also called F1 score is the weighted harmonic mean of precision and recall. This metric calculates the Top@N accuracy of the ranked results and offers a good trade-off between the precision and recall. F1 score is calculated as:

$$F1@N = 2 \cdot \frac{\text{Precision@}N \cdot \text{Recall@}N}{\text{Precision@}N + \text{Recall@}N} \tag{12}$$

4.5 Case Study Results

In this section, we discuss the results of our three research questions (RQs).

4.5.1 RQ1: Effectiveness of Pathidea Over State-of-the-art Approaches

Motivation: Most prior research that uses IRFL only considers the textual information that is available in the bug report. Although logs contain textual information of the occurred events, logs can also be further mapped to source code and assist developers with understanding the system execution during debugging and maintenance [108, 195]. Such system execution information may further help improve the performance of IRFL approaches. Therefore, in this RQ, we want to compare Pathidea with existing IRFL approaches.

Approach: As discussed in Section 4.3, for each bug report, we compute the final suspiciousness score for the files in the corresponding commit. We choose to use the commit that is prior to the bug fixing commit to avoid biases. We compare the performance of Pathidea with two baseline IRFL approaches. A recent study [101] shows that, among state-of-the-art IRFL approaches, BRTracer achieves the best results in terms of MAP and MRR. In addition, similar to Pathidea, BRTracer uses information in stack traces to improve fault localization results [180]. Hence, for the first baseline, we compare our approach with BRTracer. For the second baseline, we compare Pathidea with the vanilla approach that uses the basic vector space model (VSM) for fault localization. We apply the approaches on all eight studied systems and compare their performance in terms of precision, recall, and F1-measure at the top 1, top 5, and top 10 ranked files. We also compare the MAP and MRR scores of the approaches. Finally, we use the Wilcoxon rank-sum test to investigate whether Pathidea achieves a statistically significant improvement over the baselines. We choose the Wilcoxon rank-sum test since it is a non-parametric test that does not have an assumption on the distribution of the underlying data [127].

Result: Pathidea significantly outperforms BRTracer and VSM in all studied systems with respect to all evaluation metrics. Table 10 compares the results between VSM, BRTracer, and Pathidea. We find that for every studied system, VSM performs the worst among the three IRFL approaches. Thus, we focus our comparison between BRTracer and Pathidea. The numbers in the parentheses show the percentage of the improvement of Pathidea over BRTracer. Compared to BRTracer, Pathidea achieves an average MAP of 35% across the studied systems, which is a 13% improvement over that of BRTracer (i.e., 31%). Across the studied systems, the improvement in MAP varies from 8% to 24%. For MRR, Pathidea achieves an average of 43% across the studied systems, which is a 12% improvement over that of BRTracer (i.e., 38%).

Pathidea achieves an average Recall@10 of 50.3%, which shows that it can identify half of the faulty files in a relatively short list. Pathidea shows a large improvement in terms of Precision@N and Recall@N. Pathidea achieves, on average, 16%, 12%, and 11% improvement in Precision@$1, 5, 10$, respectively. For the average Recall@$1, 5, 10$, we see 20%, 14%, and 10% improvement, respectively. Regarding the F1-measures, Pathidea achieves an improvement between 15.5% and 31% for Top@1, and between 13.8% and 24.0% for Top@5. The average precision values indicate that 30.6% of the located files are actually faulty at Top@1, 13.6% at Top@5, and 8.2% at Top@10. The high average recall values indicate that Pathidea can locate 22.3% of

Table 10: Comparisons of results between VSM, BRTracer and Pathidea. For each metric, we calculate the percentage of improvements that Pathidea achieves over BRTracer.

System	Approach	Top@1			Top@5			Top@10			MAP	MRR
		Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)		
ActiveMQ	VSM	7.0	3.6	4.7	4.7	14.9	7.1	3.4	20.2	5.8	0.11	0.15
	BRTracer	23.3	16.6	19.4	10.9	33.5	16.5	6.9	40.1	11.7	0.28	0.34
	Pathidea	**29.1** (+25%)	**21.4** (+29%)	**24.6** (+27%)	**14.4** (+32%)	**43.7** (+30%)	**21.7** (+32%)	**9.0** (+31%)	**48.3** (+21%)	**15.1** (+29%)	**0.34** (+21%)	**0.41** (+21%)
Hadoop Common	VSM	14.0	9.8	11.5	7.5	24.3	11.4	5.1	31.9	8.8	0.20	0.23
	BRTracer	33.1	24.4	28.1	13.2	44.1	20.3	7.8	50.6	13.5	0.37	0.44
	Pathidea	**36.2** (+9%)	**27.2** (+11%)	**31.0** (+10%)	**13.9** (+6%)	**46.9** (+7%)	**21.5** (+6%)	**8.2** (+5%)	**53.4** (+6%)	**14.2** (+5%)	**0.40** (+8%)	**0.47** (+7%)
HDFS	VSM	16.2	10.6	12.8	10.1	30.3	15.2	7.1	38.9	12.0	0.23	0.29
	BRTracer	25.8	19.4	22.1	14.3	44.8	21.7	9.1	53.2	15.6	0.35	0.41
	Pathidea	**31.9** (+24%)	**23.8** (+23%)	**27.2** (+23%)	**15.8** (+10%)	**50.4** (+12%)	**24.0** (+10%)	**9.9** (+8%)	**57.7** (+9%)	**16.9** (+8%)	**0.40** (+14%)	**0.46** (+12%)
MapReduce	VSM	13.3	9.1	10.8	6.5	21.4	10.0	4.2	27.7	7.3	0.17	0.21
	BRTracer	18.7	13.8	15.9	10.7	38.4	16.8	6.4	44.4	11.2	0.27	0.32
	Pathidea	**22.3** (+19%)	**17.1** (+24%)	**19.4** (+22%)	**11.2** (+4%)	**41.1** (+7%)	**17.6** (+5%)	**6.6** (+3%)	**46.5** (+5%)	**11.5** (+3%)	**0.30** (+11%)	**0.35** (+9%)
YARN	VSM	14.9	9.7	11.7	8.4	26.1	12.7	5.6	34.2	9.6	0.20	0.25
	BRTracer	27.0	18.9	22.2	13.7	45.0	21.0	8.0	52.1	13.9	0.34	0.42
	Pathidea	**35.3** (+31%)	**26.0** (+38%)	**30.0** (+35%)	**15.4** (+12%)	**51.9** (+15%)	**23.7** (+13%)	**8.7** (+8%)	**56.8** (+9%)	**15.0** (+8%)	**0.41** (+21%)	**0.48** (+14%)
Hive	VSM	9.7	5.2	6.8	7.2	17.7	10.2	5.4	26.9	9.0	0.15	0.20
	BRTracer	37.4	23.9	29.2	13.9	40.3	20.7	7.8	44.4	13.3	0.35	0.46
	Pathidea	37.4 (+0%)	**24.1** (+1%)	29.3 (+0%)	**15.5** (+11%)	**47.5** (+18%)	**23.4** (+13%)	**8.8** (+12%)	**53.1** (+20%)	**15.1** (+13%)	**0.38** (+9%)	**0.50** (+9%)
Storm	VSM	18.0	11.7	14.2	9.8	28.2	14.6	5.9	33.8	10.0	0.22	0.28
	BRTracer	32.8	22.5	26.7	12.5	40.9	19.1	7.5	47.6	13.0	0.33	0.42
	Pathidea	**34.4** (+5%)	**25.0** (+11%)	**29.0** (+8%)	**14.1** (+13%)	**45.7** (+12%)	**21.5** (+13%)	**8.2** (+9%)	**50.5** (+6%)	**14.1** (+8%)	**0.37** (+12%)	**0.45** (+5%)
ZooKeeper	VSM	5.3	2.8	3.7	3.2	9.0	4.7	2.9	14.9	4.8	0.10	0.12
	BRTracer	13.2	9.4	11.0	7.9	26.3	12.1	5.0	32.0	8.6	0.21	0.24
	Pathidea	**18.4** (+40%)	**13.4** (+42%)	**15.5** (+41%)	**8.9** (+13%)	**30.2** (+15%)	**13.8** (+14%)	**5.8** (+16%)	**36.1** (+13%)	**10.0** (+15%)	**0.25** (+19%)	**0.29** (+21%)
Average across studied systems	BRTracer	26.4	18.6	21.8	12.1	39.2	18.5	7.3	45.6	12.6	0.31	0.38
	Pathidea	**30.6** (+16%)	**22.3** (+20%)	**25.8** (+18%)	**13.6** (+12%)	**44.7** (+14%)	**20.9** (+13%)	**8.2** (+11%)	**50.3** (+10%)	**14.0** (+11%)	**0.35** (+13%)	**0.43** (+13%)

all the faulty files at Top@1, 44.7% at Top@5, and 50.3% at Top@10. Note that since the number of faulty files is often small (i.e., the median number of faulty files is three in the studied systems), it is difficult to achieve a high precision in the ranked results. However, our approach is able to achieve a relatively high recall within a small N. Hence, by only investigating a small number of files, developers may identify around half of the faulty files. We also use the Wilcoxon rank-sum test [179] to examine whether the improvements of Pathidea over BRTracer are statistically significant. Our results show that the improvements are statistically significant in terms of MAP, MRR, recall, and precision values (*p-value* < 0.05).

Across the studied systems, Pathidea achieves an improvement that varies from 8% to 21% and 5% to 21% over BRTracer in terms of MAP and MRR across the studied systems, respectively. We also find that both Pathidea and BRTracer outperform the vanilla VSM in identifying faulty files. Moreover, Pathidea can identify faulty files with an average Recall@10 of 50.3%.

4.5.2 RQ2: Effectiveness of Path Analysis

Motivation: Previous studies [55, 180] leveraged logs to improve the ranking of the potential faulty files for further investigation. However, these approaches either directly consider logs (i.e., stack traces) as plain text, or only retrieve the files that appear directly in logs. In practice, developers not only examine the logs, but they also leverage the logs to re-construct the run-time execution paths of the system for debugging [108, 195]. Such path information may be helpful to not only Pathidea but also other IRFL approaches. Therefore, in this RQ, we study the effect of the path analysis on the performance of existing IRFL approaches.

Approach: Our goal is to study how much additional improvement can path analysis provide to IRFL approaches. Thus, we first examine the effectiveness of Pathidea with and without path analysis. Then, we further study if path analysis can help improve existing IRFL approaches. In particular, we apply path analysis to BRTracer, because it is shown to have one of the highest MAP and MRR among the IRFL approaches [101]. Moreover, BRTracer leverages logs for fault localization (i.e., the class names that are recorded in stack traces), so we can study if path analysis provides additional information to BRTracer's log analysis. We also use the Wilcoxon signed-rank test [179] to investigate whether our path analysis provides a statistically significant improvement to these two IRFL approaches.

Result: Considering path analysis improves the overall effectiveness of Pathidea by up to 20% in terms of the evaluation metrics. Table 11 compares the results of Pathidea when considering different components. We show the evaluation metrics when different components are considered and evaluate the improvement over the VSM baseline. We focus on evaluating the effectiveness of Pathidea with and without path analysis. Specifically, when considering path analysis (i.e., VSM + log + path), Pathidea has an improvement of MAP that varies from 4% to 20% over the ones without path analysis (i.e., VSM + log) across the studied systems. The improvement of MRR varies from 4% to 17% across the studied systems. We also observe an average improvement of 14%, 4%, and 4% on Precision@$1, 5, 10$, respectively. For the average recall values, the improvements are 23%, 6%, and 3% for Recall@$1, 5, 10$, respectively. The Wilcoxon signed-rank test also shows that, for all the studied systems, the improvements are statistically significant for Recall@1, Recall@5, Precision@1, Precision@5, and F1@1 (*p-value* < 0.05). Our finding shows that the path analysis is able to help promote the ranking of the faulty files in the result.

When applying path analysis on an existing IRFL approach (i.e., BRTracer), there is a 10% and 8% improvement in terms of MAP and MRR, respectively. Table 12 compares the results of BRTracer with and without considering path analysis. We observe that, when considering path analysis, the MAP and MRR of BRTracer receive a 10% and 7% improvement, respectively. We also observe an improvement on the precision, recall, and F1, and most notably on Precision@1 and Recall@1. Specifically, the path analysis improves the average precision values by 12%, 5%, 5% for Precision@$1, 5, 10$, respectively. The improvement on the average recall values are 14%, 5%, 4% for Recall@$1, 5, 10$, respectively. Our finding shows that the path analysis may provide the largest improvement to BRTracer especially when N equals to 1. The Wilcoxon signed-rank test shows that, for all studied systems, the improvements are statistically significant for precision, recall, and F1. Compared to Table 11, we find that the path analysis provides more improvement to BRTracer compared to Pathidea. For example, in YARN, adding the path analysis to BRTracer improves all the evaluation metrics when Top@1 (from 28% to 30%), where as the

Table 11: Comparisons of Pathidea's results when considering different components. For each added component, we show the percentage of improvements over the VSM baseline.

System	Approach	Top@1			Top@5			Top@10			MAP	MRR
		Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)		
ActiveMQ	VSM	7.0	3.6	4.7	4.7	14.9	7.1	3.4	20.2	5.8	0.11	0.15
	VSM + log	26.7 (+283%)	18.6 (+416%)	21.9 (+362%)	12.8 (+175%)	39.1 (+162%)	19.3 (+172%)	8.7 (+159%)	47.1 (+133%)	14.7 (+155%)	0.32 (+191%)	0.39 (+160%)
	VSM + log + path	**29.1 (+317%)**	**21.4 (+495%)**	**24.6 (+419%)**	**14.4 (+210%)**	**43.7 (+193%)**	**21.7 (+206%)**	**9.0 (+166%)**	**48.3 (+140%)**	**15.1 (+161%)**	**0.34 (+209%)**	**0.41 (+173%)**
Hadoop Common	VSM	14.0	9.8	11.5	7.5	24.3	11.4	5.1	31.9	8.8	0.20	0.23
	VSM + log	35.4 (+153%)	25.9 (+165%)	29.9 (+160%)	13.8 (+84%)	46.2 (+90%)	21.2 (+86%)	8.1 (+59%)	52.6 (+65%)	14.0 (+60%)	0.39 (+95%)	0.46 (+100%)
	VSM + log + path	**36.2 (+158%)**	**27.2 (+177%)**	**31.0 (+169%)**	**13.9 (+86%)**	**46.9 (+93%)**	**21.5 (+88%)**	**8.2 (+60%)**	**53.4 (+68%)**	**14.2 (+61%)**	**0.40 (+100%)**	**0.47 (+104%)**
HDFS	VSM	16.2	10.6	12.8	10.1	30.3	15.2	7.1	38.9	12.0	0.23	0.29
	VSM + log	30.1 (+86%)	22.3 (+110%)	25.7 (+100%)	15.8 (+56%)	50.1 (+66%)	24.0 (+58%)	9.8 (+39%)	57.4 (+47%)	16.8 (+40%)	0.38 (+65%)	0.45 (+55%)
	VSM + log + path	**31.9 (+97%)**	**23.8 (+123%)**	**27.2 (+112%)**	15.8 (+56%)	**50.4 (+67%)**	24.0 (+58%)	**9.9 (+40%)**	**57.7 (+48%)**	**16.9 (+41%)**	**0.40 (+74%)**	**0.46 (+59%)**
MapReduce	VSM	13.3	9.1	10.8	6.5	21.4	10.0	4.2	27.7	7.3	0.17	0.21
	VSM + log	20.5 (+55%)	15.6 (+71%)	17.7 (+64%)	11.1 (+70%)	40.5 (+89%)	17.4 (+74%)	6.5 (+54%)	45.5 (+65%)	11.4 (+56%)	0.28 (+65%)	0.33 (+57%)
	VSM + log + path	**22.3 (+68%)**	**17.1 (+88%)**	**19.4 (+79%)**	**11.2 (+72%)**	**41.1 (+92%)**	**17.6 (+76%)**	**6.6 (+56%)**	**46.5 (+68%)**	**11.5 (+57%)**	**0.30 (+76%)**	**0.35 (+67%)**
YARN	VSM	14.9	9.7	11.7	8.4	26.1	12.7	5.6	34.2	9.6	0.20	0.25
	VSM + log	33.2 (+122%)	23.5 (+142%)	27.5 (+134%)	14.9 (+78%)	50.4 (+93%)	23.0 (+82%)	8.6 (+54%)	56.5 (+65%)	14.9 (+56%)	0.39 (+95%)	0.47 (+88%)
	VSM + log + path	**35.3 (+136%)**	**26.0 (+169%)**	**30.0 (+155%)**	**15.4 (+83%)**	**51.9 (+99%)**	**23.7 (+87%)**	**8.7 (+56%)**	**56.8 (+66%)**	**15.0 (+57%)**	**0.40 (+100%)**	**0.48 (+92%)**
Hive	VSM	9.7	5.2	6.8	7.2	17.7	10.2	5.4	26.9	9.0	0.15	0.20
	VSM + log	36.9 (+279%)	22.7 (+336%)	28.1 (+314%)	15.3 (+113%)	46.7 (+164%)	23.0 (+125%)	8.6 (+60%)	52.0 (+93%)	14.8 (+65%)	0.37 (+147%)	0.49 (+145%)
	VSM + log + path	**37.4 (+284%)**	**24.1 (+362%)**	**29.3 (+331%)**	**15.5 (+116%)**	**47.5 (+168%)**	**23.4 (+129%)**	**8.8 (+64%)**	**53.1 (+98%)**	**15.1 (+69%)**	**0.38 (+153%)**	**0.50 (+150%)**
Storm	VSM	18.0	11.7	14.2	9.8	28.2	14.6	5.9	33.8	10.0	0.22	0.28
	VSM + log	32.8 (+82%)	23.3 (+99%)	27.3 (+92%)	14.4 (+47%)	47.3 (+68%)	22.1 (+52%)	8.2 (+39%)	50.5 (+49%)	14.1 (+40%)	0.36 (+64%)	0.44 (+57%)
	VSM + log + path	**34.4 (+91%)**	**25.0 (+113%)**	**29.0 (+104%)**	14.1 (+43%)	45.7 (+62%)	21.5 (+48%)	8.2 (+39%)	50.5 (+49%)	14.1 (+40%)	**0.37 (+68%)**	**0.45 (+61%)**
ZooKeeper	VSM	5.3	2.8	3.7	3.2	9.0	4.7	2.9	14.9	4.8	0.10	0.12
	VSM + log	15.8 (+200%)	12.0 (+325%)	13.7 (+271%)	8.4 (+167%)	27.6 (+208%)	12.9 (+176%)	5.3 (+82%)	33.3 (+124%)	9.1 (+88%)	0.23 (+130%)	0.27 (+125%)
	VSM + log + path	**18.4 (+250%)**	**13.4 (+371%)**	**15.5 (+320%)**	**8.9 (+183%)**	**30.2 (+237%)**	**13.8 (+196%)**	**5.8 (+100%)**	**36.1 (+143%)**	**10.0 (+106%)**	**0.25 (+150%)**	**0.29 (+142%)**
Average across studied systems	VSM	12.3	7.8	9.5	7.2	21.5	10.7	5.0	28.6	8.4	0.17	0.22
	VSM + log	28.9 (+135%)	20.5 (+162%)	24.0 (+152%)	13.3 (+86%)	43.5 (+102%)	20.4 (+90%)	8.0 (+61%)	49.4 (+73%)	13.7 (+63%)	0.34 (+97%)	0.41 (+91%)
	VSM + log + path	**30.6 (+149%)**	**22.2 (+185%)**	**25.8 (+170%)**	**13.6 (+90%)**	**44.7 (+108%)**	**20.9 (+95%)**	**8.2 (+65%)**	**50.3 (+76%)**	**14.0 (+66%)**	**0.35 (+106%)**	**0.43 (+97%)**

Table 12: Comparisons of BRTracer's results with and without path analysis. For each added component, we show the percentage improvement over the original BRTracer.

System	**Approach**	**Top@1**			**Top@5**			**Top@10**			**MAP**	**MRR**
		Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)		
ActiveMQ	BRTracer	23.3	16.6	19.4	10.9	33.5	16.5	6.9	40.1	11.7	0.28	0.34
	BRTracer+ path	**26.7 (+15%)**	**19.1 (+15%)**	**22.3 (+15%)**	**12.1 (+11%)**	**36.7 (+10%)**	**18.2 (+10%)**	**7.2 (+5%)**	**42.0 (+5%)**	**12.3 (+5%)**	**0.30 (+7%)**	**0.37 (+8%)**
Hadoop Common	BRTracer	33.1	24.4	28.1	13.2	44.1	20.3	7.8	50.6	13.5	0.37	0.44
	BRTracer+ path	**35.0 (+6%)**	**26.4 (+8%)**	**30.1 (+7%)**	**13.9 (+6%)**	**46.6 (+6%)**	**21.4 (+6%)**	**8.1 (+4%)**	**52.3 (+3%)**	**14.0 (+4%)**	**0.39 (+5%)**	**0.46 (+5%)**
HDFS	BRTracer	25.8	19.4	22.1	14.3	44.8	21.7	9.1	53.2	15.6	0.35	0.41
	BRTracer+ path	**30.1 (+17%)**	**22.8 (+18%)**	**25.9 (+17%)**	**15.1 (+5%)**	**47.5 (+6%)**	**22.9 (+6%)**	**9.7 (+6%)**	**56.3 (+6%)**	**16.5 (+6%)**	**0.38 (+9%)**	**0.45 (+10%)**
MapReduce	BRTracer	18.7	13.8	15.9	10.7	38.4	16.8	6.4	44.4	11.2	0.27	0.32
	BRTracer+ path	**21.1 (+13%)**	**16.3 (+18%)**	**18.4 (+16%)**	10.7 (0%)	**39.5 (+3%)**	**16.9 (+1%)**	6.4 (0%)	**44.8 (+1%)**	11.2 (0%)	**0.29 (+7%)**	**0.33 (+3%)**
YARN	BRTracer	27.0	18.9	22.2	13.7	45.0	21.0	8.0	52.1	13.9	0.34	0.42
	BRTracer+ path	**34.4 (+28%)**	**24.9 (+32%)**	**28.9 (+30%)**	**14.9 (+8%)**	**49.5 (+10%)**	**22.9 (+9%)**	**8.4 (+5%)**	**54.7 (+5%)**	**14.6 (+5%)**	**0.39 (+15%)**	**0.48 (+14%)**
Hive	BRTracer	37.4	23.9	29.2	13.9	40.3	20.7	7.8	44.4	13.3	0.35	0.46
	BRTracer+ path	37.4 (0%)	**24.3 (+2%)**	**29.5 (+1%)**	**14.3 (+2%)**	**42.0 (+4%)**	**21.3 (+3%)**	**7.9 (+1%)**	**45.6 (+3%)**	**13.5 (+2%)**	**0.36 (+3%)**	**0.47 (+2%)**
Storm	BRTracer	32.8	22.5	26.7	12.5	40.9	19.1	7.5	47.6	13.0	0.33	0.43
	BRTracer+ path	32.8 (0%)	22.5 (0%)	26.7 (0%)	12.5 (0%)	40.9 (0%)	19.1 (0%)	7.4 (-2%)	47.1 (-1%)	12.8 (-2%)	0.33 (0%)	0.43 (0%)
ZooKeeper	BRTracer	13.2	9.4	11.0	7.9	26.3	12.1	5.0	32.0	8.6	0.21	0.24
	BRTracer+ path	**18.4 (+40%)**	**13.4 (+42%)**	**15.5 (+41%)**	**8.4 (+7%)**	**27.6 (+5%)**	**12.9 (+6%)**	**6.1 (+21%)**	**36.3 (+14%)**	**10.4 (+20%)**	**0.25 (+19%)**	**0.29 (+21%)**
Average across studied systems	BRTracer	26.4	18.6	21.8	12.1	39.2	18.5	7.3	45.6	12.6	0.31	0.38
	BRTracer+ path	**29.5 (+12%)**	**21.2 (+14%)**	**24.7 (+13%)**	**12.7 (+5%)**	**41.3 (+5%)**	**19.5 (+5%)**	**7.7 (+5%)**	**47.4 (+4%)**	**13.2 (+5%)**	**0.34 (+10%)**	**0.41 (+8%)**

improvement in Pathidea is only 6% to 11%. Our finding shows that the path analysis can provide additional information to not only Pathidea, but also other IRFL approaches (e.g., BRTracer). Future studies may consider integrating the path information to improve fault localization performance.

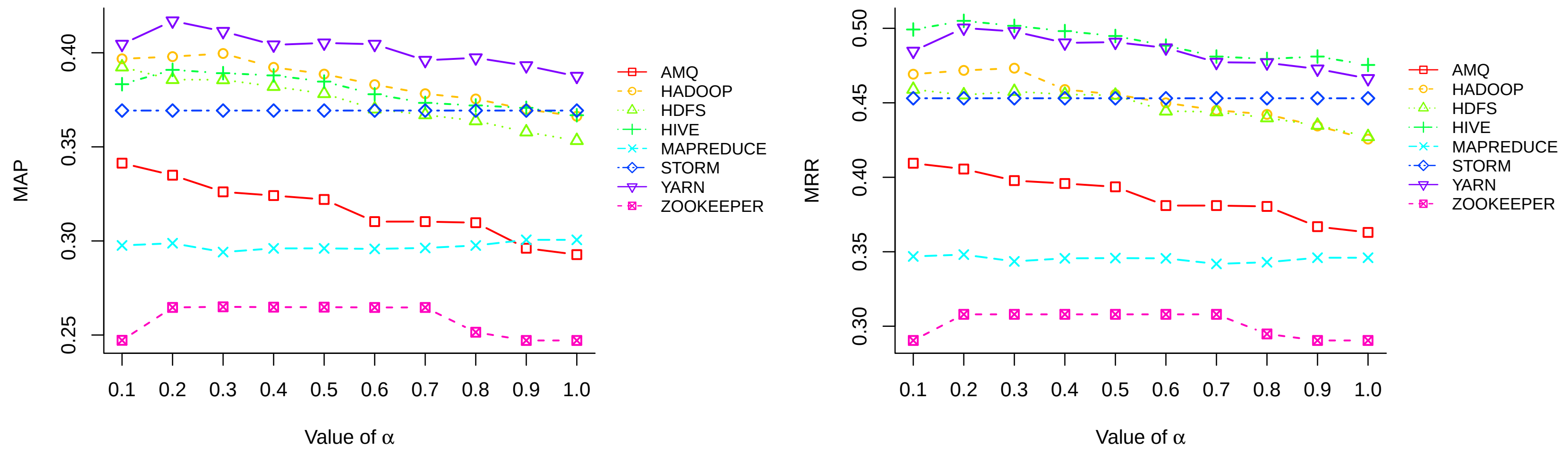
The re-constructed execution paths can complement BRTracer by providing a 10% and 8% improvement in MAP and MRR, respectively. We also find that Pathidea provides an average of 16% improvement over BRTracer on Precision@1. Future IRFL research may consider combining information in the source code (e.g., execution paths re-constructed from logs) to further improve fault localization performance.

