Triggering Modes in Spectrum-Based Multi-location Fault Localization

Tung Dao
Cvent
USA

Na Meng
Virginia Tech
USA

ThanhVu Nguyen
George Mason University
USA

ABSTRACT

Spectrum-based fault localization (SBFL) techniques can aid in debugging, but their practicality in industrial settings has been limited due to the large number of tests needed to execute before applying SBFL. Previous research has explored different trigger modes for SBFL and found that applying it immediately after the first test failure is also effective. However, this study only considered single-location bugs, while multi-location bugs are prevalent in real-world scenarios and especially at our company Cvent, which is interested in integrating SBFL to its CI/CD workflow.

In this work, we investigate the effectiveness of SBFL on multi-location bugs and propose a framework called Instant Fault Localization for Multi-location Bugs (IFLM). We compare and evaluate four trigger modes of IFLM using open-source (Defects4J) and close-source (Cvent) bug datasets. Our study showed that it is not necessary to execute all test cases before applying SBFL. However, we also found that that applying SBFL right after the first failed test is less effective than applying it after executing all tests for multi-location bugs, which is contrary to the single-location bug study. We also observe differences in performance between real and artificial bugs. Our contributions include the development of IFLM and CVent bug datasets, analysis of SBFL effectiveness for multi-location bugs, and practical implications for integrating SBFL in industrial environments.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Spectrum-based Fault Localization, CI/CD, SBFL Triggering Modes, Multi-Location Bugs, Industrial Study

ACM Reference Format:
Tung Dao, Na Meng, and ThanhVu Nguyen. 2023. Triggering Modes in Spectrum-Based Multi-location Fault Localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3613884

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12...$15.00
https://doi.org/10.1145/3611643.3613884

1 INTRODUCTION

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 [33]. To aid debugging, many automatic *fault localization* techniques have been proposed. For example, the well-known spectrum-based fault localization (SBFL) technique [12, 30] 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 [21], 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 prtical 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 [21] 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., [42], 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 inital study to develop a framework called *Instant Fault Localization for Multi-location Bugs* (IFLM) that triggers SBFL in four modes: (i) $\mathbf{IFLM}_1$ invokes SBFL right after the first failed test; (ii) $\mathbf{IFLM}_f^k$ triggers SBFL after every k=1–5 test failures (IFLM_1 is the special case when $k = 1$); (iii) $\mathbf{IFLM}_p^k$ activates SBFL after the first failed test and subsequently, k additional passed tests (k=1 − 10); and (iv) $\mathbf{IFLM}_A$ triggers SBFL after executing all tests.

We conduct the IFLM study using two multi-location bug datasets: (i) the open-source Defect4J dataset [4], 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.

In summary, our paper 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 repositiory at [5].

2 BACKGROUND

2.1 Spectrum-Based Fault Localization (SBFL)

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 $e \in P$.

SBFL uses four steps to identify and rank buggy locations:

(1) **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 [9], Clover [2], Jacoco [6], Corbetura [3] are often used in industry for this purpose. We use Clover in this paper.

(2) **Test Execution:** T is run against the instrumented P using automated test runner or framewor, such as Maven [8], TestNG [10], JUnit [7]. Once the test execution is finished, P_{spectra} is all recorded.

(3) **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 paper we used 25 popular formulae shown in Tab. 3, and described in detail in [21].

(4) **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.

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

Multi-Location Bugs. Our study focuses on 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. 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. Listing 1 is a single-location bug (incorrect predicate to check cache expiry). 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 §4.2 for definitions of accuracy metrics).

2.2 Continuous Integration/ Continuous Delivery (CI/CD)

Cvent uses 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. 1 outlines the CI/CD workflow at Cvent. Without SBFL, test failures require manual inspection to find bugs. However, with SBFL integration, bugs are automatically localized using code coverage profiling data. This accelerates the debugging process for developers as manually localizing bugs, especially multi-location ones, is highly time-consuming and difficult. In short, at Cvent we believe that an automatic fault localization approach such as SBFL is an important component of our CI/CD workflow.

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, IFLM appears to be a more suitable alternative to SBFL for scaling to our level of operations.

