Empirical Investigations of More Practical Fault Localization Approaches

Tung Dao

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Na Meng, Chair
Taejoong Tijay Chung
Muhammad Alir Gulzar
Bo Ji
Xiaoyin Wang

September 29, 2023
Blacksburg, Virginia

Keywords: Software Engineering, Testing, Debugging, Fault Localization, Spectrum-based, Information Retrieval, Slicing, Coverage, Execution Information, Abstract State Machine, Cloud Computing

Empirical Investigations of More Practical Fault Localization Approaches

Tung Dao

(ABSTRACT)

Developers often spend much of their valuable development time on software debugging and bug finding. In addition, software defects cost software industry as a whole hundreds or even a trillion of US dollars. As a result, many fault localization (FL) techniques for localizing bugs automatically, have been proposed. Despite its popularity, adopting FL in industrial environments has been impractical due to its undesirable accuracy and high runtime overhead cost. Motivated by the real-world challenges of FL applicability, this dissertation addresses these issues by proposing two main enhancements to the existing FL. First, it explores different strategies to combine a variety of program execution information with Information Retrieval-based fault localization (IRFL) techniques to increase FL's accuracy. Second, this dissertation research invents and experiments with the unconventional techniques of Instant Fault Localization (IFL) using the innovative concept of triggering modes. Our empirical evaluations of the proposed approaches on various types of bugs in a real software development environment shows that both FL's accuracy is increased and runtime is reduced significantly. We find that execution information helps increase IRFL's Top-10 by 17–33% at the class level, and 62–100% at the method level. Another finding is that IFL achieves as much as 100% runtime cost reduction while gaining comparable or better accuracy. For example, on single-location bugs, IFL scores 73% MAP, compared with 56% of the conventional approach. For multi-location bugs, IFL's Top-1 performance on real bugs is 22%, just right below 24% that of the existing FL approaches. We hope the results and findings from

this dissertation help make the adaptation of FL in the real-world industry more practical and prevalent.

Empirical Investigations of More Practical Fault Localization Approaches

Tung Dao

(GENERAL AUDIENCE ABSTRACT)

In software engineering, fault localization (FL) is a popular technique to automatically find software bugs, which cost a huge loss of hundreds of billions of US dollars on the software industry. Despite its high demanding and popularity, adopting FL in industrial software companies remains impractical. To help resolve this applicability problem, this dissertation proposed enhanced techniques to localize bugs more accurately and with less overhead runtime expenses. As a result, FL becomes more practical and efficient for software companies.

Dedication

To my family: Chinh, Toan, and Cat Tien

Acknowledgments

I still can't believe that I am making to the end of my very long Ph.D journey. Now reflecting back, there are so many people I would like to thank sincerely, to help make this impossible a reality. First and foremost, I would like to express my deepest appreciation to my advisor Na Meng. I came to her after many failed attempts doing research with many amazing professors. I still remembered my first meeting with her, she did not hesitate accepting me as her student. Though I failed her many times, she was still so kind to give me the second and many more chances. My research direction was shaped by her guidance, patience, and hard-work. She spent a countless number of meetings and discussions with me, whether it was about forming a research idea, or editing papers. Sure, she gave me tough time, but one said it is tough time that creates strong men. I am so fortunate to be her student. I am thankful to my other advisory research committee members, Taejoong Tijay Chung, Muhammad Alir Gulzar, Bo Ji, and Xiaoyin Wang for their encouraging and insightful comments and feedbacks. I would like to thank Max Wang, who saw the potential value of my research in industry, and strongly encouraged me to finish my Ph.D research. Without him, I would perhaps have never had a chance to see the light at the end of the tunnel (as he used to say so). Other persons I am deeply grateful to are Vu Nguyen and his wife, Ha Nguyen. Vu helped "drag" me to the finish line, while Ha provided us with her delicious food, when we worked on my paper. I am also deeply indebted to my Cvent co-workers, especially, my manager, Pradeep for his encouragement, support, and understanding. It won't be complete if I don't mention Bill and Hal, my previous bosses at the Assistive Technologies lab. I would like to thank Bill and Hal for providing me many years of graduate funding. Last but not least, I dearly thank my family, my wife, Chinh for her love, patience and sacrifices; and the

love of my life, my little son, Toan, and cutie little baby daughter, Cat Tien.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software bugs (defects and faults are used interchangeably) are prevalence and remains expensive and time-consuming to tackle. Software debugging (i.e., localizing and fixing bugs) takes an estimate of 50% of developers' work time and costs the whole economy more than $300 billion to $1 trillion annually [3, 63]. Before bugs are fixed, they must be accurately and quickly localized. The software defect localization itself is often the more difficult and time-consuming part of the two [18, 19, 20, 24, 34, 116, 121]. No matter how skillful and careful a developer is, he or she is always prone to introduce bugs at some point of a time during their software development. Thus, well-designed, continuous and thorough software testing is still the main hope to catch bugs. Unfortunately, however, in reality, a significant number of bugs are often leaked and uncovered into production. Therefore, the damage and consequences are far costlier and detrimental.

The major challenges for current fault localization (**FL**) techniques and tools in the real-world software development include:

(1) **Undesirable Accuracy Performance:** An enterprise-typed software application may well contain millions of lines of code (LoC), and tens of thousands of testcases. Test execution against the application generates a large amount of profiling data for FL to analyze and identify fault locations (e.g., line, method, class or file), when failing testcases happen. In addition to the dynamic execution information, static resources,

such as, bug reports, textual representation of application code and testcases should also be taken into consideration to localize faults for a better performance. Existing fault localization techniques often utilize either execution profiling information (i.e., spectrum), or static bug reports. Techniques that rely on spectrum to localize faults are called spectrum-based fault localization (SBFL) [18, 21, 50, 55, 56, 69, 77, 81, 105, 109]. Ones use the later are called information retrieval (IR-based) fault localization (IRFL) [25, 53, 62, 64, 65, 71, 78, 80, 83, 100, 121]. Analyzing either of the two spectrum or textual representation will cause a localization technique to search in a much larger searching space, resulting in lower performance in terms of accuracy.

(2) **Impractical Applicability in Industry:** SBFL techniques require all tests are executed before bugs are localized. It is common that a real-world application contains thousands of complex tests (e.g., unit tests, integration tests), whose execution often takes much time to complete (i.e., hours or days). In a highly agile software development environment, this expensive runtime overhead cost spent to execute all the tests so SBFL can be applied, is impractical if not unacceptable.

In this dissertation, we present our research to improve the state-of-the-art of **FL** by addressing the two aforementioned challenges with the following approaches:

Integration between IR-based techniques with execution information: To cope with the challenge 1, we propose to facilitate IR techniques with runtime execution information. In this research, we study and develop techniques that combine both spectrum (coverage and slicing profiling data) and textual representation information. We then conduct empirical studies to find out the best strategies for the combination, and how much accuracy improvement the proposed combination techniques achieve compared with the ex-

isting approaches. Our results show that our combining strategies increase, for example, Top-10 by 17–33% at the class level, and 62–100% at the method level.

Instant Spectrum-based Fault Localization (IFL) with triggering modes: To address the challenge 2, we study how to shorten fault localizing time, while achieving the same or better accuracy, for SBFL. In particular, we propose the novel concept of *triggering modes*, to instantly rank and output faulty locations based on a partial coverage spectrum data. We then integrated these proposed triggering mode-based SBFL techniques in a real-world software development environment using a continuous integration and continuous delivery (CI/CD) pipeline. Our empirical evaluation of this approach on both open-sourced and real-world close-sourced bug datasets shows that IFL achieves as much as 100% runtime cost reduction while gaining comparable or better accuracy. For example, on single-location bugs, IFL achieved 73% MAP, compared with 56% of the conventional approach. For multi-location bugs, IFL's Top-1 performance on real bugs is 22%, just right below 24% that of the existing SBFL approaches.

The rest of the dissertation is organized as follows. Chapter §2 introduces relevant background for our research in this dissertation. Chapter §3 gives more formal definition of related concepts used in the dissertation. Chapter §4 discusses our research on improving IRFL using execution information. Chapter §5 presents our proposed IFL and Triggering Modes for single-location bugs fault localization. Chapter §6 describes our proposed IFL and Triggering Modes for multi-location bugs fault localization. Chapter §7 reviews related work. Chapter §8 discusses our future work to improve SBFL in cloud-based applications. Finally, chapter §9 summarizes and concludes the dissertation.

Chapter 2

Background

In this section (Section 4.3.1), we first present the background of IR-based fault localization, and then discuss different execution information and how we collected them. Finally, we explain our four ways of combining execution information with IR-based techniques.

2.1 IR-Based Fault Localization (IRFL)

Given a bug report, IRFL treats the report as a *query*, and considers source code elements as a *document collection*. It ranks elements according to their textual similarity with the report [65, 70, 70, 72, 78, 82, 83, 89, 99, 100, 121]. In this section, we summarize the baseline IR-based technique, two widely used IR-based tools (BugLocator [121] and BLUiR [82]), and AML—a hybrid approach combining an IR-based technique with spectrum execution information.

2.1.1 The baseline technique

applies the IR framework Indri [92] directly without any optimization. Given a bug report and a buggy program, it preprocesses the data in three steps. First, it extracts all words except for stop words (e.g., “a”, “at”, and “which”), and programming language keywords (e.g., `while`, `for`). Second, it applies camel case splitting (`IsSigned` → {“Is”, “Signed”}) and

stemming [54] ("Signed" → "sign") to split and stem code identifiers. Third, it indexes all documents (i.e., classes or methods) by terms, and computes the term frequency (TF) for individual documents. After the preprocessing, Indri takes in the bug report query and document corpus, retrieves query-relevant documents, and ranks the documents by relevance. For our study, we used the default VSM (i.e., TF-IDF) model of Indri to do experiments.

2.1.2 The state-of-the-art techniques

BugLocator

BugLocator [121] improves the baseline with two specializations. First, instead of using the default VSM, BugLocator builds a revised VSM (rVSM) to calculate the query-document similarity differently. The specialization is based on the tool builders' observation that longer files are more likely to be buggy than shorter ones. Second, when ranking documents, BugLocator also considers bug history. Hypothetically, similar bug reports may indicate similar bug locations. If a new report is similar to some reports whose bugs are already located, then BugLocator highly ranks those bug locations. According to the evaluation with more than 3000 real bugs in prior work [121], these two customizations enabled BugLocator to outperform the known baseline techniques [73, 80].

BLUiR

BLUiR [82] improves the baseline by considering structure information. Different from prior approaches, BLUiR observes document structures and program structures. Given a bug report, it assigns more weight to terms in titles than those in summaries, because report titles usually provide more relevant information. Given a program, BLUiR assigns more weight to names of classes and methods, but less to variable names and comments, because

it assumes class and method names are more important.

Prior work [82] showed that BLUiR outperformed BugLocator and BugScout [78].

AML

AML [66] is a hybrid approach to combine spectrum execution information (see Section 2.2) with an IR-based technique. It consists of three components: AML^{Text} (the IR-based tool), $\text{AML}^{\text{Spectra}}$ (the spectrum information), and $\text{AML}^{\text{SuspWord}}$. These components independently calculate suspiciousness scores of every program element, and AMLdfdfd then computes a weighted sum of the scores to rank program elements. Although AML outperforms the state-of-the-art IR-based technique, it is still unclear *whether the outperformance is due to the approach design or extra dynamic information*.

2.2 Execution Information and Its Collection

In this section, we overview the three most widely used types of execution information: coverage, slicing, and spectrum. We also explain why they may help FL, and how we collected them.

2.2.1 Coverage information and its collection

Coverage describes all entities (i.e., classes or methods) covered by a program execution. This information can help **FL** because if a test fails, the failure run should cover some buggy entities. To collect the information, we used ASM bytecode manipulation framework [1] to instrument the entry and exit of each method. This allows us to record which methods are executed at runtime, and to identify the executed classes that own the executed methods.

We used Java Agent to insert code instrumentation on-the-fly during class loading time.

2.2.2 Slicing information and its collection

Slicing [101] describes all classes or methods that may affect the state of a program point. This information can be helpful because when a test fails, only statements responsible for the failure can be buggy. In other words, given a failure witness statement (i.e., the failed assertion or the statement that threw an uncaught exception), only statements on which it is transitively control or data dependent are responsible for the failure. Slicing information can be collected statically or dynamically [41, 95, 101]. We used JavaSlicer [2], a dynamic slicing tool, to instrument every *instruction* for trace collection, and to perform backward slicing from the failure witness statement in each trace [95].

We chose the tool for two reasons. First, we prefer dynamic slicing to static slicing, because dynamic slicing identifies all code elements that *actually* affect the failure state. Second, unlike other dynamic slicing tools, JavaSlicer is publicly available and widely used [107, 119]. Although it outputs all instructions responsible for a failure state, in our study, we mapped them to their owner methods or classes for method-level or class-level slices, because IRFL ranks buggy methods or classes. Specifically, if one method or class has at least one instruction in the failure-relevant slice, we include the entity into the method-level or class-level slice.

2.2.3 Spectrum information and its collection

Spectrum (formally defined in **??**) can be used to formulate how suspicious a program element is when the program fails. The higher suspiciousness score a code element gets, the more likely it is buggy. Intuitively, if a code element is executed solely by failed tests but never

by passed tests, the element may be buggy. Spectrum information can help IRFL, because it provides a complementary approach to localize faults.

In our study, with the ASM bytecode instrumentation mentioned above, we got both method-level and class-level coverage information by passed tests and failed tests. Then we tried four widely-used formulae to separately compute the spectrum information: Tarantula [56], Ochiai [20], Jaccard [19], and Ample [33]. Formally, given a buggy program and a set of tests, we use n_f and n_p to represent the total number of failed and passed tests. For each program element e, whether it is a class or a method, we use e_f and e_p to denote the number of failed and passed tests executing e.

All four formulae are shown below:

$$Tarantula = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f} + \frac{e_p}{e_p+n_p}} \tag{2.1}$$

$$Ochiai = \frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}} \tag{2.2}$$

$$Jaccard = \frac{e_f}{e_f+e_p+n_f} \tag{2.3}$$

$$Ample = |\frac{e_f}{e_f+n_f} - \frac{e_p}{e_p+n_p}| \tag{2.4}$$

2.3 Spectrum-Based Fault Localization (SBFL)

SBFL automatically identifies and ranks potential buggy locations when a program P fails the execution of its test suite T [57]. To use a typical SBFL approach, developers usually take three steps.

1. Given a buggy program P and its test suite T, developers instrument either the source code or compiled code (e.g., Java byte code) to monitor which program element (e.g., Java class, method, or statement) is covered by the execution of which test case.

2. Developers execute P with T, so that the instrumented code can dynamically gather and log the execution coverage of each passed/failed test. We use n_f and n_p to denote the total number of failed and passed tests by P. For each program element e, we use e_f and e_p to represent the number of failed/passed tests executing e.

3. Based on the logged data, an SBFL formula is used to compute the suspiciousness score of each program element, and to rank all elements in the descending order of those scores.

Chapter 3

Definition

3.1 Fault Classification

Listings 1 and 2 use two simplified real bug examples from *Cvent* to demonstrate single- and multi-location bugs. Lines marked with minus (-) in red represent buggy locations, and their corresponding fixes are marked with plus (+) in green.

3.1.1 Single-Location Bugs

A single-location bug has only one single line of code responsible for the bug. Listing 1 is a single-location bug (incorrect predicate to check cache expiry).

```
1 private final long validDays = 7L;
2 -private final Predicate<LocaDate> expired = cachedAt ->
  ↪  cachedAt.plusDays(validDays).isAfter(LocaDate.now());
3 +private final Predicate<LocaDate> expired = cachedAt ->
  ↪  cachedAt.plusDays(validDays).isBefore(LocaDate.now());
```

Listing 1: Single-location Bug Example

```
1  Response<DatasetResolvedForTestUse> response =
2  -client.getDatasetForTestUse(datasetId, env).execute();
3  +client.getDatasetForTestUse(datasetId, env, false).execute();// don't increment usage
   ↪  count
4  //...
5  public Object getOrCreateDatasetMinimalBlocking(String datasetId, String env,
   ↪  DatasetReusePolicy policy, List<DatasetDependency> dependencies,
   ↪  BiFunction<ResolvedDependencyCollection, DatasetHelper, List<Object>
   ↪  datasetCreationFunction) throws IOException, InterruptedException {
6  Reponse<DatasetResolvedForTestUse> response =
7  -client.getDatasetForTestUse(datasetId, env).execute();
8  +client.getDatasetForTestUse(datasetId, env, true).execute(); // do increment usage
   ↪  count
9  //...
10 }
```

Listing 2: Multi-location Bug Example

3.1.2 Multi-Location Bugs

A multi-location bug contains more than one buggy location, compared with a single-location bug that has only one single line of code responsible for the bug. Listing 2 shows a multi-location bug involving in two buggy locations, i.e., lines 2 and 7 (faulty in tracking dataset usage).

In theory, SBFL is designed to work for both cases, though finding multi-location bugs is more challenging because SBFL needs to find those locations equally well. In the example, the two buggy locations should be ranked as first and second, in which case the accuracy of SBFL is 100%. However, if SBFL ranks them as second and fifth, then its accuracy would be reduced to 45% (see §3.3 for definitions of accuracy metrics).

3.2 SBFL and Triggering Modes

3.2.1 Program and its Spectra

SBFL aims to identify bug locations or entities (e.g., statement, method, class) of a buggy program using coverage information (spectra). Let the program under investigation $P = \{e|e = entity\}$ be represented as a set of its entities. Let P's test set, $T = T_f \cup T_p$, and $n_f = |T_f|$, $n_p = |T_p|$, where T_f and T_p are the set of failed and passed tests, respectively. For an entity $e \in P$, let e_f and e_p be the numbers of distinct failed tests $\in T_f$ and passed tests $\in T_p$ that cover e, respectively. Then, e's spectrum is defined as $e_{\text{spectrum}} = (e_f, e_p, n_f, n_p)$. P's spectra, P_{spectra}, is $\{e_{\text{spectrum}}\}$, a set of all e's spectrum for all $e \in P$.

3.2.2 Fault Localization Workflow

SBFL uses the four steps below to identify and rank buggy locations:

Instrumentation: P and T are instrumented by techniques that modify the source code directly or indirectly via their compiled code (e.g., Java byte code), so that P_{spectra} can be recorded and collected. Tools such as SonarQube [16], Clover [5], Jacoco [13], Corbetura [11] are often used in industry for this purpose. We use Clover in this dissertation.

Test Execution: T is run against the instrumented P using automated test runner or framewor, such as Maven [15], TestNG [10], JUnit [14]. Once the test execution is finished, P_{spectra} is all recorded.

Suspiciousness Score Calculation: For each entity $e \in P$, its buggy suspiciousness score, e_{score}, is calculated using $e_{spectrum}$, (e_f, e_p, n_f, n_p). There are numerous spectrum-based formulae proposed to compute e_{score}, such as, $Dice = \frac{2*e_f}{e_f+e_p+n_f}$, $Goodman = \frac{2*e_f-n_f-e_p}{2*e_f+n_f+e_p}$, $Hamann = \frac{e_f+n_p-e_p-n_f}{e_f+e_p+n_f+n_p}$, and $Euclid = \sqrt{e_f+n_p}$. In this dissertation we used 25 popular formulae shown in Tab. 3.1.

Ranking: Entities in P are ranked based on their suspiciousness scores in decreasing order, where the most likely buggy locations are on top. The ranked list of P's entities are the final result of the bug localization process.

3.2.3 Ranking Formulae

The 25 SBFL common ranking formulae used in this dissertation are defined below:

3.2.4 Triggering Modes

A triggering mode is a temporal concept defined as the moment during a program's test execution at which SBFL is invoked. This dissertation defines and explores the five following triggering modes:

First-Failure Triggering (IFLS_1) invokes SBFL right after the first test failure. This is the minimal requirement for SBFL to work as it requires at least 1 failed test. While this mode uses minimal time and computing resources, it also collect fewer coverage (spectrum) information.

Table 3.1: The investigated 25 SBFL formulas

Name	Formula	Name	Formula
Ample	$\lvert\frac{e_f}{e_f+n_f}-\frac{e_p}{e_p+n_p}\rvert$	**Anderberg**	$\frac{e_f}{e_f+2*e_p+2*n_f}$
Dice	$\frac{2*e_f}{e_f+e_p+n_f}$	**Euclid**	$\sqrt{e_f+n_p}$
Goodman	$\frac{2*e_f-n_f-e_p}{2*e_f+n_f+e_p}$	**Hamann**	$\frac{e_f+n_p-e_p-n_f}{e_f+e_p+n_f+n_p}$
Hamming	e_f+n_p	**Jaccard**	$\frac{e_f}{e_f+e_p+n_f}$
Kulczynski1	$\frac{e_f}{n_f+e_p}$	**Kulczynski2**	$\frac{1}{2}*(\frac{e_f}{e_f+n_f}+\frac{e_f}{e_f+e_p})$
M1	$\frac{e_f+n_p}{n_f+e_p}$	**M2**	$\frac{e_f}{e_f+n_p+2*n_f+2*e_p}$
Ochiai	$\frac{e_f}{\sqrt{(e_f+e_p)*(e_f+n_f)}}$	**Ochiai2**	$\frac{e_f*n_p}{\sqrt{(e_f+e_p)*(n_f+n_p)*(e_f+n_p)*(e_p+n_f)}}$
Overlap	$\frac{e_f}{\min(e_f,e_p,n_f)}$	**RogersTanimoto**	$\frac{e_f+n_p}{e_f+n_p+2*n_f+2*e_p}$
RussellRao	$\frac{e_f}{e_f+e_p+n_f+n_p}$	**SimpleMatching**	$\frac{e_f+n_p}{e_f+e_p+n_f+n_p}$
Sokal	$\frac{2*e_f+2*n_p}{2*e_f+2*n_p+n_f+e_p}$	**SϕrensenDice**	$\frac{2*e_f}{2*e_f+e_p+n_f}$
Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$	**Wong1**	e_f
Wong2	e_f-e_p	**Zoltar**	$\frac{e_f}{e_f+e_p+n_f+10000*n_f*\frac{e_p}{e_f}}$
Wong3	e_f-h, where $h=\begin{cases}2+0.1*(e_p-2) & \text{if } 2<e_p<=10\\ 2.8+0.01*(e_p-10) & \text{if } e_p>10\\ e_p & \text{otherwise}\end{cases}$		

Note: n_f and n_p separately represent the total number of failed and passed tests by a program. For any executed program element e, e_f and e_p separately represent the number of failed and passed tests covering e.

Multi-Failure Triggering (IFLS$_f^k$) initiates SBFL after every k^{th} (k=1–5) test failures. As k increases, more spectrum information was collected. However, this mode requires more time and computing resources.

Failure-Pass Triggering (IFLS$_p^k$) activates SBFL after the first test failure, and subsequently k extra passed tests (k = 1–10). Compared to **IFLS$_1$**, **IFLS$_p^k$** spends more time and

resources to collect coverage data. Here we can study the trade-off between gained accuracy and time and resources required for executing more tests.

Frequency-Based Triggering (IFLS$_t^k$) IFLS$_t^k$ periodically calls SBFL to localizes faults at a predefined interval (e.g., every 2 minutes). With such periodic updates, we can observe how ranking is adjusted as more coverage data is available. One possible limitation is that when few failures happen, IFLS$_t^k$ may waste time to unnecessarily rerank locations.

Complete Execution Triggering (IFLS$_A$) is the conventional SBFL, which ranks bug locations after executing *all* available tests. IFLS$_A$ is thus expensive and might not be applicable in the real world, e.g., at Cvent with many tests.

In its essence, each triggering mode corresponds to a different approach for selecting a subset of tests from the complete test set. Our goal is to collect a partial set of spectrum data that is sufficient for SBFL to function effectively. By experimenting with these different triggering modes, we explore the trade-offs between effectiveness of SBFL and its runtime cost. Note in our study that all tests were executed sequentially in a fixed order, determined by the test executor (Maven-Clover plugin). This fixed ordering ensures deterministic results throughout our study. In addition, all triggering modes use instrumented tests. While instrumentation adds overhead, it is generally not a concern in practice, as companies (e.g., Cvent) often run instrumented tests to at least measure code coverage metrics, as part of code quality control procedure.

3.3 Effectiveness Metrics

3.3.1 Recall at Top N (Top-N)

This measures the percentage of buggy entities that are included in the top N ranked locations. For example, if an entity is ranked third, its Top-1 recall rate would be 0% (as it is not ranked first) and its Top-5 recall rate would be 100% (as it is within the top five ranks). In general, a higher Top-N recall means better performance.