4.5.3 RQ3: Parameter Sensitivity of Pathidea

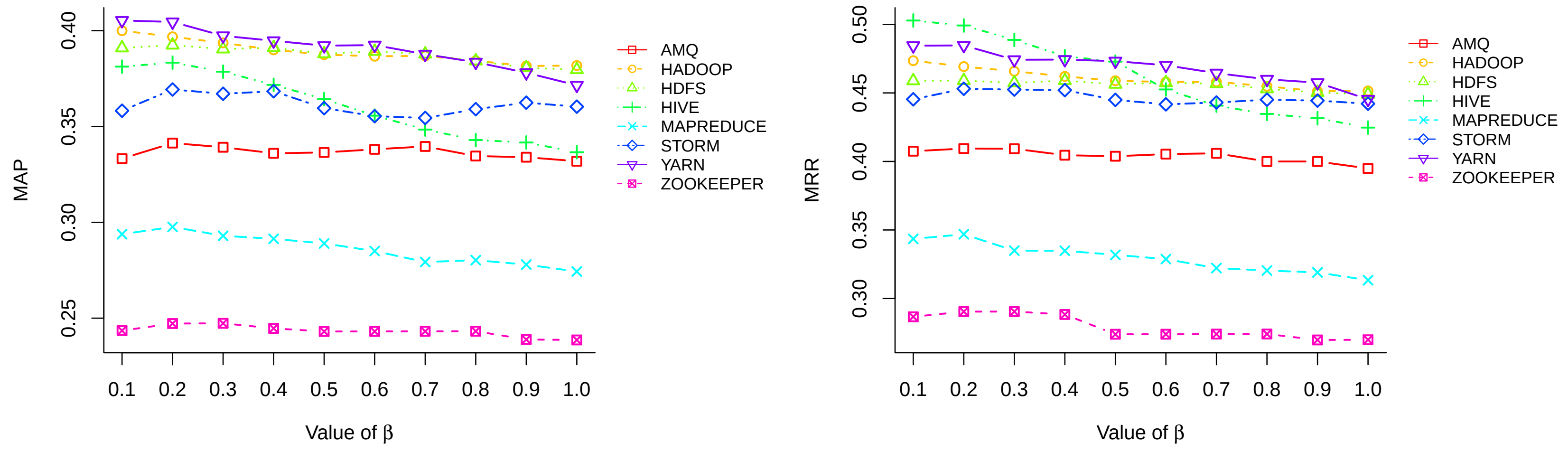
Motivation: As mentioned in Section 4.3, Pathidea uses two parameters α and β to calculate the final suspiciousness score. In each system, there may be some system-specific characteristics (e.g., lexical similarity, semantic redundancy of source code, and log density) that make the contribution of one component more important than others. For instance, if a system allocates a significant amount of effort on improving and maintaining logging statements for debugging, then the attached logs in the bug reports may contain more information compared to other systems. In such case, we may want to attribute more weight to the α parameter which is related to the logging statements. Therefore, in this RQ, we want to further investigate the sensitivity of the parameters on the overall effectiveness of Pathidea.

Approach: The parameter α serves to attribute a suspiciousness score to each file that appears in the logs (i.e., calculating *LogScore* in Equation 7). The parameter β serves as a magnifier that adjusts the weight of *VSMScore* to favor the files on the re-constructed execution paths (i.e., calculating *PathScore* in Equation 8). To understand the effect of these parameters on Pathidea, we perform a sensitivity analysis on the parameters separately by changing the values between 0.1 to 1.0, with an interval of 0.1, to quantify their effects in terms of the MAP and MRR values.

Result: Overall, the MAP and MRR values reach the highest when α and β are in the range of 0.1 and 0.2. However, we also observe some variations among the studied systems. Figure 16a shows the effectiveness of Pathidea when



(a) Effect of α value.



(b) Effect of β value.

Figure 16: Effect of α and β on Pathidea in terms of MAP and MRR.

varying the parameter α. We observe a relatively stable impact of α across the studied systems. For Hadoop Common, Hive, MAPREDUCE, YARN and ZooKeeper, when α increases from 0.1 to 0.2, we observe an improvement in terms of the MAP and MRR values. From 0.3 to 0.7, the MAP and MRR values remain relatively stable. Starting from 0.8 to 1.0, the MAP and MRR values decrease. The effect of α on ActiveMQ is different from the other systems. In ActiveMQ, the values of MAP and MRR decrease when the parameter α value increases. For MapReduce and Storm, the MAP and MRR values remain stable no matter how the parameter α varies. Figure 16b shows the effectiveness of Pathidea when varying the parameter β. Almost all systems achieve the highest MAP and MRR values when β is between 0.1 and 0.2. The further increase of β does not improve the MAP and MRR values for AMQ, MapReduce and Storm. In these three systems, the values of MAP and MRR decrease when β varies from 0.3 to 1.0. In summary, practitioners and future studies may consider setting the value of α and β between the range of 0.1 and 0.2 when applying Pathidea.

In general, Pathidea has the highest MRR and MAP values when the values of α and β are in the range of 0.1 and 0.2. Practitioners and future studies may consider choosing these values when integrating or applying Pathidea.

4.6 Discussion

Studying the effectiveness of the path analysis when added to BRTracer. In Section 4.5.2 (RQ2), we observe that Storm and Hive experience the least improvement when applying the path analysis on BRTracer (i.e., Table 12). In Hive, the improvements are 3% for MAP and 2% for MRR; and in Storm, the improvements for MAP and MRR are both 0%. After some investigation, we find that there is one possible factor that may be correlated with the relatively lower improvement when the path analysis is applied to BRTracer. We find that, among all the studied systems, Storm and Hive have the largest percentage of bug reports that contain only stack traces. 86.5% and 69.8% of the bug reports with logs contain only stack traces but no log snippets in Hive and Storm, respectively. On the other hand, there is only an average of 50.7% of such bug reports in other studied systems. A prior study [36] finds that many of the bug fixing locations may not be directly related to the reported stack traces. The stack traces may only show the symptom of the bug (e.g., NullPointerException), but the actual bug may manifest in a file that was called earlier during the execution (e.g., developers did not check the returned value in earlier method calls, which eventually results in a NullPointerException). Therefore, it may be possible that some faulty files are not related to the files on the re-constructed paths. In other words, path analysis may be less effective on the bug reports that only have stack traces than the ones that have both stack traces and log snippets. Note that the path analysis still has a relatively larger improvement for Storm and Hive when added to Pathidea. The possible reason may be that Pathidea has a different log score computation than BRTracer, which may provide larger improvements to the bug reports that contain log snippets. Future studies are needed to further understand the effect of log quality on the fault localization performance.

Another possible factor that affects the effectiveness of path analysis is the log density of a system. The log density is calculated by the ratio between thousands of lines of logging code and LOC ($\frac{lines\ of\ logging\ code}{thousands\ of\ lines\ of\ code}$). Intuitively, less noise would be introduced when re-constructing an execution paths with the reported log snippets if the log density is higher. To test our assumption, we calculate the log density of all the studied systems (Table 13). For instance, when considering path analysis on either Pathidea or BRTracer, we observe a substantial improvement in ZooKeeper (i.e., it has the highest log density among all studied systems) under all metrics. Specifically, when the path analysis is applied on BRTracer, the metrics of $Precision@1$, $Recall@1$ and $F1@1$ increase by 40%, 42% and 41%, respectively. The improvement is 19% for MAP, and 21% for MRR. ZooKeeper has the highest log density (Table 13). There is one line of logging code for every 33 lines of code. In Hive, where its log density is the lowest among the studied systems, we observe that the improvements are relatively small. Future studies are needed to examine the effect of log density on the effectiveness of the re-constructed execution paths in fault localization.

Effectiveness of segmentation. Table 14 shows the effectiveness of segmentation at different segment sizes. We evaluate the effectiveness based on Precision@1, Precision@5, Precision@10, MAP and MRR. We observe that, for most of the studied systems (i.e., ActiveMQ, Hadoop, Hive, Storm and ZooKeeper), 400 is the segment size that yields the most effective metrics, while the most effective segment size is 600

Table 13: Log density across studied systems, where LOC is referred to as lines of code, and LOLC is referred as lines of logging code. Note that we exclude code comments and empty lines.

System	**LOC**	**LOLC**	**LOLC per every thousand LOC**
ActiveMQ (5.15.13)	337,533	8,055	24
Hadoop Common (2.7.6)	189,744	2,471	13
HDFS (2.7.0)	285,071	5,971	21
MapReduce (3.1.4)	197,996	3,279	17
YARN (3.1.2)	548,043	6,854	13
Hive (2.7.0)	1,180,562	9,918	8
Storm (2.2.0)	274,860	5,620	20
ZooKeeper (3.6.0)	78,684	2,518	32
Total	3,092,493	44,686	19

for HDFS and MapReduce, and 800 for Yarn. Although the optimal segmentation size is different for each studied system, we observe a trend where smaller segmentation sizes (e.g., around or below 800) yields better localization results. Future studies and practitioners may consider using smaller segmentation sizes (e.g., 800 or below) when adopting the technique.

Parameters settings. Throughout our experiment, we have tuned these parameters to evaluate the effectiveness of our approach at different thresholds. Our experiment indicates that, for most of the studied systems, the MAP and MRR values achieve the best localization results when α and β are in the range of 0.1 and 0.2, and when the segment size is between 400 to 800. Although the optimized parameter setting can vary from system to system, some system characteristics may be related to the most effective parameter values. We observe a trend that smaller segmentation sizes (e.g., 800 or below) yield the best localization results, especially for smaller systems. For instance, Zookeeper, which is the smallest among all studied systems, has the best localization results when the segment size is 400. In contrast, Hive, which is the largest among all studied systems, has the best localization results when the segment size is 1,000. Therefore, future studies and practitioners may consider starting with

Table 14: Effectiveness of segmentation at different segment sizes, where *Size* column refers to segment size.

System	Size	Precision@1	Precision@5	Precision@10	MAP	MRR
ActiveMQ	400	10.5	4.9	4.1	0.13	0.18
	600	8.1	5.3	3.4	0.12	0.16
	800	7.0	4.7	3.4	0.11	0.15
	1000	4.7	4.4	3.3	0.09	0.12
	1200	4.7	4.7	3.4	0.09	0.12
Hadoop	400	17.1	9.3	6.1	0.22	0.28
	600	16.0	8.4	5.6	0.21	0.26
	800	14.0	7.5	5.1	0.20	0.23
	1000	13.6	7.1	5.1	0.19	0.22
	1200	12.5	6.6	4.7	0.18	0.20
HDFS	400	18.8	9.9	6.7	0.24	0.30
	600	21.4	10.7	6.9	0.26	0.32
	800	16.2	10.1	7.1	0.23	0.29
	1000	17.5	10.0	6.6	0.23	0.29
	1200	17.0	9.3	6.4	0.22	0.28
MapReduce	400	12.0	7.1	5.2	0.17	0.21
	600	12.7	6.9	5.1	0.18	0.21
	800	13.3	6.5	4.3	0.17	0.21
	1000	10.8	5.7	3.9	0.16	0.19
	1200	13.9	5.7	3.9	0.16	0.20
YARN	400	14.5	8.1	5.6	0.20	0.25
	600	12.4	8.3	5.8	0.18	0.23
	800	14.9	8.4	5.6	0.20	0.25
	1000	14.1	8.2	5.5	0.20	0.24
	1200	13.3	8.5	5.6	0.19	0.23
Hive	400	11.7	7.1	5.6	0.15	0.20
	600	10.2	7.2	5.5	0.14	0.20
	800	10.8	7.3	5.4	0.15	0.20
	1000	13.8	7.3	5.0	0.14	0.22
	1200	12.8	7.1	4.5	0.14	0.21
Storm	400	21.3	10.2	6.9	0.25	0.31
	600	19.7	9.2	6.4	0.22	0.29
	800	18.0	10.2	6.1	0.22	0.28
	1000	21.3	8.5	5.6	0.22	0.29
	1200	21.3	8.2	5.2	0.22	0.29
ZooKeeper	400	13.2	5.3	3.9	0.15	0.20
	600	7.9	4.7	2.9	0.12	0.15
	800	5.3	3.2	2.9	0.10	0.12
	1000	7.9	4.2	3.4	0.11	0.15
	1200	7.9	3.7	3.2	0.11	0.14

a smaller segment size for smaller systems and gradually increase the segment size to find the optimal value. We also observe that, when the logging statements are too far from each other (i.e., low log density), there may be more noises when we re-construct the execution path. The α value decides on how the logged classes are boosted. In general, we observe that the systems with a higher log density are more sensitive to the change of α value (i.e., the magnifier parameter given to the files found in the reported logs). In Figure 16a, the two systems with the highest log density, AMQ and ZooKeeper, have a large variation in their effectiveness as the α value varies and increases. Therefore, we suggest that future studies and practitioners may want to start with a smaller α when the log density of the system is larger. β, which serves as a magnifier for PathScore, decides on how the classes in the path are boosted. A higher β value attributes larger weight to the classes that are on the execution paths. For larger systems that have a low log density, such as Hive, we observe that the localization accuracy is the highest when the β value remains small. In Figure 16b, we observe that both the MAP and MRR values for Hive fall drastically as the β value increases. This may be that larger systems with lower log density value will have longer execution paths, which leads to a considerable amount of classes boosted by the β parameter. The localization accuracy decreases when too many classes are boosted (i.e., more noise). Thus, we suggest that future studies and practitioners may want to start with a smaller β when the system has a lower log density. In summary, we recommend future studies to set the initial parameter values small and increase them slowly (e.g., by 0.1) to find the optimal parameter values for the system.

4.7 Threats to Vaidility

External validity. Threats to external validity relates to the generalizability of our findings. To reduce this threat, we conduct our case study on eight large-scale open source systems that vary in size and infrastructures (i.e., data warehouse, realtime computation system, distributed file system). These systems are actively maintained and widely used. Although all the systems are Java-based, our approach is not limited to Java systems. We present our approach in a generic way that can easily be adapted to fit other programming languages. For uncovering the execution paths, another AST parser that fits the programming language should be used to replace Javaparser (e.g., *ast* module [3] for Python, and *cppast* library [2] for C++). Apart from execution paths, the mapping of the user-reported logs to the logging statements should be customized to fit the logging practice of the programming language. Future research is encouraged to be conducted on more bug reports from more systems written in different programming languages.

Construct Validity. Threats to construct validity refer to the suitability of the set of evaluation metrics that we use in this study. To reduce the threat, we use five evaluation metrics in our study: Recall@N, Precision@N, F1@N, MAP, and MRR. These metrics are commonly used in information retrieval and have been used to evaluate many prior fault localization techniques [101, 172, 173, 175]. We did not consider control flow analysis in our approach. In some cases, considering the control flow may provide more information. However, one challenge is that logs are relatively sparse in the code, so the accuracy of finer-grained control flow analysis will be low. Moreover, in previous chapter [38], we investigated the benefits and challenges of analyzing logs in bug reports. We found that developers may have made some code changes (i.e., the version that the user reported the issue is an older version and the code has changed), and the logging statements might be removed throughout the source code evolution (i.e., the user reported logs can no longer be found in the source code). Therefore, to reduce some noises caused by code evolution and the sparseness of logs, we decided to design the approach by generating the call graph and conduct the analysis at the file level.

Part II

Fault Localization in Continuous Integration

Chapter 5

T-Evos: A Large-Scale Longitudinal Study on CI Test Execution and Failure

Previous chapters of this dissertation have focused on addressing the lack of effectiveness in fault localization. Chapter 3 examined the use of user-reported logs as a debugging information to locate faults in the source code. Chapter 4 proposed an information-retrieval based fault localization technique, Pathidea, that leverages logs to enhance effectiveness. By employing this technique, developers can effectively identify the faulty locations when bugs occur in production, thereby minimizing their impact on users and enhancing quality assurance and reliability practices. However, resolving bugs in production is already too late, as they have already affected users. This gives rise to a specific research challenge: how can we detect problems before they impact users?

In modern software development, CI has been widely adopted for building and testing newer versions of the system, with automated execution of functional tests. For instance, in Apache Math, over 4,000 tests are automatically executed on a daily basis. As a result, CI practices provide valuable debugging information, including code changes, code coverage, and test results. These pieces of information have proven crucial for developers in understanding faults, and therefore presenting opportunities for future fault localization techniques to gain new insights.

Unfortunately, there is currently no CI benchmark available that incorporates continuous test execution and failure. This presents a challenge as developers lack knowledge on how to utilize such benchmarks, and researchers do not have access to this type of data within existing benchmarks. Building a comprehensive CI benchmark can require a significant effort. CI involves continuous data, and constructing every single commit in the version control history can be time-consuming. Additionally, if the system has been in existence for over 10 years, resolving compilation issues with outdated third-party dependencies can require extensive manual effort. Despite these challenges, the development of a large-scale CI benchmark would greatly benefit the research community by providing a realistic environment for evaluating and designing fault localization techniques.

To address the research gap, this chapter presents a large-scale longitudinal study on continuous test execution and failure, that aims to help gain a better understanding of the operational data available in CI. Moreover, we introduce T-Evos, a benchmark dataset for CI fault localization, which covers 8,093 commits from 12 Java open-source projects. The dataset comprises various operational data, including the evolution of statement-level code coverage for both passed and failed test cases, test results, build information, code changes, and corresponding bug reports. Through an initial analysis, we provide insights into the dataset, focusing on the characteristics of test failures in CI settings. Specifically, we examine the relationship between code changes and test failures, shedding light on potential directions for future automated testing research. Our findings represent a crucial initial step in comprehending the evolution of code coverage and test failures in a continuous environment.

In the next chapter, we proposed a changed-based fault localization technique by leveraging the CI data.

An earlier version of this chapter has been published at IEEE Transactions on Software Engineering (TSE) journal, 2022. Pages 2352-2365. Chen et al. [41]

5.1 Motivation

Software systems are continuously evolving. To ensure system quality, developers nowadays often execute regression tests in a CI setting. In CI, for every commit or a small set of code changes, developers would execute all the test cases to check whether the new code changes have caused any test failure. Therefore, when a test failure happens, developers may investigate the most recent code changes to identify the cause. Different from traditional software testing practices, which only run all the test cases before releases, CI allows developers to reduce the needed time to deliver the changes to the clients by catching test failures as early as possible and reducing integration overhead.

Prior studies have proposed various automated testing techniques that leverage code coverage or test execution information to assist developers with test failure diagnosis or repair. For example, researchers have proposed using code coverage information in failed test cases for tasks such as fault localization [17, 78, 95, 150, 183] and automated program repair [98, 176, 191]. These studies rely on the general assumption that the faulty code may be related to the execution path of a failed test case. However, most of prior techniques only consider snapshots of the project, while in CI settings, the code changes are integrated and tested continuously. The evolution data (e.g., recent code and coverage changes) may provide additional values that can help improve automated testing techniques and CI testing practices.

To help facilitate software testing research, many researchers have created several datasets and benchmarks. Just et al. [79] created the Defects4J benchmark that contains failed test cases caused by real faults, code coverage of the failing test cases, and the corresponding fixes. Lin et al. [110] presented the QuixBugs benchmark that contains both passed and failed test cases of known bugs. Le Goues et al. [100] presented the ManyBugs and IntroClass datasets that allow researchers to reproduce many real-world bugs and benchmark automated program repair techniques. Elbaum et al. [58] collected test execution result over time at Google. Although these datasets provide great benefits to the research community, they also have common limitations. The datasets only provide a clean snapshot of the projects: namely, selected bugs, their fixes, and the corresponding code coverage. Yet, there is no project evolution information, such as how the code coverage evolves and the relationship between recent code changes and test failures, which is crucial in CI settings.

In this chapter, we present T-Evos, a dataset that contains the evolution of test execution information across 12 Java projects over the course of 8,093 commits. For every commit, T-Evos contains the code coverage at the statement level (i.e., the specific lines that are covered), test status (i.e., pass or fail), build log, test execution stack traces, and code changes. To execute the test cases and generate individual test case coverage, we need to ensure that all the studied projects can successfully compile. To that end, we manually resolve the compilation issues, and then integrate the resolution into automation scripts. For each studied project, we also automate the manual process of adding project-specific configurations to collect code coverage. In total, we spent hundreds of hours resolving the issues that we encountered when executing the test cases. The size of the resulting dataset is around 3.3TB and took over 10 CPU years to generate. Different from prior datasets, T-Evos provides a fine-grained dataset on the evolution of test execution in CI settings, which may be used by future research that aims to leverage such continuous information to improve automated testing techniques.

5.2 Data Collection

CI is widely adopted in software evolution and most prior studies focus on test analysis at the release level or do not contain test execution information (e.g., code coverage) [58, 79, 100, 121]. However, there exists no dataset that collects fine-grained code coverage in a continuous environment (i.e., commit by commit over a period of time). Such dataset can benefit researchers in conducting various software studies (e.g., provide a realistic CI setting, or develop a new technique that leverages the development evolution). Therefore, our goals are: 1) to provide a dataset that contains continuous test execution of statement level at the commit-level, which could be used for future research and benchmarking; 2) to conduct an empirical analysis on test failure and resolution in a CI environment, which may inspire future research on how to better leverage the code evolution information (e.g., code changes in prior commits) to assist developers in various aspects, such as improving test failure diagnose, quality assurance, and CI practices (particularly in testing).

Although many projects may execute test cases in a continuous fashion, the recorded code coverage data mostly contains only general coverage results (e.g., the

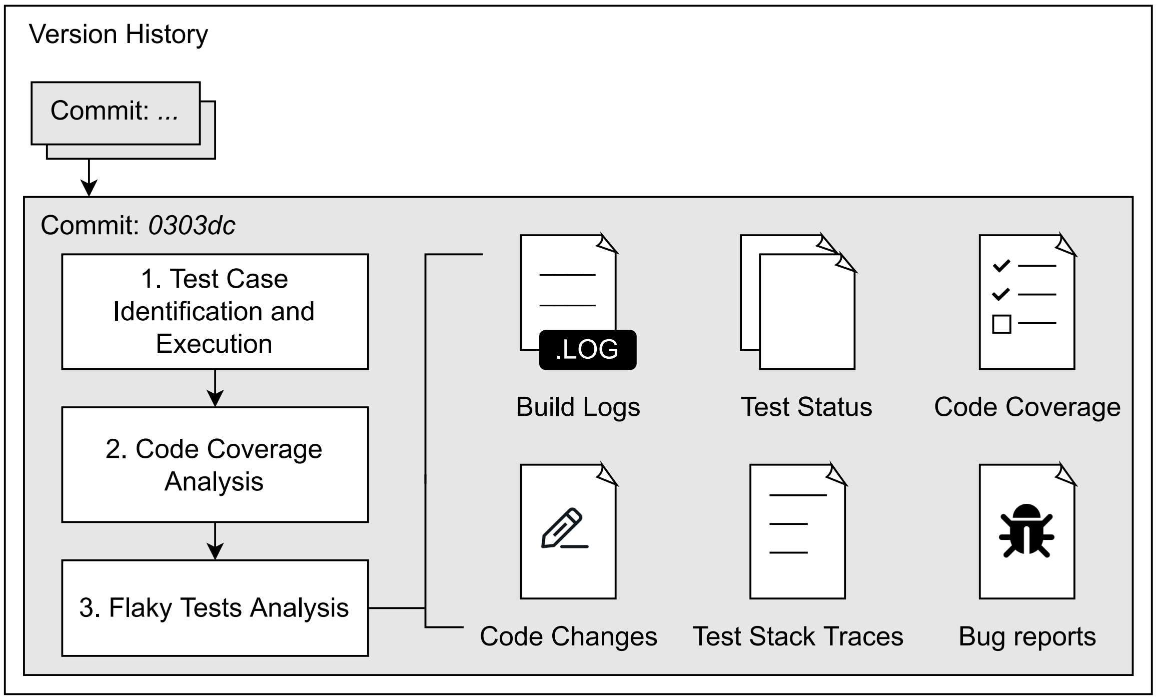


Figure 17: An overview of our data collection process and the collected data.

branch coverage) without knowing which code statements were executed. Moreover, such code coverage data is only kept for a small period of time [143]. Thus, to achieve the goals, we need to execute the test cases for every commit, collect the *individual code coverage* for each test case, and analyze the changes between commits. To collect the code coverage at the statement level for each commit of the studied projects, our approach consists of three steps, as illustrated in Figure 17. First, for every commit, we build (i.e., compile) the project with the Maven Surefire plugin [7] to identify a list of test cases for each studied project. We need to perform this step for every commit because there may be newly added test cases, and some test cases may be deleted or disabled [83, 145]. We then execute the identified test cases, and analyze the code coverage information using JaCoCo [6] for the individual test case. Finally, some test cases may be flaky (i.e., sometimes pass and sometimes fail). Flaky tests can introduce biases in the dataset when analyzing test failure [91, 120, 140, 205]. To reduce the bias, we detect and remove such flaky tests from the dataset [24].

The overall process is challenging and requires a significant amount of both manual and computing effort, because 1) a project may contain thousands of test cases and we need to run each test case for every commit to collect all the statement-level coverage information; 2) To make each test case run, we need to manually resolve many compilation issues. In the following subsections, we discuss our data collection, test execution process, and the associated challenges in detail.

5.2.1 Test Case Identification and Execution

Studied Projects. We select 10 projects that are also used in the Defects4J benchmark [79] for our study. We choose these projects because Defects4J is widely used and the research community is familiar with the projects. Note that, the data in Defects4J only contains snapshots of a subset of the test execution (e.g., the failing test cases before a bug is fixed), while our goal is to collect continuous test execution and failure data on a per-commit basis. To increase the diversity of the studied projects, our study includes additional projects (i.e., fastjson and junit4) of different software foundations. We query the top Java Maven projects on GitHub based on the number of GitHub Stars. The two projects, fastjson and junit4, have 23.7k and 8.2k GitHub Stars, respectively. The selected projects are well-maintained and contain active tests. We also manually verify that the selected projects can be successfully built and the test cases can be executed. In total, we conduct our study on 12 projects and Table 15 presents their details. For each project, we execute the test cases in all the commits (i.e., code changes) specified in the time range shown in Table 15. On average, each commit executes from 230 to over 4,000 test cases. We study the 1,000 latest commits at the time when our analysis is conducted. We chose to study 1,000 commits in each project because it is a relatively large number, but still feasible for us to manually resolve compilation issues. Based on the first and last commit of the 1,000 commits, we then compute the start and end date of our study. In other words, the time range refers to the time between the latest commit (at the time of our data collection) to 1,000 commits prior to it. We determine those commits by executing the “`git log -n 1000`” command. Note that for the project commons-cli, the repository only contains 965 commits at the time of the study.

Table 15: An overview of the studied projects, where *Total LOC* is the total lines of code, *Test LOC (%)* is the line of test code, and its percentage over total lines of code and *Average test cases* shows the average number of test cases per commit.

Project	**Total LOC**	**Test LOC (%)**	**Date range (Start, End)**	**Avg test cases per commit**	**Studied commits**	**Compiled commits**
commons-cli	12.2k	4.3k (35%)	(2002/06/10, 2020/09/19)	362	965	558
commons-codec	30.9k	14.6k (47%)	(2012/08/20, 2020/10/11)	799	1,000	709
commons-compress	65.8k	23.1k (35%)	(2017/01/07, 2020/10/13)	1,006	1,000	337
commons-csv	10.3k	7.2k (70%)	(2013/03/20, 2020/10/03)	230	1,000	936
commons-math	199.2k	76.0k (38%)	(2015/04/26, 2020/08/10)	4,200	1,000	551
gson	38.4k	15.2k (40%)	(2011/03/29, 2020/05/13)	1,103	1,000	942
jackson-core	50.8k	19.0k (37%)	2014/12/05, 2019/12/30)	320	1,000	453
jackson-dataformat-xml	564.7k	7.6k (1%)	(2014/11/12, 2020/10/16)	270	1,000	203
jfreechart	159.1k	40.0k (25%)	(2013/08/15, 2020/03/10)	2,477	1,000	439
jsoup	86.8k	9.4k (11%)	(2011/07/02, 2020/03/07)	532	1,000	990
junit4	37.9k	20.4k (54%)	(2013/02/05, 2021/02/13)	826	1,000	997
fastjson	354.6k	138.0k (39%)	(2018/09/03, 2021/04/05)	4,768	1,000	978

Handling compilation issues. Compiling the projects is not always straightforward. We perform the following steps to ensure that we can compile the projects and execute test cases in the studied commits. First, each project requires a considerable number of manual configurations to successfully compile. For instance, some projects may require a specific version of Java Runtime or Java Development Kit (JDK). Therefore, to automate the compilation and test execution process, we manually resolve all the compilation issues that we encounter and integrate the resolution in the automation scripts (e.g., changing JDK versions if needed). Second, some projects may not use JaCoCo [6] for code coverage analysis, so we need to manually configure the maven build script to add the JaCoCo dependency. Since the projects may have multiple modules, we need to identify all the maven build scripts when adding the JaCoCo dependence. Moreover, most of the studied projects use a handful of third-party libraries, so there may be other dependency issues that require manual fixes (e.g., some libraries are no longer available in the central Maven repository and need to be manually downloaded). We manually resolve the issues and update our automation scripts accordingly. Finally, even after the projects are successfully compiled, there may still be commit-specific configuration that prevents the test cases from executing. For instance, developers might refactor the structure of the configuration file (i.e., pom.xml) that specifies the dependencies. Thus, we need to modify our code accordingly to add the JaCoCo dependency. It took hundreds of hours of manual effort to resolve the compilation issues. Although we tried our best to resolve compilation issues, some commits may not be compilable even after we tried to manually resolve the issues due to reasons such as lack of dependencies. In total, we were able to successfully compile 8,093 commits.