Collecting Code Coverage. The CI/CD build pipeline at Cvent uses the open-source Clover tool [2] to collect profiling data. Fig. 2 illustrates how Clover collects code coverage from profiling data of a Java-based micro-service 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).

3 STUDY 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.

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 [5].

4.1 Datasets

We used 238 multi-location bugs, of which 211 are from Defect4J dataset [31] and 27 are mined from four close-source software products from Cvent. For the Defects4j bugs, we reused the spectrum data published in [1] 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 [31]. Out of these bugs, we found 174 multi-location bugs from 6 projects that have spectra data [1].

We also used artificial bugs from [1] 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 [1] 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.

Tab. 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).

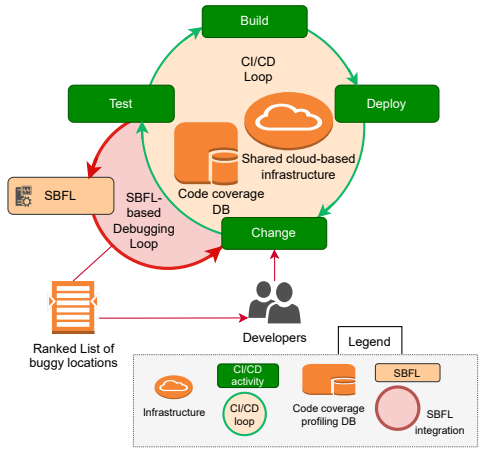


Figure 1: CI/CD and SBFL Integration Outline.

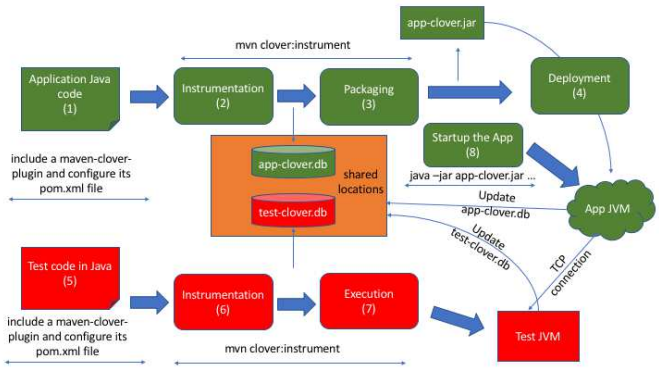


Figure 2: Collecting coverage profiling data with Clover.

Table 1: The Defect4J dataset: 174 real bugs and 37 artificial bugs from 5 open-source projects.

Project	# of Bugs		LOC Exec	# Tests	1st 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**			

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, **1st 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.

Table 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 (%)

Tab. 2 provides additional information for failed tests. Approximately 50% of the bugs have more than 1 failing test. Among these bugs, the majority of them have 2-3 failing tests. This justifies why in the 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 [5].

4.2 Effectiveness Metrics

We adopt three widely used metrics to measure the effectiveness of IFLM [39, 58].

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.

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:

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

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 kth statement is positive. Namely, $pos(k) = 1$ if the kth statement is positive, otherwise $pos(k) = 0$. For example, if four statements are ranked, and the 3rd and 4th are positive, then AP is $(\frac{1}{3} + \frac{2}{4})/2 \times 100\% = 42\%$. On the other hand, if the 1st and 2nd of the ranked list are buggy, then $AP = (\frac{1}{1} + \frac{2}{2})/2 \times 100\% = 100\%$.

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:

$$RR = \frac{1}{\text{rank}_{\text{best}}} \times 100\% \quad (2)$$

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

4.3 RQ1: Comparing Triggering Modes

We evaluate the effectiveness of different triggering modes for SBFL using **Dice** (see §2.1 for its definition) as the default ranking formula for IFLM. This is because Dice generally outperformed other formulae. Tab. 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).

4.3.1 Effectiveness of $IFLM_1$ and $IFLM_A$. Tab. 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, 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, IFLM_1 performed significantly worse than 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 IFLM_1 and IFLM_A on artificial bugs compared to real bugs. Additionally, artificial bugs appear biased towards IFLM_A and become challenging for IFLM_1. Thus, we do not advise using artificial bugs to evaluate SBFL on multi-location bugs.

