Improving Fault Localization with External Oracle by Using Counterfactual Execution

JONGCHAN PARK, FuriosaAI, Seoul, Korea
TAE EUN KIM, KAIST, Daejeon, Korea
DONGSUN KIM, Kyungpook National University, Daegu, Korea
KIHONG HEO, KAIST, Daejeon, Korea

We present Flex, a new approach to improve fault localization with external oracles. Spectrum-based fault localization techniques estimate suspicious statements based on the execution trace of the test suite. State-of-the-art techniques rely on test oracles that *internally* exist in the program. However, programs often have *external* oracles that observe their behavior from outside. This in turn hinders fine-grained and accurate estimation of suspicious statements in practice because the correctness of each execution can only be observed at termination. In this article, we aim to address this problem by observing counterfactual execution traces, which enable fine-grained estimation even without precise internal oracles. We observe two types of counterfactual scenarios related to different types of test cases: When the branch condition is set to a Boolean constant, (1) if most of the passing test cases still pass, we consider the newly executed statements in the branch statement as *unrelated* to the failure; (2) if failing test case still fails, we also consider the originally executed statements as *unrelated* to the failure. We evaluated the performance on widely used C and Java programs. Flex improves the accuracy of state-of-the-art SBFL techniques on C and Java programs by 24% and 22% on average, respectively.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**;

Additional Key Words and Phrases: Fault Localization, SBFL, Counterfactual Execution

ACM Reference format:
Jongchan Park, Tae Eun Kim, Dongsun Kim, and Kihong Heo. 2025. Improving Fault Localization with External Oracle by Using Counterfactual Execution. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 35 (January 2025), 22 pages.
https://doi.org/10.1145/3695997

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea government (MSIT) (NRF-2021R1A5A1021944 and 2021R1I1A3048013). Additionally, the research was supported by Kyungpook National University Research Fund, 2024.
Authors' Contact Information: Jongchan Park (corresponding author), FuriosaAI, Seoul, Korea; e-mail: jongchan.park@furiosa.ai; Tae Eun Kim, KAIST, Daejeon, Korea; e-mail: taeeun.kim@kaist.ac.kr; Dongsun Kim, Kyungpook National University, Daegu, Korea; e-mail: darkrsw@knu.ac.kr; Kihong Heo, KAIST, Daejeon, Korea; e-mail: kihong.heo@kaist.ac.kr.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2025 Copyright held by the owner/author(s).
ACM 1557-7392/2025/1-ART35
https://doi.org/10.1145/3695997

1 Introduction

Spectrum-based fault localization (SBFL) is a widely used technique for localizing faults in software systems. The intuition behind SBFL is that a program statement is more likely to be faulty if the statement is executed more often in failing executions than in passing executions. There have been many formulas to realize this idea such as Ochiai [1], Tarantula [8, 9], and Dstar [24].

However, not all statements executed in failing executions are faulty. Nonetheless, the accuracy of SBFL is often highly challenged by the presence of program statements that are spuriously correlated to the fault. For example, a correct statement that happens to be executed only in failing executions is unfairly assigned to a high suspiciousness score. Due to the sparse test suite and the limited number of failing test cases in practice, it is difficult to distinguish between a correct statement and a faulty statement by only using the limited execution profile.

Many recent SBFL approaches [13, 23, 29, 31] infer faulty statements by using test oracles that *internally* exist in the program under test in the form of assertions or unit tests. Such internal oracles allow users to observe the internal state of the program such as the output of a function or the value of a variable. For example, many Java libraries in a well-known fault benchmark, Defects4J [10], have a large number of unit tests as internal oracles. Those approaches improve the accuracy of SBFL by using various techniques including dynamic analysis or probabilistic models.

Unfortunately, end-user programs often lack such internal oracles but have *external* oracles, such as an expected output, that observe the behavior of the program from outside. For example, gzip, a widely used compression utility, tests the correctness of its compression and decompression functions by comparing the output of the decompression function with the original input.[1] This in turn hinders fine-grained and accurate fault localization because the internal state of the program is hidden and the correctness of each execution can only be observed at termination after long execution.

To address this problem, we present Flex,[2] a new approach to improve fault localization with external oracles by observing counterfactual execution traces. Flex first selects a set of branch statements in the original program and generates variant programs each of which has the branch predicate set to a Boolean constant. Then, we observe two types of counterfactual scenarios from the variants: (1) if the branch condition is set to a Boolean constant and most of the passing test cases still similarly pass, we consider the newly executed statements in the branch statement as *unrelated* to the failure; (2) if the branch condition is set to a Boolean constant and the failing test case still similarly fails, we also consider the originally executed statements as *unrelated* to the failure. Finally, Flex combines the execution profiles of variant programs and computes the suspiciousness scores for each statement in the original program. Thus, Flex augments insufficient coverage profile with counterfactual executions to improve the accuracy of SBFL, without relying on internal oracles.