Identifying test cases. After we successfully compile the projects, our next step is to identify a list of test cases that we need to execute. As found by Kim et al. [83], developers may disable some test cases during development. Therefore, we need to first identify the test cases that are active and will be executed in the CI process. Since all the studied projects use Maven as the build system, we track the list of active test cases by compiling the projects using the Maven Surefire plugin [7]. More specifically, when the project is built and compiled, the Maven *test* and *verify* lifecycles execute the unit and integration tests, respectively. Once the test cases are executed, the Surefire plugin then generates a *surefire-reports* folder containing the information of all the executed test suites (i.e., test class) in XML format. We then parse the XML files to identify the list of test cases (e.g., test methods annotated with `@Test` in the test class) that belong to each executed test suite.

Executing the test cases. Since we have 12 studied projects and each with hundreds of commits to compile and run test cases, we used six servers to speed up the execution. All the servers have the same hardware specification: a four-core Intel Xeon (Skylake, IBRS) CPU (2.10 GHz) and 5 GB of RAM. Four of the servers ran Ubuntu 20.04, while the other two ran Ubuntu 18.04. To collect the coverage of each individual test case, we need to run each test case one-by-one. Therefore, we implement scripts to automate the process of checking out a commit (i.e., in a Git repository), updating the Maven build file to include needed dependencies such as JaCoCo [6], compiling the project, and running each individual test case separately. We need to run each test case separately because JaCoCo only generates statement level coverage information (i.e., which specific lines a test case covers) when the test cases are executed one-by-one. To avoid dirty states that may affect the test result, we also automate test cleanup before and after running each test case. Note that we re-executed test cases in different servers to make sure that the nature of the failures is not sensitive to its environment or configuration (e.g., executing a test at a different time zone might cause a test to fail).

We created five worker threads in each of the six virtual machine servers. Each worker has its own sets of commits and projects to run to further parallelize the test execution process. Despite that the execution of the test case requires no re-compilation, Maven performs additional checks before the test execution (e.g., predefined coding style checks), which adds additional overheads to every test case execution. The building and testing time of each commit vary from 2 to 32 minutes, while the time of running a single test case is usually less than a minute. In total, our data collection and test execution took over 10 CPU years.

Distribution of compilable commits. Figure 18 summarizes the proportion of compilable commits by time interval. Most projects contain large time intervals covered by a large proportion of compilable commits. However, the overall proportion of compilable commits is smaller in three out of the 12 projects, namely, commons-codec, commons-compress and jackson-dataformat-xml. In those projects, the number of compiled commits remains zero for a specific time range, indicating that the

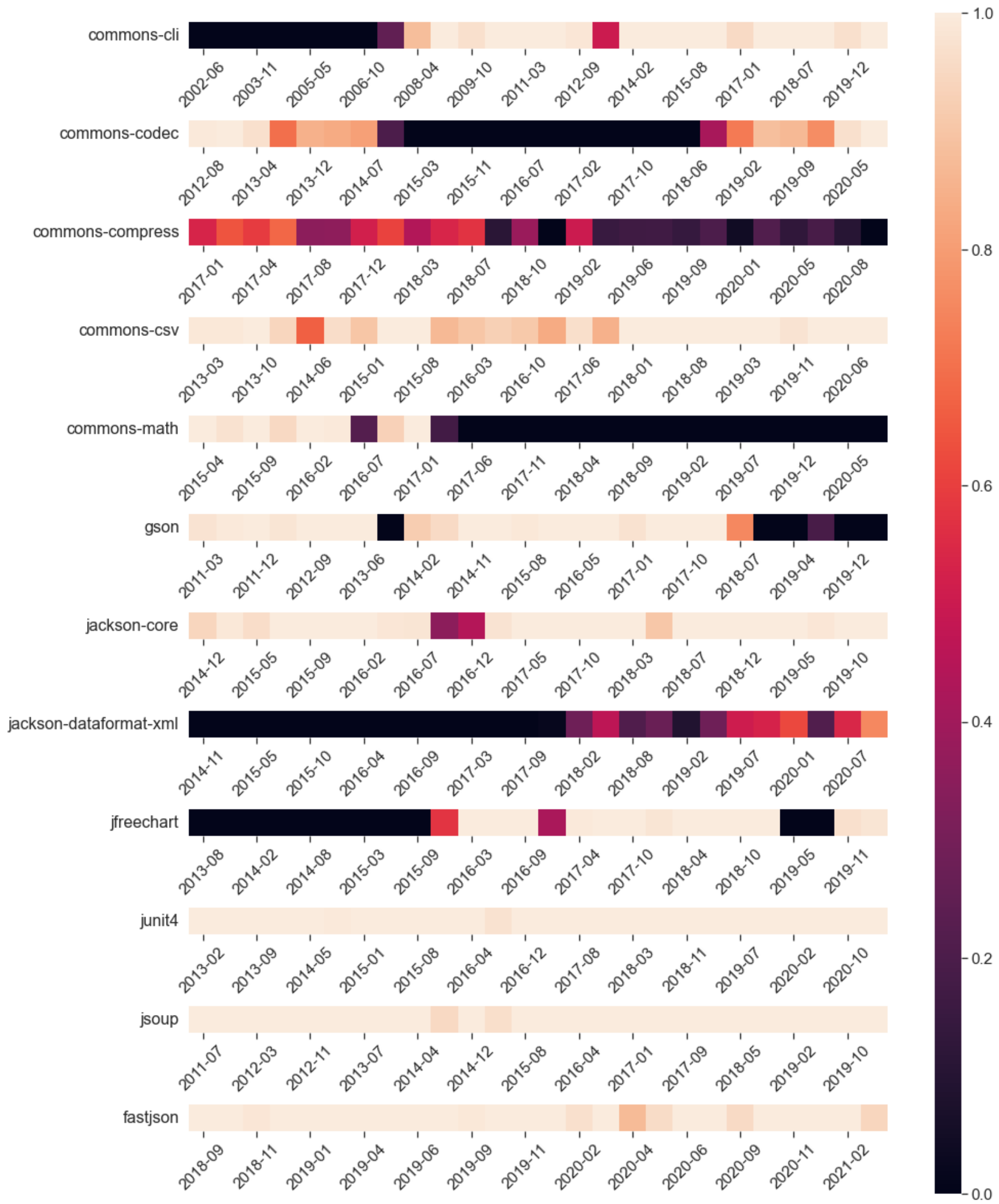


Figure 18: Proportion of compilable commits by time interval, where each square shows the proportion between the compilable commits and the uncompilable ones. The proportion ranges from 0.0 to 1.0, where values closer to 0.0 indicate less compilable commits, and values closer to 1.0 indicate more compilable commits.

compilation continuously fails for a sequence of commits. For the time range where the proportion of compilable commits are non-zeros, the average proportion is 0.8 for commons-codec, 0.4 for commons-compress, and 0.4 for jackson-dataformat-xml.

Overall, we find that on average 44% of the uncompilable commits fall into one single continuous time interval, which suggests that the uncompilable commits are grouped in the same time interval, rather than sparsely distributed at different time periods. In addition, we observed an average of 64 consecutive compilable commits between uncompilable commits. This represents a relatively large number of compilable commits in sequence, considering the average number of commits pushed to those projects in a year is 142. Therefore, we believe the impact of uncompilable commits on the continuation of our dataset is limited.

However, uncompilable commits can still impact our results in two special scenarios (as illustrated in Figure 19). First, if a compilation issue precedes or succeeds with test failure instances, then test failures can start or end on the uncompilable commit, which may introduce biases on our results. We show an example of such cases in Figure 19. Given that commit C_i is uncompilable, the test failure 1 might start either at commit C_i or C_{i+1}. To better evaluate the impact of such cases, we further investigate the number of uncompilable commits that precede or succeed test failures (i.e., continuous sequence of test failure instances). Note that this scenario only happens when there is at least one uncompilable commit before or after a test failure. Our investigation shows that, overall, only 4% of the uncompilable commits belong to this scenario, which is the upper boundary (since some commits may be uncompilable but have no test failure). Therefore, this scenario is less likely to bias our findings. Second, it is possible for test failures to exist entirely within uncompilable commits, and thus some of the uncompilable commits could have contained test failures that were dismissed in our analysis. We discuss this in Section 5.6 as a threat to our construct validity.

5.2.2 Code Coverage Analysis

We want to collect the detailed coverage of each test case in a continuous fashion. Hence, we conduct code coverage analysis on every commit.

Integrating Code Coverage Tool. We use JaCoCo [6] to generate the code coverage report. JaCoCo is one of the most popular code coverage tools that instruments

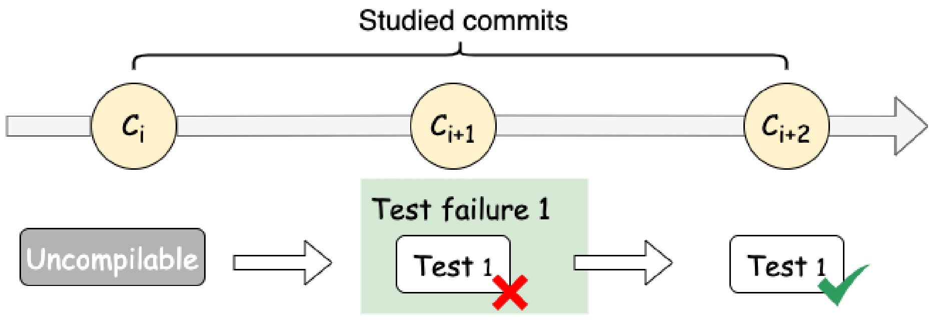


Figure 19: Example of uncompilable commit before a test failure.

bytecode to trace the execution during the test run. We integrate JaCoCo as a Maven plugin. While the integration is straightforward for most Maven projects, for the multi-module Maven projects, we need to manually modify the Maven configuration to collect code coverage. More specifically, when integrating JaCoCo into a module, the collected code coverage is limited to the classes of that module. However, in the case of a test case covering several different modules in the system (e.g., integration tests), the out-of-module class coverage will not be shown. Therefore, when some test cases cover multiple modules, we add an extra *report-aggregate* goal to the parent Maven build script (i.e., the pom file). Additionally, we add the JaCoCo plugin to every module. Every time a test is run, JaCoCo updates the covered classes in the coverage report of the module that they belong to. A coverage collector, which we implemented, will then parse and merge the coverage from every module.

Analyzing Code Coverage Results. Once the test is executed, we collect the code coverage of the individual test from the JaCoCo report at three different levels of granularity: 1) covered classes, 2) covered methods, 3) covered line of statements. In addition, we also collect the test status (i.e., passed or failed) and some general coverage metrics (i.e., branch coverage).

5.2.3 Flaky Tests Analysis

Prior studies [24, 91, 120] found that some test cases may be flaky. Namely, the test result may sometimes pass and sometimes fail, even if the code remains the same. To reduce the noise caused by flaky tests when analyzing test results, we remove flaky tests from our list of active test cases. We use DeFlaker [24] to compare the code change with the coverage of the failing test cases to identify the flaky tests. A failed test case is considered flaky if there is no overlap between the failure introducing code changes (i.e., the test case fails after the code changes) and the coverage of the failed test case. Namely, the failed test cases do not have overlap with the code changes. For every newly observed test case failure, we analyze whether any of the changed code is covered by test case, and if not, we mark the test failure as flaky.

5.2.4 Bug Reports Identification

To provide more information on the collected commits, we also identify the corresponding bug reports through the commit messages. The studied projects use the JIRA issue tracking system, where each bug report receives a unique bug report identifier (e.g., CLI-121). In addition, all 12 projects follow standards of including the bug report identifier in commit messages. Therefore, we use the git command "git show -s –format=%B" to mine the commit messages, and use regular expressions to capture bug report identifiers. For instance, for the studied project commons-cli, we use the regular expression "CLI-\d+". Then, we further leverage JIRA APIs [76] to verify that the detected identifiers are of type *Bug* rather than other types (e.g., *Feature request*). In total, we collect 221 bug reports.

5.2.5 The Collected Data

Figure 17 shows an overview of our data collection process and the collected data. At the end of our test execution and data collection process, we built 8,093 commits across 12 studied projects. The dataset includes: 1) all the build logs that were generated when compiling every studied commit; 2) the test status of every executed test case; 3) the code coverage of every passed and failed test case and how the coverage evolve overtime; 4) the code or configuration changes that developers made over time and their effect on the test result (e.g., a test case passes or fails after the change); 5) the stack traces generated by failed tests; 6) the associated bug reports. In total, the size of our collected data is 3.3 TB. Future studies may use our collected data to understand various testing practices in CI and help improve automated testing techniques.

5.3 Studying the Characteristics of Test Failures in CI Settings

As discussed in Section 5.2, we execute the test cases in over 8,093 commits to collect information such as code coverage and test status. To the best of our knowledge, we are the first to study and provide a dataset that contains commit-by-commit code coverage and failure information throughout the version control history. We believe that such dataset has great potentials for future research. Therefore, we conduct an analysis to provide an initial overview. Specifically, in this section, we provide an overview of the collected dataset and study test failure introduction and resolution across the studied time period. In particular, we answer the two following research questions:

- RQ1: How often do test failure instances occur?
- RQ2: How long does it take for developers to fix a test failure?

5.3.1 RQ1: How often do test failure instances occur?

Continuous testing in CI helps expose software faults that might negatively impact the functionality of the system under test. A prior study [26] found that test failures constitute the main reason why builds fail in CI. Labuschagne et al. [89] showed in their study that 18% of test executions in CI fail. However, previous studies only consider instances of test failure that last for exactly one commit, rather than a prolonged period of time. In such cases, we cannot have a complete picture on the prevalence of test failures throughout the project evolution, since the same test can fail repeatedly (e.g., spanning multiple consecutive commits). Therefore, in this RQ, we study test failures both in their prevalence of happening on the individual commit, and the prevalence of the same test failure lasting through consecutive commits.

To better understand the prevalence of test failures in the studied projects, we identify and track each test failure. To ease the explanation, we refer to a test failure that happens across at least one consecutive commit (i.e., without being resolved) as a *test failure*, and an instance of test failure that occurred in a single commit as a *test failure instance*. We further illustrate them using our example shown in Figure 20. Given that *Test 1* fails at commits C_{i+1} and C_{i+2}, a test failure is the consecutive

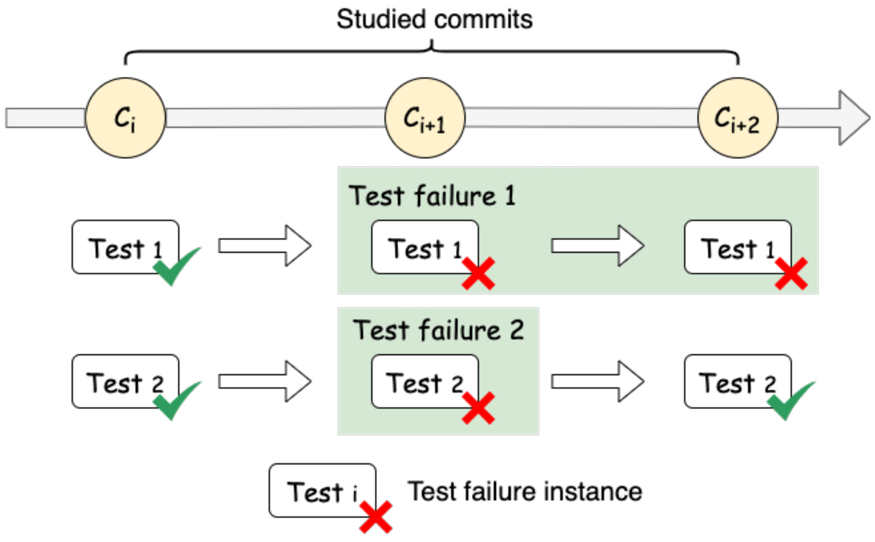


Figure 20: Example of test failures and test failure instances.

test failure happening across commits C_{i+1} and C_{i+2} (illustrated as *Test failure 1*), and the test failure instances are the individual failures happening on C_{i+1} and C_{i+2} (which may be the same test failure). We provide quantitative details of the test failures at two different levels of granularity: commits that contain failed test cases and individual failed test cases (i.e., a test case may fail multiple times before it is resolved). We further analyze the test failures to investigate how many are newly introduced and how many are resolved during the continuous process. Our findings provide a more comprehensive understanding of the test failures and their distribution over commits in the context of CI, and provide insights for future research.

A considerable number (34%) of the analyzed commits contain at least one test failure instance. Table 16 presents an overview of the prevalence of test failures for each studied project. In total, we compile and execute the test cases in 8,093 commits across the studied projects. Among these 8,093 commits, we find that 2,724 (34%) of them contain at least one test failure instance. The percentage of the commits that contain at least one test failure instance ranges from 2% to 78% among the projects. Our finding shows that although the studied projects, in general, contain a large number of failed commits, the prevalence of test failures varies noticeably among projects. Regression testing in CI aims to guarantee the quality of software with each commit. However, we observe different degrees of test failure prevalence across the studied projects. Future research may use our dataset to

Table 16: The total studied commits and the commits that contain test failure. *Test passing instances* show the number of times that a test executes without failure across all studied commits. *Test failure instances* show the number of failures that occur across all studied commits (a test failure may remain unresolved across multiple commits producing instances of test failure). *Test failures* show the number of new/resolved test failures (if a test case fails multiple times consecutively, it is counted as one test failure).

Project	**Total commits**	**Commits with failure (failure ratio)**	**Test passing instances** (in millions)	**Test failure instances**			**Test failures**	
				Total	Flaky	Non-flaky	New failure	Resolved failure
commons-cli	558	121 (22%)	0.17	183	53	130	60	60
commons-codec	709	340 (48%)	0.44	4,351	1,105	3,246	58	47
commons-compress	337	182 (55%)	0.29	3,812	3,697	115	192	191
commons-csv	936	23 (2%)	0.23	80	28	52	32	32
commons-math	551	95 (17%)	3.33	221	188	33	2	2
gson	942	253 (27%)	0.84	6,129	4,949	1,180	53	43
jackson-core	453	344 (76%)	0.48	2,672	162	2,510	99	97
jackson-dataformat-xml	203	153 (76%)	0.04	156	4	152	10	10
jfreechart	439	342 (78%)	1.02	1,253	635	618	27	13
junit4	997	321 (54%)	0.91	386	347	39	35	35
jsoup	990	42 (5%)	0.49	66	0	66	23	23
fastjson	978	333 (35%)	3.99	553	10	543	230	230
Total	8,093	2,724 (34%)	12.23	19,862	11,178	8,684	821	783

investigate the reasons that may cause such differences across projects.

We find that flaky test failure instances account for over 58% of the test failure instances. For non-flaky test cases, the same test failure might occur multiple times in the sequential commits. We find that flaky test failure instances are common in the studied projects. Out of the 19,862 test failure instances found across 8,093 commits, we observe 11,178 (56%) of them are flaky (shown in the *Test Failure Instances* column in Table 16). In contrast, 44% of the test failure instances are identified as non-flaky. Our finding shows that failures caused by flaky tests may be common. Note that a non-flaky test case may remain unsolved and fail multiple times across the studied commits. Therefore, we further study the number of new and resolved *test failures* in each project from the non-flaky test failure instances. The *New failure* column in Table 16 shows the number of newly introduced failures (i.e., a test case fails after a commit). Similarly, the *Resolved failure* column shows the number of resolved failures (i.e., the test case no longer fails after a commit). Overall, we find that there were only 821 new failures and 783 resolved failures. Given the small number of resolved test failures and the large number of test failure instances, our finding shows the same test failure might occur multiple times for the test failing repetitively in the sequential commits. Future studies may be needed to study these repeatedly failed test cases, whether flaky or not, and how they may affect software quality in general.

Most failures are concentrated in a small set of test cases. To study how the failures are distributed across the test cases, we count the number of *unique test cases* that result in failure across the studied commits. Note that we count a test case as one unique failing test case even if the test case has failed and been resolved more than once. We find that there are only 332 unique failing test cases (i.e., out of 8,684 non-flaky test failures) across all projects. This result implies that test failures are concentrated in only a small number of test cases, and most test cases never fail. After some manual investigation, we find that these test cases are often related to the main functionality or complex business logic of a project. For example, jsoup is a HTML parser and one of its core features is to parse HTML documents into *Document* Java objects. The same set of test cases in the test suite *DocumentTest* (which tests object conversation and parsing) fail multiple times during the studied period, although the fixes were applied at different locations and for different reasons.

One possible reason may be that some parts of the code undergo more changes, so the corresponding test cases are more likely to fail.

Our dataset provides a complete picture of how often test failures occur across commits in the CI context and the distribution of such test failures across commits and test cases. Future research may use the dataset to study how code evolution causes test failures and how to prioritize test execution. In addition, one interesting point as revealed by the results is that test failures are concentrated in only a small number of test cases, while most test cases never fail. Future research may further study the quality and effectiveness of the tests that never fail.

Test failure is prevalent in the evolution of the 34% of commits that contain test failures. Among the test failure instances, 44% are non-flaky test failure instances. We also find that many test cases fail multiple times across commits, and most failures are concentrated in a small set of test cases.

5.3.2 RQ2: How long does it take for developers to fix a test failure?

In the CI context, the detection of test failures presents a compensatory benefit of continuous testing, especially when failures detect real faults. However, prior research found that test failures, despite being detected, might not be resolved for various reasons. Beller et al. [25] found that up to 30% of the failing tests are not repaired immediately although developers detect them directly in IDEs. Rogers [148] found that sometimes, developers might allow known test failures into CI, as long as those failures are resolved by the end of the development iteration. However, it is unknown how long test failures last in evolutionary settings. Therefore, in this RQ, we study the resolution time it takes for developers to fix a test failure, and how failures are distributed at different resolution times.

To calculate how long it takes for developers to resolve a test failure, we analyze every test failure in the version history and look for the *failure-introducing commit* (i.e., the first commit in which the test failure occurs) and the *failure-resolving commit* (i.e., the commit where the test failure is resolved). Then, we compute the time difference between the failure-introducing commit and the failure-resolving commit. Note that if the same test case fails again after a resolution, we consider it as a

Table 17: The resolution time of test failures.

Project	Resolution time					
	Min	Max	Mean	Median	< 1 day	> 7 days
commons-cli	< 1 day	63 days	4 days	< 1 day	77%	12%
commons-codec	< 1 day	102 days	2 days	< 1 day	87%	2%
commons-compress	< 1 day	3 days	< 1 day	< 1 day	83%	0%
commons-csv	< 1 day	2 days	< 1 day	< 1 day	97%	0%
commons-math	< 1 day	< 1 day	< 1 day	< 1 day	100%	0%
gson	< 1 day	15 days	4 days	1 day	49%	23%
jackson-core	< 1 day	53 days	2 days	< 1 day	60%	4%
jackson-dataformat-xml	< 1 day	< 1 day	< 1 day	< 1 day	100%	0%
jfreechart	< 1 day	< 1 day	< 1 day	< 1 day	100%	0%
junit4	< 1 day	22 days	4 days	1 day	43%	26%
jsoup	< 1 day	17 days	< 1 day	< 1 day	96%	4%
fastjson	< 1 day	16 days	< 1 day	< 1 day	88%	1%
Average	< 1 day	21 day	1 day	< 1 day	75%	5%

different test failure as developers have already made changes to resolve the failure.

While most test failures are resolved within one day, some may require more than a week to resolve. Table 17 shows the resolution time of the test failures. The mean resolution time across all the studied projects is *1* day, while the average of the median resolution time is less than *1* day. In addition, 10 out of 12 studied projects have more than 75% of the test failures resolved within a day. More than one third of the studied projects do not have any test failure extending for more than 7 days. On average, across the projects, more than 75% of the total test failures are resolved within a day, and only 5% of the failures persist more than 7 days. While our findings show that developers are actively trying to resolve the test failure once they occur, there are still some exceptional cases.

As shown in Table 16, while most of the studied test failures (783 out of 821) are resolved, there are still 38 test failures that were introduced but never resolved. Thus, we further studied those test failures, and calculated how long they are lasting. We find that most of the failures happen near the end of the studied periods of the projects. Around half of these 38 test failures were introduced no more than three days before the end of the studied periods. The maximum time difference between their introducing time and the ending period of the studied projects is no more than 23 days. In short, it is possible that the test failures were not resolved due to the time periods that we analyzed did not include their fixes.

By calculating the mean resolution time, we observe that most test failures, if ever resolved, are resolved within one day. Even when they are not resolved, they are new failures that were introduced for no more than 21 days. However, some projects contain a maximum test resolution time that is significantly longer than the majority of the dataset (e.g., several weeks compared to within one day). Our dataset identifies the test failures that may have different characteristics, which causes the fixing time to be much longer. Future testing research may use our dataset to better understand the characteristics of such long-lasting test failures, and further assist developers with improving code quality.

While most of the test failures are resolved within one day, we still find some failures that take more than a week to resolve. Future research may use our dataset to study the characteristics of the test failures and understand the reasons for such differences.

5.4 Studying Code Changes and Their Relationship With Test Failures

Our collected dataset includes the complete code coverage evolution at the statement level and the code changes that developers made throughout the version control history. In this section, we study the changes that developers made when introducing/resolving test failures, and how code coverage change before and after fixing the failure. Our findings provide an understanding on the relationship between code coverage and test failure, and provide insights and a new dataset for future automated testing research such as fault localization. In particular, we answer the two following research questions:

- RQ3: How does the code change when the test failure first happens?
- RQ4: How does the code change when the test failure is resolved?

5.4.1 RQ3: How does the code change when the test failure first happens?

Prior research studied the characteristics of test failures in relation to the source code. Pinto et al. [145] showed that test execution might fail for three major reasons including removal of the required source class or method, catching runtime exceptions, and assertion violation. Marsavina et al. [122] further examined, in the case of introduction of test failure, how production and test code were changed. They found that many failed test cases may be added by developers while working on production code. In this RQ, we study the changes that developers made when introducing test failures, and how code coverage changes (which shows the dynamic execution information) before and after introducing the failure. Studying the change in code coverage may give insights on why test failure happens and provides an understanding of the relationship between code coverage and test failure.

To better understand the evolution of code coverage when the test failure first happens, we record the code coverage of each test failure. We provide quantitative analyses to study the test failure in CI settings where code coverage is one of the few pieces of information available to developers. First, we investigate how developers introduce test failures. In particular, we derive categories of code changes based on the types of files changed, and present the number of test failures belonging to each category. By studying what type of code changes might introduce the test failures, future research might inspire from our findings to better help developers with test failures. Then, we evaluate the impacts of test failures on code coverage. In other words, when test failures occur what can we observe from the code coverage? We present the coverage change based on the total line coverage increased and decreased.