> **Finding 1:** For real bugs, IFLM_1 performs slightly worse than 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 IFLM_A, in terms of runtime and the number of executed tests). However, for artificial bugs, IFLM_A significantly outperforms IFLM_1, suggesting that artificial bugs might not be suitable for evaluating SBFL for multi-location bugs.

4.3.2 Effectiveness of $IFLM_f^k$. We evaluated IFLM_f^k by triggering SBFL at every k^{th} occurrence of a test failure, where k= 1–5. Tab. 5 shows the average measurements. As can be seen, the effectiveness measurements 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 IFLM_f^k using 3 ranking formulae randomly selected in 3 corresponding representative groups in Tab. 3, namely **Goodman**, **Hamann**, and **Euclid** (§2.1). Fig. 3 (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. 3 (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.

> **Finding 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.*

4.3.3 Effectiveness of $IFLM_p^k$. Tab. 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, 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 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, performances increased 3% to 19% in

Table 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 [19]	2040	11	30	15	21	DICE [19]	466	12	35	23	23
KULCZYNSKI1 [37]	2040	11	30	15	21	KULCZYNSKI1 KULCZYNSKI1 [37]	466	12	35	23	23
SORENSENDICE [11]	2040	11	30	15	21	SORENSENDICE [11]	466	12	35	23	23
GOODMAN [26]	2040	11	30	15	21	GOODMAN [26]	466	12	35	23	23
ANDERBERG [11]	2040	11	30	15	21	ANDERBERG [11]	466	12	35	23	23
JACCARD [14]	2040	11	30	15	21	JACCARD [14]	466	12	35	23	23
M2 [56]	2040	10	28	14	19	M2 [56]	466	12	35	23	23
RUSSELLRAO [19]	2040	10	28	14	19	RUSSELLRAO [19]	466	12	35	23	23
AMPLE [19]	2040	8	25	12	17	AMPLE [19]	466	12	35	23	23
WONG3 [22]	2040	6	15	7	10	WONG3 [22]	466	12	27	20	20
WONG2 [48]	2040	5	15	7	10	WONG2 [48]	466	12	27	20	20
SIMPLEMATCHING	2040	5	14	6	9	SIMPLEMATCHING	466	10	22	17	17
HAMANN [19]	2040	5	14	6	9	HAMANN [19]	466	10	22	17	17
SOKAL [19]	2040	5	14	6	9	SOKAL [19]	466	10	22	17	17
ROGERSTANIMOTO [19]	2040	5	14	6	9	ROGERSTANIMOTO [19]	466	10	22	17	17
M1 [56]	2040	5	14	6	9	M1 [56]	466	10	22	17	17
EUCLID [19]	2040	0	5	3	3	EUCLID [19]	466	0	0	2	2
WONG1 [48]	2040	0	5	3	3	WONG1 [48]	466	0	0	2	2
HAMMING [19]	2040	0	5	3	3	HAMMING [19]	466	0	0	2	2
OCHIAI2 [19]	2040	0	1	1	1	OCHIAI2 [19]	466	0	0	1	1
OCHIAI [14]	2040	0	1	1	1	OCHIAI [14]	466	0	0	1	1
TARANTULA [14]	2040	0	1	1	1	TARANTULA [14]	466	0	0	1	1
KULCZYNSKI2 [19]	2040	0	1	1	1	KULCZYNSKI2 [19]	466	0	0	1	1
ZOLTAR [13]	2040	0	1	1	1	ZOLTAR [13]	466	0	0	1	1
OVERLAP [21]	2040	0	0	0	0	OVERLAP [21]	466	0	0	0	0

Table 4: Comparing $IFLM_1$ and $IFLM_A$ (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

Table 5: Effectiveness of $IFLM_f^k$ 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

Table 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

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.