3.3.2 Mean Average Precision (MAP)

This measures the accuracy and ranking quality of FL in identifying the (faulty) entities. Higher MAP value is better. The **Average Precision (AP)** of an FL task is:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{\text{number of positive instances}} \times 100\% \tag{3.1}$$

Suppose that FL ranked M statements and one of them is positive (i.e., buggy), then the number of positive instances is equal to 1. For each value of k, where k varies from 1 to M, $P(k)$ is the percentage of positive instances among the top k instances, and $pos(k)$ is a binary indicator of whether or not the k^{th} statement is positive. Namely, $pos(k) = 1$ if the k^{th} statement is positive, otherwise $pos(k) = 0$. For example, if four statements are ranked, and the 3^{rd} and 4^{th} are positive, then AP is $(\frac{1}{3} + \frac{2}{4})/2 \times 100\% = 42\%$. On the other hand, if the 1^{st} and 2^{nd} of the ranked list are buggy, then $AP = (\frac{1}{1} + \frac{2}{2})/2 \times 100\% = 100\%$.

3.3.3 Mean Reciprocal Rank (MRR)

This measures precision in a different way. Given a set of fault localization tasks, it calculates the mean of reciprocal rank values for all tasks. Overall, higher MRR value indicates better the precision. The **Reciprocal Rank (RR)** of a single task is defined as:

$$RR = \frac{1}{\text{rank}_{\text{best}}} \times 100\% \tag{3.2}$$

where $\text{rank}_{\text{best}}$ is the rank of the first actual bug located. For example, for 4 ranked statements with the 3^{rd} and 4^{th} being buggy, RR is $\frac{1}{3} \times 100\% = 33\%$.

3.4 Adopting SBFL at Cvent

3.4.1 Continuous Integration/Continuous Delivery (CI/CD)

A CI/CD build pipeline to manage its software development and operations, including build, test, and deploy stages. This reduces human effort and allows for easy and automated tasks to handle changes, integration, implementation, and delivery of software features. For example, when code changes are committed, a CI/CD script is invoked to automate tasks including compiling, running tests, packaging, and deploying. Fig. 3.1 outlines the CI/CD workflow at Cvent.

3.4.2 Integrating SBFL with CI/CD

Without SBFL, test failures require manual inspection to find bugs. However, with SBFL integration, bugs are automatically localized using code coverage profiling data. This ac-

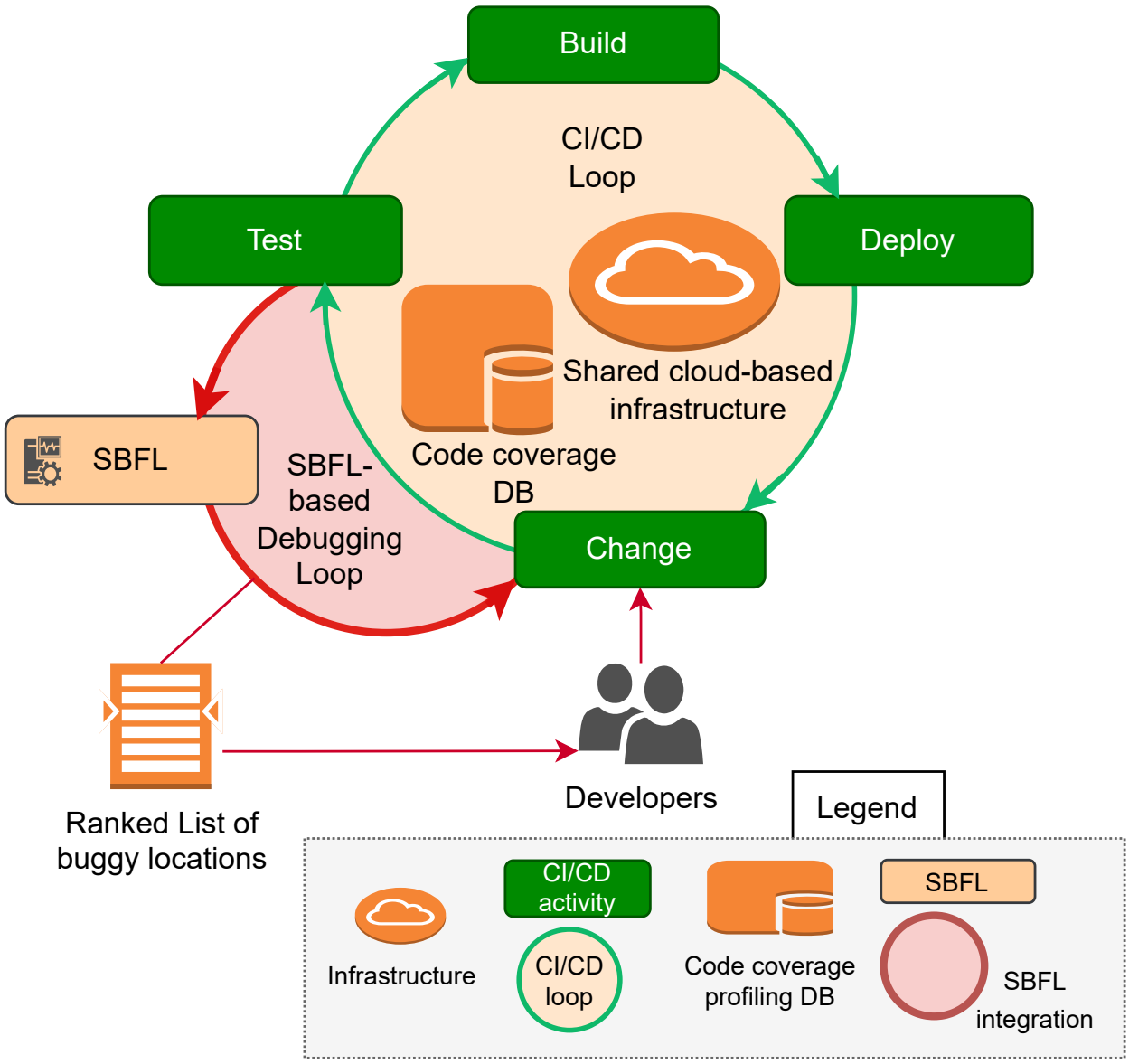


Figure 3.1: CI/CD and SBFL Integration Outline.

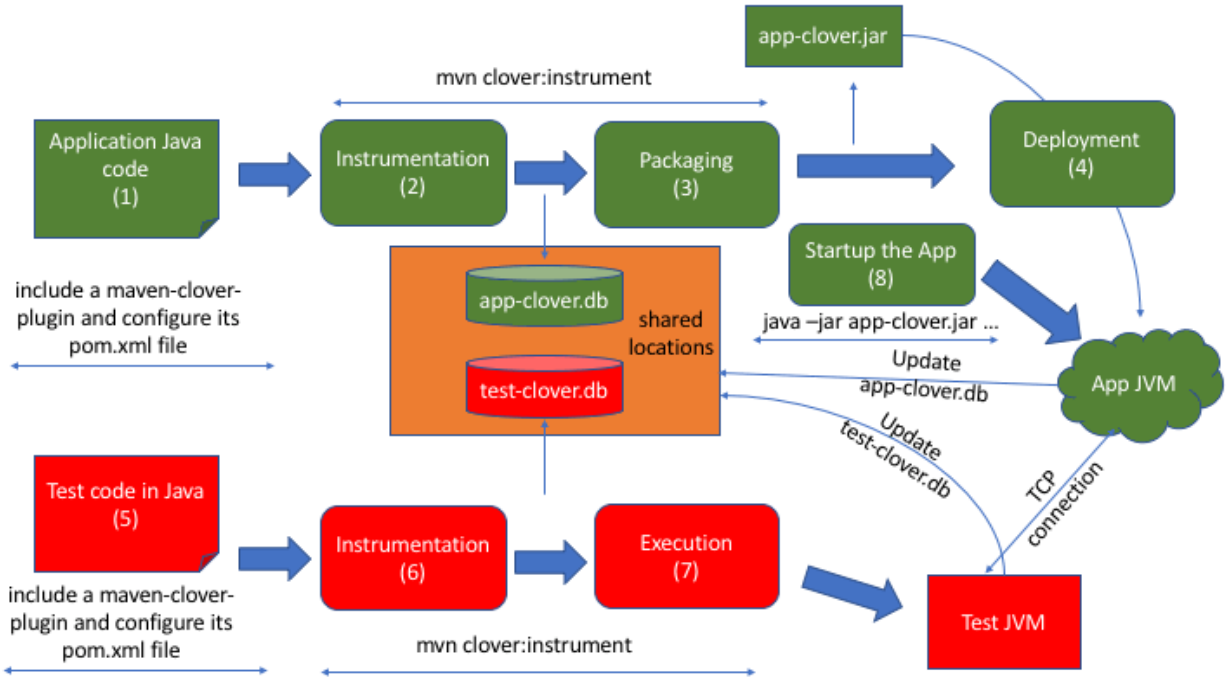


Figure 3.2: Collecting coverage profiling data with Clover.

celerates the debugging process for developers as manually localizing bugs, especially multi-location ones, is highly time-consuming and difficult.

However, with thousands of CPU hours spent daily running tests and an average failure rate of around 7%, waiting for all test executions to finish before applying SBFL becomes impractical. In this scenario, **IFLS** appears to be a more suitable alternative to SBFL for scaling to our level of operations.

3.4.3 Collecting Code Coverage in CI/CD

The CI/CD build pipeline at Cvent uses the open-source Clover tool [5] to collect profiling data.

Code coverage is the percentage of code that is covered by test execution. Code coverage measurement reflects which program elements are executed through a test run, and which elements are not [4].

Clover is an open-source code profiling tool, to gather coverage data. To collect the coverage information, Clover injects profiling logic to Java source code, and compiles the code with normal compilers to produce instrumented `.class` files. When instrumented code is executed, the profiling data (e.g., executed statements) is saved to Clover's database.

Fig. 3.2 illustrates how Clover collects code coverage from profiling data of a Java-based microservice application. Activities related to the application (under investigation) are represented by green boxes: (1) configuring the app's pom.xml, (2) instrumenting application code to enable code coverage tracking, (3) packaging the instrumented application into a deployable and executable file, and (4) deploying the application.

Activities related to testing are marked in red: (5) configuring the test's pom.xml file, (6) instrumenting the test code, (7) executing the tests, (8) running the application in a cloud-based environment, (9) shutting down the application JVM once tests are completed, (10) merging coverage databases generated by the application and the tests into a shared coverage data, and (11) finally, obtaining per-test coverage profiling data for both of the tests and the application. Of course, in the CI/CD build pipeline this is achieved automatically with build scripts (using Groovy at Cvent).

Chapter 4

Improving Fault Localization with Information-Retrieval and Execution Information

4.1 Motivation

Given a bug report and a buggy program, developers may spend tremendous time and effort understanding the bug description and code to locate faults. To facilitate fault comprehension and accelerate fault finding, researchers have proposed various information retrieval (IR) based fault localization (IRFL) techniques [78, 80, 82, 121]. By treating the bug report as a query, and the source code files as plain documents, these techniques rank software entities (i.e., classes or methods) based on their relevance or similarity to the query. The more relevant a program entity is, the higher it is ranked as a potential bug location.

These IR-based techniques can facilitate fault localization and program comprehension, because they help developers focus effort on fault-relevant code elements. In particular, Zhou et al. proposed BugLocator to use a specialized Vector Space Model (VSM), called rVSM, by considering file lengths and bug history [121]. They demonstrated that rVSM outperformed other IR models on real bugs from four open-source projects. Saha et al. further proposed

BLUiR [82] to use another revised VSM by considering code constructs, such as class and method names. Their experiments showed that BLUiR even outperformed BugLocator.

In spite of an enormous number of various IRFL techniques, their application and accuracy are inadequate. We hypothesize that the execution information of buggy programs can further help fault localization and program comprehension. Le et al. proposed the first tool, AML, to combine IRFL with spectrum execution information [66]. They used a hybrid model to encode both spectrum and textual information into a specialized VSM. They found that AML outperforms Learning-to-rank [108] (a state-of-the-art IRFL technique), and MULTRIC [106] (a state-of-the-art spectrum-based fault localization technique). However, it is still unknown *how various types of execution information can generally help with IRFL.*

To systematically investigate the impact of various execution information on IRFL techniques, we performed an extensive study on three kinds of execution information, and three state-of-the-art IRFL techniques, using an existing dataset of 157 real bugs. More specifically, we investigated the following three types of information: (1) **coverage**—the classes or methods covered by failed tests, (2) **slicing**—the classes or methods in the dynamic slice [101] of each failure witness statement (i.e., a failure assertion or an exception-throwing statement), and (3) **spectrum**—the suspiciousness score of each executed class or method, which describes the coverage ratio between passed and failed tests [56]. Hypothetically, coverage and slicing may help with IRFL techniques by refining the search space. The reason is if an entity (i.e., class or method) is not covered by a failed test or does not occur in the slice of a failure witness statement, it is unlikely to be buggy. Spectrum information may further help by ranking program entities purely based on suspiciousness scores. Its ranking can complement the ranking by IRFL techniques.

In this study, we experimented with three existing IRFL techniques: the baseline, BugLocator [121], and BLUiR [82]. To assess the impact of different execution information

on IRFL techniques, we combined IRFL techniques and execution information in four ways:

- $\mathbf{IR}_c$: *coverage* and *IR*
- $\mathbf{IR}_s$: *slicing* and *IR*
- $\mathbf{IR}_{cp}$: *coverage and spectrum* and *IR*
- $\mathbf{IR}_{sp}$: *slicing and spectrum* and *IR*

Our experiments revealed a number of interesting findings. First, we observed that coverage information can effectively reduce the search space of IR-based techniques, and thus significantly improve fault localization at both class and method levels. In particular, for all three IRFL techniques, the number of actual bug locations ranked within Top 10 was increased by 17–33% at class level, and by 62–100% at method level. This combination strategy even outperformed state-of-the-art hybrid technique AML [66] in most cases. Second, slicing information can further improve fault localization. Compared with coverage, slicing further increased the number of actual bug locations among Top 10 by 1–43% at class level, and by 9–30% at method level. Third, the additional usage of spectrum information further improved fault localization at method level. Our study shows that dynamic execution information can *generally* bring *considerable* improvement to IRFL. Some future approaches that delicately combine various execution information with IRFL techniques may further facilitate fault localization and program comprehension.

4.2 Contributions

This research makes the following contributions:

- We investigated four ways to combine execution information with IRFL by exploring three kinds of information, and three IRFL techniques.
- Our quantitative analysis shows that coverage and slicing information effectively helps with IRFL at both the class and method levels, while spectrum information further helps at method level.
- This empirical study shows for the first time that execution information can generally bring considerable improvement to IRFL, even when the combination strategies are simple and easy to understand.

4.3 Approach

4.3.1 Combining Execution Information with IRFL

Given a bug report, a buggy program, the program's passed tests and failed tests, we aim to improve IRFL with execution information using two heuristics: search space reduction and rank tuning. We systematically investigated four approaches to combine the two types of information. Intuitively, the combination approaches should always lead to considerable improvement, since both static and dynamic information is used. However, we do not know how effectively execution information can help improve IR-based techniques.

4.3.2 Search Space Reduction

According to the Propagation, Infection, and Execution (PIE) model [98], faults are triggered when a buggy element is executed. Given a buggy program and failed tests, we use coverage information of the failed tests to reduce the search space of IR-based techniques. If an element

is not covered by any failure run, it is always irrelevant to failures, and gets excluded from the scope.

Similarly, slicing information can also be used to refine the search space of IR-based techniques, because only elements affecting the runtime state of a fault witness statement may be buggy. If an element is not in any failure-relevant slice, it is unrelated to the reported bug.

4.3.3 Rank Tuning

Given a ranked list by an IR-based technique, coverage or slicing information always shortens the list, but does not change the relative ranking among covered or sliced elements. If two failure-relevant elements `A` and `B` are ranked in a wrong order, neither coverage nor slicing can correct the mistake. In comparison, spectrum information maps each element to a suspiciousness score based on execution coverage. According to the suspiciousness scores, spectrum may rank code elements very differently from IR-based techniques. When combining the two ranked lists together, we may correct the ordering mistake mentioned above in an IR-based list. Formally, given a class or method whose source code is represented as d, if its IR-based score is denoted as $Score(d, q)$, and spectrum-based score is $Susp(d)$, we define a combination factor α to control their separate weights when synthesizing an adjusted score $Score'(d, q)$ as follows:

$$Score'(d, q) = (1 - \alpha) * Score(d, q) + \alpha * Susp(d) \tag{4.1}$$

where α is configured to vary from 0 to 1, with 0.1 increment. In this way, we are able to experiment with different configurations to identify the optimal combination.

4.3.4 Four Combination Variants

We experimented four ways of combining dynamic execution information with IR-based techniques.

IR$_c$: Coverage information is used to refine the search space of IR-based techniques by filtering out un-executed entities (i.e., classes or methods). Ideally, the filtering can be applied either before or after IR-based techniques, namely (1) *Filter-then-IR* or (2) *IR-then-Filter*. If filtering is applied first, IR-based techniques only focus on documents covered by failed tests. Otherwise, IR-based techniques are applied to the whole codebase, and then coverage is used to remove entities from the ranked lists of IR-based techniques. Intuitively, both approaches should work equally well. However, according to our experiments (Chapter §4.6.1), approach (2) is generally better, so we used *IR-then-Filter* by default.

IR$_s$: Similar to IR$_c$, slicing information is used to refine the search space of IRFL.

IR$_{cp}$: Coverage information is first used to refine an IR-based list. Spectrum information is then applied to synthesize a tuned ranked list.

IR$_{sp}$: Slicing information is first used to refine an IR-based list. Spectrum information is then used to tune the ranked list.

To systematically compare different combination approaches, we evaluated their effectiveness at both class and method levels. For class-level evaluation, we check whether an approach localizes the buggy class(es). For method-level evaluation, we verify whether an approach identifies the buggy method(s). Note that since IRFL suggests buggy classes and methods, our investigated combinations also rank class- or method-level bug locations.

4.4 Research Questions

In this empirical study, we aim to answer the following research questions:

RQ1: *How does coverage information help with IRFL?* Intuitively, by refining search space, coverage information should help. However, it is unclear how effectively coverage information achieves improvement.

RQ2: *How does slicing information help with IRFL?* We are curious how slicing information helps with IRFL by reducing the search space.

RQ3: *How does spectrum information further improve **FL** over IR_c and IR_s?* Hypothetically, by integrating spectrum with IR_c and IR_s, we should localize faults more effectively. The reason is coverage and slice only focus on the execution of failed tests, but spectrum also takes passed tests into consideration. With more execution information included, we may achieve improvement in **FL** effectiveness. However, it is unclear how much improvement we can get.

RQ4: *How do our simple combinations compare with AML?* We are curious how well our combination approaches work in comparison with the state-of-the-art hybrid technique. If our approaches work equally well or even better, it means that dynamic information generally helps IR-based techniques, no matter how simply the combination is done.

4.5 Evaluation

We experimented with the existing benchmark suite published by Le et al. [66]. As shown in Tab. 4.1, the dataset consists of 157 real bugs extracted from 4 open source Java projects: AspectJ, Ant, Lucene,and Rhino. For each bug, the dataset includes a bug report, a set

Table 4.1: Dataset

Project	#Bug	Class		Method	
		#Total	#Buggy	#Total	#Buggy
AspectJ	41	4,157	67	14,218	88
Ant	53	1,063	96	9,624	197
Lucene	37	2,737	158	10,220	311
Rhino	26	191	58	4,839	145
Overall	**157**	**8,148**	**379**	**38,901**	**741**

of test cases including passed and failed tests, a buggy program,and a fixed version of the program. The bug report is used by IR-based techniques to locate bugs. The test cases are used for execution information collection. The actual bug fix, which is the textual *diff* between the buggy program and its revised version, serves as the ground truth to evaluate whether a bug is located correctly. As a bug fix may involve changes to a single or multiple classes or methods, if we consider all modified code elements as buggy locations, we have 157 bugs mapped to 379 buggy classes, or 741 buggy methods.

4.6 Results and Analysis

In this section, we first show how effectively coverage and slicing can improve IR-based techniques (Chapter §4.6.1 and §4.6.2). Then we describe the effectiveness of spectrum (Chapter §4.6.3). Finally, we compare our combination approaches with AML (Chapter §4.6.4).

4.6.1 RQ1: How does coverage information help with IRFL?

We compared IR_c with the original IR-based techniques at both class and method levels. Since coverage can be used to refine the search space either *before* or *after* IR-based techniques, we first investigated which order always produces better results.

Filter-then-IR (F-I) vs. IR-then-Filter (I-F). The former one first uses coverage information to scope a list of entities (i.e. classes or methods) executed by failed tests, and then applies IR-based techniques to rank entities relevant to a given bug report. The latter one takes the two steps in a reverse order. To understand which option is better, we tried both options to localize faults at class and method levels, and observed that I-F performed better in most cases. Due to the space limit, we only show the class-level results in Tab. 4.2. One possible reason is that I-F leverages the whole codebase to build corpus for IR techniques, while F-I only uses the executed classes or methods. With a larger document corpus, I-F better identifies both important and unimportant words, and thus ranks executed documents more precisely. Therefore, by default, we used I-F to integrate coverage or slicing with IR-based techniques.

> **Finding 2.1:** *Compared with Filter-then-IR, IRFL-then-Filter worked better to refine the search space of IRFL with execution information.*

Class-level FL identifies buggy classes. Tab. 4.3 (a) shows the comparison between IR-only and IR_c for class-level fault localization. Under each IR-based technique (Baseline, BugLocator, or BLUiR), there are two columns: **IR** and **IR_c**. Each column "IR" shows the original technique's results, while column "IR_c" presents the results of the hybrid approach. Surprisingly, coverage information alone greatly boosted the overall effectiveness for all IR-based techniques. In particular, the MAP value of BugLocator was significantly improved from 0.28 to 0.49, while the MRR value was improved from 0.34 to 0.54. Among the three techniques, BLUiR had the best effectiveness, which conformed with the findings in prior work [82]. When augmented with coverage information, BLUiR outperformed others for all metrics except for Top 1.

However, the effectiveness improvement by coverage did not evenly distribute among dif-

ferent projects. For example, compared with Baseline, IR_c improved the Top-1 metric of AspectJ from 4 to 6 with 50% improvement, but did not improve the metric for Rhino. We examined Rhino's source code, and found that the actual buggy classes were usually ranked very low (e.g., below Top 100). Therefore, even though coverage could effectively shorten ranked lists, it was not capable of removing hundreds of unexecuted classes to promote any buggy class to Top 1.

Finding 2.2: *At class level, coverage consistently improved all studied IR-based techniques. On average, MAP was increased from 0.34 to 0.49 with 44% increment, and MRR was increased from 0.40 to 0.53 with 33% increment.*

Method-level FL isolates buggy methods for developers to examine. Compared with class-level **FL**, this approach can save more manual effort, because it does not leave a whole class body for developers to delve into [66]. Tab. 4.3 (b) presents the results. Compared with class level, all three original techniques worked more poorly at method level, meaning that locating buggy methods is generally harder than locating classes. Two reasons can explain the difficulty. First, each method contains fewer terms to index, and may become less relevant to random queries. Second, there are many more methods to rank than classes, which makes it harder to rank the actual buggy methods high.

Compared with class level, the improvement by coverage was more significant at method level. For BLUiR, the overall MAP and MRR improvements were 114% (from 0.14 to 0.30) and 84% (from 0.19 to 0.35), while the class-level improvements in Tab. 4.3 (a) were 30% (from 0.40 to 0.52) and 24% (from 0.45 to 0.56). Across all subjects, coverage effectively improved IR-based techniques in most cases.

As shown in Tab. 4.3 (b), among different techniques, BLUiR performed the worst without

coverage information. This observation complements the findings in prior work [82], because Saha et al. only evaluated BLUiR's performance at class level. The reason why the observations at class level and method level do not match may be that BLUiR puts more emphasis on referred program entity names than ordinary description in bug reports. If a bug report refers to multiple bug-irrelevant methods, BLUiR is misguided to rank methods wrongly. However, once augmented with coverage, BLUiR achieved the highest MAP and MRR values, meaning that coverage improved BLUiR's effectiveness the most significantly.

Finding 2.3: *Coverage improved IRFL more significantly at method level than at class level. The average method-level MAP and MRR improvements were 107% and 79%.*

4.6.2 RQ2: How does slicing information help with IRFL?

We experimented with IR_s, and compared their results with those of IR_c. Although JavaSlicer [2] is the best dynamic slicing tool we can use, it has not been maintained for several years. It may not work well for programs requiring features newly introduced in recent JDK versions. Among all the 157 bug fixes, JavaSlicer [2] only ran successfully with 64 examples. For the other examples, JavaSlicer failed for three reasons. First, it threw an out-of-memory exception even though we allocated 8GB memory to JVM. Second, it generated huge traces without termination, violating our 100GB space limit for each subject. Third, the slicing result did not include the actual buggy element due to the tool's limitation when tracing native methods, standard library classes, and multithreaded applications[1].