We have extensively evaluated the effectiveness of Flex on a large suite of C and Java programs: *65 C programs and 393 Java programs* from the ManyBugs [5], CoreBench [2], and Defects4J [10] benchmarks. The results show that Flex significantly improves the accuracy of state-of-the-art SBFL techniques for the C benchmarks, which do not have internal oracles. The accuracy is improved by 24% on average in line-level and by 24% on average in function-level fault localization. Our results on the Java benchmark show that Flex is also effective when internal oracles are available. In particular, Flex improves the accuracy by 22% compared to SmartFL [29], the state-of-the-art technique using internal oracles. Notably, Flex remains highly effective even when internal oracles are available, especially for complex programs and test cases.

[1] https://git.savannah.gnu.org/cgit/gzip.git/tree/tests/z-suffix
[2] The name comes from *F*ault *L*ocalization with *EX*ternal oracle by using counterfactual execution.

The main contributions of this article are as follows:

— We propose a novel SBFL technique called Flex. This technique aims to improve the accuracy of SBFL with external oracles by using counterfactual executions.
— We present an effective algorithm for collaboratively computing suspiciousness scores from variant programs that have similar behavior to the original program. We also present a method to compute the similarity using approximated oracles.
— We have evaluated Flex on a large number of C and Java benchmarks and demonstrated the performance. The results show that Flex outperforms the state-of-the-art SBFL techniques.
— We publicize Flex's implementation and the experimental data to support open science: https://github.com/prosyslab/flex-artifact.

2 Motivation

2.1 Motivating Example

Figure 1(a) is a fault example designed to effectively illustrate our approach. The program `abs_sum_div` reads three integers x, y, and z from the user input, and stores the result $(|x| + |y|)/z$ into file `result.txt`. The program has a fault at Line 3 that incorrectly handles integer-overflow. Because of this faulty expression, the program produces a wrong result, an overflowed negative number instead of 2, when computing $(\texttt{MAXINT} + \texttt{MAXINT})/\texttt{MAXINT}$.

The program is tested by an external oracle (a bash script) shown in Figure 1(b). The oracle executes the program with different inputs, and checks the results in the output file. The oracle checks the result by comparing the output file with a reference result.

2.2 Background: SBFL

Fault localization is the process of identifying the location of a fault in a program. Recently, SBFL techniques [1, 9, 13, 23, 24, 29, 31] are prevailing as they can identify the fault location effectively and efficiently. These techniques compute a suspiciousness score for each program statement after analyzing the correlation between the execution of a program statement and the program failure. The intuition behind SBFL is that a program statement is more likely to be faulty if it is visited more often by failing executions than passing executions.

While there have been various techniques proposed to improve the accuracy, most of them require internal oracles that exist in the program under test such as assertions or unit tests. Therefore, such techniques cannot be directly applicable to our example. Instead, we use a widely used SBFL technique, Ochiai [1], that is based on code coverage and test results. Ochiai computes the suspiciousness score of a program statement s as follows:

$$\text{Ochiai}(s) = \frac{e_f(s)}{\sqrt{\{e_f(s) + n_f(s)\} \cdot \{e_f(s) + e_p(s)\}}}, \tag{1}$$

where $e_f(s)$ and $e_p(s)$ are the numbers of failing and passing tests that execute the program statement s while $n_f(s)$ is the number of failing tests that do not execute s.

SBFL often assigns high suspiciousness scores to correct statements whose coverage is spuriously correlated to the faulty statement. For example, Lines 20 and 8 in Figure 1(a) are correct statements. However, the statements are assigned high suspiciousness scores because the failing test case happens to cover them and one passing test does not cover them. We need an appropriate tie-breaking mechanism to address this challenge.

```c
1  int sum_div(int x, int y, int z){
2    int sum;
3    sum = x + y; // potential integer overflow
4
5    if (z == 0)
6      return 0;
7    else
8      return sum / z;
9  }
10
11 int main() {
12   int x, y, z, result;
13   x = input(); y = input(); z = input();
14
15   if (x <= 0 || y <= 0){
16     x = abs(x);
17     y = abs(y);
18   }
19   else
20     printf("Both␣are␣positive");
21
22   if (x > MAXINT/2 && y > MAXINT/2)
23     printf("Potential␣overflow!");
24
25   if (x == 0 && y == 0)
26     result = 0;
27   else
28     result = sum_div(x, y, z);
29
30   save_to_file("result.txt", result);
31   return 0;
32 }
```