4.4 RQ2: Sensitivity to SBFL Formulae

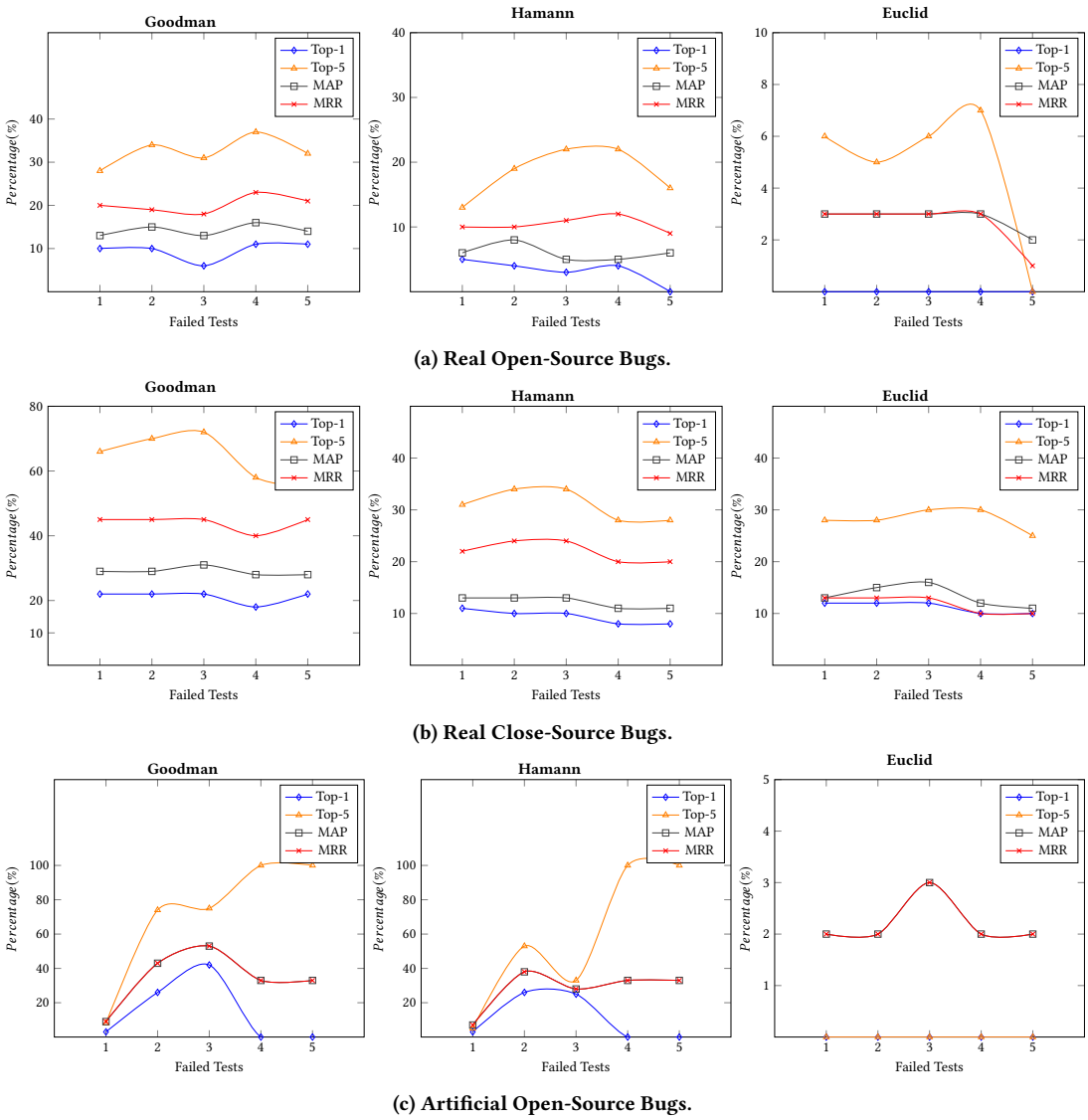
We ran IFLM using 25 popular SBFL formulae shown in Tab. 3 to ensure the generalizability of our observation comparing $IFLM_1$ and $IFLM_A$ and explore IFLM's sensitivity to different SBFL formulae.

Table 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**

Table 8: The effectiveness of $IFLM_1$ and $IFLM_A$ 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
SØrensenDice	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



(a) Real Open-Source Bugs.

(b) Real Close-Source Bugs.

(c) Artificial Open-Source Bugs.

Figure 3: Effectiveness of $IFLM_f^k$ using different SBFL formulae: Goodman, Hamann, Euclid.