Therefore, we compared IR_s and IR_c on those 64 bugs for fairness.

Tab. 4.4 (a) and (b) show the comparison between IR_c and IR_s at both class and method

[1]The limitations are also listed on JavaSlicer homepage [2].

levels. Slicing was more powerful than coverage when improving IRFL. The reason is slicing removed more irrelevant entities from the IR-based list, and further upgraded ranks of the relevant ones. Similar to the observations in Chapter §4.6.1, we found that the improvement at method level was more significant than that at class level. For BLUiR, the average MAP and MRR improvements of IR_s over IR_c at method level were 42% and 35%, while the improvements at class level were both 22%. Again, BLUiR achieved the best MAP and MRR when augmented with slicing.

Finding 2.4: *Slicing was more helpful than coverage in improving IR-based techniques. The average MAP and MRR improvements of IR_s over IR_c were both 15% at class level, with 40% and 30% at method level.*

4.6.3 RQ3: How does spectrum information further improve FL over IR_c and IR_s?

To evaluate the impact of spectrum on IR_c and IR_s, we enumerated all possible combinations between the four kinds of spectrum information (Chapter §2.2) andIR_c or IR_s. All three basic IR-based techniques were explored for complete comparison. We changed the combination factor α from 0 to 1, with 0.1 increment, to investigate how **FL** effectiveness varies with α.

For IR_{cp}, as shown in Fig. 4.1, we leveraged both coverage and spectrum to improve IR-based techniques. We evaluated MAP and MRR at both class and method levels. *X-axis* represents α. *Y-axis* represents MAP in Fig. 4.1 (a-c) and (g-i), and represents MRR in Fig. 4.1 (d-f) and (j-l). Both MAP and MRR vary within [0, 1]. Intuitively, when $\alpha = 0$,

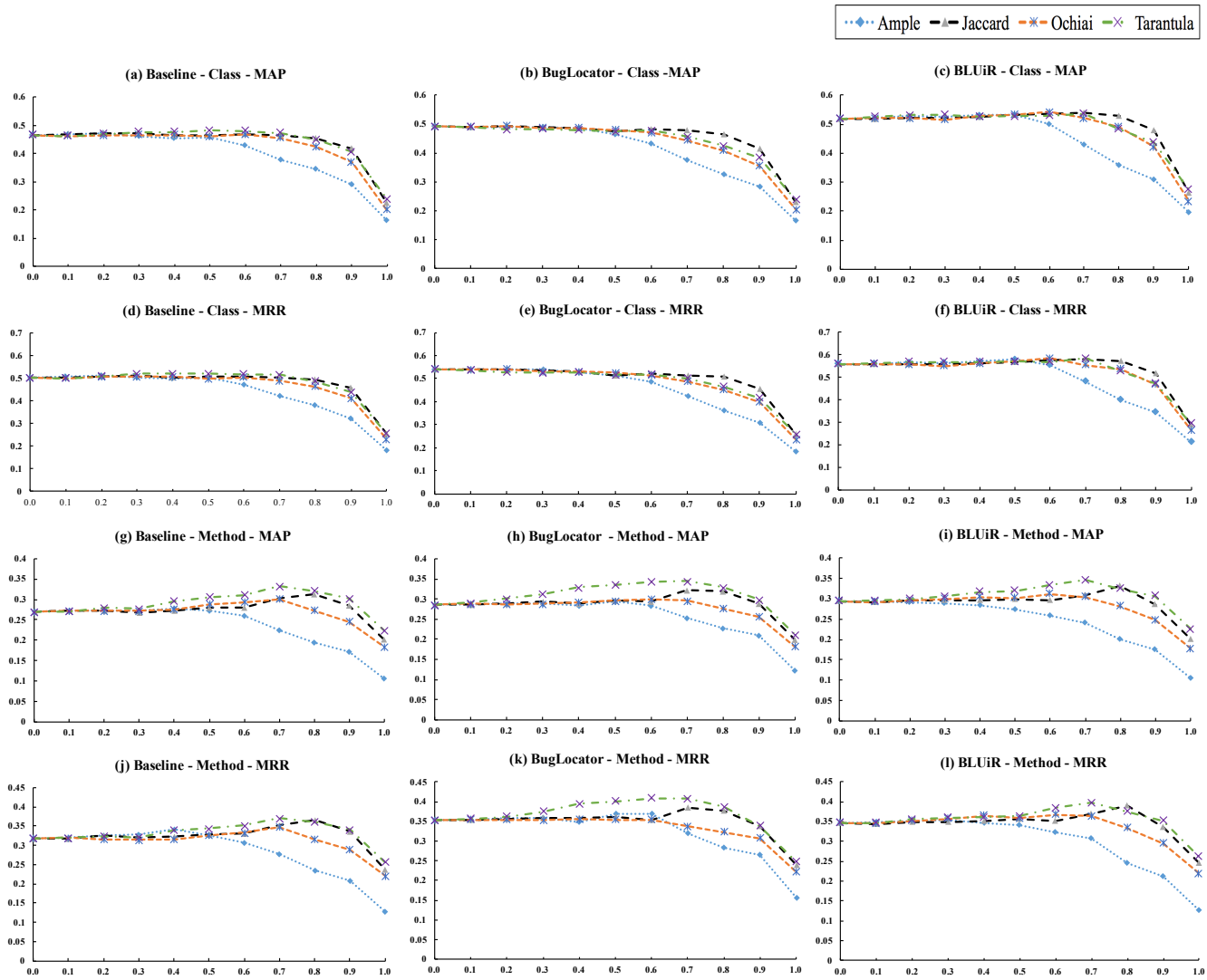


Figure 4.1: Effectiveness of IR_{cp} at the class level (a-f) and method level (g-l). The x-axis represents α. The y-axis of (a)-(c) and (g)-(i) represents MAP, while the y-axis in (d)-(f) and (j)-(l) is MRR.

the values are reported for IR_c. When $\alpha = 1$, the reported values are purely from spectrum information. We observed that IR_c always worked better than spectrum, because IR_c utilized both static and dynamic information, while spectrum was only dynamic information. The optimal combination between IR_c and spectrum always worked better than either component. Tarantula worked better than other spectrum information.

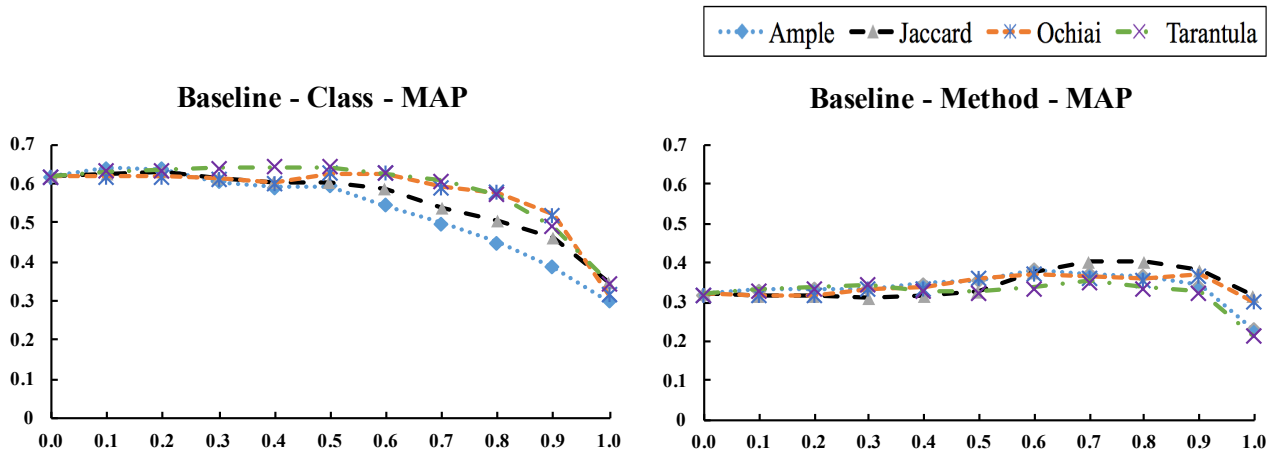


Figure 4.2: The MAP of IR_{sp} with the baseline IR technique

At class-level, IR_{cp} at most worked *slightly* better than IR_c. The reason may be that the class-level spectrum describes execution too coarsely: A class can contain many methods. Whenever a method is executed by a test, the whole class is considered covered. Such class-level spectrum makes suspiciousness scores less helpful.

When using Tarantula with $\alpha = 0.7$ for method-level **FL**, IR_{cp} significantly outperformed IR_c. As shown in Fig. 4.1(g-i) and Fig. 4.1(j-l), MAP was increased by 14–22%, while MRR was increased by16–211%. In particular, RHINO-519692 and its corresponding buggy method only shared one word in common: *transformNewExpr()*, so IR_c ranked the buggy method as 12^{th}, with a low score of 0.28. However, Tarantula considered the method as the most suspicious one, because both of the two tests executing it failed. Therefore, IR_{cp} effectively improved the method's rank as Top 1 by properly combining IR_c with Tarantula spectrum.

We also experimented with IR_{sp}, and made similar observations. With the baseline IR-based technique, IR_{sp} did not improve over IR_s at class level (on the left), but achieved noticeable improvement at method level (on the right) (Fig. 4.2).

Finding 2.5: *Spectrum information was effective to improve IRFL at method level instead of at class level. With Tarantula and $\alpha = 0.7$, IR_{cp} and IR_{sp} almost always achieved the best effectiveness.*

4.6.4 RQ4: How do our simple combinations compare with the state-of-the-art hybrid technique AML?

We directly took the AML results in prior work [66], and did similar experiments using our hybrid approaches IR_c and IR_{cp}. We experimented with the same dataset of 157 examples as AML for fair comparison. Since BLUiR always outperformed other tools when combined with execution information, it was used as the basic IR-based technique in the comparison. IR_{cp} was configured to use Tarantula spectrum with $\alpha = 0.7$, because the setting was demonstrated the best in Chapter §4.6.3.

As shown in Tab. 4.5, we found that AML ranked more Top-1 entities correctly than IR_c (31 vs. 29). Other than that, IR_c worked better in terms of Top-5, Top-10, and MAP metrics. The ***overall*** values of IR_{cp} were better than AML for all metrics. Although we did not compare IR_s and IR_{sp} directly with AML due to the JavaSlicer limitation, it is reasonable to expect both of them to perform better than AML. The reason is compared with coverage, slicing always identifies failure-relevant entities more precisely, and refines IR-based ranked lists more effectively. Our experiments with a subset of the data did demonstrate that IR_s worked better than IR_c, and IR_{sp} worked better than IR_{cp}.

The fact that our simple combination approaches worked better than AML reveals two things. First, all kinds of dynamic information can effectively improve IRFL. Second, a complex combination approach is not always necessary to better localize faults.

Finding 2.6: *IR_c outperformed AML in all metrics but one, while IR_{cp} outperformed AML. Our simple approaches worked better than AML's more complicated approach, showing that various execution information can effectively help with IRFL.*

4.7 Discussion

Our empirical study demonstrates that various kinds of execution information can *effectively* improve IRFL, as long as the information usage is *proper, but not necessarily complex*. In the study, there are still bug reports whose bugs are not located by any investigated approach. We further examined these reports and their bugs.

Among the 157 bug reports, 18 reports contain no clue about where the bug is, 77 reports mention bug-relevant code elements, such as fields or methods inside the buggy classes, and 62 reports have the exact buggy class names explicitly mentioned. For those reports without any clue of bug locations, a bug reporter usually describes the bug-triggering input(s) or bug symptoms, and the execution information does not help reveal bugs, either. Although such bugs do not count much in our dataset, they may be prevalent in reality, and require more advanced novel solutions.

For some reports with either buggy classes explicitly mentioned, or bug-relevant information (i.e., fields or methods in buggy classes) included, the investigated approaches failed for two reasons. First, some mentioned buggy entity names are so widely used that they are not distinctive, such as "*set*" and "*method*". Second, when test cases or call stacks include many entities to describe the problems, the noisy location information confuses IR-based techniques.

In the future, we will integrate static analysis-based fault prediction [48] techniques to better localize faults. For example, when a buggy method has a popular name like "set", and is covered by both passed and failed tests, neither the bug report nor execution information is helpful. We can use fault prediction to calculate various metrics (e.g., *fan-in/fan-out*[2]) to measure how likely each method is buggy. By ranking methods based on their fault prediction scores, we will obtain a ranked list, which can be further combined with the list produced by a hybrid approach of IRFL and execution information.

4.8 Threads to Validity

We reused an existing dataset of 157 real bugs from 4 open source projects to evaluate different fault localization techniques. The evaluation results may not generalize to other bugs or other open source projects. The collected execution information also strongly depends on the quality and availability of test cases.

We collected slicing information with JavaSlicer [2], because the tool is the only publicly available dynamic slicing tool based on our knowledge. The limitation of the tool may affect the generalizability of our observations. Besides, we used four most popular formulae to calculate spectrum, and used three IRFL techniques. The limited number of investigated formulae and IR-based techniques may also affect the generalizability.

[2]Fan-in denotes the number of methods invoking the method, while fan-out denotes the number of methods invoked by the method.

4.9 Summary

It is challenging to locate bugs given bug reports. In this empirical study, we investigated how various dynamic execution information (e.g., coverage, slicing, and spectrum information) can help with IRFL. We found that through refining the ranked list of suspicious locations produced by IR-based techniques, coverage and slicing information can effectively help improve fault localization. Spectrum information can further improve method-level **FL** by merging its suspicious location list with the coverage-refined or slicing-refined IR-based list.

Our investigation with the three types of execution information demonstrates that dynamic information can effectively improve IRFL, even though the information is integrated in simple ways. By comparing our combination approaches with a state-of-the-art hybrid technique of IRFL and spectrum, we observed that our simple approaches almost always worked better. It means that a combination approach does not have to be complicated for effective fault localization. When examining bugs that none of the investigated techniques can handle, we found it promising to conduct and combine static analysis-based fault prediction to further better fault localization.

Table 4.2: F-I vs. I-F at class level

Metric	Project	Baseline		BugLocator		BLUiR	
		F-I	**I-F**	**F-I**	**I-F**	**F-I**	**I-F**
Top 1	AspectJ	7	6	4	6	4	5
	Ant	29	32	31	33	27	31
	Lucene	14	14	12	15	11	14
	Rhino	3	4	3	9	8	11
	Overall	**53**	**56**	**50**	**63**	**50**	**61**
Top 5	AspectJ	18	18	16	16	17	19
	Ant	54	59	54	55	59	60
	Lucene	45	47	50	48	53	51
	Rhino	16	18	20	19	19	20
	Overall	**133**	**142**	**140**	**138**	**148**	**150**
Top 10	AspectJ	27	25	21	25	25	28
	Ant	63	64	62	63	65	68
	Lucene	61	59	67	59	72	68
	Rhino	25	26	25	28	26	25
	Overall	**176**	**174**	**175**	**175**	**188**	**189**
MAP	AspectJ	0.25	0.23	0.22	0.25	0.22	0.25
	Ant	0.63	0.73	0.70	0.71	0.67	0.72
	Lucene	0.50	0.49	0.49	0.54	0.50	0.55
	Rhino	0.33	0.40	0.34	0.47	0.50	0.55
	Mean	**0.43**	**0.46**	**0.44**	**0.49**	**0.47**	**0.52**
MRR	AspectJ	0.28	0.27	0.24	0.29	0.24	0.29
	Ant	0.67	0.75	0.73	0.76	0.69	0.75
	Lucene	0.60	0.59	0.55	0.61	0.58	0.64
	Rhino	0.32	0.40	0.36	0.51	0.51	0.56
	Mean	**0.47**	**0.50**	**0.47**	**0.54**	**0.51**	**0.56**

Table 4.3: IR vs. IR_c fault localization

Metric	Project	(a) Class Level						(b) Method Level					
		Baseline		BugLocator		BLUiR		Baseline		BugLocator		BLUiR	
		IR	**IR_c**	**IR**	**IR_c**	**IR**	**IR_c**	**IR**	**IR_c**	**IR**	**IR_c**	**IR**	**IR_c**
Top 1	AspectJ	4	6	2	6	3	5	3	3	2	4	2	3
	Ant	27	32	22	33	26	31	9	12	10	13	6	13
	Lucene	12	14	6	15	11	14	4	7	3	9	5	7
	Rhino	4	4	5	9	11	11	4	6	4	6	5	6
	Overall	**47**	**56**	**35**	**63**	**51**	**61**	**20**	**28**	**19**	**32**	**18**	**29**
Top 5	AspectJ	13	18	7	16	12	19	4	8	3	8	4	7
	Ant	48	59	44	55	45	60	21	33	20	36	16	36
	Lucene	34	47	32	48	41	51	14	30	15	33	16	32
	Rhino	15	18	13	19	19	20	7	10	8	14	7	15
	Overall	**110**	**142**	**96**	**138**	**117**	**150**	**46**	**81**	**46**	**91**	**43**	**90**
Top 10	AspectJ	18	25	14	25	20	28	5	12	6	16	6	12
	Ant	55	64	49	63	59	68	30	47	29	51	25	54
	Lucene	53	59	51	59	59	68	24	37	30	44	24	42
	Rhino	21	26	18	28	23	25	9	14	10	21	10	22
	Overall	**147**	**174**	**132**	**175**	**161**	**189**	**68**	**110**	**75**	**129**	**65**	**130**
MAP	AspectJ	0.15	0.23	0.10	0.25	0.14	0.25	0.08	0.11	0.07	0.14	0.06	0.10
	Ant	0.55	0.73	0.50	0.71	0.54	0.72	0.17	0.34	0.23	0.37	0.15	0.36
	Lucene	0.34	0.49	0.26	0.54	0.39	0.55	0.13	0.36	0.09	0.32	0.15	0.38
	Rhino	0.34	0.40	0.28	0.47	0.51	0.55	0.17	0.29	0.18	0.32	0.19	0.34
	Mean	**0.35**	**0.46**	**0.29**	**0.49**	**0.40**	**0.52**	**0.14**	**0.28**	**0.14**	**0.29**	**0.14**	**0.30**
MRR	AspectJ	0.18	0.27	0.12	0.29	0.17	0.29	0.09	0.12	0.07	0.16	0.07	0.12
	Ant	0.61	0.75	0.55	0.76	0.59	0.75	0.24	0.40	0.27	0.43	0.19	0.41
	Lucene	0.49	0.59	0.35	0.61	0.50	0.64	0.23	0.44	0.20	0.46	0.28	0.48
	Rhino	0.35	0.40	0.32	0.51	0.54	0.56	0.20	0.34	0.20	0.36	0.23	0.37
	Mean	**0.41**	**0.50**	**0.34**	**0.54**	**0.45**	**0.56**	**0.19**	**0.33**	**0.19**	**0.35**	**0.19**	**0.35**

Table 4.4: IR_c vs. IR_s fault localization

Metric	Project	(a) Class Level						(b) Method Level					
		Baseline		**BugLocator**		**BLUiR**		**Baseline**		**BugLocator**		**BLUiR**	
		IR_c	**IR_s**	**IR_c**	**IR_s**	**IR_c**	**IR_s**	**IR_c**	**IR_s**	**IR_c**	**IR_s**	**IR_c**	**IR_s**
Top 1	Ant	15	17	14	16	13	17	4	6	3	4	3	6
	Lucene	5	6	6	11	2	6	3	5	5	6	3	5
	Rhino	3	3	2	3	3	4	0	0	0	0	0	0
	Overall	**23**	**26**	**22**	**30**	**18**	**27**	**7**	**11**	**8**	**10**	**6**	**11**
Top 5	Ant	26	26	23	26	23	25	12	17	12	18	10	17
	Lucene	27	28	16	28	22	24	13	14	12	15	11	13
	Rhino	15	18	7	9	10	13	0	0	0	1	0	0
	Overall	**68**	**72**	**46**	**63**	**55**	**62**	**25**	**31**	**24**	**34**	**21**	**30**
Top 10	Ant	27	27	24	28	25	25	16	18	18	24	17	18
	Lucene	32	33	19	34	27	28	16	16	15	18	14	15
	Rhino	20	20	8	11	13	13	0	1	0	1	0	1
	Overall	**79**	**80**	**51**	**73**	**65**	**66**	**32**	**35**	**33**	**43**	**31**	**34**
MAP	Ant	0.71	0.79	0.71	0.75	0.71	0.85	0.37	0.52	0.33	0.34	0.35	0.52
	Lucene	0.50	0.58	0.44	0.62	0.44	0.54	0.30	0.45	0.35	0.45	0.34	0.46
	Rhino	0.47	0.50	0.50	0.50	0.48	0.58	0.02	0.04	0.02	0.11	0.02	0.04
	Mean	**0.56**	**0.62**	**0.55**	**0.62**	**0.54**	**0.66**	**0.23**	**0.34**	**0.23**	**0.30**	**0.24**	**0.34**
MRR	Ant	0.75	0.80	0.75	0.78	0.73	0.86	0.40	0.52	0.38	0.41	0.36	0.52
	Lucene	0.53	0.61	0.49	0.68	0.43	0.57	0.37	0.49	0.44	0.51	0.39	0.50
	Rhino	0.46	0.49	0.50	0.53	0.48	0.58	0.02	0.03	0.02	0.11	0.02	0.034
	Mean	**0.58**	**0.63**	**0.58**	**0.66**	**0.55**	**0.67**	**0.26**	**0.35**	**0.28**	**0.34**	**0.26**	**0.35**

Table 4.5: Fault localization comparison with AML at method level

Metric	Project	IR	IR_c	IR_{cp}	AML
Top 1	AspectJ	2	3	8	7
	Ant	6	13	14	9
	Lucene	5	7	6	11
	Rhino	5	6	8	4
	Overall	**18**	**29**	**36**	**31**
Top 5	AspectJ	4	7	14	13
	Ant	16	36	39	22
	Lucene	16	32	23	22
	Rhino	7	15	22	14
	Overall	**43**	**90**	**98**	**71**
Top 10	AspectJ	6	12	19	13
	Ant	25	54	57	31
	Lucene	24	42	30	29
	Rhino	10	22	27	19
	Overall	**65**	**130**	**133**	**92**
MAP	AspectJ	0.06	0.10	0.22	0.19
	Ant	0.15	0.36	0.37	0.23
	Lucene	0.15	0.38	0.30	0.28
	Rhino	0.19	0.34	0.50	0.24
	Overall	**0.14**	**0.30**	**0.35**	**0.24**

Chapter 5

Triggering Modes in Spectrum-Based Single-Location Fault Localization

5.1 Motivation

Software debugging is challenging. On average, up to 50% of developers' time is spent on finding and fixing bugs, and software bugs cost the economy $312 billion per year [3]. To simplify software debugging, various techniques have been introduced to automatically locate bugs or faults in programs [34, 57, 59, 61, 67, 82, 90, 116]. For instance, information retrieval (IR)-based fault localization (IRFL) approaches apply IR techniques to any given bug report and the corresponding buggy program, in order to retrieve and rank software entities (e.g., classes and methods) that are relevant to the report. Spectrum-based fault localization (SBFL) techniques instrument programs to (1) collect the execution coverage of each passed or failed test, and (2) compute the suspiciousness score for each executed program element (i.e., class, method, or statement).

However, existing work is insufficient to localize software bugs in industry. Specifically, IRFL works only when a report explicitly mentions the actual bug location [34, 99]. In reality, however, the bug location is not always mentioned in a report, and some serious buggy programs even require developers to urgently fix bugs before filing any report. Although SBFL tools are not limited by the availability of bug reports, they identify and rank suspicious (i.e.,

potentially buggy) locations only after executing *all* test cases. In the highly agile development environment of software companies, the runtime overhead of such all-test execution is not always acceptable. This is because there can be hundreds of thousands of tests, whose execution can last for hours or days. By waiting for SBFL to suggest any suspicious location, developers may miss the best time to fix bugs and deliver software releases.

To provide instant feedback on potential buggy locations when a program fails one or more tests, we were curious *whether SBFL can be triggered before all tests to complete their execution.* In this research, we conducted a comprehensive empirical study on (1) different SBFL techniques and (2) various modes to trigger SBFL. Specifically, we used an off-the-shelf tool—Clover [5]—to instrument source code, and to gather the statement-level coverage of each test. With the collected data for *all* tests, we applied 25 widely used SBFL formulas, computed the suspiciousness score of each statement, and ranked those statements accordingly.

In particular, we developed a framework—*Instant Fault Localization for Single-Location Bugs (***IFLS***)*—to locate bugs in five distinct modes: (i) $\textbf{IFLS}_1$ triggers SBFL right after the initial test failure; (ii) $\textbf{IFLS}_f^k$ triggers SBFL after every test failure (k=1–3); (iii) $\textbf{IFLS}_p^k$ triggers SBFL after the initial failure and several additional passed tests (k=1–10); (iv) $\textbf{IFLS}_t^k$ triggers SBFL at a fixed frequency/time (e.g., every $k = 2$ minutes); and (v) $\textbf{IFLS}_A$ triggers SBFL after executing the all or complete suite.

