Exploring the Triggering Modes of Spectrum-Based Fault Localization: An Industrial Case

Tung Dao
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
tungdm@vt.edu

Max Wang
Platform and Product Engineering Group
International SOS
Trevose, PA, USA
max.q.wang@gmail.com

Na Meng
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
nm8247@vt.edu

***Abstract*—Fault localization is important for software development and maintenance. Among existing techniques, spectrum-based fault localization (SBFL) is effective to locate bugs based on the execution coverage of passed and failed tests. However, current SBFL tools require the execution of *all* tests before suggesting any ranked list of suspicious locations. In reality, such all-test execution can be very time-consuming; SBFL's outputs based on the whole-suite execution can significantly delay developers' debugging activities and jeopardize their productivity.**

For this paper, we were curious whether we can apply SBFL immediately after seeing one or several test failures, instead of waiting for all tests to finish their run. Specifically, with 28 injected bugs and 13 real bugs in a close-sourced software product, we collected the statement-level coverage for each test case, and investigated the usage of 25 alternative SBFL formulas. We triggered SBFL in five modes: (i) after the first test failure, (ii) after the first failure and some extra passed tests, (iii) after every test failure, (iv) at a specified time interval (e.g., every 2 minutes), or (v) after the complete execution of all tests.

Our study shows interesting results. Compared with whole-suite execution, triggering SBFL formulas earlier based on partial execution helps locate bugs more effectively. Among the five modes, the first-failure-driven mode works best. Additionally, we conducted similar experiments on 57 real bugs from the Defects4J dataset and observed similar phenomena. Our observations imply that instead of waiting for the completion of all test runs, it is quite promising to apply SBFL formulas immediately after the initial test failure. In this way, developers are likely to get better suggestions within a shorter period of time. Our research will help developers better adopt SBFL in practice.

***Index Terms*—Spectrum-based fault localization (SBFL), different triggering modes, dynamic ranking update**

I. Introduction

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 [1]. To simplify software debugging, various techniques have been introduced to automatically locate bugs or faults in programs [2], [3], [4], [5], [6]. For instance, information retrieval (IR)-based fault localization (IBFL) 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, IBFL works only when a report explicitly mentions the actual bug location [4], [7]. In reality, however, the bug location is not always mentioned in a report, and some serious bugs 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 bug locations when a program fails one or more tests, we were curious *whether SBFL can be triggered before all tests to complete their execution*. For this paper, 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 [8]—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—Instance Fault Localization (IFL)—to locate bugs in five distinct modes:

- IFL_1 triggers SBFL right after the initial test failure.
- IFL_f triggers SBFL after every test failure.
- IFL_o triggers SBFL after the initial failure and several additional passed tests.
- IFL_p triggers SBFL periodically (e.g., every 2 minute).
- IFL_c triggers SBFL after executing the complete suite.

In our study, we applied IFL to a software product in the first author's company. 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.

TABLE I: 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.

We experimented IFL 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, IFL_1 outperformed the other four modes when using Ample [9] as its default SBFL formula. Among the 25 formulas, for injected bugs, IFL_c obtained 56% MAP and 66% Top-5 recall, and spent 663 seconds executing all tests on average. In comparison, IFL_1 achieved 73% MAP and 86% Top-5 recall, and spent only 135 seconds executing roughly 20% of tests. For real bugs, IFL_c acquired 46% MAP and 56% Top-5 recall based on all-test execution; while IFL_1 achieved 76% MAP and 92% Top-5 recall after executing test cases for only 106 seconds. Additionally, we conducted similar experiments on the 57 real bugs from a third-party database of software faults—Defects4J [10], [11]. These bugs are located in versions of four open-source projects: jfreechart, commons-lang, commons-math, and mockito. We observed that IFL_1 obtained better results than IFL_c (i.e., 22% vs. 19% MAP, and 32% vs. 26% Top-5), by executing only 47% of test cases.

In summary, this paper makes the following contributions:

- We built IFL, a tool infrastructure that can trigger SBFL in 5 alternative modes using 25 distinct formulas.
- We applied IFL to a close-sourced software project and four open-sourced projects, and comprehensively evaluated IFL's bug localization effectiveness when it adopted various triggering modes and calculation formulas.
- We made two interesting observations in our exploration. First, among the five modes, IFL_c did not outperform the others although its runtime cost was the highest. Second, IFL_1 worked equally well when using certain formulas.

At https://github.com/idf-icst/idfl-package, we open-sourced our code and data.

II. Background

This section introduces SBFL (Section II-A) and describes Clover [8]—a tool used for execution profiling (Section II-B).

A. Spectrum-Based Fault Localization (SBFL)

SBFL identifies and ranks potential buggy locations when a program P fails the execution of its test suite T [2]. 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.

Although existing SBFL techniques rank elements using distinct formulas, all of them require users to execute all test

cases in T before suggesting any bug location. For this paper, we intended to identify the best timing of triggering SBFL, no matter what formula is used. Therefore, to ensure the representativeness of our observations, we experimented with 25 popularly used SBFL formulas [12]. As shown in Table I, these formulas calculate suspiciousness scores based on some or all of the following variables: n_f, n_p, e_f, and e_p.