The results of $IFLM_1$ and $IFLM_A$ are given in Tab. 7, and Tab. 8 for the Defects4J's real, artificial bugs, respectively; and Tab. 9 for Cvent bugs. The **Diff** column shows the differences of measured values between $IFLM_1$ and $IFLM_A$ (larger or equal values are in bold). A positive or zero diff value (in **bold**) indicates that $IFLM_1$ performed as well as or better than $IFLM_A$.

For the real bugs in Defects4J, out of the total 25 formulae, $IFLM_1$ outperformed or achieved comparable results to $IFLM_A$ in Top-1, Top-5, MAP, and MRR in 17, 14, 10, and 14 formulae, respectively (Tab. 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. 9). On average, there was minimal distinction between $IFLM_1$ and $IFLM_A$ across all four performance metrics. Thus, for real bugs, the observation that $IFLM_1$ performed at least as well as $IFLM_A$ (in all four effectiveness metrics) was confirmed in more than 50% of the 25 investigated SBFL formulae.

> **Finding 4:** *For real bugs, in general, $IFLM_1$ performed equally or better than $IFLM_A$ 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. $IFLM_A$ outperformed $IFLM_1$ for 16/25 formulae. The remaining 9 formulae showed similar performance levels for both $IFLM_1$ and $IFLM_A$. 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.

Table 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**
SØrensenDice	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**

For sensitivity of IFLM to SBFL formulae, Tab. 7 & 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 5: *SBFL formulae can have significant influence to the performance of IFLM. There were often multiple formulae that worked equally-well for IFLM.*

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 [34, 50, 59], 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 [1]. For the close-source Cvent dataset, we modified Clover [2] 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 [5].

6 RELATED WORK

We describe several FL techniques realted to this study, including the effectiveness and applicability of SBFL; test reduction, prioritization, and generation; and bug characteristics.

Spectrum-Based Fault Localization (SBFL). Tarantula [30] 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 [12, 20, 29, 30, 38, 50, 51]. 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. [35] and Yoo et al. [53] 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 —IFLM 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 IFLM'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.

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 [16, 17, 27, 40, 44, 50]. He et al. augmented coverage information with test call graph to construct fault inducing or influencing network, which helps narrow down bug searching space [27]. Beszédes et al. used snapshots of call stack to assist SBFL to localize buggy functions [17]. Xuan et al. used machine learning to train a model by combining 25 suspiciousness score formulae [50]. 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.

Effectiveness and Applicability of SBFL. Many studies have highlighted the insufficiency of SBFL and its limited real-world application [25, 28, 32, 35, 40, 41, 45, 47, 49, 53]. 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 [47]. Sarhan et al., in a recent survey [40], 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.

Test Reduction, Prioritization, or Generation. 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 [15, 18, 23, 36, 52, 54, 57]. 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 [36]. 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 [54].

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 [52]. 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 [15].

Our research shares a similar motivation, particularly in exploring methods to reduce the running cost of SBFL. Our proposed IFLM 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.

Bug Characteristics and Relation with SBFL. Bug characteristics include various aspects such as bug type (e.g., single or multiple faults), programming language used (e.g., C++, Java, JavaScript), and program complexity (e.g., nesting level, length of methods). Previous studies have shown a correlation between bug characteristics and the effectiveness of SBFL techniques [24, 41, 46, 55]. For instance, Vancsics et al. found that for the same nesting complexity, shorter programs tend to have more accurate bug localization compared to longer ones [43, 46].

Bug type is typically classified as either single-fault or multiple-fault. In a single-fault scenario, the buggy program has only one logical bug, and all failed tests can be caused by this single bug, which may consist of one or more buggy locations. On the other hand, a multiple-fault situation refers to a buggy program containing multiple distinct single-faults. SBFL is known to be less effective for bugs in multiple-fault programs than single-fault ones [55].

Our study focused on bugs in Java programs, which constitute a major part of the codebase at Cvent. Other programming languages are left for future research. Moreover, we found it more practical and reflective of real-world industry settings to classify bugs as single-location bugs (involving only one location) or multi-location bugs (requiring fixes at more than one locations). Unlike the study evaluated on single-location bugs [21], the emphasis of this paper was on multi-location bugs as they are more common at Cvent.

7 CONCLUSION

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 [21] 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 [21], 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.