In our study, we applied **IFLS** to a software product in *Cvent*. The product contains 35,091 lines of code (LOC) for implementation and has 1,295 test cases (with 48,982 LOC). We constructed two data sets of bugs. The first set includes 28 injected bugs (i.e., the logical errors we manually introduced), while the second set has 13 real bugs. We experimented **IFLS** with different triggering mechanisms and various SBFL formulas, and evaluated the outputs by different settings in terms of precision (i.e., MAP), recall (i.e., Top-1 and Top-5),

and runtime overhead.

Based on our experiments, IFLS$_1$ outperformed the other four modes when using Ample [33] as its default SBFL formula. Among the 25 formulas, for injected bugs, IFLS$_A$ obtained 56% MAP and 66% Top-5 recall, and spent 663 seconds executing all tests on average. In comparison, IFLS$_1$ achieved 73% MAP and 86% Top-5 recall, and spent only 135 seconds executing roughly 20% of tests. For real bugs, IFLS$_A$ acquired 46% MAP and 56% Top-5 recall based on all-test execution; while IFLS$_1$ achieved 76% MAP and 92% Top-5 recall after executing test cases for only 106 seconds. With both data sets, we consistently observed considerable improvements of IFLS$_1$ over IFLS$_A$ in terms of the quality of suggestions and runtime overhead.

5.2 Contributions

This research makes the following contributions:

- We built IFLS, a tool infrastructure that can trigger SBFL in 5 alternative modes using 25 distinct formulas.
- We applied IFLS to a close-sourced software project, and comprehensively evaluated the fault localization effectiveness by enumerating different combinations between triggering modes and calculation formulas.
- We made two interesting observations in our exploration. First, among the five modes, IFLS$_A$ did not outperform the others although its runtime cost was the highest. Second, IFLS$_1$ worked equally well when using certain formulas.

We open-sourced IFLS at https://github.com/idf-icst/idfl-package.

5.3 Approach

We built a tool infrastructure—**IFLS**—to investigate the best mode of triggering SBFL. As shown in Fig. 5.1, **IFLS** has three components. The first component is Clover [5]; we configured it to instrument every Java statement and to record **per-test statement coverage**, i.e., which statement(s) are covered by a given test.

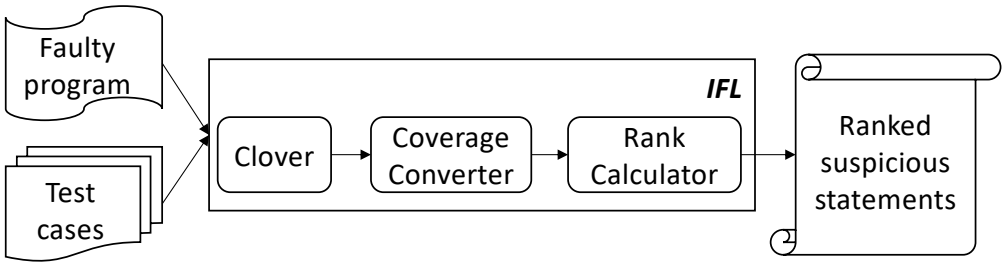


Figure 5.1: Overview of **IFLS**

Coverage Converter controls the frequency at which per-test statement coverage is converted to **per-statement test coverage**, i.e., which tests cover a particular statement. This conversion is necessary because all SBFL formulas require for the element-level coverage measurement (i.e., e_f and/or e_p). Fig. 5.2 illustrates the conversion process with a simple example. As shown in the figure, Clover stores per-test coverage data in a JSON file such that for any executed test (e.g., t1), we can easily retrieve the statements (e.g., s1) covered by that execution. **IFLS** reorganizes the data in a different JSON file such that given any statement (e.g., s1), we can obtain the number of passed or failed tests covering the statement.

Rank Calculator is invoked by *Coverage Converter* after each round of data conversion. This component applies an SBFL formula to the collected per-statement coverage data, in order to identify and rank suspicious locations. Essentially, the frequency at which *Coverage Converter* transforms data determines how SBFL is applied to partial execution data (i.e., code coverage). *Coverage Converter* can apply SBFL in the following five distinct ways: $\textbf{IFLS}_1$, $\textbf{IFLS}_f^k$, $\textbf{IFLS}_p^k$, and $\textbf{IFLS}_t^k$, and $\textbf{IFLS}_A$.

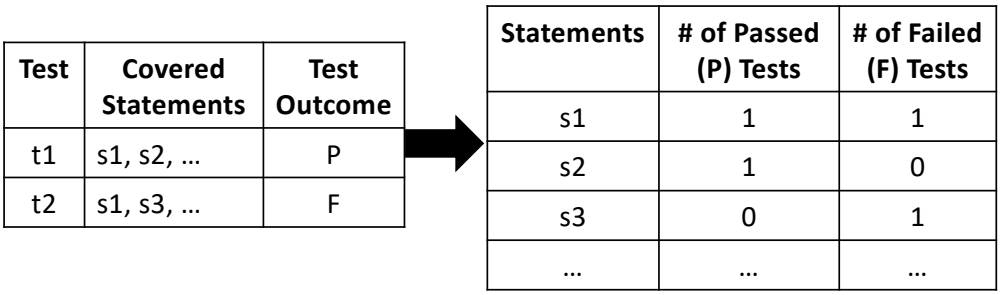


Figure 5.2: IFLS converts per-test statement coverage to per-statement test coverage

5.4 Research Questions

This study investigates the following two research questions:

RQ1 How sensitive is IFLS to different SBFL triggering modes?

RQ2 How sensitive is IFLS to the leveraged SBFL formulas?

5.5 Evaluation

This section first introduces our data sets (Chapter §5.5.1). Next, it presents the effectiveness comparison between different triggering modes of IFLS (Chapter §5.6.1). Finally, it explains our exploration of IFLS's sensitivity to the used SBFL formulas (Chapter §5.6.2).

5.5.1 Data Sets

For evaluation, we leveraged a closed-sourced software system in industry. We chose this software because (1) there are a lot of test cases (i.e., 1,295) written by developers for quality assurance, (2) the code size is large (35,091 LOC), and (3) it is from software industry and may have different program features from open-source projects. We constructed 2 bug sets,

Table 5.1: The number of failed tests triggered by different injected or real bugs

# of Failed Tests	# of Injected Bugs	# of Real Bugs
1	16	6
2	6	3
4	1	4
9	3	0
12	1	0
15	1	0

including a set of 28 injected bugs and a set of 13 real bugs. All these bugs are single and semantic faults, each of which fails at least one test case. Specifically, as shown in Tab. 5.1, there are 16 injected bugs and 6 real bugs that fail single tests. Each of the remaining bugs fail at least two tests.

To inject the 28 bugs, we first consulted with developers concerning what are the frequent bugs and usual buggy locations in their programs. Based on developers' inputs, we then manually crafted buggy programs by either substituting operators (e.g., "&&" replaced with "||"), changing constant values (e.g., "0" replaced by "1"), modifying function calls (e.g., "Math.min()" replaced with "Math.max()"), or swapping function arguments of the same data type. We decided not to use mutation testing to generate buggy programs for two reasons. First, the generated mutants may be very different from real bugs [47]. Second, the effectiveness of mutation operations can vary with the subject programs. Due to our discussion with the owner developers, we have more domain knowledge about the recurring bugs in the subject program. Therefore, the bugs we manually injected are more likely to reflect real bugs in the project.

We identified 13 real bugs by searching for single-line fixes in the software version history. Specifically, if a commit has a single-line change and contains keywords like "bug" or "fix" in the commit message, we checked out the program snapshot before that commit as a buggy program. These real bugs are mainly about incorrectly used variable names, division by

Table 5.2: Comparison between **IFLS**$_1$ and **IFLS**$_A$ in terms of effectiveness and runtime overhead

Mode	Top-1 (%)		Top-5 (%)		MAP (%)		Time Cost (second)	
	I	R	I	R	I	R	I	R
IFLS$_1$	**64**	**62**	86	**92**	**73**	**76**	135	105
IFLS$_A$	57	31	**89**	69	70	50	663	655

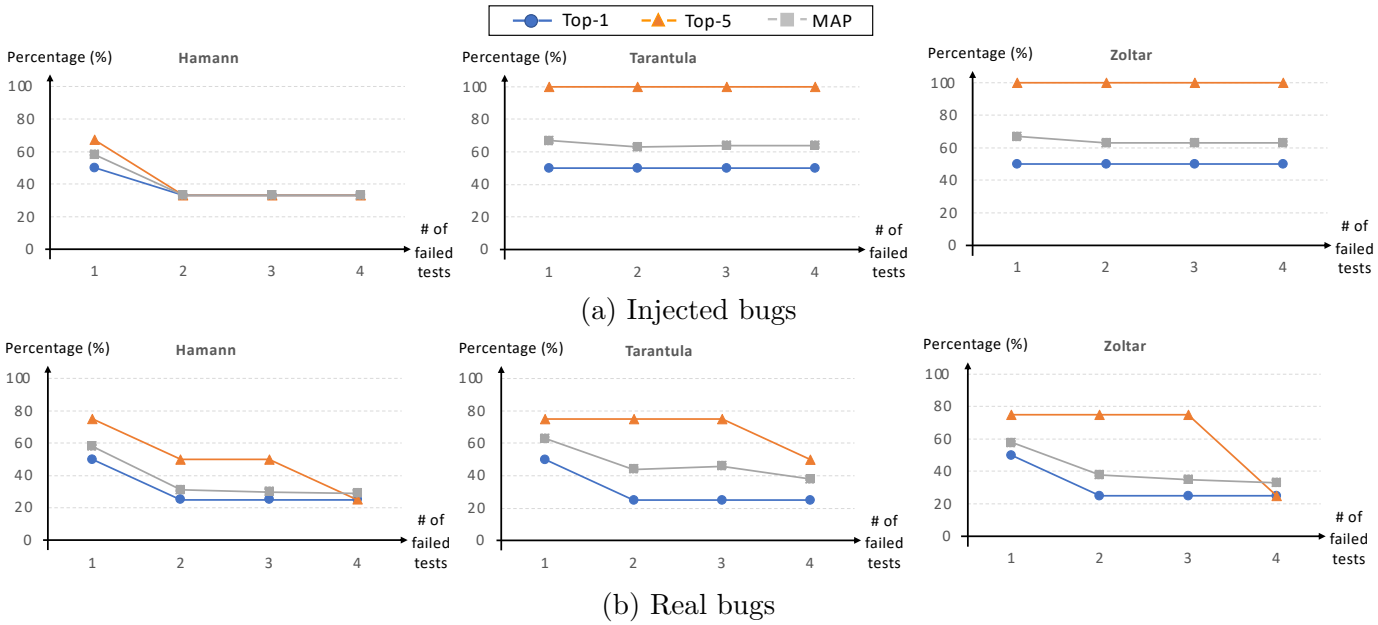
zero, incorrect calculation formulas, unhandled exceptions, and incorrect condition checks for variables' lower bounds.

5.6 Results and Discussion

5.6.1 Comparison between IFLS's Triggering Modes

To compare the fault localization effectiveness of different triggering modes, we used the Ample formula [33] as the default ranking formula in **IFLS**. This is because our other experiment (see Chapter §5.6.2) shows that Ample generally achieved a better trade-off among Top-1, Top-5, and MAP values than the other formulas.

Effectiveness of IFLS$_1$ and IFLS$_A$ Tab. 5.2 presents the results by **IFLS**$_1$ and **IFLS**$_A$, where the highest value of each effectiveness measurement is highlighted **in bold**. Because each of these two modes triggers SBFL only once during the whole execution of any buggy program version, the table has only one row to report the average effectiveness measurements for each mode. As shown in the table, **IFLS**$_1$ outperformed **IFLS**$_A$ by executing fewer tests and locating bugs more effectively. Specifically with the injected bugs, **IFLS**$_1$ spent on average 135 seconds and acquired 64% Top-1, 86% Top-5, and 73% MAP values; however, **IFLS**$_A$ spent on average 663 seconds and obtained 57% Top-1, 89% Top-5, and 70% MAP



(a) Injected bugs

(b) Real bugs

Figure 5.3: IFLS$_f^k$'s effectiveness when different formulas were used

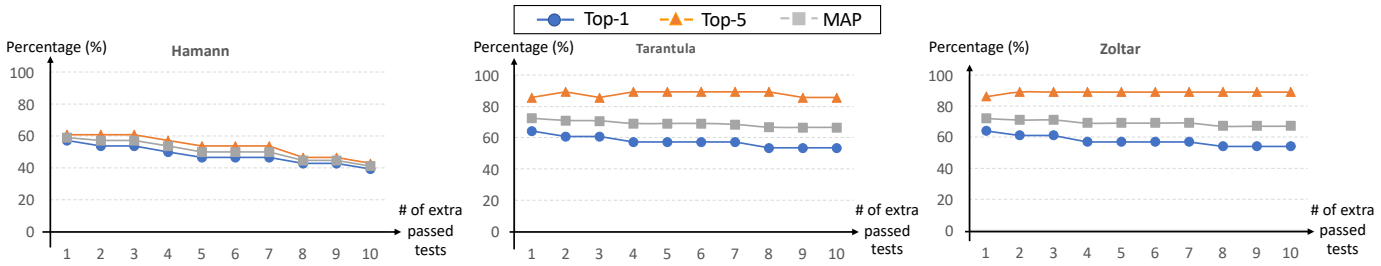
values. Additionally, with the real bugs, IFLS$_1$ spent 105 seconds and got 62% Top-1, 92% Top-5, as well as 76% MAP values; IFLS$_A$ spent 655 seconds but acquired 31% Top-1, 69% Top-5, and 50% MAP values.

Finding 3.1: *Compared with* IFLS$_A$, IFLS$_1$ *located bugs with much lower runtime overhead but a better trade-off between MAP, Top-1, and Top-5 values. It means that triggering SBFL right after the initial test failure can significantly improve fault localization.*

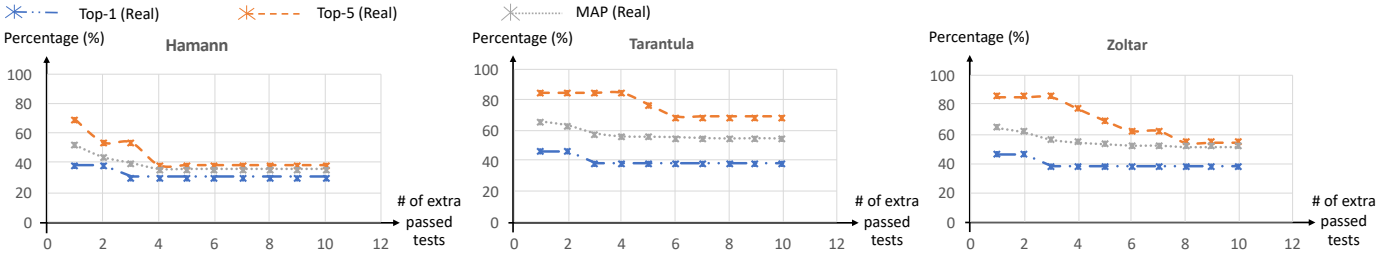
Effectiveness of IFLS$_f^k$ In our data set, there are six injected bugs and four real bugs that fail at least four test cases. To evaluate the effectiveness of IFLS$_f^k$, we triggered SBFL after one–four failed tests and reported the average measurements among these multi-failure bugs in Tab. 5.3. Hypothetically, as the number of failed tests increases, more execution

Table 5.3: IFLS_f^k's effectiveness when IFLS_f^k reranked locations after each extra test failure (12 injected and 7 real bugs)

# of Failed Tests	Top-1 (%)		Top-5 (%)		MAP (%)		Time Cost (second)	
	I	R	I	R	I	R	I	R
1	50	50	100	100	67	71	151	106
2	50	25	100	100	64	54	153	108
3	50	25	100	100	64	54	154	110
4	50	25	100	100	67	54	155	113



(a) Injected bugs



(b) Real bugs

Figure 5.4: IFLS_p^k's effectiveness when different formulas were used

information is collected and thus it is possible that SBFL can work better. However, surprisingly, Tab. 5.3 shows that the effectiveness measurements do not necessarily increase with the number of failed tests.

In particular, when more test failures were triggered by each injected bug, both Top-1 and Top-5 values remained the same while MAP first decreased and then increased. Among the real bugs, as the number of failed tests increased, both Top-1 and MAP values decreased while the Top-5 value remained. Between the first and fourth failures, on average, the runtime overhead of $\textbf{IFLS}_f^k$ increased from 151 seconds to 155 seconds for injected bugs, and increased from 106 seconds to 113 seconds for real bugs.

To validate the generality of our observation, we redid the experiment with $\textbf{IFLS}_f^k$ by using the other 24 SBFL formulas. Since some formulas produced exactly the same results (e.g., Hamann and Sokal), in Fig. 5.3, we illustrate $\textbf{IFLS}_f^k$'s effectiveness measurements for three representative formulas: Hamann, Tarantula, and Zoltar. As shown in Fig. 5.3, $\textbf{IFLS}_f^k$'s effectiveness measurements either decreased or remained the same as the number of failed tests increased. In particular, with Hamann, all measured values dropped down significantly at the second test failure. With Tarantula and Zoltar, the measured values were almost unchanged for injected bugs; however, as the number of failed tests increased, the measured values dropped considerably for real bugs. One possible reason is that although the occurrence of more failed tests can strengthen the suspiciousness signals of faulty locations, the existence of a lot more passed tests can weaken those signals, and even harm $\textbf{IFLS}_f^k$'s effectiveness in some scenarios.

Table 5.4: IFLS$_p^k$'s effectiveness when the data of 1–10 more passed tests was also included

# of Additional Passed Tests	Top-1 (%)		Top-5 (%)		MAP (%)		Time Cost (second)	
	I	R	I	R	I	R	I	R
1	64	46	86	92	73	69	135	106
2	64	46	89	92	73	66	136	107
3	64	38	89	92	73	61	137	107
4	61	38	89	92	72	59	137	108
5	61	38	89	85	72	59	138	108
6	61	38	89	77	72	59	139	109
7	61	38	89	77	71	58	139	110
8	57	38	89	77	69	58	140	110
9	57	38	89	77	69	58	141	111
10	57	38	89	77	69	54	141	112

Finding 3.2: *Compared with* IFLS$_1$, IFLS$_f^k$ *incurred more runtime overhead by profiling more execution and ranking locations multiple times. However,* IFLS$_f^k$ *does not work better when more tests fail, due to the extreme imbalance between passed and failed tests.*

Effectiveness of IFLS$_p^k$ Tab. 5.4 presents our evaluation results for IFLS$_p^k$, which triggered data processing after the initial test failure together with 1–10 additional passed tests. As shown in the table, when more passed tests were executed after the initial failure, the runtime overhead increased as expected (i.e., from 135 seconds to 141 seconds for injected bugs, and from 106 seconds to 112 seconds for real bugs).

Hypothetically, as more execution data is available, IFLS$_p^k$ should localize bugs more effectively. However, different from our expectation, both Top-1 and MAP values decreased; the Top-5 values for injected bugs increased while those values for real bugs decreased.

One possible reason to explain the unexpected trend is the leveraged Ample formula: *Ample* =

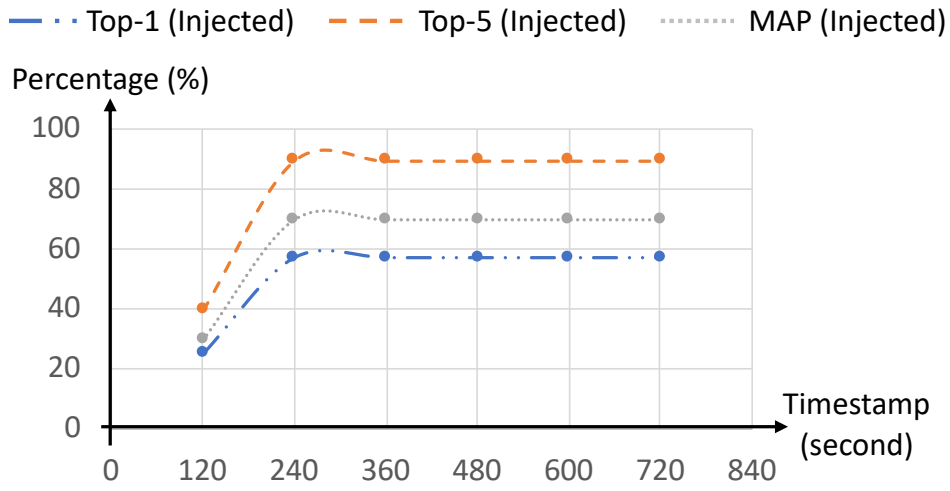


Figure 5.5: IFLS_t^k's effectiveness when SBFL is triggered every two minutes

$|\frac{e_f}{e_f+n_f} - \frac{e_p}{e_p+n_p}|$.

After the initial test failure, there are two types of locations that are likely to be highly ranked:

- **Type-I (fail-dominant)** statements covered by the failed test but rarely covered by any passed test (e.g., $\frac{1}{2} = \frac{e_f}{e_f+n_f} \gg \frac{e_p}{e_p+n_p}$),
- **Type-II (pass-dominant)** statements that are covered by many passed tests but not covered by the failed test (e.g., $\frac{e_p}{e_p+n_p} \gg \frac{e_f}{e_f+n_f} = 0$).

For better fault localization, we desire to see more Type-I but fewer Type-II statements included in top ranks. However, as one or more passed tests are provided after the initial failure, it is likely that fewer fail-dominant statements but more pass-dominant statements are highly ranked, compromising the effectiveness of IFLS_p^k. Notice that in all the experiments show in Tab. 5.4, IFLS_p^k achieved better trade-offs than IFLS_A, which fact also evidences that the execution profile of more passed tests usually does not help improve fault localization effectiveness.

For generality, we also configured IFLS_p^k to use all of the remaining 24 ranking formulae (besides Ample). Due to the space limit, here we only visualize IFLS_p^k's effectiveness with three representative SBFL formulas: Hamann, Tarantula, and Zoltar. Fig. 5.4 (a) and Fig. 5.4 (b) separately present the measurements based on injected bugs and real bugs. As more extra passed tests are added, all metric values related to Hamann go down. For Tarantula and Zoltar, although the Top-5 values increase in some scenarios (i.e., for injected bugs), both Top-1 and MAP values decrease. With different formulas explored, we observe a typical trend: the effectiveness of IFLS_p^k does not increase with the number of extra passed tests.

Finding 3.3: *Compared with* IFLS_1, IFLS_p^k *is more likely to raise the ranking of non-buggy locations and lower the ranking of buggy ones, due to its usage of the extra data for passed tests after the initial test failure.*

Effectiveness of IFLS_t^k Fig. 5.5 illustrates IFLS_t^k's effectiveness when the ranking of suspicious locations is updated every 2 minutes (i.e., 120 seconds). With IFLS_t^k, we explored how SBFL behaves when it is triggered at a fixed frequency. As shown in the figure, most measured values increase between the 120^{th} and 240^{th} seconds and decrease or remain the same afterwards. The only outlier is Top-1 for real bugs, whose value decreases in the range [120, 240] and then remains the same.

Three possible reasons can explain the relative poor results at the 120^{th} second. First, insufficient failure data was gathered during the first 120 seconds. We found that 15 injected bugs and 3 real bugs did not fail any test in that period. Due to the lack of failed tests, IFLS_t^k could not effectively locate bugs for its initial trial. Second, all faulty programs obtained the initial test failures before the 240^{th} second. The related data of failed tests boosted most measurements for IFLS_t^k. As a result, IFLS_t^k acquired the peak measurement values for

injected bugs, i.e., 57% Top-1, 89% Top-5, and 70% MAP; it obtained the highest Top-5 and MAP values for real bugs (i.e., 77% and 53%).

Third, after the 240^{th} second, even though more execution data was available, most faulty versions caused no more failed test. The extremely unbalanced increase between passed and failed tests brought no improvement for $\textbf{IFLS}_t^k$'s effectiveness.

By comparing $\textbf{IFLS}_t^k$'s peak performance against that of $\textbf{IFLS}_1$, we found that $\textbf{IFLS}_1$ worked better even though $\textbf{IFLS}_t^k$ profiled more data and ranked locations repetitively. Similar to what we observed for $\textbf{IFLS}_f^k$, the extra execution data and reranking effort did not enable $\textbf{IFLS}_t^k$ to outperform $\textbf{IFLS}_1$. Additionally, $\textbf{IFLS}_t^k$ seems to outperform $\textbf{IFLS}_A$ at certain time points (e.g., 240^{th} second) before the completion of whole-suite execution. However, since different programs and distinct bugs may have failed tests executed at different time points, it is almost impossible to conclude what is the best frequency to trigger SBFL for $\textbf{IFLS}_t^k$ and at which triggering point $\textbf{IFLS}_t^k$ is likely to work best.