B. Clover

We used Clover [8]—an open-source code profiling tool—to gather coverage 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 [13]. Such coverage information can help fault localization because if a test fails, the failure run probably covers some buggy code. 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.

III. Study Approach

We built IFL to have three components. As shown in Fig. 1, the first component is Clover [8], which is configured to instrument all Java statements and to record **per-test statement coverage**, i.e., which statement(s) are covered by a given test.

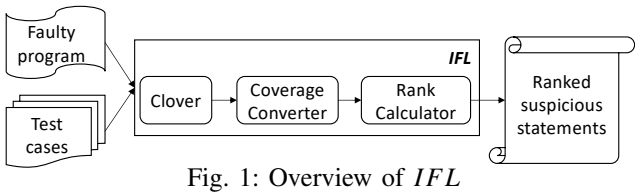


Fig. 1: Overview of IFL

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. 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. IFL 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.

Test	Covered Statements	Test Outcome
t1	s1, s2, ...	P
t2	s1, s3, ...	F

Statements	# of Passed (P) Tests	# of Failed (F) Tests
s1	1	1
s2	1	0
s3	0	1
...	...	...

Fig. 2: IFL converts per-test statement coverage to per-statement test coverage

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. IFL can apply SBFL in the following five distinct ways:

a) ***First-Failure Triggering*** *(IFL_1):* IFL_1 triggers SBFL after the first failure, because developers rarely need fault localization before seeing any failure. A potential limitation of IFL_1 is that it may have insufficient execution data.

b) ***Multi-Failure Triggering*** *(IFL_f):* IFL_f reranks bug locations after every test failure. Different from IFL_1, IFL_f calculates ranking multiple times if a bug causes multiple test cases to fail. As more failures occur, we can observe how IFL_f ranks locations differently.

c) ***Failure-Pass Triggering*** *(IFL_o):* IFL_o triggers SBFL after the first failure and a few (e.g., 10) extra passed tests. By waiting for additional passed tests to finish their execution, IFL_o gathers more coverage data than IFL_1.

d) ***Frequency-Based Triggering*** *(IFL_p):* IFL_p triggers SBFL at a user-specified 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, IFL_p may waste time to unnecessarily rerank locations.

e) ***Complete Execution-Based Triggering*** *(IFL_c):* IFL_c ranks locations after all test cases are executed. IFL_c corresponds to the existing triggering mode of all SBFL approaches. It serves as a baseline for us to decide how partial coverage information helps with fault localization.

Among the five modes mentioned above, IFL_1, IFL_f, IFL_o, and IFL_p essentially apply SBFL to different subsets of the complete execution data. By experimenting with these variant approaches, we intended to explore their separate trade-offs between diagnosis effectiveness and runtime overhead. Note that in our study, all tests were executed sequentially in a fixed order. The execution ordering was decided by the test executor (i.e., maven-clover plugin). No matter what triggering mode or which SBFL formula is in use, the execution ordering remains the same. This naturally fixed execution ordering ensures that we always observe deterministic results.

This study investigates the following two research questions:

RQ1 How sensitive is IFL to different triggering modes?

RQ2 How sensitive is IFL to the leveraged SBFL formulas?

IV. Experiments

This section first introduces our data sets (Section IV-A) and evaluation metrics (Section IV-B). Then it presents the effectiveness comparison between different triggering modes (Section IV-C). Finally, it explains our exploration of IFL's sensitivity to the used SBFL formulas (Section IV-D).

A. Data Sets

For evaluation, we used the bug data from a closed-source project in industry and four open-source projects.

Bug Data in The Closed-Source Software We chose the closed-source 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

TABLE II: The number of failed tests triggered by different injected or real bugs in closed-source software

# 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

industry and may have different program features from open-source projects. We constructed 2 bug sets, 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. As shown in Table II, there are 16 injected bugs and 6 real bugs that fail single tests. Each of the remaining bugs fail at least two tests. These bugs are related to either unchecked exceptions (e.g., `NullPointerException`), erroneous computation logic, invalid date/time format, wrongly used variable/data/function, or faulty `if`-conditions.

To inject the 28 bugs, we first consulted with developers concerning what are the frequent bugs and usual bug 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 did not use mutation testing to generate buggy programs for two reasons. First, the generated mutants may be very different from real bugs [14]. 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, our manually injected bugs are more likely to reflect real bugs. In the future, we would also like to use mutation testing to generate buggy mutants and explore our research questions accordingly.

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

Bug Data in The Open-Source Software Defects4J [10] is a third-party database of real faults, which include (1) the real bugs from open-source projects and (2) corresponding test suites to demonstrate buggy program behaviors. Due to the time limit, we randomly picked four Java projects from Defects4J: jfreechart, commons-lang, commons-math, and mockito. By removing deprecated and malformed bug data from Defects4J, we included 57 single-fault bugs into our evaluation. In Table III, **LOC Executed** shows the code size ranges among buggy program versions that are covered by test execution; **# of Tests** shows the number of tests executed for each buggy program; **Index of The 1^{st} Failed Test** presents the index of initial test failure among the tests. Each bug corresponds to a single buggy line of code.