(a) Program abs_sum_div that reads three integers x, y, and z and stores the result $(|x| + |y|)/z$ into file result.txt.

```bash
1  #!/bin/bash
2
3  # Passing test case 1
4  # Input: 4 2 1
5  # Expected output: 6
6  ./abs_sum_div 4 2 1
7  read line < result.txt
8  [[ $line -eq 6 ]] && exit 0 || exit 1
9
10 # Passing test case 2
11 # Input: 9 3 4
12 # Expected output: 3
13 ./abs_sum_div 9 3 4
14 read line < result.txt
15 [[ $line -eq 3 ]] && exit 0 || exit 1
16
17 # Passing test case 3
18 # Input: 0 0 2
19 # Expected output: 0
20 ./abs_sum_div 0 0 2
21 read line < result.txt
22 [[ $line -eq 0 ]] && exit 0 || exit 1
23
24 # Failing test case 1
25 # Input: MAXINT MAXINT MAXINT
26 # Expected output: 2
27 ./abs_sum_div MAXINT MAXINT MAXINT
28 read line < result.txt
29 [[ $line -eq 2 ]] && exit 0 || exit 1
```

(b) The external oracle (a bash script) that executes abs_sum_div and checks the results in the output file.

Fig. 1. Motivating example.

2.3 Challenges: External Oracles

However, SBFL techniques often produce inaccurate results. Table 1 shows the suspiciousness scores for each line in the example using the Ochiai formula (Equation (1)). Note that the faulty statement at Line 3 is ranked at the 6th position among 16 lines.[3] More precisely, the suspiciousness score of the faulty statement 3 is 0.57, while a correct line, Line 8 has the same score of 0.57.

While recently proposed techniques claim to address the above challenge, they highly rely on *internal* oracles such as assertions or unit tests [29]. Based on the internal oracles, they can precisely analyze the correlation between the statements and test failure by using various methods such

[3] There are various ways to address ties in fault localization [26]. However, we use a Naïve approach to rank the faulty statement at the lowest position among the tied lines.

Table 1. The Execution Profile of the Program Listed in Figure 1

Line	$\textbf{Pass}_1$	$\textbf{Pass}_2$	$\textbf{Pass}_3$	$\textbf{Fail}_1$	Score	Rank
3 (fault)	●	●	-	●	0.57	6 (5)
5	●	●	-	●	0.57	6 (5)
6	-	-	-	-	0.00	16 (4)
8	●	●	-	●	0.57	6 (5)
13	●	●	●	●	0.50	12 (6)
15	●	●	●	●	0.50	12 (6)
16	-	-	●	-	0.00	16 (4)
17	-	-	●	-	0.00	15 (3)
20	●	●	-	●	0.57	6 (5)
22	●	●	●	●	0.50	12 (6)
23	-	-	-	●	1.00	1 (1)
25	●	●	●	●	0.50	12 (6)
26	-	-	●	-	0.00	16 (4)
28	●	●	-	●	0.57	6 (5)
30	●	●	●	●	0.50	12 (6)
31	●	●	●	●	0.50	12 (6)
Result	✓	✓	✓	✗		

Columns $\textbf{Pass}_n$ and $\textbf{Fail}_n$ represent the coverage of the nth passing and failing tests. In the column **Score**, each line is assigned a suspiciousness score computed by the Ochiai formula. Column **Rank** shows the rank of each line. The number in the parenthesis is the number of lines that have the same score.

as static analysis, dynamic analysis, or probabilistic models. Suppose there exists a unit test that checks the correctness of the `sum_div` function: `assert(sum_div(MAXINT,MAXINT,MAXINT) == 2)`. Then, one can precisely estimate that Line 3 is a faulty statement because the only statement that potentially produces a negative number is the overflowed addition.

Unfortunately, programs often lack such internal oracles but have *external oracles* such as the shell script shown in Figure 1(b). Such external oracles check the behavior of the program under test by comparing various information observable from outside. For example, gzip, a file compression utility, may have a test script that the original test file is preserved after compression and decompression.

3 Approach

We propose a novel approach, Flex, to address these challenges. Flex is based on two ideas: (1) counterfactual execution and (2) approximated oracle. The intuition behind Flex is as follows:

— Suppose a branch condition (e.g., Line 15) is set to a Boolean constant (e.g., False) that is the same value as in the failing executions. Then, all the passing test cases will cover the statements in the corresponding branch in this variant program. If most of the passing test cases still *similarly* pass in this counterfactual execution, the newly executed statements (e.g., Line 20) in the branch are likely to be unrelated to the failure.
— Similarly, suppose a branch condition (e.g., Line 5) is set to a Boolean constant (e.g., True) that is the flipped value as in the failing executions. Then, all the failing test cases will cover the statements in the opposite branch in this variant program. If all the failing test cases still