Finding 3.4: *Similar to* $\textbf{IFLS}_f^k$ *and* $\textbf{IFLS}_p^k$, $\textbf{IFLS}_t^k$ *worked worse than* $\textbf{IFLS}_1$ *even if it leveraged more execution data and dynamically computed ranking more often.*

Among the five investigated variants, $\textbf{IFLS}_1$ worked best in terms of efficiency and effectiveness. Although $\textbf{IFLS}_1$ triggered SBFL only once right after the initial test failure, it localized bugs with the best trade-off among Top-1, Top-5, and MAP values. Both $\textbf{IFLS}_f^k$ and $\textbf{IFLS}_p^k$ worked less effectively than $\textbf{IFLS}_1$ but better than $\textbf{IFLS}_A$. The execution profile of additional failed tests and/or passed tests does not help improve fault localization, but usually harms the effectiveness. Finally, $\textbf{IFLS}_t^k$ did out outperform $\textbf{IFLS}_A$. By triggering SBFL at a fixed fixed frequency, $\textbf{IFLS}_t^k$ might invoke SBFL so early that no test failure was available to facilitate fault localization, or invoke SBFL so late that too many passed tests were available

Table 5.5: The effectiveness of **IFLS**$_1$ and **IFLS**$_A$ when different formulas were used

Variants of IFLS$_1$	**Top-1 (%)**		**Top-5 (%)**		**MAP (%)**		**Variants of IFLS$_A$**	**Top-1 (%)**		**Top-5 (%)**		**MAP (%)**	
	I	R	I	R	I	R		I	R	I	R	I	R
IFLS$_1$-Ample	64	**62**	**86**	**92**	**73**	**76**	IFLS$_A$-Ample	57	31	**89**	**69**	**70**	50
IFLS$_1$-Anderberg	64	**62**	**86**	85	**73**	74	IFLS$_A$-Anderberg	57	38	86	**69**	68	53
IFLS$_1$-Dice	64	**62**	**86**	85	**73**	74	IFLS$_A$-Dice	57	38	86	**69**	68	53
IFLS$_1$-Euclid	**68**	54	68	54	68	54	IFLS$_A$-Euclid	**68**	**69**	68	**69**	68	**69**
IFLS$_1$-Goodman	64	**62**	**86**	85	**73**	74	IFLS$_A$-Goodman	57	38	86	**69**	68	53
IFLS$_1$-Hamann	57	54	61	69	59	60	IFLS$_A$-Hamann	39	31	43	38	41	36
IFLS$_1$-Hamming	**68**	54	68	54	68	54	IFLS$_A$-Hamming	**68**	**69**	68	**69**	68	**69**
IFLS$_1$-Jaccard	64	**62**	**86**	85	**73**	74	IFLS$_A$-Jaccard	57	38	86	**69**	68	53
IFLS$_1$-Kulczynski1	64	**62**	**86**	85	**73**	74	IFLS$_A$-Kulczynski1	57	38	86	**69**	68	53
IFLS$_1$-Kulczynski2	0	0	0	0	0	0	IFLS$_A$-Kulczynski2	0	0	0	0	0	0
IFLS$_1$-M1	57	54	61	69	59	60	IFLS$_A$-M1	39	31	43	38	41	36
IFLS$_1$-M2	64	54	**86**	**92**	**73**	71	IFLS$_A$-M2	57	31	**89**	**69**	**70**	49
IFLS$_1$-Ochiai	64	**62**	**86**	85	72	74	IFLS$_A$-Ochiai	57	38	86	**69**	68	54
IFLS$_1$-Ochiai2	64	**62**	**86**	85	72	72	IFLS$_A$-Ochiai2	57	38	86	62	68	53
IFLS$_1$-Overlap	0	0	0	0	0	0	IFLS$_A$-Overlap	0	0	0	0	0	0
IFLS$_1$-RogersTanimoto	57	54	61	69	59	60	IFLS$_A$-RogersTanimoto	39	31	43	38	41	36
IFLS$_1$-RussellRao	64	54	**86**	85	**73**	70	IFLS$_A$-RussellRao	57	31	**89**	**69**	**70**	49
IFLS$_1$-SimpleMatching	57	54	61	69	59	60	IFLS$_A$-SimpleMatching	39	31	43	38	41	36
IFLS$_1$-Sokal	57	54	61	69	59	60	IFLS$_A$-Sokal	39	31	43	38	41	36
IFLS$_1$-SϕrensenDice	64	**62**	**86**	85	**73**	74	IFLS$_A$-SϕrensenDice	57	38	86	**69**	68	53
IFLS$_1$-Tarantula	64	**62**	**86**	85	72	73	IFLS$_A$-Tarantula	54	38	79	**69**	65	53
IFLS$_1$-Wong1	**68**	54	68	54	68	54	IFLS$_A$-Wong1	**68**	**69**	68	**69**	68	**69**
IFLS$_1$-Wong2	61	54	68	69	64	62	IFLS$_A$-Wong2	46	31	57	54	52	40
IFLS$_1$-Wong3	61	54	71	77	65	65	IFLS$_A$-Wong3	50	31	68	62	59	45
IFLS$_1$-Zoltar	64	**62**	**86**	85	72	72	IFLS$_A$-Zoltar	54	38	86	54	66	51

to weaken the signals raised by any test failure.

5.6.2 IFLS's Sensitivity to SBFL Formulas

We ran **IFLS** with all 25 alternative formulas listed in Tab. 3.1 (1) to explore **IFLS**'s sensitivity to the adopted formulas and (2) to ensure the generalizability of our observation that **IFLS**$_1$ outperforms **IFLS**$_A$ (see Chapter §5.6.1). We decided to experiment with 25 distinct formulas instead of using only a few well-known formulas, in order to make our investigation comprehensive and systematic.

Tab. 5.5 presents the results by **IFLS**$_1$ and **IFLS**$_A$ when they used distinct formulas. Accord-

ing to the table, in the experiments with $\textbf{IFLS}_1$, Ample worked best by obtaining the highest values for five out of the six measurements. Many formulas worked similarly to each other. For instance, five formulas produced identical results, including Anderberg, Dice, Goodman, Jaccard, and Kulczynski1. Three formulas obtained the same highest Top-1 value for injected bugs (i.e., 68%), including Euclid, Hamming, and Wong1. The former group of five formulas mentioned above outperformed the latter group by achieving a better trade-off among metrics. Kulczynski2 and Overlap produced pure zero values.

In the experiments with $\textbf{IFLS}_A$, we observed that Euclid, Hamann, and Wong1 worked best by achieving better trade-offs among metrics than the other formulas. Each of these three formulas obtained the highest values for four out of the six metrics. Once again, Anderberg, Dice, Goodman, Jaccard, and Kulczynski1 produced identical results, although this group of formulas worked less effectively than the three-formula group mentioned above.

Our experiments imply that **IFLS** is sensitive to the used formula, as the measurement difference between the most and least effective formulas (e.g., Ample vs. Overlap, or Euclid vs. Kulczynski2) was huge.

Finding 3.5: *Our investigation on SBFL triggering modes is sensitive to the leveraged formula. The 25 formulas can be split into several groups, each of which contain formulas generating the same results.*

In Tab. 5.5, if we compare $\textbf{IFLS}_1$ with $\textbf{IFLS}_A$ for each formula, we observe that $\textbf{IFLS}_1$ outperformed $\textbf{IFLS}_A$ in the majority of scenarios. For instance, when Anderberg was used, $\textbf{IFLS}_1$ acquired 64% Top-1, 86% Top-5, and 73% MAP for injected bugs; it obtained 62% Top-1, 85% Top-5, and 74% MAP for real bugs. On the other hand, when $\textbf{IFLS}_A$ used the same formula, it achieved 57% Top-1, 86% Top-5, and 68% MAP for injected bugs; it acquired

38% Top-1, 69% Top-5, and 53% MAP for real bugs.

IFLS_A only outperformed IFLS_1 when one of the following three formulas was used: Euclid, Hamming, and Wong1. Specifically, when Hamming was in use, both IFLS_A and IFLS_1 achieved 68% for all metrics on the injected bug set; on the real bug set, IFLS_A acquired 69% for all metrics while IFLS_1 obtained 54%. Finally, when Kulczynski2 and Overlap were in use, both IFLS_1 and IFLS_A worked equally poorly, probably because these two formulas are not effective to locate bugs.

Finding 3.6: *Among the 25 investigated SBFL formulas,* IFLS_1 *outperformed* IFLS_A *for 20 formulas. It means that our observation that* IFLS_1 *usually outperforms* IFLS_A *is generalizable.*

5.7 Threads to Validity

Threats to External Validity Our experiments were conducted based on two data sets of bugs for a software product system of a company. The evaluation results may not generalize well to other bugs, other software products of other companies, or open-source projects. Our observations also depend on the quality and quantity of test cases. In the future, we will experiment with more buggy programs of more software systems.

Threats to Construct Validity Among the investigated 41 bugs, each bug can be fixed with a single-line change and the corresponding faulty program version contains a single bug. In reality, nevertheless, there are complex buggy programs that contain multiple faults in one version. To fix a bug, developers may need to change multiple lines of code in

the same source file, and/or even modify configurations in non-source files. Our data set shares such limitations with prior work [28, 55, 57, 67]. In the future, we will diversify our approach to generate injected bugs and use more complicated real bugs to better evaluate fault localization techniques.

Threats to Internal Validity Similar to prior fault localization research [67, 106, 122], given a bug fix, we treated the location where the fixing change was applied as ground truth. However, in reality, the place where a bug is fixed is not always the place where a bug is found. For instance, when a program fails to retrieve any record from a database, the bug location lies in the code querying the database, while the bug fix may be the SQL file used to update the database records. Treating patch locations to be equivalent to bug locations can introduce bias to the evaluated results.

5.8 Summary

This research presents our exploration on the distinct triggering mode of SBFL techniques. Although researchers extensively investigated the area of (semi-)automatic fault localization, one practical problem seems to be overlooked: *Is it always necessary for SBFL techniques to wait for all test cases to finish their execution before diagnosing the root cause(s) of failed tests?* With this empirical study, we comprehensively investigated this problem.

Specifically, we built **IFLS**, a framework that triggers SBFL in five alternative modes: triggering SBFL right after the initial failure ($\textbf{IFLS}_1$), after every failed test ($\textbf{IFLS}_f^k$), after the initial failure and several extra failed tests ($\textbf{IFLS}_p^k$), at a fixed frequency ($\textbf{IFLS}_t^k$), or after the execution of all tests ($\textbf{IFLS}_A$). By comparing the Top-1, Top-5, and MAP values achieved by different triggering modes when SBFL techniques were applied to two data sets, we sur-

prisingly found that $\textbf{IFLS}_1$ worked best in most scenarios. Namely, triggering SBFL right after the initial test failure turns out to be more effective and more efficient than triggering SBFL later when more execution data is available.

Our empirical study indicates that it is neither necessary nor helpful to execute all test cases before using SBFL formulas to locate bugs. $\textbf{IFLS}_1$ demonstrates the promising adoption of SBFL for recognizing faults in large-scale systems, even though the test execution of such systems can last forever. In the CI/CD software practices nowadays, $\textbf{IFLS}_1$ is more likely to satisfy developers' need of diagnosing test failures earlier, faster, and better. Our research will shed light on new research directions, such as agile fault localization based on the stream data of execution profiles, and periodic health check for software systems that run continuously without interruption. We plan to pursue these directions in the future.

Chapter 6

Triggering Modes in Spectrum-Based Multi-Location Fault Localization

6.1 Motivation

Debugging software is a well-known and challenging task that consumes a significant amount of developers' time. The process is also expensive, and software defects alone cost the US economy $1.56 trillion in 2020 [63]. To aid debugging, many automatic *fault localization* techniques have been proposed. For example, the well-known spectrum-based fault localization (SBFL) technique [19, 57] instruments programs to (i) collect execution coverage of passed and failed tests, and (ii) compute a suspiciousness score for each program element such as classes, methods, and statements.

Despite its popularity, SBFL may not be practical for industry deployment because it typically requires the execution of all or a significant portion of test cases before it can analyze and rank buggy locations. To further understand and address this limitation, in a previous work [35], we studied the necessity of this requirement. Specificallly, we explored various trigger modes, such as applying SBFL after the first test failure or after a combination of initial failures along with additional failed or passed tests. We evaluated these trigger modes using a variety of SBFL methods and found that the application of SBFL immediately after the first test failure appears to be equally effective or even better than other trigger modes,

including those that execute all tests. These findings are interesting and promising for the practical uses of SBFL in an industrial setting with a large number of tests.

Recently, we attempted to integrate SBFL in the debugging process of *Cvent*, a Northern Virginia-based company that offers meeting and event management software solutions to clients including event planners, attendees, and hosts. While the study in [35] plays a crucial role in this decision, considering CI/CD pipeline of *Cvent* consumes thousands of CPU hours to execute integration tests, it has a major limitation in considering only *single-location bugs*, whose root causes confined to individual lines of code (i.e., the bug can be fixed by modifying a single line). However, bugs encountered at *Cvent* often have root causes that span multiple lines of code (requiring the modification of multiple lines to fix). For instance, we found that that around 72% of *Cvent*'s bugs committed and fixed during the development stage were multi-location bugs. More generally, previous studies, e.g., in [91], found that it is not practical to assume a program contains only single-location bugs.

Motivated by this challenge, in this work we assess the effectiveness of SBFL for multi-location bugs. Specifically, we extend our initial study to develop a framework called *Instant Fault Localization for Multi-location Bugs* (**IFLM**) that triggers SBFL in four modes: (i) $\textbf{IFLM}_1$ invokes SBFL right after the first failed test; (ii) $\textbf{IFLM}_f^k$ triggers SBFL after every k=1–5 test failures ($\textbf{IFLM}_1$ is the special case when $k = 1$); (iii) $\textbf{IFLM}_p^k$ activates SBFL after the first failed test and subsequently, k additional passed tests (k=1 – 10); and (iv) $\textbf{IFLM}_A$ triggers SBFL after executing all tests.

We conduct the **IFLM** study using two *multi-location* bug datasets: (i) the open-source Defect4J dataset [12], consisting of 174 real bugs and 37 artificial bugs, and (ii) the close-source *Cvent* dataset, consisting of 27 real bugs.

Similarly to our prior single-location study, we found that we do not always need to run all

test cases for SBFL to be effective for multi-location bugs. This is promising for industrial environments with a multitude of tests. Specifically, compared to **IFLM**$_A$, **IFLM**$_1$ just needs to run less than 50% of the tests to be almost as effective as **IFLM**$_A$, which requires running all tests.

We also found that **IFLM**$_1$ performed worse over artificial bugs compared to real bugs. Conversely, **IFLM**$_A$ performed better over artificial bugs compared to real ones. We believe that artificial bugs could potentially exhibit a bias that favors using **IFLM**$_A$ to evaluate SBFL techniques.

6.2 Contributions

This research made the following contributions:

- We built **IFLM**, a tool that can trigger SBFL in four different modes using 25 widely-used ranking formulae.
- We applied **IFLM** to *Cvent* close-sourced software projects (27 real multi-location bugs) and five Defects4J open-sourced projects (174 real and 37 artificial multi-location bugs).
- We found that **IFLM**$_1$ performed almost as effectively as **IFLM**$_A$, while only needing to run less than half of the tests.
- We created and provided a dataset comprising 27 real bugs from four programs currently used at *Cvent*.

We hope that this study can offer valuable insights into the integration of SBFL within

existing software processes in an industry environment. **IFLM** and all experimental data are available in a Github repository at [17].

6.3 Approach

Our goal is to evaluate SBFL on multi-location bugs at different moments, e.g., after some or all tests were executed. Our Instant Fault Localization for Multi-location Bugs (**IFLM**) framework uses four "trigger modes" to represent these moments.

1. **First-Failure Triggering (IFLM$_1$)** invokes SBFL right after the first test failure. This is the minimal requirement for SBFL to work as it requires at least 1 failed test. While this mode uses minimal time and computing resources, it also collect fewer coverage (spectrum) information.

2. **Multi-Failure Triggering (IFLM$_f^k$)** initiates SBFL after every k^{th} (k=1–5) test failures. As k increases, more spectrum information was collected. However, this mode requires more time and computing resources.

3. **Failure-Pass Triggering (IFLM$_p^k$)** activates SBFL after the first test failure, and subsequently k extra passed tests (k = 1–10). Compared to **IFLM$_1$**, **IFLM$_p^k$** spends more time and resources to collect coverage data. Here we can study the trade-off between gained accuracy and time and resources required for executing more tests.

4. **Complete Execution Triggering (IFLM$_A$)** is the conventional SBFL, which ranks bug locations after executing *all* available tests. **IFLM$_A$** is thus expensive and might not be applicable in the real world, e.g., at *Cvent* with many tests.

Each triggering mode thus corresponds to a different approach for selecting a subset of tests

from the complete test set. Our goal is to collect a partial set of spectrum data that is sufficient for SBFL to function effectively. By experimenting with these different triggering modes, we explore the trade-offs between effectiveness of SBFL and its runtime cost. Note in our study that all tests were executed sequentially in a fixed order, determined by the test executor (Maven-Clover plugin).

This fixed ordering ensures deterministic results throughout our study. In addition, all triggering modes use instrumented tests. While instrumentation adds overhead, it is generally not a concern in practice, as companies (e.g., *Cvent*) often run instrumented tests to at least measure code coverage metrics, as part of code quality control procedure.

6.4 Evaluation

We use **IFLM** to investigate the two research questions: (**RQ1**) how sensitive is **IFLM** to different triggering modes? and (**RQ2**) how sensitive is **IFLM** to different SBFL formulae?

IFLM and all data in this study are available at [17].

6.4.1 Datasets

We used 238 multi-location bugs, of which 211 are from Defect4J dataset [58] and 27 are mined from four close-source software products from *Cvent*. For the Defects4j bugs, we reused the spectrum data published in [6] to save time.

Defects4J Bugs Defects4J is a widely-used dataset of real bugs in popular open-source software. Defects4J contains 835 bugs from 17 projects [58]. Out of these bugs, we found 174 multi-location bugs from 6 projects that have spectra data [6].

We also used artificial bugs from [6] in our study. These bugs were injected through mutation into the same open-source programs in Defect4J, causing the logic or semantics of the program to fail. Our objective is to compare the evaluation results between real bugs and artificial bugs. If the results were consistent for both bug sets, it would provide more confidence in using artificial bugs for evaluating SBFL techniques when real bugs are limited. Noting that while the [6] database contains numerous injected bugs, most of them are single-location bugs. We were able to identify only 37 multi-location bugs, all of which were from the Common-Lang project and used in our study.

Table 6.1: The Defect4J dataset: 174 real bugs and 37 artificial bugs from 5 open-source projects.

Project	**# of Bugs**		**LOC Exec**	**# Tests**	**1^{st} Failed Idx**
	R	I			
Chart	13	0	25–7,057	1–428	1–248
Math	73		68–7,036	3–1,513	1–599
Mockito	26		931–4,252	25–1,111	9–273
Time	21		931–4,252	25–1,111	9–273
Lang	41	37	189–2,817	7–198	1–149
Total	**174**	**37**			

Tab. 6.1 describes our Defects4J dataset. The first column shows the **project** names. The next column shows the **# of Bugs** (R: real bugs, I: inserted/artificial bugs).

Column **LOC Executed** shows the code sizes in terms of lines of code (LOC) among buggy program versions that are covered by test cases. Column **# Tests** shows the number of tests executed for each buggy program. The last column, **1^{st} Failed Idx**, gives the index of the first failed test among the tests. Note that values in these columns are given in a range (e.g., 1–428) because each program has multiple snapshots, each of which corresponds to an individual bug.

Tab. 6.2 provides additional information for failed tests. Approximately 50% of the bugs

Table 6.2: Distribution of number of failed tests in Defect4J.

# of Failed Tests	Real Bugs		Artificial Bugs	
	#	%	#	%
1	101	58 (%)	18	49 (%)
2	41	24 (%)	7	19 (%)
3	5	3 (%)	11	30 (%)
4	8	5 (%)	0	0 (%)
≥ 5	19	11 (%)	1	3 (%)
Total	174	100 (%)	37	100 (%)

have more than 1 failing test. Among these bugs, the majority of them have 2–3 failing tests. This justifies why in the $\mathbf{IFLM}_f^k$, we set k's upper bound to 5 as bugs having more than 5 failed tests are rare.

***Cvent* Bug Dataset** This dataset consists of 27 real multi-location bugs in 4 close-source programs from *Cvent*. These bugs were identified and fixed by developers within *Cvent*'s internal CI/CD process. We collected the bugs by analyzing build logs, Git commits, Jira tickets, and test reports. The ground truths for these bugs were determined based on the fixes applied by developers. In cases where it was difficult to distinguish between code modifications made for bug fixing purposes and those made for refactoring, we excluded those fixes to ensure the accuracy of the dataset.

The four subject programs are written in Java and have a medium size, ranging from 10-20 kLOC. Among these programs, one is an internal tool software developed for *Cvent*'s own developers, while the other three are related to *Cvent*'s business domains, specifically event management, account provisioning, and planners' tools, used by *Cvent*'s external clients. The test sizes for these programs range from a few hundred to one thousand test cases, consisting of both unit tests and integration tests. Due to legal constraints, we cannot publish all the details of the dataset. However, we have made the spectrum data available at [17].

6.4.2 RQ1: Comparing IFLM's Triggering Modes

We evaluate the effectiveness of different triggering modes for SBFL using **Dice** (see §3.1 for its definition) as the default ranking formula for **IFLM**. This is because Dice generally outperformed other formulae. Tab. 6.3 gives an overview of the performance of each formula averaged across all bug and triggering mode combinations. The ranking formulae are categorized into different groups based on their performance: best (green), second (dark-gray), and the least effective (orange).

Effectiveness of IFLM$_1$ and IFLM$_A$

Tab. 6.4 shows the results from **IFLM**$_1$ and **IFLM**$_A$ on the 174 real bugs and 37 artificial bugs from open-source projects in Defect4J dataset, and 27 real bugs from *Cvent*'s close-source dataset. The five columns present the performance metrics and test execution cost for real (R) and injected/artificial (I) bugs.

Contrary to the findings for single-location bugs, **IFLM**$_A$ outperformed **IFLM**$_1$ on both real and artificial bugs. For the real open-source bugs, **IFLM**$_A$ performed slightly better than **IFLM**$_1$ with metrics such as 12% versus 10% for Top-1, 30% versus 28% for Top-5, 17% versus 13% for MAP, and 22% versus 20% for MRR. Similarly, we see the comparable performance of the two in the close-source *Cvent* bugs. However, **IFLM**$_A$ required more than twice test executions compared to **IFLM**$_1$. Similarly, for *Cvent* bugs, in average, **IFLM**$_1$ takes only 327 seconds, while **IFLM**$_A$ consumes 885 seconds. In other words, empirically, **IFLM**$_1$ runs more than twice as fast as **IFLM**$_A$, as a result of running far more fewer tests.