TABLE III: The 57 real bugs from 4 open-source projects that were included in our evaluation

Project Name	# of Bugs	Lines of Code (LOC) Executed	# of Tests	Index of The 1^{st} Failed Test
jfreechart	10	25–7,057	1–428	1–248
commons-lang	14	189–2,817	7–198	1–149
commons-math	25	68–7,036	3–1,513	1–599
mockito	8	931–4,252	25–1,111	9–273

B. Effectiveness Metrics

There are three widely used metrics to measure the effectiveness of fault localization approaches [15], [3].

Recall at Top N (Top-N) measures the percentage of buggy entities included in the top N (N = 1, 5, ...) ranked locations. For instance, suppose that there is only one actually buggy entity e_b and it is ranked at the third place. Then the Top-1 recall rate is 0/1=0%, because e_b is not ranked as top one. The Top-5 recall rate is 1/1=100%, because e_b is covered by the top five places. Intuitively, the higher Top-N recall, the better.

Mean Average Precision (MAP) calculates the mean of average precision values among a set of fault localization tasks. The higher value, the better. The **Average Precision (AP)** of one task is defined as:

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

Suppose that given a fault localization task, M statements are retrieved and only one of them is positive (i.e., buggy). Then in the formula, **the number of positive instances** is equal to 1. k varies from 1 to M. For each value of k, $P(k)$ 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 retrieved, and the 3^{rd} and 4^{th} are positive, then AP is $(\frac{1}{3} + \frac{2}{4})/2 * 100\% = 42\%$.

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

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

where $rank_{best}$ is the rank of the first actual bug located. For example, for a given task, if four statements are retrieved, and the 3^{rd} and 4^{th} are buggy, then RR is $\frac{1}{3} * 100\% = 33\%$.

Given a task, if there is only one actual bug location, then $AP = RR$. Similarly, given a set of tasks, if each task has only one actual bug location as the ground truth, $MAP = MRR$. In our data sets, because there is only one known single-line bug to locate in each task, we only reported MAP values.

C. Comparison between Triggering Modes

To compare the fault localization effectiveness of different triggering modes, we used the Ample formula [9] as the

TABLE IV: Comparison between IFL_1 and IFL_c in terms of effectiveness and runtime overhead on the closed-source data

Mode	Top-1 (%)		Top-5 (%)		MAP (%)		Time Cost (second)	
	I	R	I	R	I	R	I	R
IFL_1	**64**	**62**	86	**92**	**73**	**76**	135	105
IFL_c	57	31	**89**	69	70	50	663	655

TABLE V: Comparison between IFL_1 and IFL_c on the open-source bug data

Mode	Top-1 (%)	Top-5 (%)	MAP (%)	Portion of Tests Executed (%)
IFL_1	**14**	**32**	**22**	47
IFL_c	12	26	19	100

default ranking formula in IFL. This is because our another experiment on (see Section IV-D) shows that Ample generally achieved good trade-off among Top-1, Top-5, and MAP values.

1) Effectiveness of IFL_1 and IFL_c: Table IV presents results by IFL_1 and IFL_c on the 41 bugs from closed-source software, where the highest value of each effectiveness measurement is **bolded**. Because each of these two modes triggers SBFL only once during the whole execution of any program, the table has only one row to report the average effectiveness measurements for each mode. As shown in the table, IFL_1 outperformed IFL_c by executing fewer tests and locating bugs more effectively. Specifically with the injected bugs, IFL_1 spent on average 135 seconds and acquired 64% Top-1, 86% Top-5, and 73% MAP values; however, IFL_c spent on average 663 seconds and obtained 57% Top-1, 89% Top-5, and 70% MAP values. Additionally, with the real bugs, IFL_1 spent 105 seconds and got 62% Top-1, 92% Top-5, as well as 76% MAP values; IFL_c spent 655 seconds but acquired 31% Top-1, 69% Top-5, and 50% MAP values.

Table V shows results by IFL_1 and IFL_c on the 57 bugs from open-sourced projects. **Portion of Tests Executed (%)** presents the average percentage of tests executed by each mode. Unsurprisingly, IFL_c executes 100% of tests because it triggers SBFL only after all test execution. Similar to what we observed in Table V, IFL_1 outperformed IFL_c by locating bugs more effectively and executing a lot fewer tests.

Finding 1: *Compared with IFL_c, IFL_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 improve fault localization.*

2) Effectiveness of IFL_f: In our closed-source data set, there are six injected bugs and four real bugs that fail at least four test cases. To evaluate the effectiveness of IFL_f, we triggered SBFL after 1–4 failed tests and reported the average measurements among these multi-failure bugs in Table VI. Hypothetically, as the number of failed tests increases, more execution information is collected and SBFL may work better. However, Table VI shows that the effectiveness measurements often do not increase with the number of failed tests. In particular, as injected bugs triggered more failures, 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 IFL_f increased from 151 seconds to 155 seconds for injected bugs, and increased from 106 seconds to 113 seconds for real bugs.