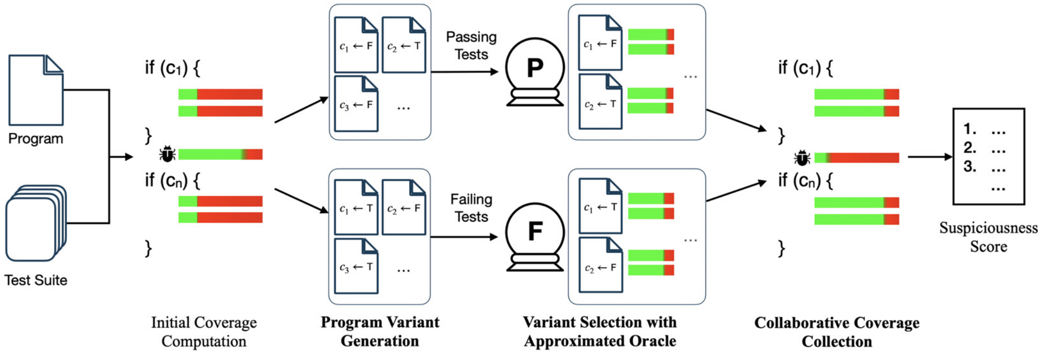

Fig. 2. Overview of FLEX.

similarly fail, the no longer executed statements (e.g., Line 8) in the branch are likely to be unrelated to the failure.

FLEX has three major steps other than ordinary SBFL techniques as shown in Figure 2: (1) program variant generation, (2) program variant selection, and (3) collaborative coverage collection. We describe each step in detail in the rest of this section.

3.1 Program Variant Generation for Counterfactual Execution

First, we generate program variants with counterfactual assumptions. FLEX identifies all conditional statements covered by the failing test cases. In the example shown in Figure 1, all the conditional statements at Lines 15, 22, 25, and 5 are selected. Then, we derive two sets of candidate variants $\mathcal{P}_{pcand} = \{P_{\overline{15}}, P_{22}, P_{\overline{25}}, P_{\overline{5}}\}$ and $\mathcal{P}_{fcand} = \{P_{15}, P_{\overline{22}}, P_{25}, P_5\}$ of program variants. We denote P_L and $P_{\overline{L}}$ as the program variants whose branch conditions at line L are set to True and False, respectively. For example, the variant $P_{\overline{15}}$ of the example is the same as the original one except that the condition of Line 15 is set to False. Each program in $\mathcal{P}_{pcand}$ is derived by changing each branch condition to be the same value as in the failing execution. This set of programs will be used for counterfactual executions with the passing test cases. Similarly, each program in $\mathcal{P}_{fcand}$ is derived by changing each branch condition to be the inverted value as in the failing execution. This set of programs will be used for counterfactual executions with the failing test cases.

3.2 Program Variant Selection Using Approximated Oracles

Next, we select the variants that exhibit similar behavior to the original program. We define two sets $\mathcal{P}_{pass} \subseteq \mathcal{P}_{pcand}$ and $\mathcal{P}_{fail} \subseteq \mathcal{P}_{fcand}$, each of which is a set of variants whose behavior is similar to the original program modulo the passing and failing test cases, respectively. In order to define the similarity, we use an approximated oracle for each set of variants.

The approximated oracle for $\mathcal{P}_{pass}$ checks whether most[4] of the passing test cases still pass with the variant P'. In the example, the variant programs $P_{\overline{15}}$ and P_{22} are selected because they preserve all the passing test cases. On the other hand, the other variant programs are discarded because they both fail with two passing test cases out of three.

The approximated oracle for $\mathcal{P}_{fail}$ checks whether all the failing test cases still fail with the variant P' and the call sequence of the failing test cases is the same to the original one. The reason we check for *all* the failing test cases is that the failing test cases are often limited in number. While programs

[4]In our experiment, we set the threshold to 90%.

often have a number of passing test cases, they often have only a few (typically one or two) failing test cases. The reason we additionally check for the *call sequence* of the failing test case is that the failure reported by the external oracle is often coarse-grained. It is difficult to determine whether the failure of a test is caused by the same reason as the original one. Thus, failing with the external oracle is a necessary but not sufficient condition for preserving the original faulty behavior. In the example, the variant program $P_{\overline{22}}$ and P_5 is selected because the call sequence does not change from the original program. Note that we only consider the call sequence of the user-defined functions (i.e., `abs`, `sum_div`) here in order to avoid the noise from the library functions (e.g., `printf`). In addition to user-defined functions, we also consider the use of macros as a function call for C programs. On the other hand, the other variants are discarded because they call different functions, or even omit originally called functions. For example, P_{25} is discarded because it omits calling the function `sum_div`, which contains the faulty statement.

3.3 Collaborative Coverage Collection

Finally, we collect the collaborative coverage across the selected variants, which is used to compute the suspiciousness score of each statement of the original program. Unlike conventional SBFL which uses a single program, we collaboratively use the execution profile of the selected variants. We aggregate the coverage of the variants into a single coverage profile. Collaborative coverage of the passing test cases is defined as the *average* passing coverage count of each statement across the variants in $\mathcal{P}_{pass}$. The collaborative coverage of the failing test cases is similarly defined with variants from $\mathcal{P}_{fail}$ including the original version, but we use the *minimum* coverage count rather than the average.