For artificial bugs, **IFLM**$_A$ demonstrated significant improvement compared to real bugs, achieving over 100% better performance across all metrics: 27% versus 12% for Top-1, 62% versus 30% for Top-5, 40% versus 17% for MAP, and 40% versus 22% for MRR. In contrast,

Table 6.3: Effectiveness of 25 ranking algorithms

Real Bugs						Artificial Bugs					
Ranking Algorithms	# Instances	Top-1 (%)	Top-5 (%)	MAP (%)	MRR (%)	Ranking Algorithms	# Instances	Top-1 (%)	Top-5 (%)	MAP (%)	MRR (%)
DICE [32]	2040	11	30	15	21	DICE [32]	466	12	35	23	23
KULCZYNSKI1 [76]	2040	11	30	15	21	KULCZYNSKI1 KULCZYNSKI1 [76]	466	12	35	23	23
SORENSENDICE [22]	2040	11	30	15	21	SORENSENDICE [22]	466	12	35	23	23
GOODMAN [46]	2040	11	30	15	21	GOODMAN [46]	466	12	35	23	23
ANDERBERG [22]	2040	11	30	15	21	ANDERBERG [22]	466	12	35	23	23
JACCARD [18]	2040	11	30	15	21	JACCARD [18]	466	12	35	23	23
M2 [114]	2040	10	28	14	19	M2 [114]	466	12	35	23	23
RUSSELLRAO [32]	2040	10	28	14	19	RUSSELLRAO [32]	466	12	35	23	23
AMPLE [32]	2040	8	25	12	17	AMPLE [32]	466	12	35	23	23
WONG3 [38]	2040	6	15	7	10	WONG3 [38]	466	12	27	20	20
WONG2 [104]	2040	5	15	7	10	WONG2 [104]	466	12	27	20	20
SIMPLEMATCHING	2040	5	14	6	9	SIMPLEMATCHING	466	10	22	17	17
HAMANN [32]	2040	5	14	6	9	HAMANN [32]	466	10	22	17	17
SOKAL [32]	2040	5	14	6	9	SOKAL [32]	466	10	22	17	17
ROGERSTANIMOTO [32]	2040	5	14	6	9	ROGERSTANIMOTO [32]	466	10	22	17	17
M1 [114]	2040	5	14	6	9	M1 [114]	466	10	22	17	17
EUCLID [32]	2040	0	5	3	3	EUCLID [32]	466	0	0	2	2
WONG1 [104]	2040	0	5	3	3	WONG1 [104]	466	0	0	2	2
HAMMING [32]	2040	0	5	3	3	HAMMING [32]	466	0	0	2	2
OCHIAI2 [32]	2040	0	1	1	1	OCHIAI2 [32]	466	0	0	1	1
OCHIAI [18]	2040	0	1	1	1	OCHIAI [18]	466	0	0	1	1
TARANTULA [18]	2040	0	1	1	1	TARANTULA [18]	466	0	0	1	1
KULCZYNSKI2 [32]	2040	0	1	1	1	KULCZYNSKI2 [32]	466	0	0	1	1
ZOLTAR [23]	2040	0	1	1	1	ZOLTAR [23]	466	0	0	1	1
OVERLAP [35]	2040	0	0	0	0	OVERLAP [35]	466	0	0	0	0

Table 6.4: Comparing $\textbf{IFLM}_1$ and $\textbf{IFLM}_A$ (R_O/I_O = Real/Artificial Defects4J bugs, R_C = Real *Cvent* bugs).

Mode	Top-1 (%)			Top-5 (%)			MAP (%)			MRR (%)			Cost (% Tests or R.Time)		
	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C (s)
IFLM_1	10	3	22	28	8	**66**	13	9	29	20	9	45	**44**	**39**	327
IFLM_A	**12**	**27**	**24**	**30**	**62**	65	**17**	**40**	**37**	**22**	**40**	**49**	100	100	885

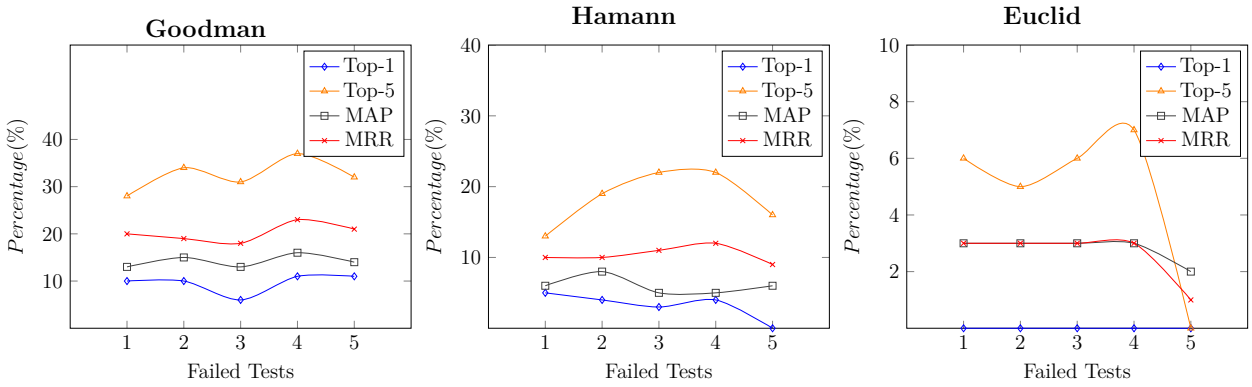
$\textbf{IFLM}_1$ performed much worse for artificial bugs compared to real bugs: 3% versus 10% for Top-1, 8% versus 28% for Top-5, 9% versus 13% for MAP, and 9% versus 20% for MRR.

Overall, for artificial bugs, $\textbf{IFLM}_1$ performed significantly worse than $\textbf{IFLM}_A$ with metrics such as 3% versus 27% for Top-1, 8% versus 62% for Top-5, and 9% versus 40% for both MAP and MRR. These results highlight the differences in performance between $\textbf{IFLM}_1$ and $\textbf{IFLM}_A$ on artificial bugs compared to real bugs. Additionally, artificial bugs appear biased towards $\textbf{IFLM}_A$ and become challenging for $\textbf{IFLM}_1$. Thus, we do not advise using artificial bugs to evaluate SBFL on multi-location bugs.

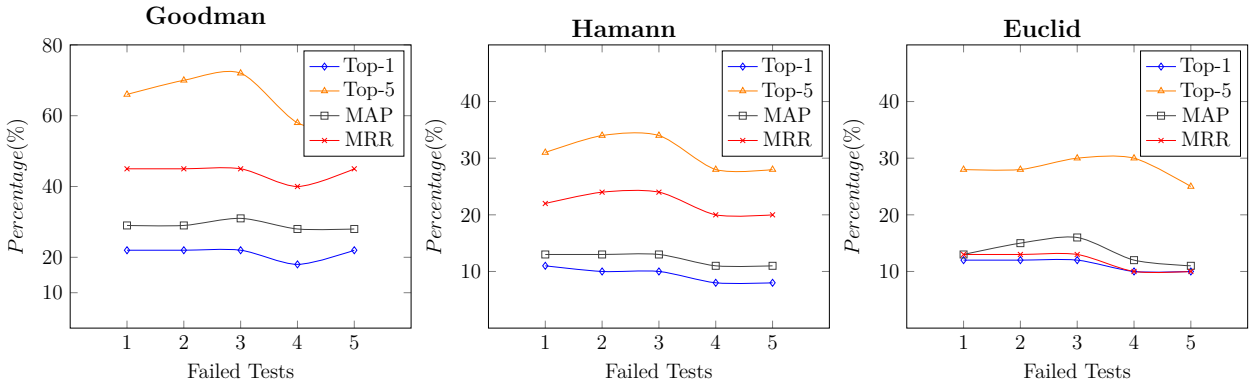
Finding 6.1: For real bugs, $\textbf{IFLM}_1$ performs slightly worse than $\textbf{IFLM}_A$ ($\approx$ 2 percentage point difference, e.g., Top-1, Top-5, MRR), but it offers a significant advantage in test execution reduction ($\approx$ 100% better than $\textbf{IFLM}_A$, in terms of runtime and the number of executed tests). However, for artificial bugs, $\textbf{IFLM}_A$ significantly outperforms $\textbf{IFLM}_1$, suggesting that artificial bugs might not be suitable for evaluating SBFL for multi-location bugs.

Effectiveness of IFLM_f^k

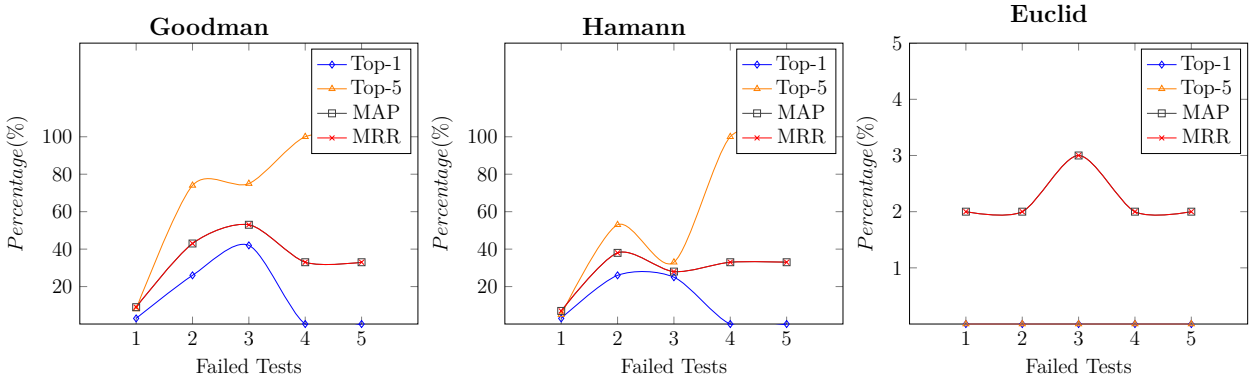
We evaluated $\textbf{IFLM}_f^k$ by triggering SBFL at every k^{th} occurrence of a test failure, where k= 1–5. Tab. 6.5 shows the average measurements. As can be seen, the effectiveness mea-



(a) Real Open-Source Bugs.



(b) Real Close-Source Bugs.



(c) Artificial Open-Source Bugs.

Figure 6.1: Effectiveness of $\mathbf{IFLM}_f^k$ using different SBFL formulae: Goodman, Hamann, Euclid.

Table 6.5: Effectiveness of $\textbf{IFLM}_f^k$ with k=1–5 (R_O/I_O = Real/Artificial Defects4J bugs, R_C = Real *Cvent* bugs).

# Failed Tests	Top-1 (%)			Top-5 (%)			MAP (%)			MRR (%)			Cost (% Tests or R.Time)		
	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C (s)
1	10	3	22	28	8	66	13	9	29	20	9	45	44	39	327
2	10	26	23	34	74	70	15	43	32	19	43	47	54	48	415
3	6	42	20	31	75	65	13	53	29	18	53	43	55	62	451
4	11	0	19	37	100	65	16	33	28	23	33	43	57	83	539
5	11	0	18	32	100	62	14	33	26	21	33	41	57	93	574

surements did not consistently improve with an increasing number of failed tests, i.e., we do not need too many failed tests for SBFL to work. For instance, for real bugs, the Top-1 measurement remained at 10% for both $k = 1$ and $k = 2$, but decreased to 6% at $k = 3$. The maximum value of Top-1, 11%, was achieved at $k = 4, 5$, which only slightly differed from the value at $k = 1$ (10%). Increasing the Top-1 measurement by 1 percent point required running 44% of tests with $k = 1$ and 57% with $k = 5$ (by 13 percent points). We observed similar results for artificial bugs.

For generality, we perform the same experiment for $\textbf{IFLM}_f^k$ using 3 ranking formulae randomly selected in 3 corresponding representative groups in Tab. 6.3, namely **Goodman**, **Hamann**, and **Euclid** (§3.1). Fig. 6.1 (a)(b) shows the results for real bugs for Defects4J and *Cvent*, respectively. In Goodman and Hamann, Top-1, Top-5, MAP, and MRR decreased when k go from 1 to 3. While in Euclid, Top-1 stayed constant against k, for all other metrics, their values decreased significantly when k reached 5.

Fig. 6.1 (c) shows the results for artificial bugs. For Goodman and Hamann, Top-1, MAP, and MRR reached optimal values when k was between 2 and 3, then decreased when k was between 3 and 5. Only Top-5 achieved optimal at $k = 5$. However, for Euclid, all metrics got optimal values at $k = 3$, and then decreased when k approached 5.

Table 6.6: Effectiveness of IFLM_p^k when k = 1–10 (R_O/I_O = Real/Artificial Defects4J bugs, R_C = Real *Cvent* bugs).

Additional Passed Tests	Top-1 (%)			Top-5 (%)			MAP (%)			MRR (%)			Cost (% Tests or R.Time)		
	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C	R_O	I_O	R_C (s)
1	11	3	22	28	14	66	14	12	29	20	12	45	46	41	327
2	12	3	22	29	14	66	14	12	29	21	12	45	47	44	329
3	11	3	22	31	22	66	14	13	29	21	13	45	48	46	330
4	12	3	22	30	22	66	15	13	29	21	13	45	48	46	331
5	11	3	22	29	31	61	15	16	26	20	16	45	48	48	333
6	10	11	22	29	39	61	15	23	26	20	23	41	48	50	335
7	11	17	22	31	47	61	16	29	26	21	29	41	49	52	336
8	11	17	20	32	42	60	16	28	26	21	28	41	50	55	339
9	10	17	21	32	42	60	16	29	26	20	29	41	49	57	340
10	11	19	21	34	44	60	17	31	26	22	31	41	49	54	341

Finding 6.2: *While increasing the number failed tests costs more to execute and collecting profiling data,* IFLM_f^k *does not work better with more failing tests.*

Effectiveness of IFLM_p^k

Tab. 6.6 shows our results for IFLM_p^k, which applies SBFL at every occurrence of additional k^{th} passed tests after the first failed test (k = 1–10). For real bugs, IFLM_p^k performed slightly better with more additional passed tests. However, the increased performance was insignificant. As shown in Tab. 6.6, Top-1 stayed relatively consistent around 11%; all other metrics slightly increased: 28–34% in Top-5, 14–17% in MAP, and 20–22% in MRR, while the cost of running tests increased from 46–49%.

Finding 6.3: *Given the execution data of extra passed tests after the initial test failure,* IFLM_p^k *did not outperform* IFLM_1 *for real bugs.*

For artificial bugs, performance gained were significant. When k increases from 1 to 10,

Table 6.7: The effectiveness of **IFLM$_1$** and **IFLM$_A$** using all 25 different formulae on Defects4J's real bugs.

Formulae	Real Bugs											
	Top-1 (%)			Top-5 (%)			MAP (%)			MRR (%)		
	IFLM$_1$	IFLM$_A$	Diff	IFLM$_1$	IFLM$_A$	Diff	IFLM$_1$	IFLM$_A$	Diff	IFLM$_1$	IFLM$_A$	Diff
Ample	**10**	8	**+2**	**25**	**25**	**0**	**12**	**12**	**0**	**18**	16	**+2**
Anderberg	10	**12**	-2	28	**30**	-2	13	**17**	-4	20	**22**	-2
Dice	10	**12**	-2	28	**30**	-2	13	**17**	-4	20	**22**	-2
Euclid	**0**	**0**	**0**	**6**	**6**	**0**	**3**	**3**	**0**	**3**	**3**	**0**
Goodman	10	**12**	-2	28	**30**	-2	13	**17**	-4	20	**22**	-2
Hamann	**5**	4	**+1**	**13**	10	**+3**	**6**	**6**	**0**	**10**	7	**+3**
Hamming	**0**	**0**	**0**	**6**	**6**	**0**	**3**	**3**	**0**	**3**	**3**	**0**
Jaccard	10	**12**	-2	28	**30**	-2	13	**17**	-4	20	**22**	-2
Kulczynski1	10	**12**	-2	28	**30**	-2	13	**17**	-4	20	**22**	-2
Kulczynski2	**0**	**0**	**0**	1	**2**	-1	1	**2**	-1	1	**2**	-1
M1	**5**	4	**+1**	**13**	10	**+3**	**6**	**6**	**0**	**10**	7	**+3**
M2	**10**	**10**	**0**	**28**	**28**	**0**	13	**15**	-2	**20**	**20**	**0**
Ochiai	**0**	**0**	**0**	1	**2**	-1	1	**2**	-1	1	**2**	-1
Ochiai2	**0**	**0**	**0**	1	**2**	-1	1	**2**	-1	1	**2**	-1
Overlap	**0**	**0**	**0**	**0**	**0**	**0**	**1**	0	**+1**	**0**	**0**	**0**
RogersTanimoto	**5**	4	**+1**	**13**	10	**+3**	**6**	**6**	**0**	**10**	7	**+3**
RussellRao	**10**	**10**	**0**	**28**	26	**+2**	13	**14**	-1	**20**	19	**+1**
SimpleMatching	**5**	4	**+1**	**13**	10	**+3**	**6**	**6**	**0**	**10**	7	**+3**
Sokal	**5**	4	**+1**	**13**	10	**+3**	**6**	**6**	**0**	**10**	7	**+3**
SørensenDice	10	**12**	-2	28	**30**	-2	13	**17**	-4	20	**22**	-2
Tarantula	**0**	**0**	**0**	1	**2**	-1	1	**2**	-1	1	**2**	-1
Wong1	**0**	**0**	**0**	**6**	**6**	**0**	**3**	**3**	**0**	**3**	**3**	**0**
Wong2	5	**6**	-1	**13**	11	**+2**	6	**7**	-1	**10**	9	**+1**
Wong3	5	**6**	-1	**13**	**13**	**0**	6	**8**	-2	**10**	**10**	**0**
Zoltar	**0**	**0**	**0**	1	**2**	-1	1	**2**	-1	1	**2**	-1
Average (%)	**5**	**5**	**0**	**15**	14	**1**	7	**8**	-1	**10**	**10**	**0**

performances increased 3% to 19% in Top-1, 14% to 44% in Top-5, 12% to 31% in MAP and MRR. There was discrepancy between the results of real and artificial bugs, suggesting that artificial bugs might not be a reliable benchmark for evaluating SBFL.

6.4.3 RQ2: IFLM's Sensitivity to SBFL Formulae

We ran **IFLM** using 25 popular SBFL formulae shown in Tab. 3.1 to ensure the generalizability of our observation comparing **IFLM$_1$** and **IFLM$_A$** and explore **IFLM**'s sensitivity to different SBFL formulae.

Table 6.8: The effectiveness of $\mathbf{IFLM}_1$ and $\mathbf{IFLM}_A$ using all 25 different formulae on Defects4J's artificial bugs.

Formulae	**Artificial Bugs**											
	Top-1 (%)			**Top-5 (%)**			**MAP (%)**			**MRR (%)**		
	IFLM_1	IFLM_A	Diff	IFLM_1	IFLM_A	Diff	IFLM_1	IFLM_A	Diff	IFLM_1	IFLM_A	Diff
Ample	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
Anderberg	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
Dice	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
Euclid	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**2**	**0**	**2**	**2**	**0**
Goodman	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
Hamann	3	**14**	-11	5	**27**	-22	7	**20**	-13	7	**20**	-13
Hamming	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**2**	**0**	**2**	**2**	**0**
Jaccard	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
Kulczynski1	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
Kulczynski2	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
M1	3	**14**	-11	5	**27**	-22	7	**20**	-13	7	**20**	-13
M2	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
Ochiai	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
Ochiai2	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
Overlap	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
RogersTanimoto	3	**14**	-11	5	**27**	-22	7	**20**	-13	7	**20**	-13
RussellRao	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
SimpleMatching	3	**14**	-11	5	**27**	-22	7	**20**	-13	7	**20**	-13
Sokal	3	**14**	-11	5	**27**	-22	7	**20**	-13	7	**20**	-13
SϕrensenDice	3	**27**	-24	8	**62**	-54	9	**40**	-31	9	**40**	-31
Tarantula	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
Wong1	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**2**	**0**	**2**	**2**	**0**
Wong2	3	**27**	-24	5	**51**	-46	7	**37**	-30	7	**37**	-30
Wong3	3	**27**	-24	5	**51**	-46	7	**37**	-30	7	**37**	30
Zoltar	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
Average (%)	2	**15**	-13	4	**32**	-28	6	**22**	-16	6	**22**	-16

Table 6.9: The effectiveness of **IFLM$_1$** and **IFLM$_A$** using all 25 different formulae on *Cvent* bugs.

Formulae	**Top-1** (%)			**Top-5** (%)			**MAP** (%)			**MRR** (%)		
	IFLM$_1$	IFLM$_A$	Diff	IFLM$_1$	IFLM$_A$	Diff	IFLM$_1$	IFLM$_A$	Diff	IFLM$_1$	IFLM$_A$	Diff
Ample	**22**	16	**+6**	**59**	54	**+5**	**27**	26	**+1**	**40**	36	**+4**
Anderberg	22	**24**	-2	**66**	65	**+1**	29	**37**	-8	45	**49**	-4
Dice	22	**24**	-2	**66**	65	**+1**	29	**37**	-8	45	**49**	-4
Euclid	**12**	0	**+12**	**28**	13	**+15**	**13**	7	**+6**	**13**	7	**+6**
Goodman	22	**24**	-2	**66**	65	**+1**	29	**37**	-8	45	**49**	-4
Hamann	**11**	8	**+3**	**31**	22	**+9**	**13**	**13**	**0**	**22**	16	**+6**
Hamming	**0**	**0**	**0**	**14**	13	**+1**	**7**	**7**	**0**	**7**	**7**	**0**
Jaccard	22	**24**	-2	**66**	65	**+1**	29	**37**	-8	45	**49**	-4
Kulczynski1	22	**24**	-2	**66**	65	**+1**	29	**37**	-8	45	**49**	-4
Kulczynski2	**0**	**0**	**0**	2	**4**	-2	2	**4**	-2	2	**4**	-2
M1	**11**	8	**+3**	**31**	22	**+9**	**13**	**13**	**0**	**22**	16	**+6**
M2	**22**	20	**+2**	**66**	61	**+5**	29	**33**	-4	**45**	**45**	**0**
Ochiai	**11**	0	**+11**	**30**	4	**+26**	**12**	4	**+8**	**13**	4	**+9**
Ochiai2	**0**	**0**	**0**	2	**4**	-2	2	**4**	-2	2	**4**	-2
Overlap	**0**	**0**	**0**	**0**	**0**	**0**	**2**	0	**+2**	**0**	**0**	**0**
RogersTanimoto	**11**	8	**+3**	**31**	22	**+9**	**13**	**13**	**0**	**22**	16	**+6**
RussellRao	**22**	20	**+2**	**66**	56	**+10**	29	**30**	-1	**45**	42	**+3**
SimpleMatching	**11**	8	**+3**	**31**	22	**+9**	**13**	**13**	**0**	**22**	16	**+6**
Sokal	**11**	8	**+3**	**31**	22	**+9**	**13**	**13**	**0**	**22**	16	**+6**
SϕrensenDice	22	**24**	-2	**66**	65	**+1**	29	**37**	-8	45	**49**	-4
Tarantula	**11**	7	**+4**	**30**	27	**+3**	**13**	**13**	**0**	21	**22**	-1
Wong1	**0**	**0**	**0**	**14**	13	**+1**	**7**	**7**	**0**	**7**	**7**	**0**
Wong2	11	**12**	-1	**31**	24	**+7**	13	**15**	-2	**22**	20	**+2**
Wong3	11	**12**	-1	**31**	28	**+3**	13	**17**	-4	**22**	**22**	**0**
Zoltar	**0**	**0**	**0**	2	**4**	-2	2	**4**	-2	2	**4**	-2
Average (%)	**13**	11	**+2**	**37**	32	**+5**	17	**18**	-1	**25**	24	**+1**

The results of $\textbf{IFLM}_1$ and $\textbf{IFLM}_A$ are given in Tab. 6.7, and Tab. 6.8 for the Defects4J's real, artificial bugs, respectively; and Tab. 6.9 for *Cvent* bugs. The **Diff** column shows the differences of measured values between $\textbf{IFLM}_1$ and $\textbf{IFLM}_A$ (larger or equal values are in bold). A positive or zero diff value (in **bold**) indicates that $\textbf{IFLM}_1$ performed as well as or better than $\textbf{IFLM}_A$.

For the real bugs in Defects4J, out of the total 25 formulae, $\textbf{IFLM}_1$ outperformed or achieved comparable results to $\textbf{IFLM}_A$ in Top-1, Top-5, MAP, and MRR in 17, 14, 10, and 14 formulae, respectively (Tab. 6.7). We observe a similar trend in the *Cvent* dataset, i.e., 17 in Top-1, 22 in Top-5, 12 in MAP, and 15 in MRR (Tab. 6.9). On average, there was minimal distinction between $\textbf{IFLM}_1$ and $\textbf{IFLM}_A$ across all four performance metrics. Thus, for real bugs, the observation that $\textbf{IFLM}_1$ performed at least as well as $\textbf{IFLM}_A$ (in all four effectiveness metrics) was confirmed in more than 50% of the 25 investigated SBFL formulae.

Finding 6.4: *For real bugs, in general,* $\textbf{IFLM}_1$ *performed equally or better than* $\textbf{IFLM}_A$ *on more than half of the 25 investigated SBFL formulae, across all of the four effectiveness metrics.*

However, the observation for the Defects4J artificial bugs contradicted that of the real bug dataset. $\textbf{IFLM}_A$ outperformed $\textbf{IFLM}_1$ for 16/25 formulae. The remaining 9 formulae showed similar performance levels for both $\textbf{IFLM}_1$ and $\textbf{IFLM}_A$. This substantial difference between the real and artificial bug datasets suggests that artificial bugs may not accurately predict the performance of SBFL techniques in localizing real bugs.

For sensitivity of **IFLM** to SBFL formulae, Tab. 6.7 & 6.8 showed that the choice of formulae played a crucial role in accurately localizing both real and artificial bugs. Formulae such as Ample, Dice, and Jaccard contributed to achieving 28% and 30% Top-5, as well

as 20% and 22% MRR for **IFLM**$_1$ and **IFLM**$_A$ respectively. Conversely, formulae such as Overlap, Kulczynski2, and Zoltar performed poorly and exhibited inaccuracy. Furthermore, there was no single formula that consistently outperformed others. Instead, multiple top-performing formulae demonstrated similar levels of performance for both **IFLM**$_1$ and **IFLM**$_A$.