ACKNOWLEDGMENTS

We thank the reviewers for their insightful comments. This work was partially funded by NSF CCF-1845446, 2238133, and 2200621.

REFERENCES

[1] 2020. Defects4J. http://fault-localization.cs.washington.edu.
[2] 2023. Clover. https://openclover.org/.
[3] 2023. Cobertura. https://cobertura.github.io/cobertura/.
[4] 2023. fault-localization. https://fault-localization.cs.washington.edu/.
[5] 2023. IFLM. https://github.com/idf-icst/sbfl-study.
[6] 2023. Jacoco. https://www.jacoco.org/jacoco/trunk/doc/index.html.
[7] 2023. JUnit. https://junit.org/junit5/.
[8] 2023. Maven. https://maven.apache.org/.
[9] 2023. SonarQube. https://www.sonarsource.com/products/sonarqube/.
[10] 2023. TestNG. https://testng.org/doc/.
[11] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035 SI: TAIC PART 2007 and MUTATION 2007.
[12] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*. 89–98. https://doi.org/10.1109/TAIC.PART.2007.13
[13] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2011. Simultaneous debugging of software faults. *Journal of Systems and Software* 84, 4 (2011), 573–586. https://doi.org/10.1016/j.jss.2010.11.915 The Ninth International Conference on Quality Software.
[14] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 89–98. https://doi.org/10.1109/TAIC.PART.2007.13
[15] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed Test Generation for Effective Fault Localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/1831708.1831715
[16] Rawad Abou Assi, Wes Masri, and Chadi Trad. 2020. Substate Profiling for Enhanced Fault Detection and Localization: An Empirical Study. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 16–27. https://doi.org/10.1109/ICST46399.2020.00013
[17] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. 2020. Leveraging Contextual Information from Function Call Chains to Improve Fault Localization. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 468–479. https://doi.org/10.1109/SANER48275.2020.9054820
[18] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. 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)*. 237–249.
[19] Seung-Seok Choi, Sung-Hyuk Cha, and Charles C. Tappert. 2010. A Survey of Binary Similarity and Distance Measures. *Journal on Systemics, Cybernetics and Informatics* 8 (2010), 43–48. https://api.semanticscholar.org/CorpusID:15289045
[20] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight Bug Localization with AMPLE. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*.
[21] Tung Dao, Max Wang, and Na Meng. 2021. Exploring the Triggering Modes of Spectrum-Based Fault Localization: An Industrial Case. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 406–416. https://doi.org/10.1109/ICST49551.2021.00052
[22] Vidroha Debroy and W. Eric Wong. 2011. On the Consensus-Based Application of Fault Localization Techniques. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*. 506–511. https://doi.org/10.1109/COMPSACW.2011.92
[23] Wenhao Fu, Huiqun Yu, Guisheng Fan, and Xiang Ji. 2016. Test Case Prioritization Approach to Improving the Effectiveness of Fault Localization. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 60–65. https://doi.org/10.1109/SATE.2016.17
[24] Debolina Ghosh and Jagannath Singh. 2021. Spectrum-based multi-fault localization using Chaotic Genetic Algorithm. *Information and Software Technology* 133 (2021), 106512. https://doi.org/10.1016/j.infsof.2021.106512
[25] Mojdeh Golagha, Alexander Pretschner, and Lionel C. Briand. 2020. Can We Predict the Quality of Spectrum-based Fault Localization?. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 4–15. https://doi.org/10.1109/ICST46399.2020.00012
[26] Leo A. Goodman and William H. Kruskal. 1979. *Measures of Association for Cross Classifications*. Springer New York, New York, NY, 2–34. https://doi.org/10.1007/978-1-4612-9995-0_1
[27] Hongdou He, Jiadong Ren, Guyu Zhao, and Haitao He. 2020. Enhancing Spectrum-Based Fault Localization Using Fault Influence Propagation. *IEEE Access* 8 (2020), 18497–18513. https://doi.org/10.1109/ACCESS.2020.2965139
[28] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, André van Hoorn, Antonio Filieri, and David Lo. 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience* 49 (2019), 1197 – 1224.
[29] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949
[30] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02)*. ACM, New York, NY, USA, 467–477. https://doi.org/10.1145/581339.581397
[31] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 4 pages. https://doi.org/10.1145/2610384.2628055
[32] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential Biases in Bug Localization: Do They Matter?. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 803–814. https://doi.org/10.1145/2642937.2642997
[33] Herb Krasner. 2021. The cost of poor software quality in the US: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)* (2021), 1–46.
[34] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2016. *Iterative User-Driven Fault Localization*. Springer International Publishing, Cham, 82–98. https://doi.org/10.1007/978-3-319-49052-6_6
[35] Lucia, David Lo, Lingxiao Jiang, and Aditya Budi. 2010. Comprehensive evaluation of association measures for fault localization. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. 1–10. https://doi.org/10.1109/ICSM.2010.5609542
[36] Wes Masri and Rawad Abou Assi. 2014. Prevalence of Coincidental Correctness and Mitigation of its Impact on Fault Localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23 (02 2014). https://doi.org/10.1145/2559932
[37] Lee Naish and Hua Jie Lee. 2013. Duals in Spectral Fault Localization. In *2013 22nd Australian Software Engineering Conference*. 51–59. https://doi.org/10.1109/ASWEC.2013.16
[38] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3, Article 11 (Aug. 2011), 32 pages. https://doi.org/10.1145/2000791.2000795
[39] R.K. Saha, M. Lease, S. Khurshid, and D.E. Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. 345–355. https://doi.org/10.1109/ASE.2013.6693093
[40] Qusay Idrees Sarhan and Árpád Beszédes. 2022. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access* 10 (2022), 10618–10639. https://doi.org/10.1109/ACCESS.2022.3144079
[41] Yui Sasaki, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. 2020. SBFL-Suitability: A Software Characteristic for Fault Localization. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 702–706. https://doi.org/10.1109/ICSME46990.2020.00076
[42] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. 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* (Lugano, Switzerland) *(ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 314–324. https://doi.org/10.1145/2483760.2483767
[43] Attila Szatmári, Béla Vancsics, and Árpád Beszédes. 2020. Do Bug-Fix Types Affect Spectrum-Based Fault Localization Algorithms' Efficiency?. In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. 16–23. https://doi.org/10.1109/VST50071.2020.9051638
[44] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. 2021. Call Frequency-Based Fault Localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 365–376. https://doi.org/10.1109/SANER50967.2021.00041
[45] Béla Vancsics, Attila Szatmári, and Árpád Beszédes. 2020. 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)*. 308–319. https://doi.org/10.1109/SANER48275.2020.9054803
[46] Béla Vancsics, Attila Szatmári, and Árpád Beszédes. 2020. 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)*. 308–319. https://doi.org/10.1109/SANER48275.2020.9054803