The choice of our aggregation scheme is based on the characteristics of the variants. Note that the level of confidence in the selection is different between the two sets, $\mathcal{P}_{fail}$ and $\mathcal{P}_{pass}$. The variants from $\mathcal{P}_{fail}$ are selected by directly comparing the faulty behavior to the original one using the failing test case. Therefore, we can be confident that the selected variants still preserve the faulty behavior and the no longer executed statements are unrelated to the fault. Thus, we aggressively aggregate the information using the minimum. On the other hand, the variants from $\mathcal{P}_{pass}$ are selected based on the similarity of the passing behavior. This is still informative, yet indirect evidence of not being related to the fault. For example, a newly executed statement of a certain variant could be the faulty statement even if the variant preserves the passing behavior. It would be dangerous to ignore the newly executed statement in this case. Thus, we conservatively aggregate the information using the average.

Table 2 shows the summarized execution profile of the program variants of the example in Figure 1. Note that variants $P_{\overline{15}}$ and P_{22} alter the passing coverage count of Lines 20 and 23 from 2 to 3, and from 0 to 3, respectively. Then, the passing test coverage count of Lines 20 and 23 for the collaborative coverage is calculated as 2.5, which is the average of 2 and 3, and 1.5 which is the average of 0 and 3, respectively. For now, let us ignore Lines 16 and 17 because they were not executed by the failing test in the original program. Using the minimum failing coverage count enables Flex to utilize the variants from $\mathcal{P}_{fail}$ more effectively. In Table 2, $P_{\overline{22}}$ and P_5 each alters the failing coverage count of Lines 23 and 8 from 1 to 0. Since we use the minimum failing coverage count, the failing collaborative coverage of Lines 23 and 8 becomes 0. Note that P_5 alters the failing coverage count of Line 6 from 0 to 1. However, since we also consider the original program to compute the minimum failing coverage count, the failing collaborative coverage of Line 6 becomes 0.

Finally, the suspiciousness score is computed with the collaborative coverage. Specifically, the score of Line 20 in the example program, which was 0.57 with Ochiai, becomes 0.53 due to the collaborative passing coverage. The score of Lines 23 and 8, which were also 0.57 with Ochiai,

Table 2. The Summarized Execution Profile of Program Variants

		$\mathcal{P}_{pass}$		$\mathcal{P}_{fail}$				
Line	P_{orig}	$P_{\overline{15}}$	P_{22}	$P_{\overline{22}}$	P_5	**Coll. Cov.**	**Score**	**Rank**
3 (fault)	(2, 1)	(2, -)	(2, -)	(-, 1)	(-, 1)	(2, 1)	0.57	3 (3)
5	(2, 1)	(2, -)	(2, -)	(-, 1)	(-, 1)	(2, 1)	0.57	3 (3)
6	(0, 0)	(0, -)	(0, -)	(-, 0)	**(-, 1)**	(0, 0)	0.00	16 (6)
8	(2, 1)	(2, -)	(2, -)	(-, 1)	**(-, 0)**	**(2, 0)**	**0.00**	16 (6)
13	(3, 1)	(3, -)	(3, -)	(-, 1)	(-, 1)	(3, 1)	0.50	10 (6)
15	(3, 1)	(3, -)	(3, -)	(-, 1)	(-, 1)	(3, 1)	0.50	10 (6)
16	(1, 0)	**(0, -)**	(1, -)	(-, 0)	(-, 0)	**(0.5, 0)**	0.00	16 (6)
17	(1, 0)	**(0, -)**	(1, -)	(-, 0)	(-, 0)	**(0.5, 0)**	0.00	16 (6)
20	(2, 1)	**(3, -)**	(2, -)	(-, 1)	(-, 1)	**(2.5, 1)**	**0.53**	4 (1)
22	(3, 1)	(3, -)	(3, -)	(-, 1)	(-, 1)	(3, 1)	0.50	10 (6)
23	(0, 1)	(0, -)	**(3, -)**	**(-, 0)**	(-, 1)	**(1.5, 0)**	**0.00**	16 (6)
25	(3, 1)	(3, -)	(3, -)	(-, 1)	(-, 1)	(3, 1)	0.50	10 (6)
26	(1, 0)	(1, -)	(1, -)	(-, 0)	(-, 0)	(1, 0)	0.00	16 (6)
28	(2, 1)	(2, -)	(2, -)	(-, 1)	(-, 1)	(2, 1)	0.57	3 (3)
30	(3, 1)	(3, -)	(3, -)	(-, 1)	(-, 1)	(3, 1)	0.50	10 (6)
31	(3, 1)	(3, -)	(3, -)	(-, 1)	(-, 1)	(3, 1)	0.50	10 (6)

Columns P_n denotes the number of passing (x) and failing (y) tests that cover each line by the notation of (x, y). The collaborative coverage, **Coll. Cov.**, is derived according to Section 3.3. Each suspiciousness score in column **Score** is computed with the collaborative coverage. Any coverage or score that has been changed from the original version is highlighted in red.

becomes 0.00 mainly due to the collaborative failing coverage. Such degradation of the scores of Lines 20, 23, and 8 successfully puts Line 3, which is the faulty statement, at the top of the ranking with two ties (i.e., 3rd rank) among 16 lines while the original rank was the 6th.