To investigate which files were modified, for each test failure, we conduct quantitative analyses on the failure-introducing commits. We categorize the files with *.java* extension as either source code or test files, and otherwise as non-code files (e.g., data and configuration files). We use the list of active tests identified in Section 5.2.1 to further distinguish between source code and test files.

To investigate how the code coverage changes after failure introduction, we first compute the per-method line increased and decreased from the failure-introducing commit and the commit before it. Rather than checking whether the overall coverage increases or decreases, we calculate the individual line increased and decreased in each covered method. In this way, we can obtain finer-grained results and identify the case where the line coverage has changed but the overall coverage remains the same. Then, we sum up the per-method line increased and decreased from all the covered methods to get the total lines increased and decreased. We quantify the coverage change based on the total line increased and decreased. As the line coverage counts the lines of code without including the conditional statements (e.g., `if` and `while`), we further compute the branch coverage in the case where we observe no difference in line coverage. Similar to the line coverage calculation, we calculate the individual branch increased and decreased from each covered method.

We find that many failed test cases may be added by developers while trying to resolve an issue. Figure 21 shows the types of files that were modified in failure-introducing commits. Overall, we find that many failure-introducing changes modify both source code and test files (52% on average), while just 35% modify only source code files. We also find that many of the commits that modify both source code and test files may be adding new test cases to address newly reported bugs. For instance, in fastjson, which is a JSON processor for data streaming, 63% of its failure-introducing commits (i.e., 147/233) modify both test and source code files. We find that nearly 99% (146/147) of those modifications added new test files to the project. Those new test files either contain or are named after some bug ID (e.g., *Issue2133*), which may indicate that the test files were added when developers were trying to resolve the bugs. Overall, we find that 18% (150/821) of the test failures in the studied projects either modified or added test files that contain the failed test cases. Moreover, 13% of the failure-introducing commits modify only test files. In other words, developers may be fixing a bug while modifying or adding test files.

Automated testing techniques (e.g., fault localization) use the information of running test cases to assist developers in identifying bugs. Our finding shows that, if a dataset is collected without considering such newly added test cases that are used to resolve the faults, there may be potential biases in the result. Namely, developers were already trying to address the fault by adding new test cases based on their knowledge. For instance, if such newly added (and failing) test cases were used for evaluating techniques such as fault localization, one may be implicitly using developers' knowledge of the faulty location to assist the automated techniques. In contrast, our dataset specifically shows whether a failure-introducing commit modifies either

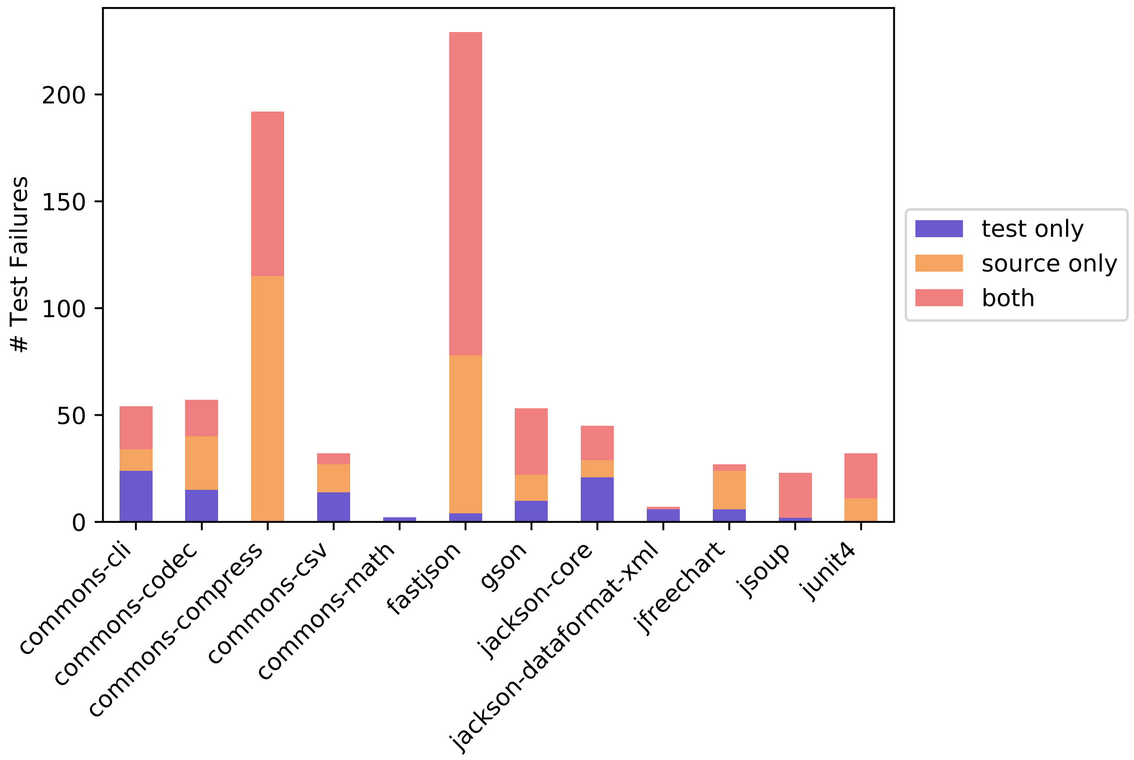


Figure 21: The categories of the files that were modified in failure-introducing commits.

Table 18: Average code coverage before and after the failure-introducing commit.

Project	Method level		Line level	
	Increased	Decreased	Increased	Decreased
commons-cli	10	12	36	68
commons-codec	4	8	18	56
commons-compress	5	16	16	82
commons-csv	2	26	7	136
fastjson	15	27	329	511
gson	0	28	1	511
jackson-core	15	17	89	101
jfreechart	7	18	49	119
junit4	13	29	54	124
Average	8	20	67	190

source code, test file, or both. Future studies may use our dataset to filter out those newly added or modified test cases (e.g., the test is being modified by developers to capture the fault, so it is failing) to better evaluate automated testing techniques such as fault localization.

Test execution may stop prematurely when it encounters a failure in test files (e.g., assertion statement is invalid), which results in a decrease in code coverage. Table 18 shows the changes in the average code coverage (i.e., averaged across all failure-introducing commits in a project) before and after the failure-introducing commit. We report the changes in both method-level and line-level coverage. To note that we perform our coverage analysis on 468 (57%) new failures instead of all the new failures (821), as not all failures have code coverage change between the failure-introducing commit and the prior commit. In addition, there were some test cases that did not exist before the failure-introducing commit (i.e., new test cases).

Overall, we find that the failure-introducing commits decreased more coverage than increased. For example, the average decreased covered lines are 190, while the increased covered lines were only 67. We notice that since we generate the coverage based on the executed test code, when a test case fails, the part of the test code

Table 19: Test failures with and without coverage change for different introduction categories.

Test Only		**Source Only**		**Both**	
Changed	Unchanged	Changed	Unchanged	Changed	Unchanged
26	38	140	74	159	31

that is beyond the failing location will not be executed and will not be included in the coverage data. If a test case fails at the beginning of the execution, the code coverage may be empty in some cases (e.g., the pre-test check fails). For example, we observe a serialization test (i.e., *Issue3436#test_for_issue*) from fastjson, which fails with a *JSONException*. The failure happens at the beginning of the test case, so the remaining test code is not executed and no code coverage report is generated.

Our dataset provides both the code coverage before the failure-introducing commits and the code coverage of the failed commit. Future studies may use our dataset to study how the coverage evolution data may help locate where a fault is introduced when the coverage of the failed test cases is incomplete.

Around 6% of the failure-introducing changes do not change the line coverage, nor the branch coverage. When analyzing the coverage change with different categories of failure-introducing changes, we find that around 44% (200/468) of the failure-introducing changes do not change line coverage, even though the changes modify source code or test files, as described in Table 19. While those failures do not have changes on the line coverage, 87% (174/200) of them do have a different branch coverage. The remaining 13% (26/200) of the test failures do not involve coverage change at all (e.g., the failed assertion statement is the last line in the test case and the test failure is due to incorrect test setup or variable value).

As we found, the coverage (i.e., line or method coverage) might not always change when the test failures are first introduced. Our dataset and findings may inspire researchers and practitioners to further investigate the prevalence of software regression and refactoring that caused test failures.

Many failed test cases may be added when developers are trying to resolve an issue, and 13% of the failure-introducing commits only modify test files. We also find that, compared to the prior passing commit, the code coverage generally decreases in failure-introducing commits.

5.4.2 RQ4: How does the code change when the test failure is resolved?

Understanding the changes that developers apply when resolving a test failure may help improve future automated testing techniques. In this RQ, we study the types of files that are modified in failure-resolving commits, and how do code coverage changes when the failure is resolved. Specifically, we answer the RQ by answering two sub-RQs: *What types of changes do developers apply when fixing a test failure?* and *Are there overlaps between the code coverage of the failed test cases and the failure-resolving location?*

RQ4.1: What types of changes do developers apply when fixing a test failure?

We study the test failure resolution in two steps: 1) we investigate which files were modified during the resolution, and then 2) we study how the coverage changes after failure resolution. To investigate which files were modified, for each test failure, we examined the changed files in failure-resolving commits. Same as RQ3, we categorize the files with *.java* extension as either source code or test files, and otherwise as non-code files (e.g., data and configuration files). To investigate how the code coverage changes after failure resolution, similar to RQ3, we quantify the coverage change based on the total line increased and decreased. In other words, we sum up the per-method line added and deleted from all the covered methods to get the total increased and decreased lines. Knowing which files were modified during the resolution and how the coverage changes may help better understand how do developers fix a test failure and improve automated testing techniques.

Developers often resolve test failures by modifying non-code files (21%) or only test files (14%). Figure 22 shows the distribution of the files that were modified in failure-resolving commits. Overall, we find that 34% of the failure-resolving commits modify both test and source code files, and 31% of the commits modify only

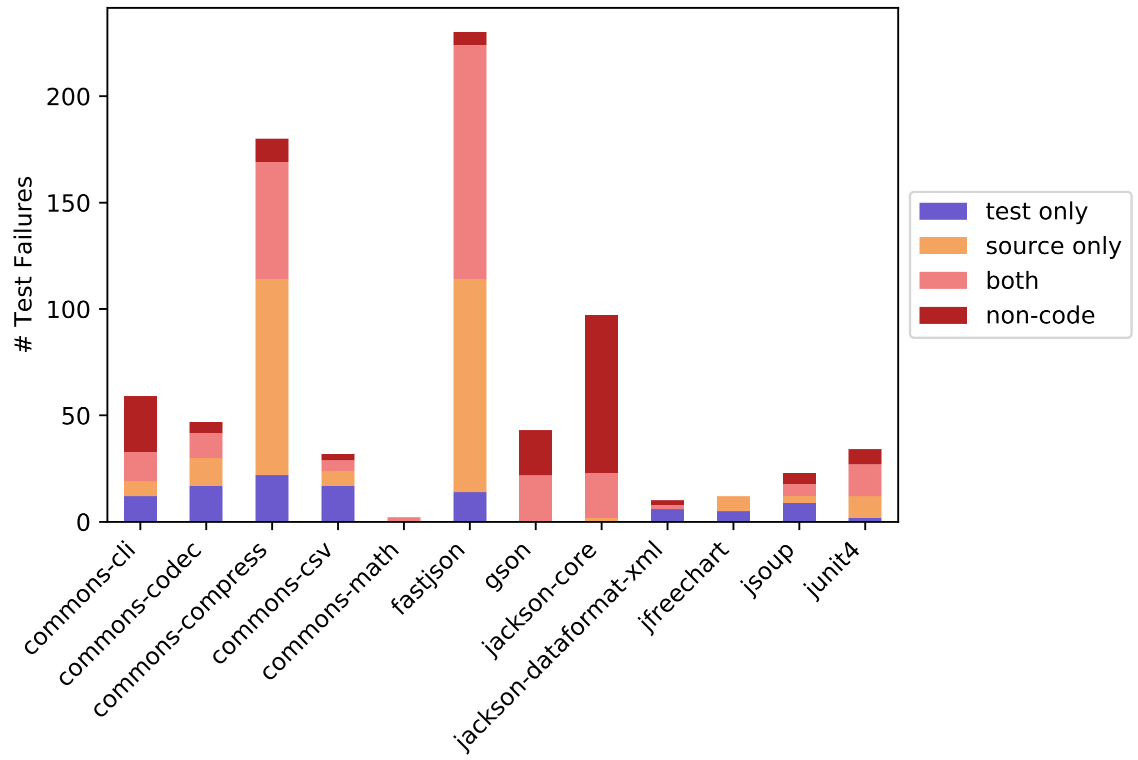


Figure 22: The types of files that were modified in failure-resolving commits.

the source code files. We also find that developers commonly modify non-code files (21%), or only the test files (14%) in failure-resolving commits. The types of modified files vary across the studied projects. For instance, in fastjson, we observe 91% of the failure-resolving commits modify either only source code files, or both test and source code files. Only 6% of failure-resolving commits modify test files. However, in another project (i.e., commons-codec), we observe that more than 36% of the test failures are resolved by modifying only test files. In other projects, such as gson and jackson-core, developers might also only modify the non-code files, such as data or configuration files, to fix the test failures. We observe more than 49% and 76% of the failures are fixed through non-code changes in gson and jackson-core, respectively. Our findings show that the resolution of the test failure does not only limit to source code files, but also the test and non-code files. When studying test failures, future research might consider non-code files as a potential fix for failures, as well as the test files that might already contain some issues related to the failures.

Around 19% of the failure-resolving changes do not alter code coverage,

Table 20: Test failures with and without coverage change count for different resolution categories.

Test Only		**Source Only**		**Both**	
Changed	Unchanged	Changed	Unchanged	Changed	Unchanged
71	22	183	46	130	21

even though the changes modify source code or test files. In Table 20, we show the number of resolved test failures with and without coverage change. We find that there is a non-negligible number of failure-resolving changes that did not change code coverage. 23.7% (22/93), 20.1% (46/229), and 14.0% (21/151) of the test-resolving commits did not change code coverage in each resolution category (i.e., modify only test files, only source code files, or both), respectively. To note that we perform our coverage analysis on 473 (60%, out of 783) among all the resolved failures because the code coverage information may not be available for the failure-resolving commit or the commit before (i.e., the failing commit). Some test cases may be removed in the failure-resolving commit (as found by previous research [82, 145]), and there may be no coverage for some failing test cases (e.g., a test case fails early when no coverage is available yet). In addition, as discussed above, developers may modify non-code files (e.g., test configuration or data files) that are not visible through code coverage. The data files may be used in test cases to verify the expected test output, and some configuration issues may cause test cases to fail.

To better understand in what situation do failure-resolving commits have no code coverage change, we manually studied the test failures. We performed our manual study on all 310 resolved test failures in which the failure-resolving commits did not introduce any code coverage change. We manually examined the code changes applied to the failure-resolving commit, and the code coverage. This information provides hints on the relationship between the changed code and test execution. By leveraging this information, we uncovered a list of categories of code changes that may result in no code coverage change. Then, we systematically verified the assigned categories. In case of any discrepancy, we further carried on discussions to reach a consensus. We observed four main types of changes: 1) *test assertion changes*, 2) *method parameter changes*, 3) *invisible dynamic changes*, 4) *conditional statement* *changes.* Each of these four types of changes can result in resolved test failures without coverage change. We briefly present the four types of changes as follows:

Test assertion changes: changes performed on assertion statements in the test cases. As only the assertion statement is modified, the dynamic execution is unchanged. For instance, one test failure captured in fastjson modified the expected string value in the assertion statement `assertEquals(expected_str, actual_str)` to adapt to recent changes in the source code that necessitate a new expected string value in the test.

Method parameter changes: changes that modified the parameter value inside method invocations. An example of such is correcting a wrong parameter to fix the test failure.

Invisible dynamic changes: changes that modified the flow of the third-party code which is not reported in the coverage report. For example, modifying the string format of a datetime object before passing it to a special third-party json datetime formatter (where the software will use the output).

Conditional statement changes: changes that modified the conditional statements (e.g., `ifelse` or `while`) to fix logical errors. For example, developers may add a new condition (e.g., from `while(condition1)` to `while(condition1 && condition2)`) to fix a logical error. However, since the change does not add new code and the test may not be updated, there is no coverage change.

Our findings show that the resolution of test failure is not always visible through code coverage change. Moreover, the system dynamic behavior may change, even if developers fix the test failure and the code coverage remains the same. Future research may use our dataset to further evaluate whether existing automated testing techniques (e.g., fault localization or automated program repair) need to be tailored to work on this type of issues.

RQ4.2: Are there overlaps between the coverage of the failed test cases and the failure-resolving location?

Many automated testing techniques (e.g., fault localization) leverage code coverage to help developers identify faulty code [17, 78, 95, 150, 183]. The general assumption of such techniques is that the faulty code is on the execution path of a failed test. In this RQ, we wish to investigate whether the coverage of the failed test can provide useful insights on the failure-resolving location. By understanding this, we may provide insights for future research.

Here, we analyze the results of 473 test failures that have the code coverage information. We first extract a list of covered files (f_{covered}) from the coverage information of the commits that have failing test cases. Then, we compare f_{covered} with the changed files in failure-resolving commits (f_{changed}) and compute an overlap between them. We calculate the percentage of the overlap based on the total number of changed files as follows:

$$\text{Percentage Overlap} = \frac{\#f_{\text{covered}} \cap \#f_{\text{changed}}}{\#f_{\text{changed}}} \tag{13}$$

In all the studied projects except one, many modified files in the failure-resolving commits do not have overlap with the executed files in the failing commit. Table 21 shows the overlaps between f_{covered} and f_{changed}. We find that a notable number of the changed files are not on the execution path covered by the test cases in the failure-resolving commit. The average percentage overlap across the studied projects is 66% (except for jfreechart, where the overlap is 100%). We observe that the reason for a relatively low overlap may be that the coverage of failed test cases may be incomplete if the test case fails during the early execution stage. Such incomplete code coverage might present a limitation to existing automated testing techniques that analyzes code coverage (e.g., fault localization). We also find a non-trivial number of instances (i.e., 10% of the overlaps) where the failure-resolving commits only modify the failed test case and the coverage did not change. By examining these instances, we observe that some of the test failures were resolved in unconventional ways. For instance, as also reported in a prior study [83], developers may disable the assertion statements in the test code (e.g., the code is commented out), but the issue remains unsolved.

Developers often resolve test failures by modifying non-code files (21%) or only test files (14%). Even when modifying source code or test files, around 19% of the failure-resolving changes do not alter code coverage. In addition, in all studied projects except one, we observe that a notable number of the changed files are not on the execution path covered by the test cases in the failure-resolving commit.

Table 21: Overlaps between the coverage of the failed test case and the failure-resolving location.

Project	commons-cli	commons-codec	commons-compress	commons-csv	fastjson	gson	jackson-core	jfreechart	junit
Overlaps(%)	64	76	65	90	71	14	58	100	58

5.5 Future Research Directions

As described in Section 5.2, we executed the test cases on consecutive sequences of commits in 12 studied projects. We collected the data and made it publicly available [12]. Our data is 3.3TB and may be used in future testing research or benchmarking different techniques. Below, we discuss some possible research directions using our data.

Studying and understanding quality issues in test code: In RQ4, we found that developers may only modify test files to resolve test failures. We also found that some test cases may be failing already when they were first added. Our findings indicate that there may be pre-existing issues causing test cases to fail. Future studies may use our dataset to study the quality of test code throughout software evolution.

Studying code coverage evolution: We observed that some test failures do not involve any coverage change either in the failure introduction or the failure resolution in all studied projects in RQ3 and RQ4. Future studies may use our dataset to study the evolution of such test failures and their coverage throughout their entire lifetime. We observe that the time of resolving test failures and how the failures are resolved is project-specific and our study provides the first-step insights towards studying the relationship between test failure and code coverage. Future studies may investigate whether there exists any correlation between the increased coverage and faster failure resolution time. It is also interesting to investigate whether better coverage helps to detect and resolve the failures.

Studying the roles of test cases that failed repetitively: In RQ1, our results show that most test failures are concentrated in a small number of test cases. Namely, the test failure may be resolved but the same test case may fail again multiple times during the evolution due to other reasons. Our dataset may be used to study the characteristics of such test cases, and whether it is possible to help developers enhance the quality of both the test code and the tested source code to prevent future bugs.

Studying how code change history may assist fault localization and program repair techniques: In RQ3, our findings show that test execution may stop prematurely when it encounters a test failure, which might result in a decrease in code coverage. As there exist many automated testing techniques that leverage code coverage (i.e., fault localization and program repair techniques), future studies may use our dataset to propose new or enhance current techniques. Future studies may mine the past code coverage data to complement the decrease in coverage. In addition, future studies may analyze the code changes that we collected in the dataset and examine how recent code changes contribute to test failures. Future studies might even use our dataset for benchmarking automated testing techniques.

Studying how build configurations may affect test result: Our dataset contains all the build logs that we collected when compiling and executing the test cases in the studied commits. As we found in RQ4, some failure-resolving commits only modify non-code files such as configuration files. Future studies may analyze the build logs to study how the quality of the build scripts contributes to test failures.

5.6 Threats to Validity

External validity. Our studied projects are all open source and implemented in Java, so our findings may not be generalizable to other projects. To minimize the threat, we try to choose the projects that are well studied in the research community or are commonly used by many systems around the world (e.g., junit4). Future research may consider collect similar datasets for projects that are implemented in other programming languages and verify with our findings. We base our findings on the data in the studied time range from each project. In some cases, studying a different time range may lead to slightly different results. To generalize our results as much as possible, we select a large range of commits (1,000 commits per studied project). This time range of commits was chosen because it is a relatively large number but still feasible to manually resolve compilation issues. Note that for some projects (e.g., commons-cli), the total number of commits is less than 1,000 commits as of September 2020. In total, we compile and execute the test cases in 8,093 commits across the studied projects.

Construct validity. In our study, the starting date varies from 2002 to 2018, which implies that the studied projects may be at different stages of development. While this can increase the diversity of the studied systems, it can also be a threat to the construct validity of our results. Nevertheless, our findings are consistent in general. We encourage future research to leverage our dataset and further explore the differences among the projects and their relationship between different time ranges. When analyzing code coverage, we notice that some test cases may have empty coverage. To ensure the validity of our results, we re-run all the failing tests in a new environment. Then, if there is still any test failure without coverage, we randomly select some tests to run manually. Based on our manual study, we observe that the code coverage might not be generated when the test case fails in the early stage of the execution where no source code is yet covered. Future studies should consider this situation when applying automated testing techniques that leverage coverage information. Even though we tried our best to compile and run the test cases in the studied projects, some of the excluded commits (e.g., we cannot compile) may still be compilable. Nevertheless, our dataset still includes over 8,000 compiled commits and test execution results, which we share with the research community. There are uncompilable commits between sequences of compilable commits, which might affect the continuation of our dataset. In Section 4.3, we conducted analyses and showed that, despite the presence of uncompilable commits in some projects, in general, our dataset contains long and consecutive sequences of compilable commits. We encourage future studies to further investigate those compilation issues.

There are many factors that can influence the compilation of the commits (e.g., availability of past dependencies, inappropriate Java Development Kit version). For instance, most of the studied projects use a handful of third-party libraries, so there may be other dependency issues that require manual fixes (e.g., some libraries are no longer available in the central Maven repository and need to be manually downloaded). We manually resolve the issues and update our automation scripts accordingly. We spent our best effort to manually resolve the compilation issues. To provide better confidence on the accuracy of our results and allow continuous improvement on our dataset, we made all the data that we collected publicly available [12], as transparent as possible with this chapter. Due to the size of the code coverage data, we published it separately in a Zenodo repository [13].

5.7 Related Work

Testing in Continuous Integration. Some prior research [26, 89, 148] have aimed to study CI testing practices. Rogers [148] found that developers might allow known test failures into CI, as long as those failures are resolved by the end of the development iteration. In this chapter, we also found that, despite the presence of CI environment, there are a non-negligible number of test failures that persist over multiple days. Beller et al. [26] observed that testing constitutes the main reason why builds fail in CI, with test failures responsible for 59% of broken builds. Labuschagne et al. [89] showed that 18% of test executions in CI fail and that 13% of these test failures are flaky. In their study, they categorized the resolution of failed tests into three categories: code fixes, test fixes, and combination of code and test fixes. In this chapter, we found that non-code changes (e.g., data files) might also constitute the resolution of test failure.

Previous studies [122, 145] also discussed some characteristics of test failure by exploring how the test code evolved over time. Marsavina et al. [122] discussed co-evolution patterns of production and test code. Our study further examines how code coverage, production and test code change upon the introduction of a test failure. We find that, compared to the prior passing commit, the code coverage generally decreases. They also found that, in some studied projects, up to 47% of all code changes are performed on test files. In our manual study, our goal is to study how the code changes when developers resolve the test failures. We find that 48% of the code changes modified test files when resolving test failures (i.e., 34% modifying both test and source code files, 14% on only the test files). Pinto et al. [145] studied the effect of newly added tests on code coverage, and found that, on average, 56% of the newly added tests do not change the previous code coverage (i.e., branch coverage). In this chapter, we observe that test failures might impact code coverage, since the test execution may stop prematurely which results in a decrease in code coverage.

Testing Practices. Prior studies conducted empirical studies on test code, and proposed suggestions to help improve testing practices [67, 80, 82, 114]. Just et al. [80] evaluated the developer-provided tests (from version history) and the user-provided tests (from bug reports) on fault localization and automated program repair techniques. They found that developer-provided tests contain more information to detect bugs, as the tests are specifically tailored to cover the buggy code. Kim et al. [82] conducted an empirical study on the evolution and maintenance of test annotations. They found that developers may use test annotations to remove or disable failed tests. Liu et al. [114] discussed the potential bias of over-fitting issue in automated program repair, where test failure may be resolved without actually fixing the bug. Our results confirm this finding as we found that developers may disable the assertion statements to make failed tests pass again. Hilton et al. [67] evaluated the coverage change between project revisions and assessed the impact of code changes on test quality. Zaidman et al. [200] studied the co-evolution of source and test code. They investigated the test coverage evolution based on its relation with test-writing activity. Beller et al. [25] suggested in their study that the production and test code have some tendency to change together, but the production code change does not always involve test change and vice versa. Catolino et al. [35] surveyed developers to understand how assertion density relates to the quality of test code. In this chapter, we present the test result and test coverage in the context of CI where the code changes are integrated and tested continuously.

Bug and Test Datasets. Many studies [58, 79, 100, 121, 154] have proposed benchmark or dataset on failed (and passed) tests to facilitate research on automated testing techniques. Just et al. [79] proposed Defects4J which records the information on the failed tests before and after the failure resolution. Le Goues et al. [100] proposed ManyBugs and IntroClass for C projects that have the test failure, the version in which it occurs, and the repairs to the failure that describes expected behavior. Elbaum et al. [58] collected a dataset at Google that includes over 3.5M records of test suite executions. Madeiral et al. [121] shared BEARS-BENCHMARK that contains the test failures before and after the resolution. Saha et al. [154] presented Bugs.jar that contains test failures for existing bugs. We present T-Evos as a dataset that contains the evolution of test execution. As the code changes are integrated and tested continuously, the evolution data provide additional values in CI settings. Our dataset may also complement existing datasets and benchmarks.

Chapter 6

Extending Fault Localization via Code Change Information in Continuous Integration

Chapter 5 explores the evolution of code coverage and test failures in a continuous environment, using the T-Evos dataset. This dataset offers new possibilities for future research on automated testing techniques in continuous integration settings. We discuss a specific fault localization technique that can leverage operational data in continuous integration.