TABLE VI: IFL_f's effectiveness when IFL_f reranked locations after 1–4 test failures for the closed-source software

# 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

TABLE VII: IFL_f's effectiveness when IFL_f reranked locations after 1–2 test failures for the open-source software

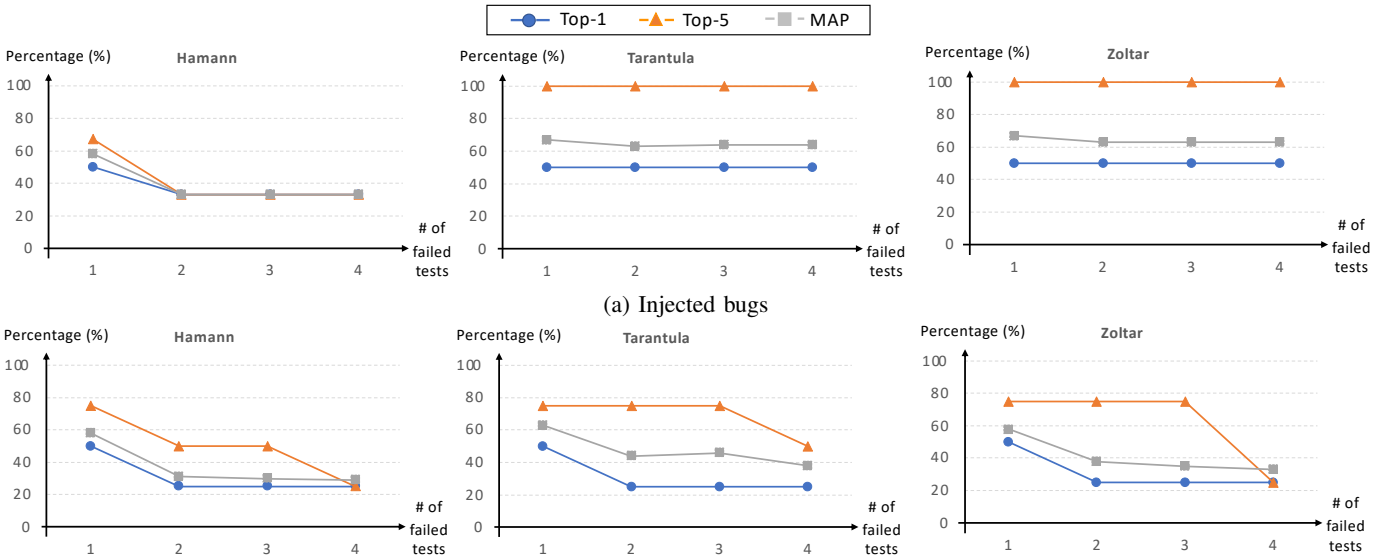
# of Failed Tests	Top-1 (%)	Top-5 (%)	MAP (%)	Portion of Tests Executed (%)
1	25	25	26	26
2	25	25	25	58

To validate the generality of our observation, we redid the experiment with IFL_f by using the other 24 SBFL formulas. As some formulas produced exactly the same results (e.g., Hamann and Sokal), due to the space limit, in Fig. 3, we show IFL_f's effectiveness measurements for three representative formulas: Hamann, Tarantula, and Zoltar. Please refer to our project website for the results by all formulas. According to Fig. 3, IFL_f's measurements did not increase with the number of failed tests. 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 but dropped considerably for real bugs. One possible reason is that although the occurrence of more failed tests can strengthen the suspiciousness signals of bug locations, the existence of a lot more passed tests can weaken those signals and even harm IFL_f's effectiveness in some cases.

Among the 57 studied bugs for open-source projects, there are 8 bugs that fail at least 2 tests. We triggered SBFL after 1–2 failed tests and present the average measurements among these multi-failure bugs in Table VII. Similar to what we observed in Table VI, IFL_f's effectiveness decreased slightly as the number of test failures increased.

Finding 2: *Compared with IFL_1, IFL_f incurred more runtime overhead by profiling more execution and ranking locations multiple times. However, IFL_f does not work better when more tests fail, due to the extreme imbalance between passed and failed tests.*

3) Effectiveness of IFL_o: Tables VIII and IX present our evaluation results for IFL_o, which triggers SBFL after the initial test failure together with 1-10 additional passed tests. As shown in Table VIII, 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, IFL_o should localize bugs more effectively. However, different from our expectation, all effectiveness measurements decreased except



(a) Injected bugs

(b) Real bugs

Fig. 3: IFL_f's effectiveness when different formulas were used (on the closed-source data)

TABLE VIII: IFL_o's effectiveness when 1–10 more passed tests were also included for the closed-source software

# 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

TABLE IX: IFL_o's effectiveness when 1–10 more passed tests were included for 39 bugs in open-source projects

# of Additional Passed Tests	Top-1 (%)	Top-5 (%)	MAP (%)	Portion of Tests Executed (%)
1	15	28	22	37
2	15	28	22	38
3	15	28	22	40
4	15	28	22	41
5	15	26	21	42
6	15	26	21	44
7	15	26	21	45
8	15	26	21	46
9	15	26	21	48
10	15	26	21	49

for the Top-5 of injected bugs.