4 The Flex Framework

In this section, we formalize Flex. The overall algorithm is presented in Algorithm 1. Flex takes as input a program $\mathcal{P}$, a passing test suite $\mathcal{T}_{pass}$, a failing test suite $\mathcal{T}_{fail}$, the approximated oracles O_{pass} and O_{fail} each for the passing and failing test cases, respectively, and a fault localization formula Ψ.

Flex first generates variant programs with counterfactual assumptions. We collect all predicates in the conditional statements (Line 1) and the coverage information (Line 2) from the original program. Based on the execution results of the failing test cases, Flex generates two sets of candidate variant programs: $\mathcal{P}_{pcand}$ and $\mathcal{P}_{fcand}$ (Lines 3, 4). For each predicate for which all the failing test cases evaluate to the same Boolean value, Flex generates a variant program for $\mathcal{P}_{pcand}$ by setting the predicate to the evaluated value (Line 3), and for $\mathcal{P}_{fcand}$ by setting the predicate to an inverted value (Line 4).

Next, Flex selects the variants based on the approximated oracles. For each variant in $\mathcal{P}_{pcand}$, Flex checks whether the variant has a similar passing behavior modulo O_{pass} (Line 7). If so, the variant is added to the set of passing variants $\mathcal{P}_{pass}$ (Line 8). Similarly, for each variant in $\mathcal{P}_{fcand}$, Flex checks whether the variant has a similar failing behavior modulo O_{fail} (Line 10). If so, the variant is added to the set of failing variants $\mathcal{P}_{fail}$ (Line 11).

Algorithm 1: FLEX $(\mathcal{P}, \mathcal{T}_{pass}, \mathcal{T}_{fail}, O_{pass}, O_{fail}, \Psi)$ where $\mathcal{P}$ is the Program Under Test, $\mathcal{T}_{pass}$ and $\mathcal{T}_{fail}$ are the Passing and Failing Test Suites, O_{pass} and O_{fail} are the Approximated Oracles for the Passing and Failing Test Cases, and Ψ is the Fault Localization Formula

1 $Preds \leftarrow \text{GetAllPredicates}(\mathcal{P})$
2 $Cov_P \leftarrow \text{Run}(P, \mathcal{T}_{pass}, \mathcal{T}_{fail})$
3 $\mathcal{P}_{pcand} \leftarrow \text{GenPassVariantCandidate}(P, Cov_P)$
4 $\mathcal{P}_{fcand} \leftarrow \text{GenFailVariantCandidate}(P, Cov_P)$
5 $\mathcal{P}_{pass}, \mathcal{P}_{fail} \leftarrow \langle \emptyset, \emptyset \rangle$
6 **foreach** $P_{cand} \in \mathcal{P}_{pcand}$ **do**
7   **if** $O_{pass}(P_{cand}, \mathcal{T}_{pass})$ **then**
8         $\mathcal{P}_{pass} \leftarrow \mathcal{P}_{pass} \cup \{P_{cand}\}$ // Variants selected by O_{pass}
9 **foreach** $P_{cand} \in \mathcal{P}_{fcand}$ **do**
10   **if** $O_{fail}(P_{cand}, \mathcal{T}_{fail})$ **then**
11         $\mathcal{P}_{fail} \leftarrow \mathcal{P}_{fail} \cup \{P_{cand}\}$ // Variants selected by O_{fail}
12 $Cov_{fail} \leftarrow \text{GetMinCov}(\mathcal{P}, \mathcal{P}_{fail}, \mathcal{T}_{fail})$
13 $Cov_{pass} \leftarrow \text{GetAvgCov}(\mathcal{P}_{pass}, \mathcal{T}_{pass})$
14 $Cov \leftarrow \text{CombineCov}(Cov_{pass}, Cov_{fail})$ // Collaborative coverage
15 **return** $\Psi(Cov)$

We define the approximated oracle for the passing test cases O_{pass} as follows:

$$O_{pass}(P_{cand}, \mathcal{T}_{pass}) \iff \left[\frac{NumberOfPass(P_{cand}, \mathcal{T}_{pass})}{|\mathcal{T}_{pass}|} \geq \sigma \right].$$

We check whether the ratio of the number of passing test cases to the total number of passing test cases is greater than or equal to σ. The intuition is that if the variant passed with the same test cases as the original program, it is more likely to have a similar behavior to the original program. The threshold σ is a parameter of FLEX and is set to 0.9 in our evaluation.