Previous studies [15, 16, 17, 49, 72, 203] have proposed SBFL techniques (presented in Section 2.4) to identify faulty locations at the statement or method level, which may be used for resolving test failures. However, most of these studies evaluate SBFL techniques in traditional software development settings, where only a single snapshot of the system is considered. In contrast, modern software development, particularly in the context of CI, involves continuous and fine-grained changes to the system. Therefore, when a new test failure occurs, the detailed information associated with these changes can provide additional insights for locating the fault. Furthermore, the atomic nature of code changes limits the unintended consequences and makes fault isolation more feasible with accessible and cost-effective diagnosis metrics.

In this chapter, we conduct an empirical study to evaluate the effectiveness of integrating change information for fault localization in CI settings. Specifically, we examine two types of change information: code changes and coverage changes. These two types capture both the static and dynamic aspects of the system. We analyze real-world faults to characterize change information and derive design insights for leveraging such information in fault localization. Our key idea is to consider code changes and coverage changes as valuable debugging knowledge that can enhance fault localization, given their availability in systems following CI practices.

Through extensive evaluation, we demonstrate that code changes can significantly reduce effort and play a crucial role in fault localization. To the best of our knowledge, this study is the first to integrate change information from CI to improve SBFL techniques. Our findings highlight the importance of considering change information for effective fault localization. Future research may explore ways to leverage change information to complement existing fault localization techniques. Furthermore, our study emphasizes the need to leverage CI practices and the valuable information it provides for improving fault localization techniques in modern software development.

An earlier version of this chapter has been published at IEEE/ACM International Conference on Automated Software Engineering (ASE), 2022. Michigan, USA. Chen et al. [40]

6.1 Motivation

CI is a software development practice by which developers regularly deliver into a central repository. CI has been widely established in modern software systems [59, 66, 134, 170]. With continuous delivery, developers automatically build and test new code changes incrementally. This practice helps to identify faults early in the development cycle, making them less expensive to fix.

Prior studies [15, 16, 44, 45, 94, 183, 186] focused on studying the use of code coverage in locating faults under traditional software development settings. However, under CI settings, the changes are incremental. Those finer-grained change information may provide helpful hints on the faulty locations. Typically, there are two types of change information: **code changes**, which track the modified code statements in a commit; and **coverage changes**, which record the changes in code coverage before and after the commit.

As pointed out by prior studies [31, 175, 178], changes to the system may help reveal hints on the faults. In this chapter, we examine the two aforementioned change information and study whether they can provide additional information compared to code coverage, which is widely used in traditional SBFL techniques. We examine code and coverage change information in the CI context where changes are continuous and finer-grained. We consider these two types of change information since they are less expensive to obtain and may be readily available for systems following CI practices. Studying such change information may open new directions to improve fault localization. Below, we discuss the two types of changes in detail.

Coverage Changes. Coverage changes are the effect of the code changes from the coverage aspect. Coverage changes include two types of changed statements: statically and dynamically changed statements. While both types lead to changes in coverage execution, they are different in how the statements are changed. For the statically changed statements, changes happen because the original statements are modified based on the code changes. For instance, when developers modify a statement that is part of the code coverage, the coverage execution naturally changes from the original statement to the recently updated statement. For the dynamically changed statements, changes happen because the dynamic execution (e.g., control flow) in the system is different. The coverage changes as a result of the system taking an alternative execution path. Together, statically and dynamically changed statements provide different in-depth information on the changes performed on the system.

There are two benefits to leveraging such change information. First, the coverage changes can help characterize the source code from a new perspective — the perspective of the statements statically and dynamically influenced by the code changes. This perspective may gain new insights into existing problems, such as tie issue in fault localization. Tie issue is a well-investigated problem in traditional spectrum-based fault localization techniques [15, 16, 17, 49, 72, 203], which is caused by the exceeding number of statements within the code coverage. The coverage changes may help prioritize faulty locations within the coverage based on the statically and dynamically changed statements. Another advantage of using the coverage changes is that it helps limit the search space (i.e., coverage change is a subset of code coverage), which offers more opportunities for improving the precision in fault localization.

Let us illustrate the coverage changes through an example adapted from fault Time 2 in the Defects4J benchmark. In Figure 23, we show the source code, and the coverage of the test failing in the fault inducing commit (denoted as Faulty) and passing in the prior commit (denoted as Prior). First, when updating from the prior commit to the fault-inducing commit, we observe that the statements are modified at line 463 and 464. Those modifications change the code coverage by covering a different *Partial* constructor, as well as introducing new coverage based on the statements within that new constructor. We highlight (in red) those coverage changes in Figure 23. While the code changes reveal what is being done to the system, the aforementioned coverage changes show the effect of the code changes on the system. In this fault, the faulty statements locate at line 218 in the Partial.java file, and at line 227 in the UnsupportedDurationField.java file where the coverage changes. A prior study [19] suggests there exist correlations between the statements dynamically affected by code changes and the faulty locations. This study finds that by inspecting only the dynamically changed statements, developers may reduce the inspection cost and find faults faster. The above observations motivate us to study the usefulness of coverage changes in fault localization.

```
// Commit: 3ba9ba7
// Partial.java
189: Partial(DateTimeFieldType[] types, int[] values, Chronology
    chronology){
212:   DurationField lastUnitField = null;
217:   int compare = lastUnitField.compareTo(loopUnitField);
218:   if (compare < 0 || (compare != 0 && ..)) {
219:     throw new IllegalArgumentException(..);
...
426: public Partial with(DateTimeFieldType fieldType, int value) {
463: - Partial newPartial = new Partial(iChronology, newTypes, newValues);
463: + // use public constructor to ensure full validation
464: + Partial newPartial = new Partial(newTypes, newValues, iChronology);
465:   iChronology.validate(newPartial, newValues);
466:   return newPartial;
474: }
// UnsupportedDurationField.java
226: public int compareTo(DurationField field){
227:   return 0;
228: }
```

s	Prior	Faulty
	Partial.java	
189		
212		🔴
217		🔴
218		🔴
219		🔴
426		
463	●	
464		🔴
465	●	●
466	●	●
474		
	UnsupportedDurationField.java	
226		
227		🔴
228		

Figure 23: Coverage of the fault-triggering test from Time-2, where *Faulty* denotes the fault-inducing commit and *Prior* denotes the commit prior to the fault-inducing commit.

Code Changes. In CI, code changes are one of the mostly used information by developers. They show the modified methods/code statements that cause the fault-triggering test to fail. Such change information is important to developers in practice, as it can provide developers with better understanding on the root cause of the faults. For instance, in the example illustrated in Figure 23, the code changes at line 463 and 464 provide a new perspective on the static change of the system, orthogonal to the coverage changes. Prior studies [31, 175] analyze the change metric (e.g., the complexity of the introduced code hunks), and suggest that code changes can be closely correlated to the faulty locations. However, the usefulness of code changes remains unknown in the context of CI and fault localization, where the changes are continuous and finer-grained. Based on the above observations, we further investigate how the code changes might contribute to enhancing existing fault localization techniques in CI.

6.2 Experimental Setup

In this section, we first present the studied systems and fault dataset. Then, we discuss the data collection process, and the challenges that we encountered in test execution.

6.2.1 Studied Systems and Fault Dataset

Although there are several open source fault datasets such as Defects4J [79], none of them includes the code evolution details (e.g., the test result and code coverage information prior to the fault). Therefore, we collect the dataset using five studied systems from the Defects4J benchmark (version 1.0) and two additional systems (i.e., Fastjson and Jackson-core). In total, we collected 192 faults and the corresponding test failures across seven studied systems.

We choose the Defects4J benchmark because it has been widely used in prior fault localization studies [105, 142, 162, 177, 203]. The benchmark contains a clean dataset that allows researchers to reproduce the faults easily. For each fault, it provides the faulty commit, the fix commit, and fault-triggering tests. However, Defects4J is not applicable for studies in CI context due to the following reasons. First, the *faulty commit* identified by Defects4J does not fit the CI setting. The faulty commit is defined as one of the commits where fault happens, but not the first commit. However, testing begins as soon as the commits are submitted into CI, and if some tests fail, developers will investigate the issue at the failing commit, which differs from the faulty commit identified by Defects4J. To simulate a CI setting, we need to conduct the study on the the fault-inducing commit (i.e., where test failure occurs). Following prior studies [31, 177], we identify the matching fault-inducing commits by using "git bisect" to run the fault-triggering tests on previous commits of the system.

Second, the fault-triggering test at the fault-inducing commit may have different points of failure and reason of failing compared to the fault-fixing commit provided by Defects4J. As these provided fault-fixing commits serve as the ground truth for evaluating the effectiveness of fault localization techniques, they indicate the location where developers should change to fix the faults. However, there might have been code changes between the fault-fixing commit provided by Defects4J and the fault-inducing commit. Therefore, we need to make sure that the test is failing due to the same reason on both commits. To address this challenge, we extract the fault-triggering test from the fault-fixing commit and execute it on the fault-inducing commits by following a prior study [178]. Specifically, we first execute the fault-triggering test on the commit prior to the fault-fixing commit (i.e., where the fault still occurs) to obtain the point of failure (e.g., assertion statement). We then execute the same test on the fault-inducing commit and exclude the fault if the point of failure is different. We use this approach on all the 357 faults from the Defects4J 1.0 benchmark. At the end of this process, we collect 83 faults from the Defects4J 1.0 benchmark.

To further increase the size of the fault dataset, we added two additional systems (i.e., Fastjson and Jackson-core, which follow the CI practices) and increase the fault data in three Defects4J systems (i.e., Chart, Lang, and Time) that have the least number of faults after our previous data validation step. Fastjson is a popular open source Java system used for JSON object conversion (with 24k stars on GitHub). Fastjson has been used in prior research [57, 165] to study code evolution in the CI context. Jackson Project is a well-known Java JSON library, and its fundamental component, Jackson-core, is frequently used in prior fault localization studies [74, 130].

To collect the fault dataset in these five systems, we automatically compile and execute the tests for the 1,000 latest commits at the time when we conduct our case

Table 22: An overview of studied systems.

System	**# Faults**	**KLOC**	**# Test cases**
Fastjson	88	183	4,736
Lang	6	22	2,245
Math	22	85	3,602
Closure	41	147	7,927
JacksonCore	12	45	664
Time	5	28	4,130
Chart	18	96	2,205
Total	192	606	25,509

study in May 2021. Our goal is to find the fault-inducing commit where a fault is first introduced. We sort the commits by their creation time and find the first occurrence for each test failure. Because a test failure may continuously occur in sequential commits until the fault is fixed, we isolate and identify the first occurrence of the test failure as the fault-inducing commit. At the end of this process, we collect 109 additional faults (with a total of 192 faults, as shown in Table 22) and their corresponding fault-inducing commit.

6.2.2 Data Collection Process

In the previous subsection, we discuss the dataset that we collected and used in our study. In this subsection, we provide a detailed explanation of our data collection process.

6.2.2.1 Identifying the Commit Prior to Fault-Inducing Commit

To obtain the code and coverage changes that resulted in test failure, we need to identify the commit prior to the fault-inducing commit. Namely, the commit where the fault has not yet been introduced or triggered by the tests. We identify the prior passing commit using the Git command “git rev−parse commit^” where commit refers to the fault-inducing commit. In the case where there are multiple parent commits, the caret annotation (^) helps to locate the first immediate parent.

6.2.2.2 Collecting Code Changes

As mentioned in Section 6.1, we want to analyze the code changes to study the benefits of leveraging such change information in fault localization. To collect code change information, we first use the "git diff" command to capture the change information between the fault-inducing commit and the prior commit. This change information includes the modified files, the modified code statements, and their corresponding line numbers. To perform a more comprehensive analysis, we also trace higher granularity information (i.e., method in which the modified line belongs to). We use a static analysis tool (i.e., JavaParser [137]) to derive the per-method abstract syntax tree (AST) for each modified file. Since the generated ASTs contain the starting and ending line number for each method, we check whether the line number of the modified statement is within the range of the starting and ending line numbers of the ASTs to determine its corresponding method.

6.2.2.3 Collecting Code Coverage

Our goal is to compare the change information to the conventional code coverage when leveraged in fault localization. Therefore, we collect the code coverage on the fault-inducing commit, and also on the prior commit, to identify the changes in code coverage. To automate this process, we integrate GZoltar [9] into each studied system as a Maven plug-in. GZoltar is a Java framework for automatic debugging and coverage generation. On every test execution, GZoltar instruments the source files to obtain a coverage matrix. The coverage matrix provides information on which statements were executed and by which tests. Thus, we collect the coverage matrix to compute the code coverage for each test.

6.2.2.4 Identifying Coverage Changes

In addition to code changes, we also want to study whether changes in code coverage help improving fault localization techniques. As our goal is to identify the code that is likely to be affected by faults, we compute the changes based on the coverage of the fault-triggering test, between the fault-inducing commit and the prior commit.

We first represent each covered statement using the code statements (e.g., Reducer r = new Reducer(..)), rather than the conventional location information (e.g., Reducer.java, line 33). When comparing the code coverage between two different commits, the location information is not reflective on what is the exact code statements been covered, which might introduce bias. For instance, if the code statement at line 33 changes, then there is a coverage change at line 33, despite the location information remaining the same (i.e., line 33 is covered) on the fault-inducing commit and the prior commit. Therefore, we map the code statements to code coverage and denote the code coverage as $Cov = \{s_1, s_2, ..., s_n\}$, where s represents the code statement covered. Then, we compare the statements covered on the fault-inducing commit (i.e., Cov_{fail}) with the prior commit where the test passed (i.e., Cov_{pass}). We locate the changes by identifying the newly added and changed code statements on the Cov_{fail}. We do not consider the deleted statements, because when we localize faults on the fault-inducing commit, only the existent statements are helpful for analysis. For instance, given $Cov_{fail} = \{s_1, s_2, s_3\}$ and $Cov_{pass} = \{s_2, s_3, s_4\}$, the change is $Cov_{change} = \{s_1\}$. Once we locate the changes, we describe them with the location representation.

6.2.3 Resolving Challenges in Test Execution

Building the systems and running the tests require non-trivial effort [166, 169]. As neither the fault-inducing commit nor the prior commit is readily available on Defects4J, we first need to find the two commits, build and compile the systems, and execute the tests multiple times. In total, we spent hundreds of hours of manual effort compiling the code, executing the tests, and collecting code coverage. To encourage future studies in the area and ease the replication of our study, we made the replication package publicly available [12]. It should be noted that while the data collection has been challenging, gathering the code coverage information requires lower overheads in practice. In Section 6.4.2, we further discuss about the time costs associated with using the change information in fault localization.

Below, we share how we resolve the challenges that we encountered, which may help future studies create benchmarks in CI settings.

Automatically compiling evolving code. The project structure may change as the system evolves. As a result, we need to update the location of the build file in our automation scripts accordingly. For instance, in the earlier versions of the system Time, the build file and the source files were placed inside a nested directory rather than at the root. In order to compile the system on the earlier versions, we need to manually resolve the issue and update our automation script to include the new location of the build file.

Fixing test execution issues. Compiling fault-triggering tests on the fault-inducing commit is not always straightforward. For instance, the JUnit 3 framework is not able to evaluate test annotations with excepted exception (e.g., @Test(expected = Exception) that is featured in JUnit 4. Hence, running a fault-triggering test that is implemented with JUnit 4 syntax on the fault-inducing commit that still uses JUnit 3 will result in a test compilation error. To solve this error, we manually refactor the tests to ensure there is no compilation issue.

Handling JDK compilation. Some studied systems may depend on specific versions of the Java Development Kit (JDK). To address this challenge, we manually determine the required JDK version for each studied system and build an automated script to switch between versions when needed.

Handling flaky tests. To ensure the reliability of our results, we need to remove flaky tests from the fault-inducing commit and the prior passing commit. Flaky tests generate inconsistent code coverage because of their non-deterministic nature. We run Deflaker [8, 24], a state-of-the-art flaky tests detection tool, on both the fault-inducing commit and the prior commit to detect flaky tests, and exclude them from the suspiciousness score computation.

6.2.4 Evaluation Metrics

To measure the effectiveness of leveraging fine-grained change information for fault localization, we consider the following three evaluation metrics discussed in Section 2.5: top ranked N (Top-N), mean average precision (MAP), and mean reciprocal rank (MRR), as they have been widely used in fault localization [39, 172, 175, 180, 192, 206]. Below, we briefly discuss each metric.

Top-N: Given a number N, the Top-N metric defines the number of faults whose faulty program elements (i.e., methods in our experiment) are ranked in the top n ranking positions. Top-N evaluates the ability to find relevant methods among the top ranked n methods. When the suspiciousness score is the same, we randomly break the tie, and repeat the process three times to calculate the average result.

MAP: The MAP metric first computes the average precision for each fault, then calculates the mean of the average precision. We define the average precision (AP) as the average of precision values at all ranks where relevant methods are found. MAP assesses the ability in finding all relevant methods.

$$AP = \frac{\sum_{i=1}^{m} i/Pos(i)}{m} \tag{14}$$

MRR: The MRR metric calculates the mean of the reciprocal position at which the first relevant method is found. MRR assesses the ability to find the first relevant method.

$$MRR = \frac{1}{K}\sum_{i=1}^{K}\frac{1}{rank_i} \tag{15}$$

6.3 Experiment Results

In this section, we present our experiment results by answering three research questions (RQs). For each RQ, we present the motivation, approach, and results and discussion.

RQ1: What Are the Overlaps Between the Change Information and Faulty Locations

Motivation: Prior research has widely studied code coverage-based fault localization techniques (e.g., SBFL) [16, 181, 201, 203, 208]. Despite their popularity, these techniques suffer from precision issues due to the broad search space [16, 78, 162, 201]. Intuitively, coverage changes are subsets of code coverage which implies a smaller search space. The code changes can also help restrict the search space while providing new information (e.g., the changed code may not have corresponding tests to cover it). Hence, in this RQ, we investigate how the change information overlaps with the faults, and whether they are helpful in fault localization.

Approach: To understand how the change information contributes to finding faulty locations, we analyze the number of ***faulty methods*** covered by each type of change information, and code coverage. We define faulty methods as the methods that were modified by developers to fix the faults (i.e., faulty locations). For each fault, we first identify a set of faulty methods from the fault-resolving commits. Then, we study how many faulty methods have code changes or coverage changes, and compare them to code coverage. Furthermore, we use ***faulty method ratio*** to study the percentage of faulty methods among all the covered methods. A higher faulty method ratio means that the identified search space has more faulty methods, which may be leveraged to improve the precision of fault localization techniques.

Results: Although coverage changes cover only 67% of the faulty methods from the code coverage, its faulty method ratio is 7 times higher. As shown in Table 23, coverage changes have overlaps with 170 faulty methods in the reduced search space (since coverage changes are a subset of code coverage) and code coverage has overlaps with 254 faulty methods. Although coverage changes cover fewer faulty methods, the covered methods have a much higher faulty method ratio (i.e., 7 times higher, 5.7% compared to 0.7% from code coverage) and significantly fewer methods compared to code coverage (i.e., 2,963 methods v.s. 35,483 methods). The results show that the coverage changes, as a subset of the code coverage, cover 12 times fewer total methods than the code coverage, which may help in the ranking of faulty locations. Nevertheless, the coverage changes provide as much as 67% of the faulty methods within the reduced search space. This means that, when leveraging the coverage changes, we can perform the fault localization on a much smaller number of methods, while identifying a good percentage of faulty methods. The above findings suggest initial evidence for the potentials of leveraging coverage changes to improve the precision of fault localization.

Code changes cover additional faulty methods over code coverage, and provide faulty method ratio that is 14 times higher. In Table 23, code changes overlap with 173 faulty methods in the reduced search space while code coverage overlaps with 254 faulty methods. Even though code changes cover fewer faulty methods, its faulty method ratio is 14 times higher (i.e., 10.7%, compared to 0.7% from code coverage). Code changes' search space is also smaller, covering 22 times fewer methods compared to code coverage (i.e., 1,614 methods v.s. 35,483 methods). We also find that code changes have overlap with 14% *additional* faulty methods that code coverage is not able to cover. After some manual investigation, we find that the reason is these faulty methods do not have a corresponding test and code coverage (i.e., not tested in the system). Hence, these additional faulty methods that code changes have overlap with may further help coverage-based fault localization

Table 23: The number of total methods covered by code coverage, code changes and coverage changes, and the number of faulty methods captured in each information. *Total* and *Faulty* denote the number of total and faulty methods. *Faulty ratio* denotes faulty method ratio, which is the percentage of faulty methods per the total methods covered.

System	**Code Coverage**			**Code Changes**			**Coverage Changes**		
	Total	Faulty	Faulty Ratio	Total	Faulty	Faulty Ratio	Total	Faulty	Faulty Ratio
Fastjson	7,538	135	1.8%	283	88	31.1%	443	82	18.5%
Lang	41	6	14.6%	7	6	85.7%	8	5	62.5%
Math	827	18	2.2%	264	15	5.7%	90	14	15.6%
Closure	22,572	27	0.1%	293	12	4.1%	2,169	12	0.6%
JacksonCore	1,022	40	3.9%	185	31	16.8%	95	31	32.6%
Time	1,872	7	0.4%	41	5	12.2%	90	7	7.8%
Chart	1,611	21	1.3%	541	16	3.0%	68	19	27.9%
Total	35,483	254	0.7%	1,614	173	10.7%	2,963	170	5.7%

techniques identify more faults.

In short, we find that both types of change information have a higher percentage of faulty method ratio within the identified search space. Moreover, code changes overlap with faulty methods that code coverage fails to identify. These findings shed lights on the potentials of incorporating change information to improve coverage-based fault localization in CI settings.

Both types of change information cover a higher percentage of faulty methods compared to code coverage in their reduced search space. Code changes also cover additional faulty methods that do not have code coverage.

RQ2: How Does Change Information Perform in Fault Localization?

Motivation: In RQ1, we found that change information achieves a higher faulty method ratio in the reduced search space. However, it is yet to explore whether both types of change information can be used for fault localization. Therefore, in this RQ, we propose three change-based techniques derived from change information and evaluate their effectiveness in fault localization techniques under CI settings.

Approach: Our goal is to systematically study the effectiveness of each type of change information in fault localization. We adopt three change-based techniques to characterize the change information with fault proneness. These change-based techniques are based on the size of code changes, the size of coverage changes, and the size of the statements affected by coverage changes. *CodeChange*, *CoverageChange*, and *CoverageExecution* denote these three change-based techniques respectively, and they each exclusively leverages one of the aforementioned change-based metrics. We want to investigate how each change-based technique performs in fault localization.

We conduct our analysis at the **method level**. For each method, we compute its suspiciousness score by computing and aggregating the suspiciousness scores across all the statements within the method. Prior studies [103, 115, 162, 203, 204] have also demonstrated that such method level aggregation helps better distinguish the non-faulty statements from the faulty ones. We compare the change-based techniques with *Ochiai*, a commonly used SBFL technique [62, 94, 204, 208]. We choose *Ochiai* since it outperforms other SBFL formulas in terms of fault localization performance [94, 118, 203]. For evaluating the results, we examine the Top-1, Top-5 and Top-10 accuracy, MAP, and MRR values (defined in Section 6.2.4).

We design *CodeChange* to rank methods with the most changes to be more suspicious. Namely, a method is ranked to be more suspicious if it contains more modified statements. For instance, we rank the method with the most changed statements at position 1, indicating it is the most suspicious method. For methods without any code changes, the technique considers them as non-suspicious, and removes them from the ranking to reduce noise. We design *CodeChange* this way since we want to study how vanilla code changes may be used for fault localization, and previous research [129, 132, 185] observe that the size of a change is a good indicator of fault proneness.

We design *CoverageChange* to rank methods with most coverage changes as more suspicious. Within a method, each code statement with coverage change receives a suspiciousness score of 1. Within a method, we count the number of code statements with coverage change, and rank the method with the most changed statements at position 1. Hence, a method is ranked more suspicious if more changes happen in code coverage. A previous research [19] found that the size of changes gives good indication on the fault proneness. Therefore, we apply the same concept to study the effect of cover changes on fault localization. For the methods that do not have any coverage change, the technique considers them as non-suspicious, and removes them from the ranking to reduce noise.

We design *CoverageExecution* to rank methods with the most statements affected by the coverage changes to be more suspicious. Previous studies [124, 174, 183] found that the size of the execution affected by the faults can provide additional guidance towards the faulty locations. Intuitively, if there is a coverage change (either dynamic or static change) at any statement within a method, the internal state (i.e., dynamic execution) of the subsequent statements is likely affected. Therefore, we design this technique to boost the methods that are "likely affected" by the change. We identify the methods with the most affected statements to be more suspicious. For instance, if the first occurrence of the coverage changes locates at line 33 of a given method, then starting from line 33, we count the number of statements that were executed by the code coverage. If the number of statements executed (affected) in that method is higher than other methods, then it is considered as the most suspicious method. The methods without any coverage change are considered as non-suspicious, and removed from the ranking to reduce noise.

Results: On average, *CodeChange*, *CoverageChange* and *CoverageExecution* achieve 13%, 7% and 23% improvement over *Ochiai* for MAP, respectively, and 17%, 17% and 24% improvement for MRR, respectively. Table 24 shows the fault localization results in terms of MAP, MRR, Top-1, Top-5, and Top-10. We observe that all three techniques derived from change information, on average, perform better than *Ochiai* in fault localization. In particular, *CoverageExecution* has the best overall Top-5 and Top-10 (i.e., locating 109 and 118 faults), and the highest average MAP and MRR (i.e., with an average MAP of 0.37 and MRR of 0.52). *CoverageExecution* achieves an improvement of 23% for average MAP and 24% for MRR.

CoverageChange achieves the second best overall performance, improving the average MAP and MRR by 13% and 17% respectively (i.e., with an average MAP of 0.37 and MRR of 0.52.). *CodeChange* achieves an improvement of 7% and 17% for average MAP and MRR (i.e., with an average MAP of and MRR of 0.49). The results show that even simple techniques that rank by the size of code changes or coverage changes tend to perform well.

Table 24: Effectiveness of Ochiai, CodeChange, CoverageChange and CoverageExecution in terms of Top-1, Top-5, Top-10, MAP and MRR. For each project, we show the best MAP and MRR in bold. The last rows of the table show the sum values for Top-N, and the weighted average for MAP and MRR across the studied systems. The last row of the table shows the sum values for Top-N, and the weighted average for MAP and MRR.