One possible reason to explain the unexpected trend is the leveraged Ample formula: $Ample = |\frac{e_f}{e_f+n_f} - \frac{e_p}{e_p+n_p}|$. After the initial test failure, there are two types of locations that are likely to be highly ranked:

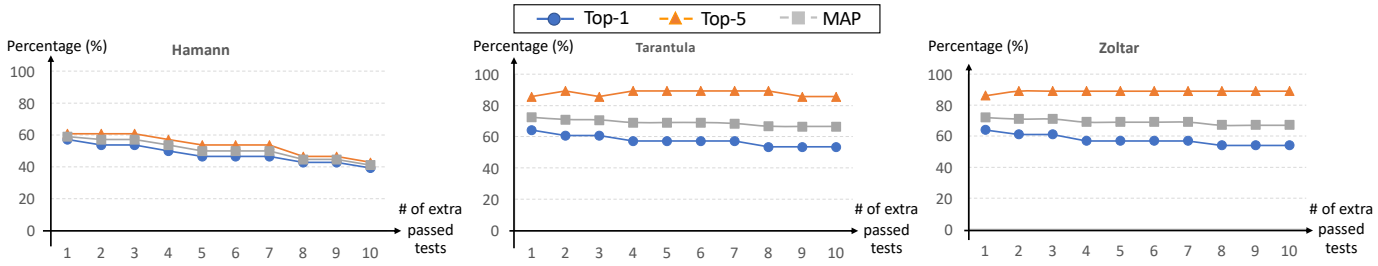
- **Type-I (fail-dominant)** statements that are covered by the failed test but rarely covered by any passed test (e.g., $\frac{1}{2} = \frac{e_f}{e_f+n_f} >> \frac{e_p}{e_p+n_p}$),
- **Type-II (pass-dominant)** statements that are covered by many passed tests but not covered by the failed test (e.g.,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

For better bug 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 IFL_o. Notice that in all the experiments shown in Table VIII, IFL_o achieved better trade-offs than IFL_c, which fact also evidences that the execution profile of more passed tests usually does not help improve fault localization effectiveness.

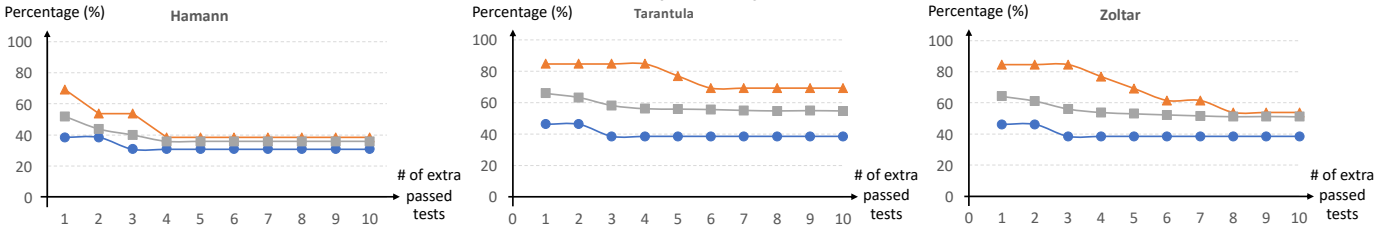
For generality, we also configured IFL_o to use all of the remaining 24 ranking formulae (besides Ample). Due to the space limit, here we only visualize IFL_o's effectiveness with three representative SBFL formulas: Hamann, Tarantula, and Zoltar. Fig. 4 (a) and Fig. 4 (b) separately present the measurements 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 observed a typical trend: the effectiveness of IFL_o does not increase with the number of extra passed tests.

Among the 57 bugs from open-source projects, there are 39 bugs whose initial test failures are followed by at least 10 passed tests. Thus, we evaluated the effectiveness of IFL_o with these bugs and presented results in Table IX. Similar to what we observed in Table VIII, the extra passed tests do not help improve IFL_o's effectiveness.

Finding 3: *Compared with IFL_1, IFL_o 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.*

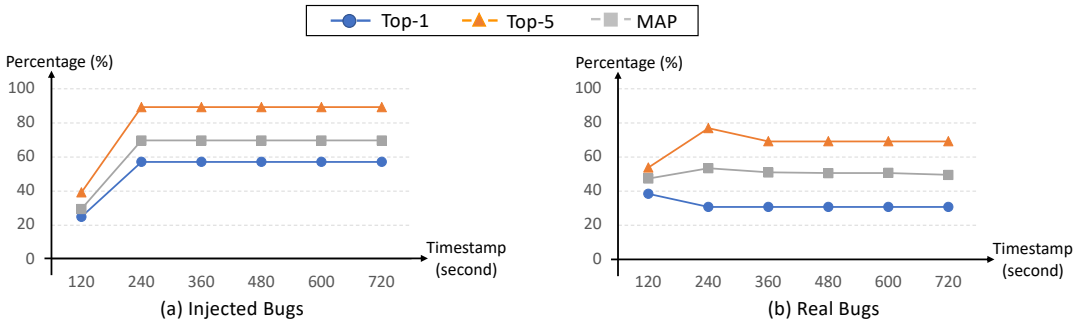


(a) Injected bugs



(b) Real bugs

Fig. 4: IFL_o's effectiveness when different formulas were used based on the closed-source data



(a) Injected Bugs (b) Real Bugs

Fig. 5: IFL_p's effectiveness when SBFL is triggered every two minutes for the closed-source project

4) Effectiveness of IFL_p: Fig. 5 illustrates IFL_p's effectiveness when the ranking of suspicious locations was updated every 2 minutes (i.e., 120 seconds). 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, IFL_p 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 IFL_p. As a result, IFL_p 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 IFL_p's effectiveness.