The approximated oracle for the failing tests is formally defined as follows:

$$O_{fail}(P_{cand}, \mathcal{T}_{fail}) \iff \forall t \in \mathcal{T}_{fail}.\big(CallSeq(P_{cand}, t) = CallSeq(\mathcal{P}, t)\big).$$

We check whether the subsequent call sequence after the altered predicate is the same as the original program with each of the failing tests. Note that the altered predicate may introduce a new fault. In such cases, it is difficult to check whether the same fault still exists in the variant without internal oracles. Thus, FLEX uses this approximated oracle to estimate the presence of the same fault in the variant.

Finally, FLEX collects the collaborative coverage from the variants in $\mathcal{P}_{fail}$ and $\mathcal{P}_{pass}$, then computes the suspiciousness scores. For the collaborative coverage of the failing test cases, we compute the minimum coverage across the variants of $\mathcal{P}_{fail}$ and the original program $\mathcal{P}$ (Line 12). The reason we include the original program is to provide a coverage of 0 for the statements that were not covered in the original program, thus filtering it out. For the collaborative coverage of the passing test cases, we compute the average coverage across the variants of $\mathcal{P}_{pass}$ (Line 13). Then, we compute the suspiciousness scores for each statement in the original program using the given fault localization formula Ψ (Line 14). Table 3 shows the examples of the SBFL formulas that can be used as Ψ.

Table 3. Examples of SBFL Formulas

Ochiai [1]	$\frac{e_f(s)}{\sqrt{\{e_f(s)+n_f(s)\}\cdot\{e_f(s)+e_p(s)\}}}$
Tarantula [9]	$\frac{\frac{e_f(s)}{e_f(s)+n_f(s)}}{\frac{e_f(s)}{e_f(s)+n_f(s)}+\frac{e_p(s)}{e_p(s)+n_p(s)}}$
Dstar [24]	$\frac{e_f(s)^*}{e_p(s)+n_f(s)}$

Here, $e_f(s)$ and $e_p(s)$ are the numbers of failing and passing tests that execute the program statement s while $n_f(s)$ and $n_p(s)$ are the number of failing and passing tests that do not execute s. The symbol $*$ in Dstar's formula is a variable, which is set to 2 in this article according to the previous work [20].

5 Evaluation

We designed experiments to answer the following **research questions (RQs)**:

RQ1. How effectively does FLEX improve the state-of-the-art SBFL techniques?
RQ2. How effective is FLEX with internal oracles?
RQ3. How does the approximated oracle impact the performance of FLEX?
RQ4. How does the aggregation scheme for the collaborative coverage impact the performance of FLEX?
RQ5. How does the variants from $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$ each contribute to the performance of FLEX?

5.1 Setting

Environment. All experiments were performed on Ubuntu 20.04 with Intel Xeon 2.9 GHz CPU and 500 GB RAM.

Implementation. We implemented FLEX with approximately 4,800 lines of OCaml code and 600 lines of Java code. FLEX uses the CIL [17] and Spoon [19] frameworks to instrument target C and Java programs, respectively, and invert conditions.

Benchmark. Our experiments utilize multiple C and Java defect benchmarks as listed in Table 4 to evaluate FLEX. The benchmarks include 65 real-world C programs and 393 Java programs. For C, we use the gzip, libtiff, and php from ManyBugs [5] and make, and grep from CoreBench [2] benchmarks that are widely used in the fault localization literature [3, 6, 11, 15, 23]. Each of the faults is a real-world fault in popular C programs that are associated with external oracles provided as shell scripts. Since FLEX is based on branch condition manipulation, we excluded faults that involve unconditional jumps such as `goto`. We further excluded faults in the benchmarks for practical issues such as unavailable docker images and build errors. Note that we do not list all the faulty versions for the sake of the presentation. The comprehensive list can be found in our artifact. For Java, we used Chart, Closure, Lang, Math, Mockito, and Time from Defects4J 1.0.1 [10] that contains 393 faults in 6 different Java projects. Defects4J is a set of real-world faults in popular Java programs with internal oracles as unit tests. This is also a widely used benchmark in the fault localization literature [12, 21, 29, 33]. In total, we use 393 faults for Java benchmarks. We use both C and Java benchmarks to answer RQ1. For the rest of the RQs, we focus on the C benchmarks.

Table 4. Benchmark Characteristics

Language	Project	Faults	Avg. # Tests	Avg. Size (KLOC)
C	gzip	5	7	470
	libtiff	11	14	382
	php	31	7,716	1,059
	make	9	109	38
	grep	9	33	561
	Total	65	3,702	689
Java	Chart	26	222	132
	Closure	133	2,981	216
	Lang	64	173	50
	Math	106	162	104
	Mockito	38	125	45
	Time	26	2,552	27
	Total	393	1,276	124

Faults denotes the number of faults in the project, each associated with a specific version. **Avg. # Tests** denotes the average number of test cases in each version. **Avg. Size** denotes the average lines of code per faulty version.