System	**Approach**	**Top-N**			**MAP**	**MRR**
		N=1	N=5	N=10		
Fastjson	Ochiai	31	45	45	0.20	0.36
	CodeChange	45	53	55	0.29 (+45%)	0.56 (+56%)
	CoverageChange	54	62	65	**0.38 (+90%)**	**0.65 (+81%)**
	CoverageExecution	46	60	65	0.35 (+75%)	0.61 (+69%)
Lang	Ochiai	4	4	4	0.69	0.71
	CodeChange	4	5	5	**0.78 (+13%)**	**0.90 (+27%)**
	CoverageChange	4	5	5	0.72 (+4%)	**0.90 (+27%)**
	CoverageExecution	4	5	5	0.72 (+4%)	**0.90 (+27%)**
Math	Ochiai	9	10	13	0.51	0.51
	CodeChange	10	12	12	**0.53 (+4%)**	**0.54 (+6%)**
	CoverageChange	8	13	13	0.47 (-8%)	0.49 (-4%)
	CoverageExecution	8	14	14	0.50 (-2%)	0.52 (+2%)
Closure	Ochiai	4	10	13	0.18	0.16
	CodeChange	7	10	11	**0.22 (+22%)**	**0.24 (+20%)**
	CoverageChange	2	3	5	0.07 (-61%)	0.09 (-55%)
	CoverageExecution	2	8	9	0.12 (-33%)	0.13 (-35%)
JacksonCore	Ochiai	0	1	1	0.02	0.02
	CodeChange	7	7	7	**0.21 (+950%)**	**0.58 (+2800%)**
	CoverageChange	1	5	6	0.20 (+900%)	0.26 (+1200%)
	CoverageExecution	4	5	6	0.19 (+850%)	0.39 (+1850%)
Time	Ochiai	3	4	4	**0.63**	**0.50**
	CodeChange	2	2	4	0.29 (-54%)	0.46 (-8%)
	CoverageChange	1	2	2	0.24 (-62%)	0.32 (-36%)
	CoverageExecution	2	2	3	0.43 (-32%)	0.43 (-14%)
Chart	Ochiai	14	16	16	**0.80**	**0.83**
	CodeChange	6	7	8	0.50 (-38%)	0.54 (-35%)
	CoverageChange	7	15	16	0.57 (-29%)	0.62 (-25%)
	CoverageExecution	14	15	16	0.76 (-5%)	0.81 (-2%)
Sum/Avg.	Ochiai	65	90	96	0.30	0.42
	CodeChange	81	96	102	0.32 (+7%)	0.49 (+17%)
	CoverageChange	77	105	112	0.34 (+13%)	0.49 (+17%)
	CoverageExecution	80	109	118	**0.37 (+23%)**	**0.52 (+24%)**

All three techniques achieve improvements in the overall Top-N values. *CodeChange* locates the most faults at Top-1 (i.e., locating 81 faults at Top-1), followed by *CoverageExecution* (i.e., locating 80 faults at Top-1), and *CoverageChange* also achieves improvements over *Ochiai* (i.e., locating 77 faults at Top-1). In terms of the Top-5 and Top-10, *CoverageExecution* achieves the most faults (i.e., locating 109 at Top-5, and 118 at Top-10), followed by *CoverageChange* (i.e., locating 105 at Top-5, and 112 at Top-10), and *CodeChange* (i.e., locating 96 at Top-5, and 102 at Top-10).

In short, the change-based techniques have a better fault localization performance compared to Ochiai. Even though change information has a reduced search space, our findings show that change information may be better at ranking the faulty methods and reducing possible investigation effort from developers. Future fault localization studies should consider change information due to its effectiveness and availability in CI settings.

The three change-based techniques achieve an improvement that varies from 7% to 23% and 17% to 24% over *Ochiai* for the average MAP and MRR, respectively. The results also indicate that all three change-based techniques outperform *Ochiai* in locating faults across all studied Top-N metrics.

RQ3: Can Change Information Complement Existing Fault Localization Techniques?

Motivation: In RQ2, our findings show that the change-based techniques achieve better fault localization results compared to the coverage-based baseline (*Ochiai*). However, as found in RQ1, code coverage still covers more faulty methods compared to coverage and code changes. Therefore, we hypothesize that the two types of information (i.e., coverage and change information) may complement coverage-based SBFL techniques when combined together. In this RQ, we experiment with different combinations of *Ochiai* and the three proposed change-based techniques, and then we discuss their fault localization results.

Approach: To answer this RQ, we study the effectiveness of adding five different combinations of the change-based techniques to *Ochiai*. These combinations includes *Ochiai + CC*, *Ochiai + CovC*, *Ochiai + CovE*, *Ochiai + CovC + CC*, and *Ochiai + CovE + CC*, where we denote the size of code changes and coverage changes as *CC* and *CovC* respectively, and *CovE* as the size of the execution affected by coverage changes. Similar to RQ1 and RQ2, we conduct the fault localization at the method level by aggregating the suspiciousness scores of the code statements within a method (by taking the highest score). We compare the results of Top-1, Top-5, Top-10, MAP and MRR. Below, we describe how we combine *Ochiai* with *CC*, *CovC* and *CovE*.

Ochiai + CC: We combine the size of code changes with *Ochiai* by following a similar equation (Equation 16 below) defined in prior studies [39, 173, 180] to calculate a boost score for each code statement.

$$BoostScore(s) = \begin{cases} \frac{1}{rank} & \text{if } s \in \text{ RankedStatements} \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

The intuition is that the methods with more changes are ranked higher, and thus the corresponding statements receive a higher boost score. If a method is ranked second, then the boost score is 0.5 (1/2). If a method is not part of the coverage changes (and therefore not ranked), then the boost score is 0. We calculate the suspiciousness score for each code statement by adding the boost score to the initial suspiciousness score computed by *Ochiai*. Finally, we aggregate the suspiciousness score for all code statements within a method, and calculate method-level ranking.

Ochiai + CovC: We combine the size of coverage changes with *Ochiai* by also following Equation 16. The methods with more coverage changes are ranked higher, and thus more likely to be faulty. We attribute a higher boost score to the code statements within that method. Similarly, we calculate the suspiciousness score for each code statement by adding the boost score to the initial suspiciousness score computed by *Ochiai*.

Ochiai + CovE: We combine the size of the execution affected by coverage changes with *Ochiai* by following Equation 16. The methods with more affected execution are ranked higher, and thus more likely to be faulty. Similarly, we add the boost score to the initial suspiciousness score computed by *Ochiai* to come up with a final suspiciousness score.

Ochiai + CovC + CC and ***Ochiai + CovE + CC:*** We combine the change-based techniques from different change information together to examine the effect of each change metric on the performance *Ochiai*. To combine both change information in *Ochiai*, for each code statement, we add the boost scores calculated from each of technique to the suspiciousness score computed by *Ochiai*. We base our recommendation of results on the resulting suspiciousness score.

Results: Overall, *Ochiai + CovE + CC* achieves the best performance, improving the MAP and MRR values from *Ochiai* by 53% and 52% respectively. Table 25 compares the performance of *Ochiai* with different change information considered. We calculate the evaluation metrics for each combination and compute its improvement over Ochiai. On average, all techniques outperform *Ochiai*. Specifically, adopting both the size of affected statements and the size of code changes in Ochiai (i.e., *Ochiai + CovE + CC*) achieves the best overall MAP and MRR, and the highest Top-N values. *Ochiai + CovE + CC* achieves the optimum improvement of 53% and 52% for MAP and MRR over Ochiai (i.e., with an average MAP of 0.46 and MRR of 0.64). *Ochiai + CovC + CC* achieves the second best improvement of 40% and 45% for MAP and MRR (i.e., with an average MAP of 0.42 and MRR of 0.61).

Adopting the individual change metric also yields better results for *Ochiai*. Specifically, for *Ochiai + CC*, *Ochiai + CovC* and *Ochiai + CovE*, the improvements for MAP are 32%, 20% and 33%, respectively, and the improvements for MRR are 38%, 26% and 36%, respectively. We observe similar results in terms of the Top-N values. Our finding shows that any type of the studied change information provides noticeable benefits when combined with *Ochiai*.

We find that adopting both change information in *Ochiai* achieves the best performance on average. Either of the two combinations (i.e., *Ochiai + CovC + CC* and *Ochiai + CovE + CC*) provides better results than adopting individual change information. In particular, *Ochiai + CovC + CC* improves the overall MAP and MRR by 20% and 19% when compared to *Ochiai + CovC*, and 8% and 7% when compared to *Ochiai + CC*. We observe that combining *CovE* with *CC* produces better results compared to combining *CovC* with *CC*. The results suggest that *CovE* with *CC* complement each other better and can help further improve the fault localization results.

Adopting the size of code changes alone (i.e., *Ochiai + CC*) can significantly improve *Ochiai*. We observe an improvement of 33% and 38% for average MAP and MRR respectively. Moreover, *Ochiai + CC* achieves 92 faults at Top-1, locating 27 more faults than *Ochiai*. This result suggests that, by only investigating the first

Table 25: Effectiveness of *Ochiai* when applied different combination of change information. For each combination, we evaluate the performance in terms of MAP and MRR. *CC* denotes the code change information. *CovC* denotes the coverage change information. *CovE* denotes the coverage execution information. The best performing approach is marked in bold. The last row of the table shows the sum values for Top-N, and the weighted average for MAP and MRR.

System	**Approach**	**Top-N**			**MAP**	**MRR**
		N=1	N=5	N=10		
Fastjson	Ochiai	31	45	45	0.20	0.36
	Ochiai + CC	46	61	61	0.28 (+40%)	0.58 (+61%)
	Ochiai + CovC	48	67	74	0.35 (+75%)	0.63 (+75%)
	Ochiai + CovE	46	68	72	0.35 (+75%)	0.62 (+72%)
	Ochiai + CovC + CC	54	74	76	**0.39 (+95%)**	**0.70 (+94%)**
	Ochiai + CovE + CC	55	75	75	0.38 (+90%)	0.69 (+92%)
Lang	Ochiai	4	4	4	0.69	0.71
	Ochiai + CC	4	5	5	**0.85 (+23%)**	**0.90 (+27%)**
	Ochiai + CovC	4	5	5	0.83 (+20%)	**0.90 (+27%)**
	Ochiai + CovE	4	5	5	0.83 (+20%)	**0.90 (+27%)**
	Ochiai + CovC + CC	4	5	5	0.83 (+20%)	**0.90 (+27%)**
	Ochiai + CovE + CC	4	5	5	0.83 (+20%)	**0.90 (+27%)**
Math	Ochiai	9	10	13	0.51	0.51
	Ochiai + CC	14	15	16	**0.71 (+39%)**	**0.73 (+43%)**
	Ochiai + CovC	9	13	15	0.53 (+4%)	0.56 (+10%)
	Ochiai + CovE	9	13	15	0.53 (+4%)	0.55 (+8%)
	Ochiai + CovC + CC	12	15	16	0.65 (+27%)	0.67 (+31%)
	Ochiai + CovE + CC	13	15	16	0.68 (+33%)	0.71 (+39%)
Closure	Ochiai	4	10	13	0.18	0.16
	Ochiai + CC	8	12	16	0.27 (+50%)	0.29 (+81%)
	Ochiai + CovC	4	10	13	0.17 (-6%)	0.20 (+25%)
	Ochiai + CovE	5	11	16	0.22 (+22%)	0.24 (+50%)
	Ochiai + CovC + CC	6	11	16	0.24 (+33%)	0.27 (+69%)
	Ochiai + CovE + CC	9	13	16	**0.30 (+67%)**	**0.33 (+106%)**
JacksonCore	Ochiai	0	1	1	0.02	0.02
	Ochiai + CC	1	7	7	0.14 (+600%)	0.33 (+1550%)
	Ochiai + CovC	1	6	6	0.08 (+300%)	0.22 (+1000%)
	Ochiai + CovE	4	6	6	0.10 (+400%)	0.38 (+1800%)
	Ochiai + CovC + CC	6	7	7	0.19 (+850%)	**0.53 (+2550%)**
	Ochiai + CovE + CC	6	7	7	**0.22 (+1000%)**	0.53 (+2550%)
Time	Ochiai	3	4	4	**0.63**	0.50
	Ochiai + CC	2	4	4	0.51 (-19%)	**0.51 (+2%)**
	Ochiai + CovC	2	4	4	0.42 (-33%)	**0.51 (+2%)**
	Ochiai + CovE	2	3	4	0.48 (-31%)	0.48 (-4%)
	Ochiai + CovC + CC	2	4	4	0.44 (-30%)	0.49 (-2%)
	Ochiai + CovE + CC	2	3	4	0.48 (-24%)	0.48 (-4%)
Chart	Ochiai	14	16	16	0.80	0.83
	Ochiai + CC	17	18	18	0.90 (+13%)	**0.96 (+16%)**
	Ochiai + CovC	8	18	18	0.66 (-18%)	0.71 (-14%)
	Ochiai + CovE	16	18	18	0.88 (+10%)	0.93 (+8%)
	Ochiai + CovC + CC	9	18	18	0.69 (-14%)	0.74 (-11%)
	Ochiai + CovE + CC	17	18	18	**0.92 (+14%)**	**0.96 (+16%)**
Sum/Avg.	Ochiai	65	90	96	0.30	0.42
	Ochiai + CC	92	122	127	0.40 (+33%)	0.57 (+38%)
	Ochiai + CovC	76	123	135	0.36 (+20%)	0.53 (+26%)
	Ochiai + CovE	86	124	136	0.40 (+33%)	0.56 (+36%)
	Ochiai + CovC + CC	93	134	142	0.42 (+40%)	0.61 (+45%)
	Ochiai + CovE + CC	106	136	141	**0.46 (+53%)**	**0.64 (+52%)**

position, the developers might locate 48% (i.e., 92 out of 192 faults) of the faults. The above findings illustrate the effort reduction for developers in practice, and show the usefulness of code changes metric in fault localization. Compared to the use of two other change-based techniques (i.e., *Ochiai + CovC* and *Ochiai + CovE*), *Ochiai + CC* achieves better performance in fault localization. Particularly, in three out of the seven studied systems (i.e., Math, Closure and Chart), *Ochiai + CC* locates more faults at Top-1 than the two other metrics. This is because, different from the two other change-based techniques that are both based on the code coverage, the code changes leverage a different search space. The additional information as discussed in RQ1 helps to cover more faults.

We observe that the system, Time, experiences a decrease in fault localization performance when leveraging some of the change-based techniques. For instance, all of the combinations locate one less fault at Top-1, and up to one less fault at Top-5. After our investigation, we find that one specific fault, Time 16, contributes to this result. As the code changes and coverage changes are large in size (due to potential refactoring that happens when fixing the fault), it introduces much noise (i.e., non-faulty statements), making it more challenging to locate the faulty locations. Nevertheless, in *Ochiai + CovE*, *Ochiai + CovC + CC* and *Ochiai + CovE + CC*, the faulty statement is only ranked at a slightly lower position (i.e., from position 1 to 4). Note that considering there are only five faults in Time, this difference is further magnified when looking at the percentages of difference in MAP and MRR values, but having a small impact on the overall results. The results still demonstrate the usefulness of the change information in locating faults, considering the overall performance improvement. Future studies are needed to explore different techniques to leveraging change-based techniques in fault localization.

The change information can complement *Ochiai* by providing up to 53% and 52% improvement in MAP and MRR respectively, and locating 41 more faults at Top-1. Future fault localization techniques may consider combining both change information to further improve the performance.

6.4 Discussion

6.4.1 Effectiveness of Change Metrics

Although the combination of the change metrics and *Ochiai* achieves promising fault localization results, the change metrics do not contribute to locating some of the faults. In this section, we discuss the reasons and hope the finding can provide insights for future studies in CI and fault localization.

Conventional statement coverage may not be sufficient for capturing the behaviour changes in some code statements (20/32). While common code coverage tools (e.g., JaCoCo, Cobertura, and GZoltar) report coverage at various levels such as code statement or branch (aggregated per method), they do not report the condition coverage within each code statement. Therefore, coverage change that happens at the condition-level may be missed. For example, in fault Closure-85, based on the coverage change analysis, the covered code statements are identical between the fault inducing commit and the prior passing commit. However, based on our manual study, the condition coverage changes at line 199. The statement at line 199 was:

```
if (n.isEmpty() —— (n.isBlock() && !n.hasChildren()))
```

Although both the fault inducing commit and prior passing commit cover this code statement, the condition coverage is different. In fact, the prior commit only covers the first condition n.isEmpty() while the fault inducing commit covers both conditions. This change in condition coverage is relevant to the root cause of the fault, but has not been captured because of the absence of condition coverage information. Note that the above-mentioned condition coverage refers to *per-statement condition coverage*, which is different from the term “condition coverage” used in Cobertura [5] (also called “branch coverage” in JaCoCo [6]) which shows the percentage of conditions covered throughout a method.

Our findings show that despite the advantages of leveraging coverage change information, there is a need for finer-granularity coverage information. Future studies are encouraged to study the usefulness of finer-granularity coverage information in fault localization.

Noise introduced when combining the change metrics (12/32). To better examine the effect of each change metric on the performance, we adopt the design of our approach to combine the change metrics following prior studies [39, 180]. We find that, in some faults, such combinations may introduce noises (i.e., prioritizing non-faulty statements). For instance, on fault Closure 120, we observe that neither the code changes, nor the coverage changes contain the faulty statements, thus the approach boosts the non-faulty statements to a higher position. This causes the rank of the faulty statements to be further pushed down in the ranking, which reduces the performance. While faults are affected by noises, the result demonstrates that the effect on the overall performance is trivial.

6.4.2 Overheads of Change-based Techniques

In this subsection, we discuss the overheads for integrating change-based fault localization techniques into CI.

To evaluate the overheads, we measure the processing time in seconds for locating a fault. The localization breaks down into four steps. In particular, step 1 refers to collecting and analyzing code change. Step 2 refers to collecting code coverage. Step 3 refers to performing code coverage change analysis. Finally, step 4 refers to ranking suspicious methods. On average, it takes less than 42 seconds in total to determine the final ranking of suspicious methods. In practice, this processing time only represents a small overhead considering a single build can take more than 12 minutes to run in some projects (i.e., Fastjson). The main source of overheads originates from the second step to collect code coverage. This step requires compilations from the fault-inducing commit and the prior passing commit. Therefore, depending on the size of the system, it takes 13 to 43 seconds to successfully compile both commits. Collecting the code coverage of failing tests can take up to 8 to 24 seconds. While this step contributes to significant time cost, the compilation of the system is a necessary procedure in CI to run the builds. And thus, in practice, the compiled source code can be directly used to collect the code coverage when test failures are identified. This can help to reduce the overheads associated with the compilation time. Eventually, as part of the continuous practice, change-based techniques can be automatically triggered when test failures happen during nightly builds. As some tests might take longer time to run, this procedure allows the collection of additional information on the test failures overnight (e.g., list of suspicious methods, change information), which can assist developers in debugging later.

6.5 Threats to Validity

External Validity. One potential threat to external validity is the generalizability of our results based on the studied systems. To mitigate this threat, we conduct our experiments on seven real-world open source Java systems each with different characteristics and infrastructures. On top of the five studied systems provided by the Defects4J benchmark, we carefully select two additional systems that are actively maintained, widely used, and follow the CI practices. While we cannot confirm the generalization of our findings to fault localization approaches written in other programming languages, we design our approaches of leveraging the change metrics as generic as possible. Future study can easily adopt our approaches to fit other programming languages.

Construct Validity. One potential threat is our design decision of combining the change information to *Ochiai* in RQ3. There may be better ways of combining the change information that further explore their benefits. We adopt the approaches following existing studies [39, 173, 180] to provide insights on the effects of combining the change information together. Such design helps better illustrate the improvement over the baseline (i.e., *Ochiai*), and can be easily integrated into the CI context. We encourage future studies to explore other ways of combining the change information, and release the data online [12] to facilitate replication. In addition, we did not consider mutation analysis in our approach. While the mutation analysis may provide more information, one challenge is that mutation is very costly, and thus does not fit the CI scenario. Moreover, in this study, we focus on two change information as they provide accurate information on the internal execution changes of the system. However, there are other change information that may be leveraged in fault localization. Future study is needed to investigate other change metrics.

Part III

Fault Localization Data Cleanness

Chapter 7

Studying Data Cleanness in Defects4J and its Impact on Fault Localization

Previous chapters illustrated that locating and fixing bugs is an expensive and time-consuming task, especially due to the ever-increasing complexity of modern software. Particularly, understanding the cause and locating the bug is often the most challenging step [64, 111, 141, 183]. Developers need to analyze all the available information, such as test failures and program execution details, to identify potential buggy code statements in the system and develop a bug fix. To assist developers in locating and fixing bugs, prior research has proposed techniques such as spectrum-based fault (SBFL) localization [113, 142, 162, 173, 203] and automated program repair [104, 106, 153].

Given the increasing community attention on software testing research, prior studies introduced benchmarks for automatic program repair and fault localization. For example, there are many benchmarks available for both C and Java, such as ManyBugs [100] QuixBugs [110], Bugs.jar [154], Bugswarm [167], and Defects4J [4]. These benchmarks often provide test coverage, test failure information, and the corresponding bug fix. Hence, researchers can easily apply the proposed techniques for evaluation by using these benchmarks. In Chapter 5, we also proposed T-Evos, a benchmark on continuous integration test execution and failure.

Among these existing benchmarks, Defects4J is one of the most widely-used for evaluating the effectiveness of fault localization and program repair techniques [123, 131, 142, 162, 203]. Defects4J aims to provide a controlled environment to study real bugs collected from real systems. The latest version of Defects4J, v2.0, contains 835 bugs from 17 open source Java systems. Each bug in the Defects4J benchmark is accompanied by at least one failing test that triggers the bug (i.e., fault-triggering test), where the tests and bug fixes are mined from the system commit history.

Although Defects4J has significantly helped advance software testing research, prior studies [80, 87, 113] pointed out concerns that some tests in Defects4J may contain developer knowledge of the bugs. As found by Liu et al. [113], some fault-triggering tests may be taken from a version of the system where the bug has already been resolved. These tests may provide hints to fault localization techniques about the location of the bug. Therefore, having such developer knowledge in the tests may impact the evaluation of existing fault localization techniques.

In this chapter, we conduct a study on the fault-triggering tests in Defects4J. The goal of this study is to provide insights into the prevalence of developer knowledge in fault-triggering tests and its impact on fault localization techniques. First, we classify the tests in relation to developers' bug-fixing activities (e.g., a test is newly added or modified after a bug was reported), which may help future studies to better utilize Defects4J when evaluating SBFL techniques. Second, we analyze developers' modifications on the fault-triggering tests and identify different categories of reasons why developers modify them. Finally, we study the impact of developer knowledge on the results of fault localization techniques. We apply state-of-the-art SBFL techniques on the commit before and after the bugs were fixed. Our experimental results show that the state-of-the-art spectrum-based fault localization techniques perform significantly worse in the absence of developer knowledge in the tests.

An earlier version of this chapter has been submitted to an international conference on software engineering, 2023.

Table 26: An excerpt of the Bug report CLI-51 from the Defects4J benchmark.

BugID	CLI-51
Summary	Parameter value "-something" misinterpreted as a parameter
Developer's comment	"*Fix so parser doesn't burst options which are not defined. (-s) in the above case. Includes unit test [BugCLI51Test].*"

7.1 Motivation

The Defects4J v2.0 benchmark contains 835 bugs from 17 Java open source systems [4]. All 835 bugs are extracted from different phases of software development, and the 17 projects span a wide range of domains and maturities. The objective of this benchmark is to facilitate research in software testing and debugging. Due to its ease of use and the realistic nature of the bugs, Defects4J has been widely used for conducting research in fault localization [113, 142, 162, 173, 203], automated program repair [73, 123, 131, 189], and automated test generation [61, 159, 194].

In Defects4J, each bug comes with at least one failed test to ensure reproducibility. This failed test is known as the *fault-triggering test*. As not all bugs are guaranteed to have a fault-triggering test at the time they are first uncovered, Defects4J utilizes an automated step to mine candidate fault-triggering tests from the bug fix and buggy commit of the system [4]. Specifically, a fault-triggering test must deterministically pass on the fixed commit and fail on the buggy commit. Every bug and fault-triggering test is then manually examined to eliminate irrelevant code changes, such as the addition of new features. By default, Defects4J uses developer-written tests as fault-triggering tests to reproduce the bugs.

However, fault-triggering tests may be mined from the bug-fixing commit where developers may have already fixed the bug or were in the process of fixing it. This can include developer knowledge (i.e., information about the bug that was not available at the time the bug was reported) in the tests. Table 26 provides an excerpt of a bug, CLI-51, from the Defects4J benchmark with a test that has developer knowledge. CLI-51 is a bug from Commons-Cli where the code misinterpreted parameter values as new parameters. When developers first received this bug report, there was no test failure in the system. Developers provided a patch to fix the bug and commented in the report that a fault-triggering test (i.e., BugCLI51Test) was developed as part of the patch for regression testing. The fault-triggering was introduced after the corresponding bug fixes. In particular, developers developed the test based on the content of the bug report and the bug fix. The fault-triggering test verifies the parameter value "-t -something" that triggers the bug, as mentioned in the bug report. As a result, the test contains developer knowledge, which provides hints on the causes and location of the bug in the source code.

Although prior research studies [40, 87, 177] suggest that some tests in Defects4J may contain developer knowledge, there is no systematic study on the prevalence of such tests and their impact on the results of fault localization techniques. Fault localization is an important step in assisting developers locate faults [23, 116, 142, 208] and in guiding automated program repair techniques [96, 113, 136]. Under- or over-estimating the effectiveness of fault localization techniques may affect their adoption in practice. Hence, in this chapter, we study the fault-triggering tests in Defects4J in relation to the bug fixes, and their impact on the fault localization results. Our experimental results reveal that a majority of the fault-triggering tests (77%) are affected by developer knowledge. We also provide a classification of the bugs in Defects4J based on our findings and future research may consider our dataset when evaluating fault localization techniques.

7.2 Research Questions

In this section, we first discuss an overview of the studied systems. Then, we present the motivation, approach, and results of the three research questions (RQs) that we seek to answer.

7.2.1 Overview of the Studied Systems

We performed our study on the Defects4J (V2.0.0) benchmark [4], which includes real and reproducible faults from a wide range of systems. Defects4J forms the basis of many prior studies on fault localization [103, 113, 142, 162, 177, 203], where these studies use Defects4J as the benchmark dataset for evaluation and comparison with state-of-the-art. Table 27 provides an overview of our studied systems (we collected the metrics from the HEAD version of each system). The sizes of the studied systems in Defects4J range from 4K to 90K lines of code, and the benchmark contains 1,655 fault-triggering tests. We excluded the system, Chart, from Defects4J since it does not use Git as the version control system, and we rely on analyzing the development history in Git repositories to understand the changes on the fault-triggering tests for our study. In total, we conducted our experiments on 809 bugs from 16 systems. Note that, one bug may have more than one fault-triggering test, so there are more fault-triggering tests than the number of bugs.

7.2.2 RQ1: Were Fault-triggering Tests Added/Modified After a Bug Was Reported?

Motivation. Defects4J is commonly used by prior SBFL research as a benchmark to evaluate the accuracy of fault localization techniques [14, 15, 40, 177, 187, 203]. When localizing faults, SBFL techniques rely mainly on analyzing the coverage of the fault-triggering tests. However, prior research [40, 87, 177] suggests that Defects4J may contain some tests that were added by developers *after the bug was reported or fixed*. Such tests may contain developer knowledge of the bug, which compromises the reliability of the results of fault localization techniques. Therefore, in this RQ, we study the timelines of test modification/addition for every bug, starting from the creation of the bug report until its resolution. We also study whether or not the changes in tests bring developer knowledge of the bugs to the test. The findings will provide initial evidence on how many bugs from Defects4J whose fault-triggering tests may have developer knowledge.

Approach. We conducted a tool-assisted manual study on the timelines of events for every bug. We first collected the bug report, fault-triggering tests, and bug-fixing patches for all of the 809 studied bugs. We then identified the events (e.g., bug report creation) associated with each piece of information. We analyze these events automatically in relation to bug resolution and manually reviewed the test modifications to ensure their relevance to the bug. Below, we discuss our data collection process in detail.

Bug report creation and resolution date: To determine the time interval of the bug

Table 27: An overview of our studied systems from Defects4J v2.0.

System	#Bugs	LOC	#Tests	Fault-triggering Tests
Cli	39	4K	94	66
Closure	174	90K	7,911	545
Codec	18	7K	206	43
Collections	4	65K	1,286	4
Compress	47	9K	73	72
Csv	16	2K	54	24
Gson	18	14K	720	34
JacksonCore	26	22K	206	53
JacksonDatabind	112	4K	1,098	132
JacksonXml	6	9K	138	12
Jsoup	93	8K	139	144
JxPath	22	25K	308	37
Lang	64	22K	2,291	121
Math	106	85K	4,378	176
Mockito	38	11K	1,379	118
Time	26	28K	4,041	74
Total	809	409K	25,708	1,655

resolution and identify whether the fault-triggering tests were modified during this period, we collected the creation and resolution time of the bug reports. We retrieved the bug reports from the bug tracking systems (i.e., Jira and GitHub) using REST APIs and the bug IDs provided in Defects4J. We then extracted the creation and resolution time from the "Created" and "Resolved" fields (or the issue creation and closed dates on GitHub) of each bug report. By using the creation and resolution date, we are able to determine if a test was added/modified after a bug was reported or fixed.