Finding 6.5: *SBFL formulae can have significant influence to the performance of* **IFLM**. *There were often multiple formulae that worked equally-well for* **IFLM**.

6.5 Threats to Validity

External Validity Our findings depend on the quality and characteristics of the bug datasets used in our experiments. However, our benchmark, Defects4J, a well-known dataset with real Java bugs, and the four current *Cvent* projects, can help mitigate this threat.

Construct Validity We focused on multi-location bugs, whose ground truths were constructed by comparing the buggy and fixed versions of the programs. Locations (except comment) that were modified (i.e., add, delete, change) were considered locations of the bug. In reality, developers often mixed between bug fixing and refactoring in a commit, and it is not trivial to distinct the two, especially in *Cvent* dataset. However, the first-author who constructed the dataset is familiar with the selected programs and thus help address this concern.

Internal Validity Similar to prior fault localization research [67, 106, 122], given a bug fix, we treated the location where the fixing change was applied as ground truth. However, in reality, the place where a bug is fixed is not always the place where a bug is found.

Another concern was the reliability of the coverage data. This data collection process was time-consuming and hard to validate.

For the open-source Defects4J dataset, we reused the data published in [6]. For the close-source *Cvent* dataset, we modified Clover [5] and ran tests to gather the profiling data. For accuracy, we repeated the collection process three times. We publish the close-source dataset in standardized spectrum format and the tool we built on top of Clover in [17].

6.6 Summary

This paper explored opportunities of reducing overhead cost of running tests in SBFL while maintaining its accuracy. We experiment with the concept of triggering modes proposed in [35] but focus on multi-location bugs, which are common in real-world settings, e.g., at our company *Cvent*. While there were minor disagreements with the single-location study in [35], our work generally confirms that it is not always necessary to execute all test cases before using SBFL formulae to locate bugs. The results in this study are useful for *Cvent* and hopefully other industrial companies who seek to adopt **IFLM** into their CI/CD development pipeline to automate and speedup software debugging. In the future, we plan to conduct a user study to empirically measure productivity (e.g., developer's debugging time reduction) would be actually gained with the integration between **IFLM** and *Cvent*'s CI/CD pipeline.

Chapter 7

Related work

In this section, we will discuss related work in spectrum-based fault localization, IR-based fault localization, and empirical studies on fault localization techniques, and testing optimization.

7.1 Spectrum-Based Fault Localization (SBFL)

SBFL techniques identify bug locations using the execution information of buggy code [19, 33, 55, 77, 106, 109]. For instance, given a buggy program and test cases, Tarantula instruments code to collect the execution coverage of passed and failed tests, counts the number of passed/failed tests covering each program element (i.e., class, method, or statement), and computes suspiciousness scores [57]. Xuan et al. used machine learning to train a model by combining 25 suspiciousness calculation formulae [106]. Although they did not observe any single formula to work universally better than the others, the trained model outperforms the state-of-the-art formulae, such as Tarantula, Ochiai, and Ample.

The SBFL approaches mentioned above only provide a static ranked list after the execution of all tests. Some researchers further improved SBFL approaches by taking in developers' feedback on the initial ranked list to dynamically tune ranking accordingly [45, 49, 67]. In particular, Li et al. leveraged SBFL to rank suspicious methods, and then generated high-level queries to ask developers about the correctness of specific executions for the most

suspicious methods [67]. If developers determine that a method's execution is correct, the approach labels the execution tree rooted at this method invocation node as "correct" instead of "buggy", and performs suspiciousness recalculation accordingly.

Our research does not define any new SBFL formula. Instead, we reused 25 existing SBFL formulas. We built a framework—**IFLS**—to systematically investigate diverse triggering modes of SBFL, and to understand how each triggering mode balances the effectiveness and efficiency of fault localization. Our exploration compared **IFLS**'s effectiveness given (1) different SBFL formulas and (2) distinct triggering mechanisms for SBFL formulas. By revealing bug locations early, $\textbf{IFLS}_1$ turned out to be the most effective and efficient way to help developers fix bugs.

Tarantula [57] was the first SBFL technique that identifies buggy locations by leveraging the execution information or code coverage profiling data gathered by running tests against a program under investigation. Since then there have been many other variations of the SBFL approach, such as, Ochia, Jaccard, Dice [19, 33, 55, 57, 77, 106, 109]. The main difference among these techniques is how a program spectra are formulated into a metric called suspiciousness score (i.e., SBFL formula) to predict how buggy each location of the program is. Lucia et al. [68] and Yoo et al. [111] compared different formulae defined for SBFL approaches, and concluded that there was no optimal formula that always worked better than others.

This paper does not define any new SBFL formula but instead reuses 25 existing SBFL ones. We built the —**IFLS** framework to investigate diverse triggering modes of SBFL, and to understand how each triggering mode balances the effectiveness and efficiency of bug localization. Our exploration compared **IFLS**'s effectiveness given (i) different SBFL formulae and (ii) distinct triggering mechanisms for SBFL formulae. Our ultimate goal was to find optimal triggering modes that worked best with SBFL in industrial settings.

7.2 Information Retrieval-Based Fault Localization (IRFL)

IRFL approaches locate bugs based on bug reports [65, 82, 108, 121]. For example, BLUiR treats a bug report as a document query and considers source code as documents [82]. Given a bug report, BLUiR searches for program entities that are relevant to the report, and ranks those entities as candidate bug locations. To better retrieve and rank documents, BLUiR assigns more weights to bug report titles, and to any class or method name referred to by a report. Learning-to-rank integrates domain knowledge of bug history and API specification to train a model for bug location prediction [108].

One limitation of IRFL tools is the implicit assumption that a bug report has certain document relevance with the buggy code. However, such assumption does not always hold. To overcome the limitation, some researchers proposed hybrid approaches that combine IRFL with other approaches [34, 122]. For instance, Dao et al. combined IRFL with SBFL by assigning different weights to the separately generated ranked lists [34]. Zou et al. combined IRFL with another six kinds of techniques: SBFL, mutation-based fault localization, dynamic program slicing [24], stack trace analysis [102], predicate switching [118], and history-based fault localization [60]. The combination is achieved via machine learning so that the results by distinct techniques are given appropriate weights.

Compared with the above-mentioned IR-based approaches, **IFLS** does not rely on the existence of any bug report, neither does it require for the execution of all test cases.

7.3 Empirical Studies on Fault Localization Techniques

Researchers empirically studied fault localization techniques in various ways [62, 68, 99, 103, 111]. Specifically, Lucia et al. [68] and Yoo et al. [111] compared different formulae defined

for SBFL approaches, and concluded that there was no optimal formula that always worked better than others. Kochhar et al. [62] and Dao et al. [34] independently manually inspected bug reports whose bugs were either fully, partially, or not localized by IRFL approaches. They found that the quality of bug reports can substantially impact IRFL results. If bug reports explicitly contain buggy file names, IRFL techniques are more likely to identify the bugs. Additionally, Wang et al. conducted user studies with developers to examine how developers perceived the usefulness of IRFL tools [99]. The study revealed that developers did not find such tools to be quite useful and were unsatisfied by IRFL techniques.

In our evaluation, we constructed two data sets of bugs and leveraged the known bug locations as ground truth. By comparing the ranked list by any **IFLS** variant against the ground truth, we determined the approaches' effectiveness. In the future, we will also conduct a user study with developers to learn about their opinions on **IFLS**, and design better fault localization approaches accordingly.

7.4 Test Optimization and Generation

Some approaches were proposed to reduce, prioritize, or generate test cases in order to facilitate fault localization [26, 75, 110, 112]. For instance, Masri et al. introduced *coincidental correctness* to describe the scenarios where buggy statements are executed but the execution does not lead to a test failure [75]. The researchers proposed a technique to identify all coincidentally correct tests in a given test suite, and to remove these tests in order to improve the effectiveness of SBFL approaches. Yu et al. investigated how test-suite reduction strategies influence the effectiveness of fault localization techniques [112]. When reducing the number of test cases that cover the same statement, the researchers observed existing SBFL techniques to usually work worse. Thus, they proposed a new test-suite reduction

strategy that reduces the number of test cases covering the same statement set but causes negligible impacts on fault localization.

Yoo et al. proposed FLINT, an information theoretical approach to prioritize statements and test cases [110]. In particular, statements are ordered by suspiciousness, while test cases are ordered by the degree to which they reduce the entropy inherent in fault localization. Artzi et al. developed a test generation approach to maximize the effectiveness of SBFL [26]. Specifically, they defined a *similarity criterion*, which is used to measure how similar the execution characteristics associated with two tests are, and is used to direct concolic execution towards generating tests whose execution characteristics are similar to those of a given failed test.

Our research shares the same motivation with all prior work, which is to explore ways to improve fault localization. However, we did not propose any new approach to selectively reduce, prioritize, or generate test cases. Instead, we conducted an empirical study to compare different SBFL triggering modes, and revealed that triggering SBFL right after the initial test failure is the most effective and efficient mode. Our research complements prior work. It can be used together with existing approaches of test reduction, prioritization, or generation. In the future, we will also explore how distinct triggering modes work with existing approaches to influence the effectiveness of fault localization.

These topics not only improve fault localization accuracy but also reduce its overhead costs. Several approaches have been proposed to facilitate fault localization through test case reduction, prioritization, and generation [26, 31, 42, 75, 110, 112, 120]. For instance, Masri et al. introduced "coincidental correctness" to describe scenarios where buggy statements are executed but do not result in test failures. They proposed a technique to identify and remove these coincidentally correct tests from a given test suite, aiming to improve SBFL approaches [75]. Yu et al. investigated the influence of test suite reduction strategies on the

effectiveness of fault localization techniques. They observed that existing SBFL techniques tend to perform worse when the number of test cases covering the same statement is reduced. They then proposed a new test suite reduction strategy that minimally impacts fault localization while reducing test case redundancy [112].

Yoo et al. presented FLINT, an information-theoretic approach that prioritizes statements and test cases. Statements are ordered based on their suspiciousness, while test cases are ordered by their ability to reduce the inherent entropy in fault localization [110]. Artzi et al. developed a test generation approach that aims to maximize the effectiveness of SBFL. They defined a "similarity criterion" to measure the similarity in execution characteristics between two tests. This criterion guides concolic execution to generate tests with similar execution characteristics to a given failed test [26].

Our research shares a similar motivation, particularly in exploring methods to reduce the running cost of SBFL. Our proposed **IFLS** triggering modes serve as a practical technique for reducing test execution, and in the future, we plan to investigate test prioritization to further improve triggering modes in practice.

7.5 Enhanced SBFL Techniques

Unlike standard SBFL methods that use only program's coverage information and one single ranking metric, recent enhanced SBFL approaches leverage other program analysis inputs, such as, dependency and execution graphs, contextual information, types of program entities (e.g., branch, predicate) to localize bugs more accurately [27, 30, 51, 84, 97, 106]. He et al. augmented coverage information with test call graph to construct fault inducing or influencing network, which helps narrow down bug searching space [51]. Beszédes et al. used snapshots of call stack to assist SBFL to localize buggy functions [30]. Xuan et al. used

machine learning to train a model by combining 25 suspiciousness score formulae [106]. Our study is different in that it focuses on how to reduce the overhead cost of applying the existing SBFL techniques in the real-world with as minimal accuracy loss as possible.

7.6 Effectiveness and Applicability of SBFL

Many studies have highlighted the insufficiency of SBFL and its limited real-world application [44, 52, 62, 68, 84, 85, 96, 99, 103, 111]. Wang et al. conducted user studies involving developers to assess the usefulness of Information-Based Fault Localization (IBFL) tools, revealing developers' dissatisfaction with these techniques [99]. Sarhan et al., in a recent survey [84], provided reasons for the limited adoption of SBFL, including the unavailability of supported tools, high cost of collecting execution information, and inaccurate results.

These concerns regarding the overhead cost of collecting spectrum data were shared by our team at Cvent and served as motivation for our study. However, in contrast to these work that rely on older open-source datasets for evaluation, we further validated our findings using an industry-scaled dataset benchmark that is up-to-date.

7.7 Fault Localization in Cloud-based Environments

Despite there have been many approaches proposed to improve **FL** in cloud-based applications, the problem remains hard and requires more innovative ideas and research [29, 36, 39, 79, 86, 87, 94, 113]. In this section, we discuss works that are relevant and related to our research. Most researches introduced for **FL** in cloud-based applications, uses three different types of techniques in order to identify abnormalities. More conventional approaches address the issue by applying statistical models to reason about abnormal cloud operations, using

data (e.g., logs, metrics) collected over a period of time [88, 93, 123]. The second type of techniques are based on machine learning algorithms and graph-based theories to localize faults, have been emerging recently [36, 43, 74]. LOUD [74] used machine learning to dictate correct executions, just incorrect runs can be then reasoned and referenced from the trained model, and the tool can be applied to lightweight applications. Errin et al. used support vector machine (SVM) to predict system failures, such as hard disk errors [43]. UBL [36] applied unsupervised machine learning algorithms to detect abnormal behaviors in virtualized cloud systems. In terms of problem domains, most of the proposed approaches focus on individual areas, such as, infrastructure, configuration, network, security, resources [37, 93, 94, 115, 117]. Sai Zang et al. proposed an statistical analysis model, ConfDiagnoser, to detect and root causes undesired and problematic configurations in a software system [117].

Chapter 8

Future Work

Enterprise software companies typically have large number of products lines serving various customers. Each product may well contain millions of lines of code (LoC). The software is deployed and operates in a large and complex environment. To meet business demand, companies have to utilize a very complex infrastructure, including a grid of hundreds (if not thousands) of servers or virtual machines (VM), a huge array of services (microservices), and complicated ecosystems and tools for automation of build, testing, deployment, and operation. This complex infrastructure needs to be systematically managed and monitored. Many specialized tools and software, such as, Splunk [9], New Relic [8], Datadog [7], are deployed to the IT infrastructure to accomplish this task. In addition to these expensive tools, a large number of engineers are required to run and monitor the tools. In spite of these expensive investments and expenses, results are limited. Failing services, unstable VM capacities, unsuccessful deployments, under-desired load performances, etc., remain difficult to localize and identify their root causes [29, 37, 39, 39, 40, 86, 87, 113, 117, 123]. In addition, those faults (e.g., failing services, low VM memory) due to environment will further complicate localizing faults in software applications.

In this chapter, we will discuss our future research on localizing faults in a cloud-based system or application using abstract state machine approach. A fault in this context is defined as an environment related fault, such as, insufficient memory, high CPU usage, network disconnection, misconfiguration. FL techniques, such as, spectrum-based or information

retrieval, or their combinations, assume tests are executed under a system's normal or ideal condition, i.e., environment. Because under such an abnormal environment, a failing test case does not necessarily indicate there exist a bug in the system. Thus, these techniques are often not applicable if the environment is abnormal. In addition, after a system is deployed and starts operating in a live environment, profiling information in terms of test execution is no longer available. We need a different technique with different profiling data to detect and localize environment faults. The need for being able to predict and localize accurately and quickly such a fault becomes a very challenging and expensive problem for cloud-based software development and operation companies. They often rely on manual efforts by hiring a large number of engineers, for example, site reliability, network, quality engineers, who constantly and manually monitor and analyze the data to tell if there is any abnormal behavior in the operating environment.

There have been many approaches and techniques proposed by research community to alleviate the problem. However, none of them adequately and effectively address the issue, to best of our knowledge. There are two main essential and inherent challenges need to overcome. First, data capturing the state of a cloud-based environment is large, composing of a wide variety of monitoring profiling information, such as a virtual machine (VM)'s health, database, network, exceptions. Second, the number of states an environment to examine can be exponentially large. In the following sections, we describe our approach to solve the problem, what has been done, and what remains to be done.

8.1 Fault Localization with Abstract State Machine

We explore the idea of applying an abstract state machine (AST) to model the state space of a cloud-based system to narrow down the exponential number of possible states, if otherwise,

to find an abnormal state and its transition from a normal counterpart (Fig. 8.1). When a state is being transitioned from normal to abnormal, we know that a fault is detected. More importantly, by analyzing the transition, which is composed of multiple individual events, we are able to identify what is possibly the root cause of the fault with accuracy. Thus, by utilizing an abstract state machine, we can be able to not only detect and localize a possible fault, but also identify its root cause effectively.

We define informally below the main conceptual elements of our proposed solution:

- **Abstract State Machine:** a state machine that captures and models operation of a cloud-based system. A state is an abstraction of whether a system is working as it is specified and expected, or it is behaving unexpectedly. In other words, a state captures the abstract idea of if a system is operating normally, or abnormally. At any point of time, a state can be either in normal or abnormal, and can be transitioned to either of them, triggered by time, or some event. A transition that bring a state to normal state is called a *normal transition*, and one that bring a state to an abnormal one is call *abnormal transition*. The system can be terminated and therefore transitioned to a

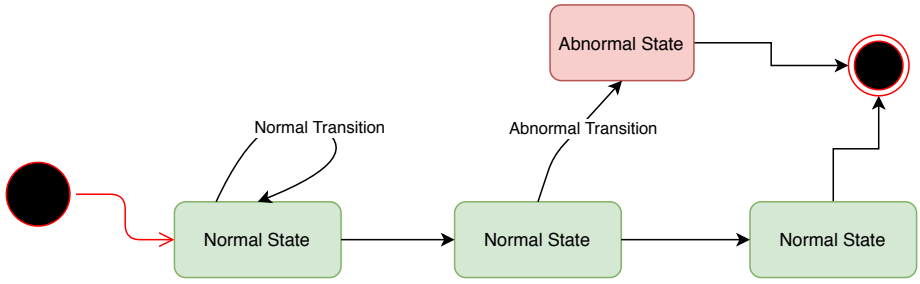


Figure 8.1: Abstract State Machine to Localizing Faults in A Cloud-based System

stop state, from either abnormal or normal state. Often, we want to see the system at a live normal state, however, when it is in an abnormal state, and cannot be recovered, it will eventually move to a *stop state* (Fig. 8.1).

- **Compound State:** is the computational modeling or representation of an abstract state. In other words, an abstract state is an abstraction of its corresponding compound state. A compound state is a n-component vector, *compound state* $= v_c(v_1, v_2, \ldots, v_n)$, or v_c in short, where $v_i \in V_i = \{v_i^k, k = \overline{1, m_i}\}$, set of all possible m_i values of a connotation component attribute A_i. We define, $A = \{A_1, A_2, \ldots, A_n\}$, thes set of n

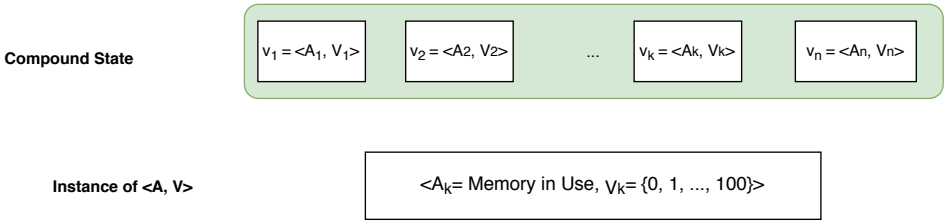


Figure 8.2: Compound State

attribute components constituting a system's whole internal state semantics. Such an A_i as, memory, disk, network sent/received, etc. We denote, $V = \{V_1, V_2, \ldots, V_n\}$, the set of domain values of A, or V_i is the set of all all possible values of attribute A_i. For example, a possible memory in use in percentage can be in the range, $[0, 100]$. Thus, a compound state vector v_c is an instance of the meta vector $< A, V >$ (Fig. 8.2).

- **State Partitioning:** The domain space of V_i, corresponding to A_i attribute, in theory and reality can be infinite, for example, *bytes_sent/bytes_received* metric can be any positive value. Therefore, *state partitioning* is used to map a compound state into an abstract state. For example, $< V_k, A_k >$, a domain space of k^{th} component of

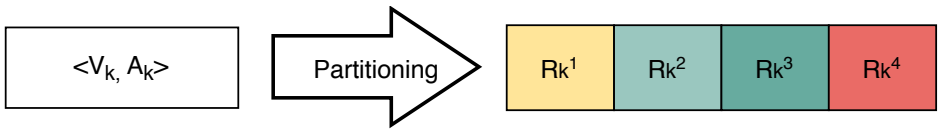


Figure 8.3: State Partitioning

$< V, A >$, is partitioned into 4 regions, $\{R_1, R_2, R_3, R_4\}$, where yellow and red colors mean a possibility of problematic values, and green and blue colors mean normal

operation possible values (Fig. 8.3). We formally define $P_k :< V_k, A_k > \xrightarrow{partitioning} R_k$, a partition operator on k^{th} component of the meta vector $< V, A >_k$, to partition its domain value into s regions, or a set R_k of s possible values, where $s = |R_k| \leq |V_k|$. Then, $P = \{P_k, k = \overline{1,n}\} :< V, A > \xrightarrow{partitioning} (R, A)$, is a partition operator on the complete meta vector $< V, A >$ is transformed into the abstract meta vector (R, A).

- **Abstract State:** an *abstract state* $= v_a(a_1, a_2, \ldots, a_n)$ is an instance of the meta vector (R, A), where $a_i \in R_i$, a set of all possible region abstract values of the attribute A_i. Often, R_i consists of two kinds of regions, one whose values mean something "normal"– *normal or green regions*, denoted as R_i^{green}. The other type is composed of regions, whose values mean "abnormal"– *abnormal or red regions*, denoted as R_i^{red}.

- **Normal State:** is an abstract state, $v_a(a_1, a_2, \ldots, a_n)$, where $a_j \in R_j^{green}, for\ all\ j = \overline{1,n}$.

- **Abnormal State:** is an abstract state, $v_a(a_1, a_2, \ldots, a_n)$, where $\exists j \in [1, n], such\ that, a_j \in R_j^{red}$.

- **Transition:** is mapping $ts : (R, A) \xrightarrow{events} (R, A), v_a \mapsto ts(v_a)$. A transition is normal– ts^{green}, if its destination abstract state is *normal*, otherwise it is called *abnormal*, and denoted as ts^{red}. A transition is defined as *single transition* if the difference between

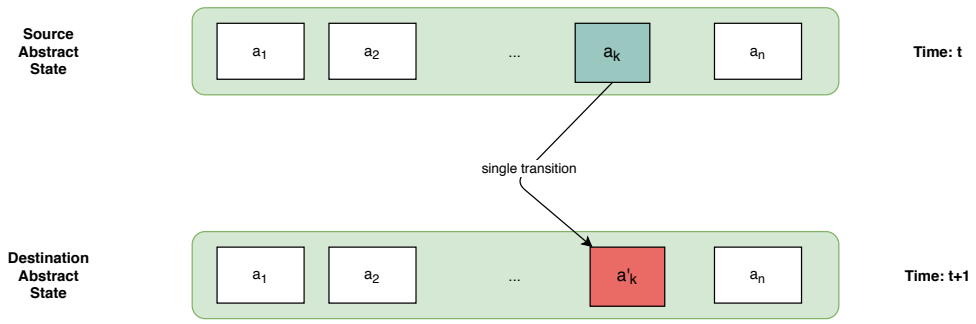


Figure 8.4: Single Transition

destination state and source state happens only one attribute A_k (Fig. 8.4), otherwise, is called *compound transition* (Fig. 8.5)

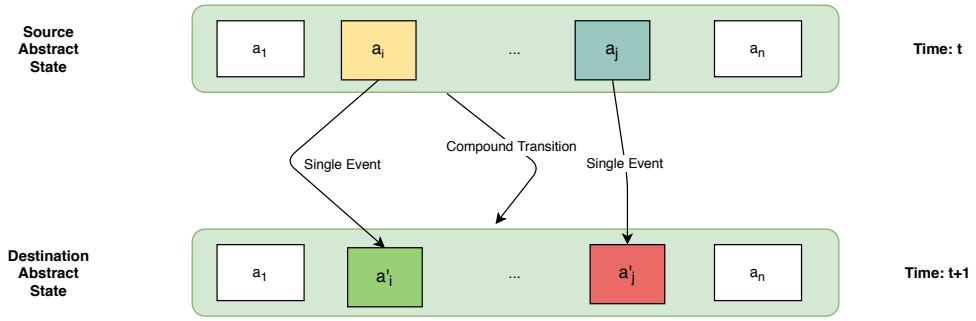


Figure 8.5: Compound Transition

- **Overall Workflow:** This section describes the overall workflow of our proposed solution, called Cloud-based Fault Localization with Abstract State Machine (CFA). CFA involves in two steps, construction and deployment (Fig. 8.6).