By comparing IFL_p's peak performance against that of IFL_1, we found that IFL_1 worked better even though IFL_p profiled more data and ranked locations repetitively. Similar to what we observed for IFL_f, the extra execution data and reranking effort did not enable IFL_p to outperform IFL_1. Additionally, IFL_p seems to outperform IFL_c at certain time points (e.g., 240^{th} second) before the completion of whole-suite execution. We did not explore how IFL_p works if the ranking is updated at a different frequency (e.g., every one minute). However, since different programs and distinct bugs may produce test failures at different time points, it is almost impossible to conclude what is the best frequency to trigger SBFL for IFL_p and at which triggering point IFL_p is likely to work best. IFL_p works less flexibly than IFL_1 mainly because it blindly triggers SBFL periodically without considering the execution status of tests.

Finding 4: *Similar to IFL_f and IFL_o, IFL_p worked worse than IFL_1 even if it leveraged more execution data and dynamically computed ranking more often.*

Among the five investigated variants, IFL_1 worked best in terms of efficiency and effectiveness. Although IFL_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 IFL_f and IFL_o worked less effectively than

TABLE X: The effectiveness of IFL_1 and IFL_c when different formulas were applied to bugs of the closed-source software

Variants of IFL_1	Top-1 (%)		Top-5 (%)		MAP (%)		Variants of IFL_c	Top-1 (%)		Top-5 (%)		MAP (%)	
	I	R	I	R	I	R		I	R	I	R	I	R
IFL_1-Ample	64	**62**	**86**	**92**	**73**	**76**	IFL_c-Ample	57	31	**89**	**69**	**70**	50
IFL_1-Anderberg	64	**62**	**86**	85	**73**	74	IFL_c-Anderberg	57	38	86	**69**	68	53
IFL_1-Dice	64	**62**	**86**	85	**73**	74	IFL_c-Dice	57	38	86	**69**	68	53
IFL_1-Euclid	**68**	54	68	54	68	54	IFL_c-Euclid	**68**	**69**	68	**69**	68	**69**
IFL_1-Goodman	64	**62**	**86**	85	**73**	74	IFL_c-Goodman	57	38	86	**69**	68	53
IFL_1-Hamann	57	54	61	69	59	60	IFL_c-Hamann	39	31	43	38	41	36
IFL_1-Hamming	**68**	54	68	54	68	54	IFL_c-Hamming	**68**	**69**	68	**69**	68	**69**
IFL_1-Jaccard	64	**62**	**86**	85	**73**	74	IFL_c-Jaccard	57	38	86	**69**	68	53
IFL_1-Kulczynski1	64	**62**	**86**	85	**73**	74	IFL_c-Kulczynski1	57	38	86	**69**	68	53
IFL_1-Kulczynski2	0	0	0	0	0	0	IFL_c-Kulczynski2	0	0	0	0	0	0
IFL_1-M1	57	54	61	69	59	60	IFL_c-M1	39	31	43	38	41	36
IFL_1-M2	64	54	**86**	**92**	**73**	71	IFL_c-M2	57	31	**89**	**69**	**70**	49
IFL_1-Ochiai	64	**62**	**86**	85	72	74	IFL_c-Ochiai	57	38	86	**69**	68	54
IFL_1-Ochiai2	64	**62**	**86**	85	72	72	IFL_c-Ochiai2	57	38	86	62	68	53
IFL_1-Overlap	0	0	0	0	0	0	IFL_c-Overlap	0	0	0	0	0	0
IFL_1-RogersTanimoto	57	54	61	69	59	60	IFL_c-RogersTanimoto	39	31	43	38	41	36
IFL_1-RussellRao	64	54	**86**	85	**73**	70	IFL_c-RussellRao	57	31	**89**	**69**	**70**	49
IFL_1-SimpleMatching	57	54	61	69	59	60	IFL_c-SimpleMatching	39	31	43	38	41	36
IFL_1-Sokal	57	54	61	69	59	60	IFL_c-Sokal	39	31	43	38	41	36
IFL_1-SϕrensenDice	64	**62**	**86**	85	**73**	74	IFL_c-SϕrensenDice	57	38	86	**69**	68	53
IFL_1-Tarantula	64	**62**	**86**	85	72	73	IFL_c-Tarantula	54	38	79	**69**	65	53
IFL_1-Wong1	**68**	54	68	54	68	54	IFL_c-Wong1	**68**	**69**	68	**69**	68	**69**
IFL_1-Wong2	61	54	68	69	64	62	IFL_c-Wong2	46	31	57	54	52	40
IFL_1-Wong3	61	54	71	77	65	65	IFL_c-Wong3	50	31	68	62	59	45
IFL_1-Zoltar	64	**62**	**86**	85	72	72	IFL_c-Zoltar	54	38	86	54	66	51

IFL_1 but better than IFL_c. The execution profile of additional failed tests and/or passed tests usually does not help improve bug localization, but can harm the effectiveness in most cases. Finally, IFL_p did out outperform IFL_c. By triggering SBFL at a fixed fixed frequency, IFL_p might invoke SBFL so early that no test failure was available to help locate bugs, or invoke SBFL so late that too many passed tests were already executed and could weaken the signals raised by any test failure.