Date of fault-triggering test creation and modification: We identified all the commits that are associated with the fault-triggering tests and analyzed when the commits happened (e.g., before or after the bug was reported/fixed). We used the git command "`git log -L:[funcname]:[file]`" to identify the list of commits that modified the fault-triggering test and the modification date.

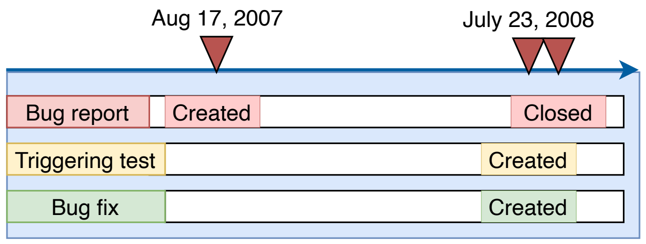
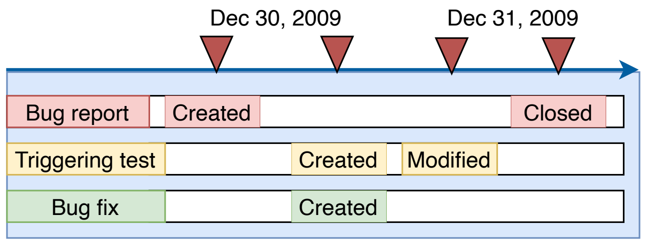
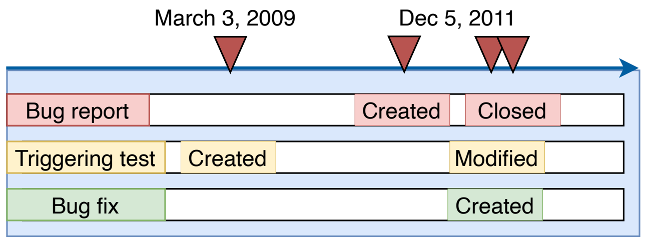
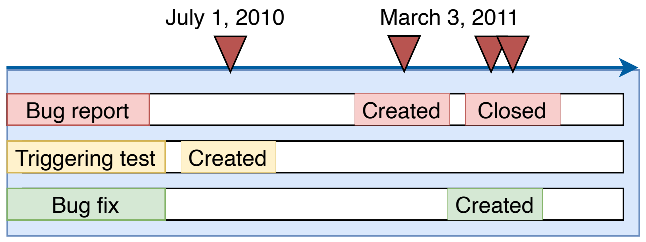
Bug fix date: In addition to the bug report creation/resolution time, we study the time of the bug fix to understand if a test was modified before or after the fix became available. We used the "`git log [commit]`" command to determine the exact date and time of the bug fixes. We then align the time of the bug fixes with modifications made to the fault-triggering tests.

Change patterns of the tests and their relevance to the bugs: We arranged the events – bug report creation and resolution, creation and modification of fault-triggering tests, and bug fix time – in chronological order to reconstruct the timeline. In particular, we focused on the creation and modifications of fault-triggering tests with respect to bug resolution. We used an automated script to sequence the events into timelines. We then manually studied the commit messages and related code changes to examine if the added/modified tests include developer knowledge about the bug. Our collected data is publicly available [1].

Results. ***We find that 77.2% of the fault-triggering tests in Defects4J include developer knowledge of the bugs, bringing hints on the groundtruth of the buggy location in the data.*** In total, we uncovered four timelines of change patterns on the fault-triggering tests. Table 28 shows the uncovered change patterns and corresponding timelines of the events. In summary, through our manual analysis, we find that developers' modification or addition of tests adds bug-related information to all of the tests from Pattern 1 and Pattern 2, and most of the tests from Pattern 3 (357/362). Below, we discuss each pattern in detail.

Pattern 1: Test created after bug report creation (53%). When developers are trying to fix a bug, they often rely on fault-triggering tests to understand and replicate the problem [20, 40, 52, 163, 178]. However, as illustrated in Table 28, we found that in Defects4J, a large portion of these fault-triggering tests may not exist before the bug report was created. The most common change pattern is that the tests were added to the system only after developers began to fix the bug. As an example, in CLI-144, the bug report was created on August 17, 2007. On July 23, 2008, developers created a new fault-triggering test called BugCLI144Test as part of the bug fixing commit (the test was named using the bug report ID). This new fault-triggering test was added because existing tests were unable to capture the reported bug. Thus, the test

Table 28: Timelines of the changes on the fault-triggering tests. Note that the same bug may belong to more than one pattern because a bug may have more than one fault-triggering test.

Patterns	Description	Example Timeline	#Tests	#Bugs
Pattern 1	Fault-triggering tests were newly added after the bug was reported.	 Aug 17, 2007 July 23, 2008 Bug report Created Closed Triggering test Created Bug fix Created	872 (53%)	558
Pattern 2	Fault-triggering tests were newly added then modified, after the bug was reported.	 Dec 30, 2009 Dec 31, 2009 Bug report Created Closed Triggering test Created Modified Bug fix Created	43 (2%)	30
Pattern 3	Fault-triggering tests were modified after the bug was reported and before the bug was marked as fixed.	 March 3, 2009 Dec 5, 2011 Bug report Created Closed Triggering test Created Modified Bug fix Created	362 (22%)	155
Pattern 4	Fault-triggering tests were not modified after the bug was reported and before the bug was marked as fixed.	 July 1, 2010 March 3, 2011 Bug report Created Closed Triggering test Created Bug fix Created	378 (23%)	150
Total			1,655	-

contains information on the groundtruth of the bug-fixing location. However, this test was marked as the fault-triggering test in Defects4J. Through manual inspection, we found that all the fault-triggering tests that belong to this pattern were added by developers as part of the bug-fixing process. Namely, nearly 50% of the fault-triggering tests in Defects4J were regression tests that were added deliberately to replicate/prevent the bug.

Pattern 2: Test created and modified after bug report creation (2%). As shown in Table 28, after a bug report is created, developers may create initial versions of the fault-triggering tests that require further enhancement (e.g., the initial version is not able to cover all possible scenarios). Nevertheless, in our manual analysis, similar to Pattern 1, we found that these fault-triggering tests were created for regression testing purposes and contained developer knowledge of the bug. As an example, in MATH-320, the developer initially added a bug fix and a new test to reproduce the bug. However, the developer commented that the fix was incomplete and shared the test to facilitate discussions with other developers. Later, the developer applied a patch to fix the bug along with the updated test. For all the tests that belong to Pattern 2, either the commit messages or the names of these fault-triggering tests contain the ID of the bug report, further confirming that they contain developer knowledge of the bug.

Pattern 3: Test modified after bug report creation (22%). In practice, some bugs reported by users or other developers cannot be revealed by existing tests. Hence, when fixing these bugs, developers may modify and enhance the tests for regression testing purposes. For example, in COMPRESS-10, the test UTF8ZipFilesTest was enhanced as part of the bug fix. Developers modified the assertions to better capture the bug. It is possible that the test modification is not related to the bug fix. However, after our manual analysis, we found that 99% (357/362) of the tests contained changes that alter the test execution for replicating the bug, while the remaining 1% (5/362) did not introduce new knowledge to the tests (e.g., code re-styling, and enabling or disabling a failed test). In short, we find that most modifications to the fault-triggering tests were done after the bug was reported and were adding developers' knowledge of the bug in the tests.

Pattern 4: Test unmodified during bug resolution (23%). We found that developers may fix the bug without making any changes to the fault-triggering tests. As an example

(CLOSURE-79), when the problem was initially uncovered, the fault-triggering test testPropReferenceInExterns3 failed. Developers work on the bug fix without having to change the test as it was working as intended (i.e., failing upon unexpected behaviour). Therefore, the fault-triggering tests were not changed during the resolution of the bug report. In total, we found 378 manually-verified fault-triggering tests that belong to this pattern.

Discussion. In our manual analysis, we identified four common change patterns on the fault-triggering tests. Based on the results of Pattern 1 and 2, *we observed that 55% (915/1,655) of the fault-triggering tests were created after the bug report was created.* These tests did not exist before the bugs were reported, so they could not have helped in identifying the bugs. Moreover, these tests contain developers' knowledge of the bug, adding hints on the groundtruth of the bug location. Leveraging these tests in fault localization introduces biases in how well code coverage can help locate the faults. We also observed that developers may modify the tests after the creation of the bug report, which is the case in 22% (362/1,655) of the studied fault-triggering tests. This can lead to a similar problem where tests were modified with bug specifications (i.e., developer knowledge). In particular, the modified version of the test can differ significantly from the initial version, and it may be modified to specifically address the problem described in the bug report. In short, we find that only 23% of the studied fault-triggering tests were actually able to detect the bugs (i.e., the tests failed).

The majority of the fault-triggering tests in Defects4J (77%) contain developer knowledge. These tests were either newly added or modified to replicate the bug or to prevent future regression. Only 23% of the tests were able to detect the bugs as intended (the tests failed).

7.2.3 RQ2: How Do Developers Modify the Fault-triggering Tests?

Motivation. When fixing a bug, developers may look for test failures to help in debugging and understanding the root causes of the bug [41, 70, 84, 89]. However, as we found in RQ1, many fault-triggering tests may not exist before the bug report. Even when they do exist, they can only trigger the bug after developers made modifications to them during the bug-fixing process. In this RQ, we manually study the modifications made by developers to the fault-triggering tests and discuss the common reasons behind them. Studying why developers modify these tests can help us understand what motivates them to make changes and how these changes relate to fixing the bugs.

Manual Study Process. We conduct our manual study on all 16 studied systems. For a 95% confidence level and a 5% confidence interval, we randomly sample 300 fault-triggering tests from 1,361 modified tests that were collected from Patterns 1, 2, and 3. We use the stratified sampling strategy [135] to obtain the number of samples for each studied system that is proportional to their total number of tests. For each test, we study its code changes, code comments, commit message, and corresponding bug report and bug fix to understand the potential motivation leading to its modification. Following prior studies [56, 92, 107, 109], we conduct our manual study in three phases using the open coding method.

Phase 1: We manually review the tests to create a preliminary list of categories for 100 fault-triggering tests that are randomly selected. We additionally list the justifications for creating each category in terms of its code changes, code comments, commit message, bug report, and bug fix. We revise the list of categories and address any discrepancies.

Phase 2: We review the remaining of the 300 tests independently based on the discussed categories and assign the test to one of the uncovered categories from Phase 1.

Phase 3: We compare their assigned categories and discuss any disagreements until we reach a consensus. During this phase, our results show that we reached a substantial level of agreement, achieving a Cohen's Kappa of 0.86 [51].

Results. ***Other than test addition, most changes on the fault-triggering tests are related to improving test oracles or updating tests in response to source code changes.*** In our manual analysis, we uncover five categories of reasons why developers changed the fault-triggering tests as shown in Table 29. Below, we discuss each category in detail.

Table 29: List of categories of the modifications to fault-triggering tests.

Category	**Description**	**Count**	**Percentage**
Adding New Test	Developers added a new test to reproduce the bug.	189	63%
Test Co-evolution	Developers modified the expected output in tests to cope with source code evolution.	58	20%
Adding New Assertion	Developers added a new assertion to replicate the bug and for regression testing purposes.	41	13%
Improving Test Logic During Bug Fixes	While modifying a test to replicate the bug, developers also enhance the structure or the logic in the test.	7	2%
Others	Developers reformated the style in the test (e.g., indentation), or eliminated code that is not essential to the reproduction of faults.	2	1%

Adding New Test (189/300, 63%). We found that some tests are specifically created to aid in the reproduction of a bug. As developers work towards resolving a bug, they may create new tests designed to replicate the problematic behavior. Once the bug is fixed, developers often incorporate these tests into their patch to ensure that the bug does not re-occur in future releases (i.e., regression testing). All test changes from this category belong to Patterns 1 and 2 that we found in RQ1. These tests are often named after the bug report ID to ease traceability and management.

Test Co-evolution (58/300, 19%). We find that developers may change the test code to accommodate the bug-fixing changes in the source code. For example, developers fixed bug #207 in JacksonCore by modifying the calcHash method in the source code (as shown below). In addition to the bug fix, developers patched the test testSyntheticWithBytesNew responsible for verifying the stability of the hash code, which failed after the new bug fix. One of the developers highlighted in a comment that this bug fix improved "*hashing [with regards to] existing test*". The system is expected to have a different distribution of collision count. Therefore, developers modified the test to match its expected output to the actual behavior of the system.

```
// ByteQuadsCanonicalizer.java
public int calcHash(int q1){
    int hash = q1 ^ _seed;
    hash += (hash >>> 16); // to xor hi- and low- 16-bits
    - hash ^= (hash >>> 12); // as well as lowest 2 bytes
    + hash ^= (hash << 3); // shuffle back a bit
    + hash += (hash >>> 12); // and bit more
    return hash;
}

// TestSymbolTables.java
public void testSyntheticWithBytesNew() throws IOException{
    ...
    // anywhere between 70-80% primary matches
    - assertEquals(8524, symbols.primaryCount());
    + assertEquals(8534, symbols.primaryCount());
}
```

Adding New Assertion (41/300, 14%). During the bug-fixing process, developers may add new assertions to help reproduce a bug (belongs to Pattern 3 from RQ1). Typically, as we found in our manual study, the modification to the test involves adding single-line assertion statements. For example, as seen in the test testCreateNumber below (LANG-822), developers added a new assertion to reproduce the bug. The assertion checks whether the method createNumber can execute as expected if the input starts with the string "--". According to the developer's comment, there are numerous edge cases that can expose problematic behaviors when calling the method, so whenever a new edge case arises, it is added to the test as a new assertion statement.

```
public void testCreateNumber() {
   // a lot of things can go wrong
...
   // LANG-693
   assertEquals("createNumber(String) LANG-693 failed", Double.valueOf(Double.MAX_VALUE),
        NumberUtils.createNumber("" + Double.MAX_VALUE));
+
+ // LANG-822
+ assertFalse("createNumber(String) LANG-822 succeeded", checkCreateNumber("--1.1E-700F"));
}
```

Improving Test Logic During Bug Fixes (7/300, 2%). Developers may modify tests to change the logic of the tests when trying to fix a bug. Typically, these modifications happen when developers are trying to replicate the bug in a test. Examples of modifications include simplifying complex logic and reorganizing the code structure, which can alter the execution of the test. An example of a test modification in this category is the JsonAdapterNullSafeTest from issue #800 in GSON. During the bug-fixing process, developers discussed providing a "simpler" test to reproduce the bug. They incorporated new changes that simplified the initialization of the Device class from the test and removed logic that was irrelevant in triggering the fault. This contributed to improving the quality and maintenance of the test.

```
public void testNullSafeBugDeserialize() throws Exception {
  - String json = "\"\\\"id\\\":\\\"ec57803e2\\\",\\\"category\\\":2\"";
  - Device device = gson.fromJson(json, Device.class);
  + Device device = gson.fromJson("'id':'ec57803e2'", Device.class);
  ...
  @JsonAdapter(Device.JsonAdapterFactory.class)
  private static final class Device {
    String id;
    - int category;
    - Device(String id, int category)
    + Device(String id)
      ...
  }
```

Others (2/300, 1%). We find two other reasons why developers may modify the tests, which do not belong to any of the above categories. In particular, we observe that developers may reformat the source code files without necessarily changing any of the logic in the code. For instance, developers removed the extra indentation in the test to improve its readability. We also observe that developers may clean up the test which entails eliminating any obsolete variables or comments.

Developers often add new tests (63%) or assertions (14%) to replicate the bug, and sometimes improve tests (21%), e.g., reflecting the fix in source code or improving test logic, to regression test the bugs that they are fixing.

7.2.4 RQ3: How does Having Developer Knowledge in Fault-triggering Tests Affect Fault Localization Results?

Motivation. In the previous RQs, we found that many tests are newly added or modified for replicating the bugs and regressing testing. Such tests contain developer knowledge of the bugs that may affect the effectiveness of fault localization techniques. Therefore, in this RQ, we study the impact of using developer-modified tests for fault localization. The findings will provide insight into the impact of using these tests for fault localization, and whether there is a need to improve the benchmark.

Approach. We study the impact of having developer-modified tests on the results of state-of-the-art SBFL techniques. Among existing fault localization techniques, SBFL is one of the most widely studied techniques as it provides an accurate ranking of the potentially buggy statements [14, 81, 103]. SBFL techniques are also an important building block for automated program repair techniques since they rely on SBFL to provide a list of potentially buggy locations to start repairing [113]. SBFL techniques differentiate the buggy statements from the non-buggy ones through the program spectrum (code coverage profile). Intuitively, a statement covered by more failed tests but fewer passed tests is more likely to contain the bug. Therefore, developer-modified fault-triggering tests can significantly impact the performance of SBFL, as these tests may not have existed when the bug was first reported, or might not have been able to trigger the bug (i.e., do not fail) when initially introduced to the system.

In particular, we study the effectiveness of SBFL techniques on the bugs whose fault-triggering tests existed before the bug report and were modified after the bugs were reported. This is done to enable a comparison of fault localization techniques with and without developer knowledge of the bug. The tests that belong to Pattern 3 existed before the bug report and were modified after the bugs were reported, and we found that most of the modifications added developer knowledge of the bug to the tests (RQ2). Therefore, we perform our study on bugs that belong to Pattern 3. To study the impact of including developer knowledge on SBFL, we compare the performance of SBFL techniques in two versions: 1) *vBuggy:* when the bug was first reported, and 2) *vD4J:* the version provided by Defects4J (the tests were modified during bug fixes). We perform our comparison on all 155 bugs that contain fault-triggering tests belonging to Pattern 3.

Since Defects4J only provides the test coverage and test execution result for *vD4J*, we need to collect the data for *vBuggy*. To collect *vBuggy*, for each bug, we extract the timestamp when the bug report was created and identify the nearest commit *before* the bug was reported. In practice, the bugs remain unfixed in these commits, and if there are any test failures, they are more likely to be related to the bug. We checkout these commits on Git, compile the systems, and execute the tests. Note that, if a bug does not have any failed tests in either *vBuggy* or *vD4J*, we exclude the bug from our analysis for a more direct comparison between the two versions. Since most systems do not report test coverage, we manually modified the systems to use JaCoCo to collect the coverage information. In total, we spent hundreds of hours configuring JaCoCo and resolving compilation issues such as missing dependencies and incompatible environment settings. We release the data publicly to encourage replication and future research [1].

After collecting the data for *vBuggy*, we apply SBFL techniques on both *vBuggy* and *vD4J* and compare the localization result. In particular, we use four following commonly used SBFL techniques [33]: Ochiai [14], Tarantula [77], DStar [182], and Barinel [17]. To evaluate the fault localization techniques, we use the following metrics:

Top-N is defined as the number of faults with at least one faulty statement correctly identified within the first N statements in the ranking. Therefore, a better Top-N result indicates that developers are able to find faulty statements by examining fewer statements. We set N to 1, 3, and 5 in our evaluation.

Mean Average Rank (MAR) first calculates the average rank of all the faulty statements for a bug. Then, MAR calculates the average of the ranks from all the bugs. A smaller value means that the faulty statements are ranked earlier.

Mean First Rank (MFR) calculates, for all the bugs, the mean of the first faulty statement in the ranked result. Therefore, a smaller value means that the technique, on average, is able to identify a faulty statement early in the ranked list.

In contrast to MAP and MRR discussed in the previous chapters, MAR and MFR represent the ranked positions instead of the inverse of the rank positions. The use of these two metrics in this chapter enables a direct comparison of the impact of future tests on the rank position of the faulty statements.

Results. ***For all the four fault localization techniques that we studied, the***

Table 30: Fault localization results for the bugs that belong to Pattern 3 (fault-triggering tests exist but were modified). We compare the results of running the techniques in the commits before the bugs were reported (*vBuggy*) and in the commits that were provided by Defects4J (*vD4J*). Top-N is the number of times a faulty element ranked in Top-N.

Technique	**Version**	**Bugs**	**Top-1**	**Top-3**	**Top-5**	**MFR**	**MAR**
Ochiai	*vBuggy*	118	3	3	3	2,965 (-415%)	3,254 (-268%)
	vD4J	118	49	64	70	714	1,214
Tarantula	*vBuggy*	118	4	4	4	2,988 (-424%)	3,244 (-262%)
	vD4J	118	47	62	66	705	1,238
D-Start	*vBuggy*	118	10	11	15	2,827 (-104%)	3,231 (-701%)
	vD4J	118	29	39	44	273	461
Barinel	*vBuggy*	118	3	3	3	2,954 (-378%)	3,282 (-272%)
	vD4J	118	45	60	65	781	1,208

localization results degrade significantly on vBuggy compared to vD4J on bugs in Pattern 3. Table 30 shows the fault localization (FL) results on *vBuggy* and *vD4J* for the bugs whose fault-triggering tests belong to Pattern 3 (fault-triggering tests exist but were modified after the bug was reported). Out of the 155 unique bugs that we considered, 118 of them caused test failures. The remaining 25 bugs either did not lead to any test failures or their commits before the bug report (*vBuggy*) were too old and could not be retrieved. Thus, we perform our analysis on 118 bugs. Table 30 shows that the results in *vBuggy* are significantly worse than that of *vD4J* for all the metrics. All four FL techniques in *vBuggy* have much fewer times of a faulty element ranked in Top-1, Top-3, and Top-5 than that in *vD4J*. The finding indicates that in the best scenario 15 of the bugs have faulty statements that are ranked early in the list. In comparison, for *vD4J*, the number of a faulty element being ranked in Top-1 and Top 5 is around 40 and 60, respectively. The MFR and MAR results for *vBuggy* are in the range of 3,000, whereas the results for *vD4J* are in the range of 200 to 1200. In other words, most faulty statements are ranked very low in *vBuggy* and the ranking results cannot help developers with locating the bugs. The decrease in MRF and MAR value are in the range of 100% to 700%. In short, the findings

Table 31: Fault localization results for the bugs that belong to Pattern 4 (the fault-triggering tests were not modified). We compare the results of running the techniques in the commits before the bugs were reported (*vBuggy*) and in the commits that were provided by Defects4J (*vD4J*). Top-N is the number of times a faulty element ranked in Top-N.

Technique	**Version**	**Bugs**	**Top-1**	**Top-3**	**Top-5**	**MFR**	**MAR**
Ochiai	*vBuggy*	105	2	2	6	2,097 (-245%)	2,561 (-203%)
	vD4J	105	10	18	26	855	1,264
Tarantula	*vBuggy*	105	2	2	6	2,100 (-247%)	2,587 (-206%)
	vD4J	105	12	17	28	851	1,254
D-Start	*vBuggy*	105	2	2	5	2,073 (-250%)	2,553 (-195%)
	vD4J	105	10	16	23	829	1,309
Barinel	*vBuggy*	105	2	2	5	2,075 (-234%)	2,564 (-185%)
	vD4J	105	9	17	28	888	1,389

indicate that, in *vBuggy*, the fault-triggering tests (without Pattern 3 tests having prior developer knowledge) are not able to help locate the bugs.

As a comparison, we also study the effectiveness of fault localization (FL) techniques on *vBuggy* and *vD4J* for the bugs that belong to Pattern 4 (the fault-triggering tests were not modified). Different from the bugs that belong to Pattern 3, as shown in Table 28, Pattern 4 consists of 150 bugs where the fault-triggering tests were not modified by developers. This implies that the bugs of Pattern 4 may provide a more realistic setting for fault localization. We perform similar analysis on bugs in Pattern 4. Out of the 150 unique bugs we analyzed, 105 had test failures, while the remaining 45 bugs either had no test failures or the commits before the bug report (vBuggy) were too old to be retrieved. Surprisingly, we find that the fault localization results for the bugs that belong to Pattern 4 also have become worse. Table 31 shows that for Top-1, Top-3, and Top-5, *vBuggy* ranges from 2 to 6 while *vD4J* ranges from 9 to 28. Similarly, MFR and MAR are much higher for *vBuggy* (in the range of 2,000), meaning that developers need to investigate an average of 2,000 ranked results before find the faulty ones. The MFR and MAR are in the range of 800 to 1,300 for *vD4J*. The decrease in MRF and MAR value is around 200%.

The fault localization results on *vBuggy* (MFR and MAR around 3,000) are significantly worse than that on *vD4J* (MFR and MAR around 700 and 1200). Our findings show that the fault localization results using Defects4J may be significantly different from practical settings.

Discussion. Since the effectiveness of fault localization techniques degrades significantly, we further investigate the possible causes, other than that the tests contain developer knowledge of the bug. We examine how many bugs whose fault-triggering tests (from Pattern 3 and 4) actually failed in *vBuggy* (i.e., the version when the bug was first reported). Note that our study focuses on 263 out of 305 bugs, as some earlier versions of the system could not be retrieved. Table 32 shows the number of bugs with failed fault-triggering tests in *vBuggy*. We noticed that in both Pattern 3 and Pattern 4, a small percentage (15%) of fault-triggering tests did not cause any failure in *vBuggy*, indicating that they were unable to detect the fault. On the other hand, the remaining fault-triggering tests (85%) failed and assisted in fault localization. Through manual analysis of bugs in Pattern 4, we observe that even though developers did not modify the fault-triggering tests, they may still have introduced developers' knowledge affecting fault localization results. Developers may introduce their knowledge in different ways, such as modifying functions involved in test execution or adjusting the test setup configuration. Let's take bug Lang 57 as an example. In this bug, the test suite LocaleUtilsTest was modified by adding an extra setUp method that runs before each test case. This modification was made after the bug report was submitted. The purpose of the added setUp method is to ensure that fault-triggering tests fail if the fields are not correctly initialized before the test execution starts. Thus, the results of fault localization is overestimated in vD4J as developers' knowledge was introduced in the test setup configuration. Our finding shows that developers' knowledge may not always be directly incorporated into the fault-triggering tests themselves. Future research may explore bugs from Pattern 4 to investigate and characterize the instances where developers' knowledge is introduced outside of the fault-triggering tests.

Future studies may consider utilizing the fault-triggering tests that were not modified by developers after the bug report to investigate and characterize instances where developers' knowledge is introduced outside of the fault-triggering tests.

Table 32: The number of bugs with failed fault-triggering tests on the buggy commit (*vBuggy*). We consider the bugs whose fault-triggering tests belong to Pattern 3 and Pattern 4 (Table 28).

Project	**Total bugs**	**#Bugs with no failed tests**		**#Bugs with failed tests**	
		Pattern 3	Pattern 4	Pattern 3	Pattern 4
Cli	17	0	2	9	6
Closure	93	11	3	32	47
Codec	5	1	0	3	1
Collections	0	0	0	0	0
Compress	14	1	2	8	3
Csv	2	1	0	1	0
Gson	9	1	6	1	1
JacksonCore	7	2	3	1	1
JacksonXml	1	1	0	0	0
JacksonDatabind	17	0	0	10	7
JxPath	1	1	0	0	0
Lang	27	0	0	22	5
Math	36	2	2	30	2
Mockito	31	1	0	2	28
Time	3	0	0	2	1
Total	263	22	18	121	102

7.3 Discussion and Future Work

In this chapter, we studied the bugs and their corresponding fault-triggering tests in Defects4J. We classified the bugs based on whether their tests contain developer knowledge and also the test coverage data on the buggy commit (i.e., the commit prior to the bug report creation). We made the dataset publicly available [1] and we believe that the dataset can inspire future research. Below, we discuss the implication of our findings and potential future research directions.

7.3.1 Future research may use our annotated data to re-evaluate fault localization and automated program repair techniques.