[47] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2771783.2771797

[48] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[49] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug 2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[50] J. Xuan and M. Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. 191–200. https://doi.org/10.1109/ICSME.2014.41

[51] Shin Yoo. 2012. Evolving Human Competitive Spectra-based Fault Localisation Techniques. In *Proceedings of the 4th International Conference on Search Based Software Engineering* (Riva del Garda, Italy) *(SSBSE'12)*. Springer-Verlag, Berlin, Heidelberg, 244–258.

[52] Shin Yoo, Mark Harman, and David Clark. 2013. Fault Localization Prioritization: Comparing Information-Theoretic and Coverage-Based Approaches. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22 (07 2013). https://doi.org/10.1145/2491509.2491513

[53] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2014. *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.

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

[55] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology* 124 (2020), 106312. https://doi.org/10.1016/j.infsof.2020.106312

[56] Abubakar Zakari, Sai Peck Lee, and Ibrahim Abaker Targio Hashem. 2019. A single fault localization technique based on failed test input. *Array* 3-4 (2019), 100008. https://doi.org/10.1016/j.array.2019.100008

[57] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, and Xin Xia. 2022. Improving Fault Localization Using Model-domain Synthesized Failing Test Generation. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 199–210. https://doi.org/10.1109/ICSME55016.2022.00026

[58] Jian Zhou, Hongyu Zhang, and D. Lo. 2012. 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*. 14–24. https://doi.org/10.1109/ICSE.2012.6227210

[59] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2019. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* (2019).