Finding 5: *IFL_1 generally worked better than other variants. Our comparison provides two insights. First, more execution data might not necessarily lead to better fault localization results. Second, the first failed test is often more important than passed tests and any later failed test for fault localization.*

D. Sensitivity to SBFL Formulas

We ran IFL with all 25 alternative formulas listed in Table I (1) to explore IFL's sensitivity to the adopted formulas and (2) to ensure the generalizability of our observation that IFL_1 outperforms IFL_c (see Section IV-C). 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.

Table X presents the results by IFL_1 and IFL_c when they used distinct formulas to locate bugs of the closed-source software. According to the table, in the experiments with IFL_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 IFL_c, 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 IFL 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.

In Table X, if we compare IFL_1 with IFL_c for each formula, we observe that IFL_1 outperformed IFL_c in the majority of scenarios. For instance, when Anderberg was used, IFL_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 IFL_c 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.

IFL_c only outperformed IFL_1 when one of the following three formulas was used: Euclid, Hamming, and Wong1. Specifically, when Hamming was in use, both IFL_c and IFL_1 achieved 68% for all metrics on the injected bug set; on the real bug set, IFL_c acquired 69% for all metrics while IFL_1 obtained 54%. Finally, when Kulczynski2 and Overlap were in use, both IFL_1 and IFL_c worked equally poorly, probably because these two formulas are not effective to locate bugs.

Additionally, we applied IFL to the 57 bugs by configuring it to use 25 alternative formulas. Please find the detailed results on our project website. According to this experiment, both IFL_1 and IFL_c are sensitive to the adopted formulas,

although some formulas led to exactly the same results. The 25 formulas can be actually divided into 7 groups based on their result mappings. Two of these groups (i.e., eight formulas) enabled IFL_1 to outperform IFL_c; three groups (i.e., five formulas) enabled IFL_1 to work equally well with IFL_c even though IFL_1 executed a lot fewer tests; and two groups (i.e., 12 formulas) enabled IFL_c to achieve slightly higher MAP values. There are a group of formulas (e.g., 11 formulas) that produced better results than Ample. For instance, IFL_1 acquired 16% Top-1, 42% Top-5, and 27% MAP when using Ochiai, but obtained 14% Top-1, 32% Top-5, and 22% MAP when using Ample. This observation corroborates Yoo et al.'s finding [16] that there is no optimal formula that always outperforms others.

Finding 6: *Among the 25 investigated SBFL formulas, IFL_1 worked equally well with or even outperformed IFL_c for the majority of formulas.*

V. Related Work

The related work of our research includes spectrum-based fault localization (Section V-A), information retrieval-based fault localization (Section V-B), empirical studies on fault localization approaches (Section V-C), and test reduction, prioritization, as well as generation (Section V-D).

A. Spectrum-Based Fault Localization (SBFL)

SBFL techniques identify bug locations using the execution information of buggy code [17], [18], [19], [20], [9], [12]. 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 [2]. Xuan et al. used machine learning to train a model by combining 25 suspiciousness calculation formulae [12]. 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 [21], [22], [23]. 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 [21]. 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.

This paper does not define any new SBFL formula. Instead, we reused 25 existing SBFL formulas. We built a framework—IFL—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 IFL's effectiveness given (1) different SBFL formulas and (2) distinct triggering mechanisms for SBFL formulas. By revealing bug locations early, IFL_1 turned out to achieve the best trade-off between effectiveness and efficiency.

B. Information Retrieval-Based Fault Localization (IBFL)

IBFL approaches locate bugs based on bug reports [15], [3], [24], [25]. For example, BLUiR treats a bug report as a document query and considers source code as documents [3]. 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 [24].

One limitation of IBFL 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 IBFL with other approaches [4], [26]. For instance, Dao et al. combined IBFL with SBFL by assigning different weights to the separately generated ranked lists [4]. Zou et al. combined IBFL with another six kinds of techniques: SBFL, mutation-based fault localization, dynamic program slicing [27], stack trace analysis [28], predicate switching [29], and history-based fault localization [30]. 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, IFL does not rely on the existence of any bug report, neither does it require for the execution of all test cases.