We find that a majority of the fault-triggering tests in Defects4J contain developer knowledge, and such knowledge can degrade the fault localization results. Our findings provide insights into the effectiveness of fault localization techniques in a more realistic setting where developers may use these techniques in practice (e.g., analyze the failing tests associated with a reported bug). Our dataset also annotates the fault-triggering tests and whether or not they contain developer knowledge. We believe the dataset can be used in three directions. First, future studies can leverage the dataset to re-evaluate the effectiveness of fault localization or automated program repair techniques based on fault-triggering tests that do not have developer knowledge. Second, future research may use our dataset to have a separate evaluation of the bugs (e.g., with and without developer knowledge) in Defects4J. Third, future studies may use our dataset to investigate the characteristics of fault-triggering tests that require less maintenance during bug fixes, such as tests that involve little to no code changes (e.g., tests in Pattern 4).

7.3.2 Future studies may need to consider developers' bug-fixing process when creating benchmarks.

In RQ2, we found that fault-triggering tests may not be available when developers receive a bug report. Developers may then develop fault-triggering tests while working on a bug fix to replicate the bug. Hence, our findings call for more empirical studies to further understand developers' bug-fixing activities, and how to consider such activities when creating benchmark datasets. Future studies may use our dataset as an initial step to understand the bug-fixing process and expand the study on other systems and settings.

7.3.3 There is a need for more comprehensive and realistic benchmarks.

To help facilitate software testing research, researchers have proposed several bug benchmarks. Among these benchmarks, Defects4J [79] is the most popular and widely used benchmark. However, as we found in this chapter, some fault-triggering tests in Defects4J contain developer knowledge of the bug, which affect the result of fault localization techniques. The bugs and tests from Defects4J were manually extracted and may contain data from "future" commits. As a result, we believe that there is a need for benchmarks that are more comprehensive and realistic. The T-Evos dataset, discussed in Chapter 5, compiles and executes a consecutive sequence of commits in multiple systems while collecting test coverage data. The continuous data from benchmarks like T-Evos provides a more realistic setting for evaluating fault localization and automated program repair techniques under modern software development. It also highlights the need for more comprehensive and realistic benchmarks in the field of software testing research.

7.4 Threats to Validity

Internal Validity. Threats to internal validity are related to experiment errors or biases. One main threat comes from the human analysis in our study. In RQ1, we conducted a manual analysis to assess the relevance of test modifications to bugs. However, the majority of the work in extracting events related to bug resolution was performed automatically, with manual verification limited to identifying any errors in the generated results. We manually analyzed and categorized the modifications on fault-triggering tests in RQ2. However, we adopted the manual analysis approaches used in prior studies [56, 92, 107, 109] to mitigate subjectivity. Three phrases were used and we were independently involved in the analysis. The analysis results achieve a Cohen's Kappa of 0.86, which indicates a substantial level of agreement [51].

External Validity. Threats to external validity are related to the generalization of our results. In this chapter, we mainly focus on the Defects4J dataset. Conducting the study on different datasets with test cases and bug information may have different results. The Defects4J dataset is the most used benchmark in the important software engineering tasks, such as fault localization and automated program repair. Quite a large body of literature, e.g., in automated program repair, heavily rely on the Defects4J dataset for the evaluation on Java programs. The discussions, such as developer knowledge in fault-triggering tests and the identified categorizes of modifications on fault-triggering, derived from the analysis of Defects4J in this chapter could be beneficial to future research, such as evaluating the research in the settings of no developer knowledge that is known in the tests.

Chapter 8

Conclusion

This chapter provides a summary of the key ideas presented in this dissertation. In the following section, we propose future work related to mining software repositories and leveraging program analysis to improve the debugging process.

Software debugging is a challenging and time-consuming task [32, 38, 41, 46]. To expedite this process, researchers have developed automated debugging techniques. One such technique that has gained significant attention is fault localization. By utilizing fault localization techniques, developers can automatically identify and locate faults in the source code, thereby speeding up the debugging process. While the industry has shown interest in these techniques, their effectiveness may be limited, hindering their widespread adoption. Furthermore, the integration of fault localization in modern software development has not been extensively studied, further limiting their evaluation in practical settings. We believe that leveraging operational data can enhance the effectiveness and applicability of fault localization in modern software development, particularly in continuous integration settings. In this dissertation, we highlight two types of operational data that can extend the adoption of fault localization in practice: user-reported logs, which facilitate the reconstruction of execution paths and provide deeper debugging information, and continuous and finer-grained changes, which enable the detection of new faults and enhance the adaptability of fault localization in modern software development. Our results demonstrate the value of our approaches for software engineering practitioners and highlight the importance of extending the reach of automated debugging solutions.

8.1 Contributions

This dissertation focuses on first understanding the debugging information, then proposing approaches to help build more effective and efficient fault localization techniques. The contributions of this dissertation also fill the research gap of missing continuous integration fault localization benchmark, as demonstrated in our large-scale longitudinal study in Chapter 5. In our study conducted in Chapter 6, we also identify the bias that may exist, in which state-of-the-art fault localization techniques perform significantly worse in the absence of developer knowledge in the tests. Below, we outline the key contributions of this dissertation:

1. **Leveraging User-reported Logs in Debugging.** We conduct an empirical study on the challenges that developers may encounter when analyzing the user-provided logs and their benefits. Our findings show that: 1) BRWL takes longer time (median ranges from 3 to 91 days) to resolve compared to BRNL (median ranges from 1 to 25 days). We also find that reporters may not attach accurate or sufficient logs (i.e., developers often ask for additional logs in the Comments section of a bug report), which extends the bug resolution time. 2) Logs often provide a good indication of where a bug is located. Most bug reports (73%) have overlaps between the classes that generate the logs and their corresponding fixed classes. However, there is still a large number of bug reports where there is no overlap between the logged and fixed classes. 3) Our manual study finds that there is often missing system execution information in the logs. Many logs only show the point of failure (e.g., exception) and do not provide a direct hint on the actual root cause. In fact, through call graph analysis, we find that 28% of the studied bug reports have the fixed classes reachable from the logged classes, while are not visible in the logs attached in bug reports. To facilitate the reproducibility, we have made the data available online [11].

2. **Adopting Logs in Fault Localization Techniques.** To assist developers, we first propose Pathidea, an IRFL approach that utilizes logs from bug reports to reconstruct execution paths and enhance fault localization results. Pathidea uses static analysis to create a file-level call graph, and re-constructs the call paths from the reported logs. We find that Pathidea achieves a high recall (up to 51.9% for Top@5). On average, Pathidea achieves an improvement that

varies from 8% to 21% and 5% to 21% over BRTracer in terms of Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) across studied systems, respectively. Moreover, we find that the re-constructed execution paths can also complement other IRFL approaches by providing a 10% and 8% improvement in terms of MAP and MRR, respectively. Finally, we conduct a parameter sensitivity analysis and provide recommendations on setting the parameter values when applying Pathidea. To facilitate the reproducibility, we have made the data available online [37].

3. **Benchmarking for Automated Testing Research.** To fill the research gap of missing CI benchmark, we introduce T-Evos [12], a large-scale dataset on test result and coverage evolution, covering 8,093 commits across 12 open-source Java projects. Our dataset includes the evolution of statement-level code coverage for every test case (either passed and failed), test result, all the builds information, code changes, and the corresponding bug reports. We conduct an initial analysis to demonstrate the overall dataset. In addition, we conduct an empirical study using T-Evos to study the characteristics of test failures in CI settings. We find that test failures are frequent, and while most failures are resolved within a day, some failures require several weeks to resolve. We highlight the relationship between code changes and test failure, and provide insights for future automated testing research. Our dataset may be used for future testing research and benchmarking in CI. Our findings provide an important first step in understanding code coverage evolution and test failures in a continuous environment.

4. **Automatically Locating Faults in Continuous Integration.** While the continuous nature of CI requires the code changes to be atomic and presents fine-grained information on what part of the system is being changed, traditional SBFL techniques do not benefit from it. We propose to integrate the code and coverage change information in fault localization under CI settings. First, code changes show how faults are introduced into the system, and provide developers with better understanding on the root cause. Second, coverage changes show how the code coverage is impacted when faults are introduced. This change information can help limit the search space of code coverage, which offers more opportunities for improving fault localization techniques. Based on the above

observations, we propose three new change-based fault localization techniques, and compare them with Ochiai, a commonly used SBFL technique. Our results show that all three change-based techniques outperform Ochiai on the Defects4J dataset. In particular, the improvement varies from 7% to 23% and 17% to 24% for average MAP and MRR, respectively. Moreover, we find that our change-based fault localization techniques can be integrated with Ochiai, and boost its performance by up to 53% and 52% for average MAP and MRR, respectively. To facilitate the reproducibility, we have made the data available online [10].

5. **Studying Data Cleanness in Fault Localization.** To facilitate research in software testing, researchers have proposed several benchmark datasets for the community to evaluate fault localization techniques. Among them, Defects4J is the most widely used benchmark due to its clean and controlled environment for studying real bugs collected from popular open source systems. However, prior research suggests that Defects4J may contain some tests that were added by developers after the bug was reported or fixed. Such tests may contain developer knowledge of the bug, which compromises the effectiveness of fault localization techniques. Thus, we conduct a comprehensive study on the fault-triggering tests in Defects4J. We study the timelines of the changes that developers made to the fault-triggering tests with respect to bug report creation time. Then, we study the effectiveness of SBFL techniques without developer knowledge in the tests. We found that 1) 55% of the fault-triggering tests were newly added to replicate the bug or to test for regression; 2) 20% of the fault-triggering tests were modified after the bug reports were created, containing developer knowledge of the bug; 3) developers often modify the tests to include new assertions or change the test code to reflect the changes in the source code; and 4) the performance of SBFL techniques degrades significantly (up to –415% for Mean First Rank) when evaluated on the bugs without developer knowledge. Our results can help future studies conduct in-depth evaluations of their SBFL techniques in different settings when using Defects4J. Our findings also provide insight into the development activities that need to be considered when creating future bug benchmarks. To facilitate the reproducibility, we have made the data available online [1].

8.2 Future Research Directions

In this dissertation, we have proposed several fault localization techniques that are effective and easy to integrate within modern software development settings. However, there are still many challenges that can be addressed in future research. In the following sections, we discuss each of these challenges.

8.2.1 Providing Assistance on Which Logs to Include in Bug Reports

The earlier chapters of this dissertation investigate the use of user-reported logs in bug reports, focusing on their role in debugging (Chapter 3) and fault localization (Chapter 4). However, more often than not, reporters are unable to provide accurate logs that effectively illustrate the problem. Therefore, future research should assist reporters in providing more accurate logs that can better facilitate the debugging process.

8.2.2 Detecting and Understanding Logging Code Evolution

As shown in Chapter 3, even when reporters provide high-quality logs, it is not uncommon for logging statements or methods mentioned in stack traces to be removed from the source code when developers start debugging. For developers who are not familiar with the system, such changes in logging statements can pose additional challenges during debugging. Future studies should consider analyzing the software development history and assisting developers in locating user-provided logs, specially when the corresponding logging statements or methods have been deleted or moved. Additionally, understanding the evolution of logging code in relation to the source code (i.e., logs and source code co-evolution) is crucial. Future studies should investigate the mismatches between user-provided logs and the logging statements in most recent code repository, taking into account the continuous and frequent changes that occur in modern software development.

8.2.3 Leveraging Operational Data to Improve Software Quality

As demonstrated in Chapter 3 and Chapter 4, we can leverage user-reported logs to assist developers in debugging and improving software quality. However, modern software development offers additional operational data, particularly in the context of CI, which contain rich debugging information. For instance, in Chapter 5, we introduced T-Evos, a dataset that captures the continuous evolution of test status and code coverage. This dataset can be utilized to study the quality of test code throughout software evolution. Future studies can further explore this dataset to investigate the evolution of test code over time and to help improve it.

8.2.4 Evaluating Data Cleanness in Fault Localization and Automated Program Repair

As highlighted in Chapter 7, a majority of the fault-triggering tests in Defects4J contain developer knowledge, which can affect fault localization results. These findings provide insights into the effectiveness of fault localization techniques in a more realistic setting, where developers may utilize these techniques in practice (e.g., analyzing failing tests associated with reported bugs). Our dataset also annotates whether fault-triggering tests contain developer knowledge. Future studies can leverage this dataset to re-evaluate the effectiveness of fault localization or automated program repair techniques based on fault-triggering tests that do not involve developer knowledge. Additionally, the dataset can be used to investigate the characteristics of fault-triggering tests that require less maintenance during bug fixes, such as tests that involve minimal or no code changes.

Bibliography

[1] Public repository on data cleanness in defects4j. https://github.com/\double-anonymous-submission/ASE-paper442. 2022.

[2] Library to parse and work with the c++ ast, . URL https://github.com/foonathan/cppast.

[3] ast — abstract syntax trees — python 3.9.1 documentation, . URL https://docs.python.org/3/library/ast.html.

[4] The defects4j dataset version 2.0.0. https://github.com/rjust/defects4j. 2022.

[5] Cobertura. https://cobertura.github.io/cobertura/, 2021. Last accessed May 5 2021.

[6] Jacoco. https://www.eclemma.org/jacoco/, 2021. Last accessed May 5 2021.

[7] Maven surefire report plugin. https://maven.apache.org/surefire/maven-surefire-report-plugin/, 2021. Last accessed May 5 2021.

[8] Deflaker. https://www.deflaker.org/, 2022. Last accessed February 28 2022.

[9] Gzoltar. https://gzoltar.com/, 2022. Last accessed February 28 2022.

[10] Leveraging-change-information repository. https://github.com/anonymized-datascientist/Leveraging-Change-Information, 2022. Last accessed March 7 2022.

[11] Log in bug reports empirical data. https://github.com/SPEAR-SE/LogInBugReportsEmpirical_Data, 2023. Last accessed June 1 2023.

[12] T-evos. `https://github.com/T-Evos/T-Evos`, 2023. Last accessed June 1 2023.

[13] T-evos zenodo repository. `https://zenodo.org/record/5500821`, 2023. Last accessed June 1 2023.

[14] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[15] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[16] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[17] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, 2009.

[18] Akira Ajisaka. How to contribute to apache hadoop. `https://cwiki.apache.org/confluence/display/HADOOP/How+To+Contribute`. Last accessed June 1 2023.

[19] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 520–523. IEEE, 2011.

[20] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. Fonte: Finding bug inducing commits from failures. *arXiv preprint arXiv:2212.06376*, 2022.

[21] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 361–370, 2006.

[22] Apache. Aapache JIRA, 2019. URL `https://issues.apache.org/jira/?` Last accessed: Feb. 1, 2019.

[23] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188, 2016.

[24] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444. IEEE, 2018.

[25] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190, 2015.

[26] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367. IEEE, 2017.

[27] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, November 2008.

[28] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 27–30, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370757.

[29] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. Orca: Differential bug localization in large-scale services. In *13th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 493–509. USENIX Association, 2018.

[30] Francesco A. Bianchi, Mauro Pezzè, and Valerio Terragni. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 705–716, 2017.

[31] Marcel Böhme and Abhik Roychoudhury. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 105–115, 2014.

[32] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 229, 2013.

[33] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*, pages 378–381, 2012.

[34] Yu Cao, Hongyu Zhang, and Sun Ding. Symcrash: Selective recording for reproducing crashes. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 791–802, 2014.

[35] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. How the experience of development teams relates to assertion density of test classes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 223–234. IEEE, 2019.

[36] An Ran Chen. Studying and leveraging user-provided logs in bug reports for debugging assistance. Master's thesis, Concordia University, 2019.

[37] An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. Replication package. URL `https://github.com/SPEAR-SE/Pathidea_Data`.

[38] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. Demystifying the challenges and benefits of analyzing user-reported logs in bug reports. *Empirical Software Engineering*, 26(1):1–30, 2021.

[39] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, 2021.

[40] An Ran Chen, Tse-Hsun Chen, and Junjie Chen. How useful is code change information for fault localization in continuous integration? In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[41] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. T-evos: A large-scale longitudinal study on ci test execution and failure. *IEEE Transactions on Software Engineering*, 2022.

[42] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing and detecting anti-patterns in the logging code. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 71–81. IEEE / ACM, 2017. doi: 10.1109/ICSE.2017.15.

[43] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, pages 305–316, 2018.

[44] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 223–234, 2019.

[45] Junjie Chen, Haoyang Ma, and Lingming Zhang. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 78–89, 2020.

[46] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 82–91, 2014.

[47] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919, 2016.

[48] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 243–252, 2017.

[49] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 184–191. IEEE, 2018.

[50] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, November 1993.

[51] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[52] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor S Trivedi. Fault triggers in open-source software: An experience report. In *2013 IEEE 24th International symposium on software reliability engineering (ISSRE)*, pages 178–187. IEEE, 2013.

[53] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436, 2007.

[54] Tung Dao, Lingming Zhang, and Na Meng. How does execution information help with information-retrieval based bug localization? In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 241–250, 2017.

[55] Steven Davies and Marc Roper. Bug localisation through diverse sources of information. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 126–131. IEEE, 2013.

[56] Zishuo Ding, Jinfu Chen, and Weiyi Shang. Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet? In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1435–1446. IEEE, 2020.

[57] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, 2020.

[58] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, 2014.

[59] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[60] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE-SEIP '14, pages 24–33, 2014.

[61] Gregory Gay and René Just. Defects4j as a challenge case for the search-based software engineering community. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*, pages 255–261. Springer, 2020.

[62] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2017.

[63] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[64] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[65] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, 2018.

[66] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 426–437, 2016.

[67] Michael Hilton, Jonathan Bell, and Darko Marinov. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 53–63, 2018.

[68] IEEE. Ieee definitions. `https://standards.ieee.org/standard/610_12-1990.html`, 2020. Last accessed June 1, 2023.

[69] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014.

[70] Bo Jiang, Zhenyu Zhang, TH Tse, and Tsong Yueh Chen. How well do test case prioritization techniques support statistical fault localization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 99–106. IEEE, 2009.

[71] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, TH Tse, and Tsong Yueh Chen. How well does test case prioritization integrate with statistical fault localization? *Information and Software Technology*, 54(7):739–758, 2012.

[72] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 502–514. IEEE, 2019.

[73] Jiajun Jiang, Yingfei Xiong, and Xin Xia. A manual inspection of defects4j bugs and its implications for automatic program repair. *Science china information sciences*, 62:1–16, 2019.

[74] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. Extracting concise bug-fixing patches from human-written patches in version control systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 686–698. IEEE, 2021.

[75] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, 2012.

[76] Jira. Jira rest apis, 2020. URL `https://developer.atlassian.com/server/jira/platform/rest-apis/`. Last accessed: Feb. 1, 2020.

[77] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 273–282, 2005.

[78] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

[79] René Just, Darioush Jalali, and Michael D Ernst. 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*, pages 437–440, 2014.

[80] René Just, Chris Parnin, Ian Drosos, and Michael D Ernst. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 287–297, 2018.

[81] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125. IEEE, 2017.

[82] Dong Jae Kim, Nikolaos Tsantalis, Tse-Hsun Peter Chen, and Jinqiu Yang. Studying test annotation maintenance in the wild. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 62–73. IEEE, 2021.

[83] Dong Jae Kim, Bo Yang, Jinqiu Yang, and Tse-Hsun (Peter) Chen. How disabled tests manifest in test maintainability challenges? In *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21, pages 187–196, 2021.

[84] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.

[85] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proc. of the 21st Int. Conference on Automated Software Engineering (ASE*, 2006.

[86] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176, 2016.

[87] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: Bug report driven program repair. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 314–325, 2019.

[88] Herb Krasner. The cost of poor software quality in the us: A 2022 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2023.

[89] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 821–830, 2017.

[90] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Bug localization with combination of deep learning and information retrieval.

In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229. IEEE, 2017.

[91] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1471–1482, 2020.

[92] Maxime Lamothe and Weiyi Shang. When apis are intentionally bypassed: An exploratory study of api workarounds. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 912–924. IEEE, 2020.

[93] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 185–194, 2010.

[94] Tien-Duy B Le, Ferdian Thung, and David Lo. Theory and practice, do they match? a case with spectrum-based fault localization. In *2013 IEEE International Conference on Software Maintenance*, pages 380–383. IEEE, 2013.

[95] Tien-Duy B Le, Richard J Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 579–590, 2015.

[96] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[97] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 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)*, pages 3–13. IEEE, 2012.

[98] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. doi: 10.1109/TSE.2011.104.

[99] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software quality journal*, 21:421–443, 2013.

[100] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015. ISSN 0098-5589.

[101] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. Bench4bl: reproducibility study on the performance of IR-based bug localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 61–72, 2018.

[102] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering*, 2020.

[103] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180, 2019.

[104] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 602–614, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380345. URL `https://doi.org/10.1145/3377811.3380345`.

[105] Yi Li, Shaohua Wang, and Tien N Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 661–673. IEEE, 2021.

[106] Yi Li, Shaohua Wang, and Tien N Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 511–523, 2022.

[107] Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang, and Weiyi Shang. Dlfinder: characterizing and detecting duplicate logging code smells. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 152–163. IEEE, 2019.

[108] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. DLfinder: Characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 152–163, 2019.

[109] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. Where shall we log? studying and suggesting logging locations in code blocks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 361–372, 2020.

[110] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56, 2017.

[111] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 393–403. IEEE, 2017.

[112] Bing Liu, Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability*, 26(6):431–459, 2016.

[113] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pages 102–113. IEEE, 2019.

[114] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and

Tegawendé F Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021.

[115] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87, 2020.

[116] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 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*, pages 664–676, 2021.

[117] Pablo Loyola, Kugamoorthy Gajananan, and Fumiko Satoh. Bug localization by learning to rank and represent bug inducing changes. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, pages 657–665, 2018.

[118] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26(2):172–219, 2014.

[119] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *2008 15Th working conference on reverse engineering*, pages 155–164. IEEE, 2008.

[120] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, 2014.

[121] Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, SANER'19, pages 468–478, 2019.

[122] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204. IEEE, 2014.

[123] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22:1936–1964, 2017.

[124] Wes Masri. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability*, 20(2):121–147, 2010.

[125] Wes Masri, Rawad Abou-Assi, Marwa El-Ghali, and Nour Al-Fatairi. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 1–5, 2009.

[126] Mathias Meyer. Continuous integration and its tools. *IEEE software*, 31(3): 14–16, 2014.

[127] D.S. Moore, G.P. MacCabe, and B.A. Craig. *Introduction to the Practice of Statistics*. W.H. Freeman and Company, 2009.

[128] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval-based bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 151–160. IEEE, 2014.

[129] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[130] Manish Motwani and Yuriy Brun. Automatically repairing programs using both tests and bug reports. *arXiv preprint arXiv:2011.08340*, 2020.

[131] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? In *Proceedings of the 40th International Conference on Software Engineering*, pages 25–25, 2018.

[132] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2.

[133] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[134] Steve Neely and Steve Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *2013 Agile Conference*, pages 121–128. IEEE, 2013.

[135] Jerzy Neyman. On the two different aspects of the representative method: the method of stratified sampling and the method of purposive selection. In *Breakthroughs in statistics*, pages 123–150. Springer, 1992.

[136] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[137] Smith Nicholas, Bruggen Danny van, and Tomassetti Federico. javaparser, 2017. URL `https://leanpub.com/javaparservisited`.

[138] NLTK. Nltk corpora. `https://www.nltk.org/nltk_data/`. Last accessed Feb 1 2020.

[139] Oracle. Java language keywords. `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html`. Last accessed Feb 1 2020.

[140] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, ICSME 2017, pages 1–12, 2017.

[141] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[142] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[143] Z. Peng, T. Chen, and J. Yang. Revisiting test impact analysis in continuous testing from the perspective of code dependencies. *IEEE Transactions on Software Engineering*, (01):1–1, dec 2021. ISSN 1939-3520.

[144] Alexandre Perez, Rui Abreu, and Eric Wong. A survey on fault localization techniques. 2014.

[145] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, pages 1–11, 2012.

[146] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, 2014.

[147] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52, 2011.

[148] R Owen Rogers. Scaling continuous integration. In *Extreme Programming and Agile Processes in Software Engineering: 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004. Proceedings 5*, pages 68–76. Springer, 2004.

[149] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and

cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[150] Carl Martin Rosenberg and Leon Moonen. Spectrum-based log diagnosis. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2020.

[151] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 345–355, 2013.

[152] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.

[153] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, pages 648–659, Oct 2017. doi: 10.1109/ASE.2017.8115675.

[154] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13, 2018.

[155] Qusay Idrees Sarhan and Árpád Beszédes. A survey of challenges in spectrum-based software fault localization. *IEEE Access*, 10:10618–10639, 2022.

[156] Kiavash Satvat and Nitesh Saxena. Crashing privacy: An autopsy of a web browser's leaked crash reports. *CoRR*, abs/1808.01718, 2018.

[157] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121. IEEE, 2010.

[158] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017.

[159] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015.

[160] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 402–411, 2013.

[161] Bunyamin Sisman and Avinash C. Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 50–59, 2012.

[162] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283, 2017.

[163] Jeongju Sohn, Yasutaka Kamei, Shane McIntosh, and Shin Yoo. Leveraging fault localisation to enhance defect prediction. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–294. IEEE, 2021.

[164] M. Soltani, A. Panichella, and A. Van Deursen. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[165] Xuezhi Song, Yun Lin, Siang Hwee Ng, Ping Yu, Xin Peng, and Jin Song Dong. Constructing regression dataset from code evolution history. *arXiv preprint arXiv:2109.12389*, 2021.

[166] Matúš Sulír and Jaroslav Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25, 2016.

[167] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349. IEEE, 2019.

[168] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 131–144, 2007.

[169] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, 2017.

[170] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 805–816, 2015.

[171] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. pages 53–63, 2014.

[172] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63, 2014.

[173] Shaowei Wang and David Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28 (10):921–942, 2016.

[174] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns

to improve fault localization. In *2009 IEEE 31st International Conference on Software Engineering*, pages 45–55. IEEE, 2009.

[175] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 262–273. IEEE, 2016.

[176] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172*, 2017.

[177] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering*, 2019.

[178] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 326–337, 2019.

[179] Frank Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

[180] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE international conference on software maintenance and evolution*, pages 181–190. IEEE, 2014.

[181] W Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

[182] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63: 290–308, 2014.

[183] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42 (8):707–740, 2016.

[184] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 204–214, 2014.

[185] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*, 23(5):2866–2900, 2018.

[186] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4): 1–40, 2013.

[187] Xiaoyuan Xie, W Eric Wong, Tsong Yueh Chen, and Baowen Xu. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866–879, 2013.

[188] Xiaofeng Xu, Vidroha Debroy, W Eric Wong, and Donghui Guo. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering*, 21(06):803–827, 2011.

[189] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F Bissyandé. Where were the repair ingredients for defects4j bugs? exploring the impact of repair ingredient retrieval on the performance of 24 program repair systems. *Empirical Software Engineering*, 26:1–33, 2021.

[190] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 831–841, 2017.

[191] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering*, 23(5):2948–2979, 2018.

[192] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. Bug localization based on code change histories and bug reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 190–197. IEEE, 2015.

[193] Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177–192, 2017.

[194] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24:33–67, 2019.

[195] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 143–154, 2010.

[196] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *ASPLOS '11: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 3–14, Newport Beach, California, USA, 2011. ACM. ISBN 978-1-4503-0266-1.

[197] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 293–306, 2012.

[198] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices

in open-source software. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 102–112, 2012.

[199] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, 2014.

[200] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[201] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology*, 124:106312, 2020.

[202] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Peter Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, 24(6):3394–3434, 2019.

[203] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–272, 2017.

[204] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. An empirical study of boosting spectrum-based fault localization via pagerank. *IEEE Transactions on Software Engineering*, 47(6):1089–1113, 2019.

[205] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications.

[206] Jian Zhou, Hongyu Zhang, and David Lo. 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)*, pages 14–24. IEEE, 2012.

[207] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, Sep. 2010. ISSN 2326-3881. doi: 10.1109/TSE.2010.63.

[208] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2019.

[209] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? page 1–5, 2005.