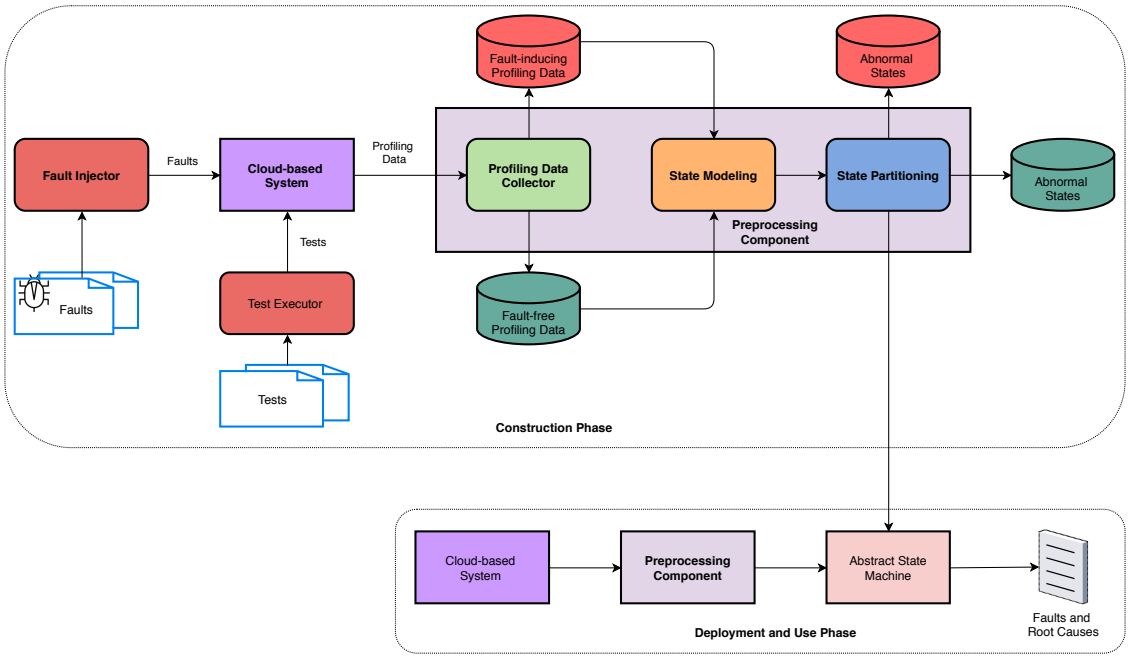


Figure 8.6: CFA Overall Workflow

Design and Implementation Plan This is the process of constructing and training

an ASM which is then used in the later phase.

1. **Fault Injection:** Faults are injected into the system via the Fault Injector component to make sure the system is faulty. Such faults as, excessive CPU usage, network connection drop, out-of-memory, etc.
2. **Test Execution:** For each fault injected, we execute some verification tests until there are some failing tests to make sure that a fault is really introduced into the system.
3. **Profiling Data Collecting:** When the system is in faulty state, its behavior is collected using a data collector, and labeled with *faulty-inducing*. When no fault is injected and all verification tests are passing, data is labeled with *fault-free*. All profiling data at this step is raw data in aJSON format.
4. **State Modeling:** Raw profiling data in JSON format is then modeled into a compound state vector (Sec. 8.1, Fig. 8.2).
5. **State Partitioning:** Compound states are partitioned using a k-clustering procedure to map them into a much smaller space of abstract states (Sec. 8.1, Fig. 8.3).
6. **ASM:** The final outcome of this process is the pre-computed or trained ASM and ready to be deployed and used for detecting faults and analyzing their root causes.

Deployment Plan Once an ASM is trained, it is deployed to monitor and analyze in real-time a cloud-based system to localize faults and root causes.

k-State Tuning How states are abstracted by partitioning into k-state with a k-clustering procedure is the key factor influencing accuracy of the approach. This step is necessary to find an optimal ASM to best localize faults and root causes.

8.2 Research Questions

In this research, we aim to answer the following research questions:

- **RQ1:** How does CFA help localize faults in a cloud-based system? Is it able to both localize faults and root causes?
- **RQ2:** How CFA is compared with the state-of-the-art and baseline tools, in terms of performance and accuracy ?

8.3 Approach

We have implemented the following components of CFA:

- **Implementation:** main components of CFA:
  (1) **Fault Injector:** We have collected and classified the most common faults happened in a real-world cloud-based environment. We then write automated scripts to inject the faults into the cloud whenever the scripts are executed.
  (2) **Test Executor:** Our cloud system is guarded with a set of tests, called Business Verification Test (BVT), and Production Verification Test (PVT) which test the most common behaviors of a cloud-based system, such as, if database is connected, HTTP requests return 200 status code, etc. All of the tests are executed via the TestNG framework [10].
  (3) **Data Collector:** We have implemented a data collector that is capable of collecting a cloud-based system profiling data in real-time. The metrics it can collect include, virtual machine (VM) usage, such as, CPU, memory, and disk usage;

network traffic, such as, bytes_sent, bytes_received; Database metrics, such as, number of connections, query-processing-time; exceptions or stack-traces thrown in the environment; and many others.

(4) **State Modeling:** We have implemented this component which is capable of cleaning, standardizing, and formatting the profiling data into a format called *compound state vector*, that we define above.

(5) **State Partitioning:** Finally, we have implemented this state partitioning component that uses k-clustering algorithms to map *compound states* to *abstract states*.

The code of CFA is open-sourced and can be found in Github [1].

- **Experiment data sets:** We have prepared a large clean experiment data sets of ten of GBs of profiling data of a cloud-based system that dictate *normal behavior* of the system. The data set can then be used to train and build an ASM for CFA.

8.4 Evaluation Plan

In order to answer the proposed research questions, we have to complete the following work:

- **Running Experiments:** design and run experiments to evaluate how CFA performs on a real-world cloud-based system.

- **Analyzing Results:** collect and analyze the results from the experiments to provide insights into the proposed research questions.

[1] The code repository will be open to access when we publish this work.

8.5 Summary

In this chapter, we describe the idea and concept of using abstract state machine to localize faults in a cloud-based system. We discuss how to implement the proposed solution into a tool called CFA. We summarize what have been done and remained to complete in order to evaluate the solution and provide answers to our proposed research questions.

Chapter 9

Conclusions

Fault localization remains a challenging, time-consuming, and expensive task. Despite a huge loss in the software companies due to software defects, adopting automated fault localization in industry is still difficult and even impractical.

This dissertation research improved fault localization, in terms of increasing accuracy and reducing runtime overhead cost. In addition, our research aimed at easing the applicability of fault localization in the software industry. More specifically, this dissertation enhanced the existing IRFL techniques with dynamic runtime execution information to improve the accuracy of fault localization. Moreover, this dissertation introduced a novel concept of *triggering mode* and *Instant Fault Localization (IFL)* techniques that significantly reduces the runtime cost of SBFL, ultimately making its applicability in industry more practical. We hope the results of this research can be beneficial to research community, especially to industrial companies who seek to adopt *IFL* into their development processes to uncover bugs as efficiently as possible.

Finally, the dissertation planned out the future work to further extend and improve fault localization techniques to highly complex cloud-based applications using SBFL and the abstract state machine concept.

Bibliography

[1] ASM. http://asm.ow2.org.

[2] Javaslicer. https://www.st.cs.uni-saarland.de/javaslicer/.

[3] Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year. http://www.prweb.com/releases/2013/1/prweb10298185.htm, 2013.

[4] About Code Coverage. https://confluence.atlassian.com/clover/about-code-coverage-71599496.html, 2017.

[5] Clover. https://www.atlassian.com/software/clover, 2019.

[6] Defects4j. http://fault-localization.cs.washington.edu, 2020.

[7] Data dog. https://www.datadoghq.com/, 2020.

[8] New relic. https://newrelic.com/, 2020.

[9] Splunk. https://www.splunk.com/, 2020.

[10] Testng. https://testng.org/doc/, 2020.

[11] Cobertura. https://cobertura.github.io/cobertura/, 2023.

[12] fault-localization. https://fault-localization.cs.washington.edu/, 2023.

[13] Jacoco. https://www.jacoco.org/jacoco/trunk/doc/index.html, 2023.

[14] Junit. https://junit.org/junit5/, 2023.

[15] Maven. https://maven.apache.org/, 2023.

[16] Sonarqube. https://www.sonarsource.com/products/sonarqube/, 2023.

[17] Iflm. https://github.com/idf-icst/sbfl-study, 2023.

[18] R. Abreu, P. Zoeteweij, and A. J. C. 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, Sep. 2007. doi: 10.1109/TAIC.PART.2007.13.

[19] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98, Sept 2007. doi: 10.1109/TAIC.PART.2007.13.

[20] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2724-8. doi: 10.1109/PRDC.2006.18. URL http://dx.doi.org/10.1109/PRDC.2006.18.

[21] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, November 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.06.035. URL http://dx.doi.org/10.1016/j.jss.2009.06.035.

[22] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2009.06.035. URL https://www.sciencedirect.com/science/article/pii/S0164121209001319. SI: TAIC PART 2007 and MUTATION 2007.

[23] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. Simultaneous debugging of software faults. *Journal of Systems and Software*, 84(4):573–586, 2011. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2010.11.915. URL https://www.sciencedirect.com/science/article/pii/S0164121210003183. The Ninth International Conference on Quality Software.

[24] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151, Oct 1995. doi: 10.1109/ISSRE.1995.497652.

[25] Mohammad Alshayeb and Wei Li. An empirical study of system design instability metric and design evolution in an agile software process. *J. Syst. Softw.*, 74(3):269–274, February 2005. ISSN 0164-1212. doi: 10.1016/j.jss.2004.02.002. URL http://dx.doi.org/10.1016/j.jss.2004.02.002.

[26] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 49–60, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588230. doi: 10.1145/1831708.1831715. URL https://doi.org/10.1145/1831708.1831715.

[27] Rawad Abou Assi, Wes Masri, and Chadi Trad. Substate profiling for enhanced fault detection and localization: An empirical study. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 16–27, 2020. doi: 10.1109/ICST46399.2020.00013.

[28] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program depen-

dence graph and its application to fault diagnosis. *IEEE Transactions on Software Engineering*, 36(4):528–545, July 2010. ISSN 0098-5589. doi: 10.1109/TSE.2009.87.

[29] Eric Bauer and Randee Adams. *Reliability and Availability of Cloud Computing*. Wiley-IEEE Press, 1st edition, 2012. ISBN 1118177010.

[30] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. Leveraging contextual information from function call chains to improve fault localization. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–479, 2020. doi: 10.1109/SANER48275.2020.9054820.

[31] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 237–249, 2020.

[32] Seung-Seok Choi, Sung-Hyuk Cha, and Charles C. Tappert. A survey of binary similarity and distance measures. *Journal on Systemics, Cybernetics and Informatics*, 8: 43–48, 2010. URL https://api.semanticscholar.org/CorpusID:15289045.

[33] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with ample. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, 2005.

[34] 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, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-0535-6. doi: 10.1109/ICPC.2017.29. URL https://doi.org/10.1109/ICPC.2017.29.

[35] Tung Dao, Max Wang, and Na Meng. Exploring the triggering modes of spectrum-based fault localization: An industrial case. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 406–416, 2021. doi: 10.1109/ICST49551.2021.00052.

[36] D. Dean, Hiep Nguyen, and Xiaohui Gu. Ubl: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *ICAC '12*, 2012.

[37] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut. Perfcompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1742–1755, 2016. doi: 10.1109/TPDS.2015.2444392.

[38] Vidroha Debroy and W. Eric Wong. On the consensus-based application of fault localization techniques. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pages 506–511, 2011. doi: 10.1109/COMPSACW.2011.92.

[39] W. Eberle and L. Holder. Applying graph-based anomaly detection approaches to the discovery of insider threats. In *2009 IEEE International Conference on Intelligence and Security Informatics*, pages 206–208, 2009. doi: 10.1109/ISI.2009.5137304.

[40] Peter Feiler, Kevin Sullivan, Kurt Wallnau, Richard Gabriel, John Goodenough, Richard Linger, Thomas Longstaff, Rick Kazman, Mark Klein, Linda Northrop, and Douglas Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, 2006.

[41] Margaret Ann Francel and Spencer Rugaber. The value of slicing while debugging. *Sci. Comput. Program.*, 40(2-3):151–169, July 2001. ISSN 0167-6423. doi: 10.1016/S0167-6423(01)00013-2. URL http://dx.doi.org/10.1016/S0167-6423(01)00013-2.

[42] Wenhao Fu, Huiqun Yu, Guisheng Fan, and Xiang Ji. Test case prioritization approach to improving the effectiveness of fault localization. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 60–65, 2016. doi: 10.1109/SATE.2016.17.

[43] Errin W. Fulp, Glenn A. Fink, and Jereme N. Haack. Predicting computer system failures using support vector machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs*, WASL'08, page 5, USA, 2008. USENIX Association.

[44] Mojdeh Golagha, Alexander Pretschner, and Lionel C. Briand. Can we predict the quality of spectrum-based fault localization? In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 4–15, 2020. doi: 10.1109/ICST46399.2020.00012.

[45] Liang Gong, Hongyu Zhang, Lingxiao Jiang, and David Lo. Interactive fault localization leveraging simple user feedback. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 67–76, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-2313-0. doi: 10.1109/ICSM.2012.6405255. URL http://dx.doi.org/10.1109/ICSM.2012.6405255.

[46] Leo A. Goodman and William H. Kruskal. *Measures of Association for Cross Classifications*, pages 2–34. Springer New York, New York, NY, 1979. ISBN 978-1-4612-9995-0. doi: 10.1007/978-1-4612-9995-0_1. URL https://doi.org/10.1007/978-1-4612-9995-0_1.

[47] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189–200, Nov 2014. doi: 10.1109/ISSRE.2014.40.

[48] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012.

[49] D. Hao, L. Zhang, H. Mei, and J. Sun. Towards interactive fault localization using test information. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 277–284, Dec 2006. doi: 10.1109/APSEC.2006.59.

[50] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. *SIGPLAN Not.*, 33(7):83–90, July 1998. ISSN 0362-1340. doi: 10.1145/277633.277647. URL http://doi.acm.org/10.1145/277633.277647.

[51] Hongdou He, Jiadong Ren, Guyu Zhao, and Haitao He. Enhancing spectrum-based fault localization using fault influence propagation. *IEEE Access*, 8:18497–18513, 2020. doi: 10.1109/ACCESS.2020.2965139.

[52] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, André van Hoorn, Antonio Filieri, and David Lo. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience*, 49:1197 – 1224, 2019.

[53] Thomas Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '99, pages 50–57, New York, NY, USA, 1999. ACM. ISBN 1-58113-096-1. doi: 10.1145/312624.312649. URL http://doi.acm.org/10.1145/312624.312649.

[54] David A. Hull. Stemming algorithms: A case study for detailed evaluation. *J. Am. Soc. Inf. Sci.*, 47(1):70–84, January 1996. ISSN 0002-8231. doi: 10.1002/(SICI)

1097-4571(199601)47:1<70::AID-ASI7>3.3.CO;2-Q. URL http://dx.doi.org/10.1002/(SICI)1097-4571(199601)47:1<70::AID-ASI7>3.3.CO;2-Q.

[55] 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, pages 273–282, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101949. URL http://doi.acm.org/10.1145/1101908.1101949.

[56] 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*, 2002.

[57] 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 '02, pages 467–477, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi: 10.1145/581339.581397. URL http://doi.acm.org/10.1145/581339.581397.

[58] 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*, ISSTA 2014, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL https://doi.org/10.1145/2610384.2628055.

[59] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. 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, July 2017. doi: 10.1109/QRS.2017.22.

[60] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498, May 2007. doi: 10.1109/ICSE.2007.66.

[61] Feyzullah Koca, Hasan Sözer, and Rui Abreu. Spectrum-based fault localization for diagnosing concurrency faults. In Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich, editors, *Testing Software and Systems*, pages 239–254, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-41707-8.

[62] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 803–814, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642997. URL http://doi.acm.org/10.1145/2642937.2642997.

[63] Herb Krasner. The cost of poor software quality in the us: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, pages 1–46, 2021.

[64] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481, Nov 2015. doi: 10.1109/ASE.2015.73.

[65] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *ASE*, pages 476–481. IEEE, 2015. ISBN 978-1-5090-0025-8.

[66] 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, ESEC/FSE 2015, pages 579–590, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786880. URL http://doi.acm.org/10.1145/2786805.2786880.

[67] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. *Iterative User-Driven Fault Localization*, pages 82–98. Springer International Publishing, Cham, 2016. doi: 10.1007/978-3-319-49052-6_6.

[68] Lucia, David Lo, Lingxiao Jiang, and Aditya Budi. Comprehensive evaluation of association measures for fault localization. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010. doi: 10.1109/ICSM.2010.5609542.

[69] Lucia, David Lo, and Xin Xia. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 127–138, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642983. URL http://doi.acm.org/10.1145/2642937.2642983.

[70] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972 – 990, 2010. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2010.04.002. URL http://www.sciencedirect.com/science/article/pii/S0950584910000650.

[71] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-13360-1.

[72] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

[73] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[74] L. Mariani, C. Monni, M. Pezzé, O. Riganelli, and R. Xin. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 262–273, 2018. doi: 10.1109/ICST.2018.00034.

[75] Wes Masri and Rawad Abou Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23, 02 2014. doi: 10.1145/2559932.

[76] Lee Naish and Hua Jie Lee. Duals in spectral fault localization. In *2013 22nd Australian Software Engineering Conference*, pages 51–59, 2013. doi: 10.1109/ASWEC.2013.16.

[77] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, August 2011. ISSN 1049-331X. doi: 10.1145/2000791.2000795. URL http://doi.acm.org/10.1145/2000791.2000795.

[78] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 263–272. IEEE Computer Society, 2011. ISBN 978-1-4577-1638-6. URL http://dblp.uni-trier.de/db/conf/kbse/ase2011.html#NguyenNANN11.

[79] Caleb C. Noble and Diane J. Cook. Graph-based anomaly detection. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, page 631–636, New York, NY, USA, 2003. Association for

Computing Machinery. ISBN 1581137370. doi: 10.1145/956750.956831. URL https://doi.org/10.1145/956750.956831.

[80] 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*, MSR '11, pages 43–52, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985451. URL http://doi.acm.org/10.1145/1985441.1985451.

[81] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC '97/FSE-5, pages 432–449, New York, NY, USA, 1997. Springer-Verlag New York, Inc. ISBN 3-540-63531-9. doi: 10.1145/267895.267925. URL http://dx.doi.org/10.1145/267895.267925.

[82] R.K. Saha, M. Lease, S. Khurshid, and D.E. Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355, Nov 2013. doi: 10.1109/ASE.2013.6693093.

[83] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975. ISSN 0001-0782. doi: 10.1145/361219.361220. URL http://doi.acm.org/10.1145/361219.361220.

[84] Qusay Idrees Sarhan and Árpád Beszédes. A survey of challenges in spectrum-based software fault localization. *IEEE Access*, 10:10618–10639, 2022. doi: 10.1109/ACCESS.2022.3144079.

[85] Yui Sasaki, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. Sbfl-suitability: A software characteristic for fault localization. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 702–706, 2020. doi: 10.1109/ICSME46990.2020.00076.

[86] Carla Sauvanaud, Kahina Lazri, Mohamed Kaâniche, and Karama Kanoun. Anomaly Detection and Root Cause Localization in Virtual Network Functions. In *27th International Symposium on Software Reliability Engineering (ISSRE 2016)*, Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE 2016), pages 196 – 206, Ottawa, Canada, October 2016. doi: 10.1109/ISSRE.2016.32. URL https://hal.archives-ouvertes.fr/hal-01419014.

[87] Abhishek B. Sharma, Haifeng Chen, Min Ding, Kenji Yoshihira, and Guofei Jiang. Fault detection and localization in distributed systems using invariant relationships. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–8. IEEE Computer Society, 2013. doi: 10.1109/DSN.2013.6575304. URL https://doi.org/10.1109/DSN.2013.6575304.

[88] Kai Shen, Christopher Stewart, Chuanpeng Li, and Xin Li. Reference-driven performance anomaly identification. *SIGMETRICS Perform. Eval. Rev.*, 37(1):85–96, June 2009. ISSN 0163-5999. doi: 10.1145/2492101.1555360. URL https://doi.org/10.1145/2492101.1555360.

[89] 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, Piscataway, NJ, USA, 2012.

IEEE Press. ISBN 978-1-4673-1761-0. URL http://dl.acm.org/citation.cfm?id=2664446.2664454.

[90] Higor Souza, Danilo Mutti, Marcos Chaim, and Fabio Kon. Contextualizing spectrum-based fault localization. *Information and Software Technology*, 94, 10 2017. doi: 10.1016/j.infsof.2017.10.014.

[91] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 314–324, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321594. doi: 10.1145/2483760.2483767. URL https://doi.org/10.1145/2483760.2483767.

[92] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. *Proceedings of the International Conference on Intelligence Analysis*, 2004.

[93] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 285–294, 2012. doi: 10.1109/ICDCS.2012.65.

[94] Yongmin Tan, Xiaohui Gu, and Haixun Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, page 173–182, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588889. doi: 10.1145/1835698.1835741. URL https://doi.org/10.1145/1835698.1835741.

[95] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[96] Béla Vancsics, Attila Szatmári, and Árpád Beszédes. Relationship between the effectiveness of spectrum-based fault localization and bug-fix types in javascript programs. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 308–319, 2020. doi: 10.1109/SANER48275.2020.9054803.

[97] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Call frequency-based fault localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 365–376, 2021. doi: 10.1109/SANER50967.2021.00041.

[98] J. M. Voas. Pie: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, Aug 1992. ISSN 0098-5589. doi: 10.1109/32.153381.

[99] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 1–11, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771797. URL http://doi.acm.org/10.1145/2771783.2771797.

[100] 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*, ICPC 2014, pages 53–63, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2879-1. doi: 10.1145/2597008.2597148. URL http://doi.acm.org/10.1145/2597008.2597148.

[101] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[102] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. 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, Sep. 2014. doi: 10.1109/ICSME.2014.40.

[103] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016. ISSN 2326-3881. doi: 10.1109/TSE.2016.2521368.

[104] 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. doi: 10.1109/TSE.2016.2521368.

[105] 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 Trans. Softw. Eng. Methodol.*, 22(4):31:1–31:40, October 2013. ISSN 1049-331X. doi: 10.1145/2522920.2522924. URL http://doi.acm.org/10.1145/2522920.2522924.

[106] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 191–200, Sept 2014. doi: 10.1109/ICSME.2014.41.

[107] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63, 2014.

[108] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports

using domain knowledge. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 689–699, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635874. URL http://doi.acm.org/10.1145/2635868.2635874.

[109] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In *Proceedings of the 4th International Conference on Search Based Software Engineering*, SSBSE'12, pages 244–258, Berlin, Heidelberg, 2012. Springer-Verlag.

[110] Shin Yoo, Mark Harman, and David Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22, 07 2013. doi: 10.1145/2491509.2491513.

[111] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. Technical report, University College London and Swinburn University, 2014.

[112] Y. Yu, J. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 201–210, 2008.

[113] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, page 375–388, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933220. doi: 10.1145/1217935.1217972. URL https://doi.org/10.1145/1217935.1217972.

[114] Abubakar Zakari, Sai Peck Lee, and Ibrahim Abaker Targio Hashem. A single fault localization technique based on failed test input. *Array*, 3-4:100008, 2019. ISSN 2590-0056. doi: https://doi.org/10.1016/j.array.2019.100008. URL https://www.sciencedirect.com/science/article/pii/S2590005619300086.

[115] Hui Zhang, Junghwan Rhee, Nipun Arora, Sahan Gamage, Guofei Jiang, Kenji Yoshihira, and Dongyan Xu. CLUE: system trace analytics for cloud service performance diagnosis. In *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, pages 1–9. IEEE, 2014. doi: 10.1109/NOMS.2014.6838348. URL https://doi.org/10.1109/NOMS.2014.6838348.

[116] 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*, ISSTA 2017, pages 261–272, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092731. URL http://doi.acm.org/10.1145/3092703.3092731.

[117] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 312–321, 2013. doi: 10.1109/ICSE.2013.6606577.

[118] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 272–281, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134324. URL http://doi.acm.org/10.1145/1134285.1134324.

[119] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite

effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 214–224, 2015.

[120] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, and Xin Xia. Improving fault localization using model-domain synthesized failing test generation. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 199–210, 2022. doi: 10.1109/ICSME55016.2022.00026.

[121] Jian Zhou, Hongyu Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 14–24, June 2012. doi: 10.1109/ICSE.2012.6227210.

[122] 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*, 2019.

[123] Ma łgorzata Steinder and Adarshpal S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165 – 194, 2004. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2004.01.010. URL http://www.sciencedirect.com/science/article/pii/S0167642304000772. Topics in System Administration.