C. Empirical Studies on Fault Localization Techniques

Researchers empirically studied fault localization techniques in various ways [31], [32], [7], [33], [34]. Specifically, Lucia et al. [31] and Yoo et al. [32] compared different formulae defined for SBFL approaches, and concluded that there was no optimal formula that always worked better than others. Kochhar et al. [33] and Dao et al. [4] independently manually inspected bug reports whose bugs were either fully, partially, or not localized by IBFL approaches. They found that the quality of bug reports can substantially impact IBFL results. If bug reports explicitly contain buggy file names, IBFL 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 IBFL tools [7]. The study revealed that developers did not find such tools to be quite useful and were unsatisfied by IBFL 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 IFL 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 IFL, and design better fault localization approaches accordingly.

D. Test Reduction, Prioritization, or Generation

Some approaches were proposed to reduce, prioritize, or generate test cases in order to facilitate fault localization [35], [36], [16], [37]. 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 [37]. 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 [35]. 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-theoretic approach to prioritize statements and test cases [16]. 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 [36]. Specifically, they defined a *similarity criterion*, which is used to measure how the execution characteristics of two tests are similar to each other. The criterion is also used to direct concolic execution to generate 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.

VI. Threats to Validity

a) Threats to External Validity: Our experiments were conducted based on two bug sets for a closed-source project and one bug set from four open-sourced projects. The evaluation results may not generalize well to other bugs, other software products of other companies, or other 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.

b) Threats to Construct Validity: Among the investigated bugs, each bug can be fixed with a single-line change and the corresponding faulty program version contains a single known 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 some of these limitations with prior work [2], [17], [38], [21]. In the future, we will diversify our approach to generate injected bugs and use more complicated real bugs to better propose and evaluate fault localization techniques.

c) Threats to Internal Validity: Similar to prior fault localization research [12], [21], [26], 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.

VII. Conclusion

This paper presents our exploration on the distinct triggering modes 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?* We explored this problem in this study.

Specifically, we built IFL, a framework that triggers SBFL in five alternative modes: triggering SBFL right after the initial failure (IFL_1), after every failed test (IFL_f), after the initial failure and several extra failed tests (IFL_o), at a fixed frequency (IFL_p), or after the execution of all tests (IFL_c). By comparing the Top-1, Top-5, and MAP values achieved by different triggering modes when SBFL techniques were applied to distinct datasets, we surprisingly found that IFL_1 worked better or at least equally well with other variants in most scenarios. Namely, triggering SBFL right after the initial test failure turns out to be often more effective and/or more efficient than triggering SBFL later.

Our empirical study indicates that it is not always necessary to execute all test cases before using SBFL formulas to locate bugs. IFL_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, IFL_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.

Acknowledgment

We thank reviewers for their insightful comments, and thank the Cvent management team for their support. We also thank Hanwen Liu for his help on experiment data. This work was partially supported by NSF CCF-1845446.

REFERENCES

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

[2] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477. [Online]. Available: http://doi.acm.org/10.1145/581339.581397

[3] R. Saha, M. Lease, S. Khurshid, and D. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 345–355.

[4] T. Dao, L. Zhang, and N. Meng, "How does execution information help with information-retrieval based bug localization?" in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 241–250. [Online]. Available: https://doi.org/10.1109/ICPC.2017.29

[5] H. Souza, D. Mutti, M. Chaim, and F. Kon, "Contextualizing spectrum-based fault localization," *Information and Software Technology*, vol. 94, 10 2017.

[6] 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)*, July 2017, pp. 114–125.

[7] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771797

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

[9] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, 2005.

[10] R. Just, D. Jalali, and M. 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*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[11] "Defects4J," http://fault-localization.cs.washington.edu, 2020.

[12] 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*, Sept 2014, pp. 191–200.

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

[14] 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*, Nov 2014, pp. 189–200.

[15] J. Zhou, H. 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*, June 2012, pp. 14–24.

[16] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, 07 2013.

[17] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101949

[18] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000791.2000795

[19] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Proceedings of the 4th International Conference on Search Based Software Engineering*, ser. SSBSE'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 244–258.

[20] R. Abreu, P. Zoeteweij, and A. 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*, Sept 2007, pp. 89–98.

[21] X. Li, M. d'Amorim, and A. Orso, *Iterative User-Driven Fault Localization*. Cham: Springer International Publishing, 2016, pp. 82–98.

[22] L. Gong, H. Zhang, L. Jiang, and D. Lo, "Interactive fault localization leveraging simple user feedback," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 67–76. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2012.6405255

[23] 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)*, Dec 2006, pp. 277–284.

[24] X. Ye, R. Bunescu, and C. 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*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 689–699. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635874

[25] 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 *ASE*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE, 2015, pp. 476–481.

[26] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[27] 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*, Oct 1995, pp. 143–151.

[28] 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*, Sep. 2014, pp. 181–190.

[29] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 272–281. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134324

[30] 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)*, May 2007, pp. 489–498.

[31] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Sept 2010, pp. 1–10.

[32] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist," University College London and Swinburn University, Tech. Rep., 2014.

[33] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: Do they matter?" in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 803–814. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642997

[34] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug 2016.

[35] 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*, 2008, pp. 201–210.

[36] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 49–60. [Online]. Available: https://doi.org/10.1145/1831708.1831715

[37] W. Masri and R. Abou Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, 02 2014.

[38] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 528–545, July 2010.