Baselines. For the C benchmarks, we apply Flex to three different SBFL techniques: Ochiai [1], Tarantula [8, 9], and Dstar [24]. They all compute the suspiciousness score of each statement using the code coverage and their own formulas. We excluded other tools because they either do not work with external oracles or do not support C programs. For Java, we additionally use SmartFL [29], a state-of-the-art fault localization technique that utilizes internal oracles. SmartFL extracts the internal state of the program from the oracle and derives the probabilistic model for fault localization. Thus, we could not run SmartFL on the C benchmarks because most of them do not have such internal oracles but only external oracles. We excluded Flip [27] because Flip is not publicly available, and we did not receive a response from the authors despite our attempts. However, the conceptual comparison between Flex and Flip is discussed in Section 7.

Evaluation Criteria. We evaluate the effectiveness of Flex in terms of the number of statements to examine before finding the faulty statement. More precisely, for each project, we sum up the number of statements that are ranked higher than the actual faulty statements. In case of multiple faulty statements, we consider the highest ranked faulty statement. Following the previous works [29, 31, 33], we also evaluate the effectiveness of Flex in function-level ranking with C benchmarks. We do not use statistical tests such as **Mann-Whitney U (MWU)** test because Flex, along with the baseline tools, is deterministic while the purpose of the MWU test is to compare two distributions influenced by randomness to determine if they are significantly different. Furthermore, previous literature on SBFL [27, 29, 33] has not utilized statistical tests either, including the MWU test. One factor that hinders the evaluation of fault localization tools is the presence of ties in the ranking. For example, Ochiai assigns a suspiciousness score to a faulty statement while assigning up to 600 statements with the same score.[5] Thus, handling these ties is crucial for the evaluation. We follow the commonly used practices in the fault localization literature. We assume the worst case and report the faulty statement ranked as the last among its techniques following the previous work [6–8, 18].

[5]Case 2010-11-27-eb326f9-eec7ec0 of the Manybugs benchmark, in libming project.

Table 5. Performance of Baseline Tools and FLEX on the C Benchmarks

Level	Program	Ochiai			Tarantula			Dstar		
		Vanilla	FLEX	Improv.	Vanilla	FLEX	Improv.	Vanilla	FLEX	Improv.
Line	gzip	2,233	1,881	16%	2,233	1,881	16%	2,233	1,881	16%
	libtiff	5,914	4,827	18%	6,193	5,170	17%	5,805	4,825	17%
	php	1,396	1,135	19%	1,443	1,094	24%	1,442	1,088	25%
	make	430	292	32%	430	292	32%	430	292	32%
	grep	1,819	846	53%	1,819	842	54%	1,819	842	54%
	Total	11,792	8,981	24%	12,118	9,279	23%	11,729	8,928	24%
Function	gzip	172	166	3%	172	166	3%	172	166	3%
	libtiff	632	531	16%	646	540	16%	632	531	16%
	php	552	391	29%	552	388	30%	552	388	30%
	make	1,190	866	27%	1,190	866	27%	1,190	866	27%
	grep	311	95	69%	311	95	69%	311	95	69%
	Total	2,857	2,049	28%	2,871	2,055	28%	2,857	2,046	28%

Each baseline is augmented with **FLEX**, and is compared to the original version, denoted as **Vanilla**. Both columns report the total number of statements that must be examined to locate all the faults in the project. **Improv.** reports the improvement ratio of FLEX over the vanilla version of the baselines. We report the rank of the faulty statement as the last among its ties.

5.2 Effectiveness of FLEX

5.2.1 C Benchmarks. This section evaluates the effectiveness of FLEX compared to the baseline systems for the C benchmarks. Note that all the programs have external test oracles such as bash scripts. We instantiate FLEX upon three different baseline SBFL techniques: Ochiai [1], Tarantula [8, 9], and Dstar [24]. We regularized the scores produced by Dstar between 0 and 1 with the standard min-max normalization because its maximum score exceeds 1 by the nature of its formula.

Table 5 shows the overall evaluation results in the statement level ranking for the cases. For 64 cases out of 65 faults, FLEX significantly improves the quality of ranking across all the baselines. In the statements level ranking, FLEX can improve the rank by 24%, 23%, and 24% on average over each baseline. Overall, FLEX is consistent in improving the ranking quality across all the baselines.

It is notable that FLEX shows an exceptional tie-breaking ability. While the baseline tools rely on the coverage profile of the failing test cases, FLEX augments such tools with counterfactual executions, providing a way to break ties by reducing the number of statements with the same score as the faulty statement. In the case of grep, the vanilla version of Ochiai suffers from ties of 2,000 statements. On the other hand, FLEX effectively breaks the ties and improves the ranking by 1,300. This result demonstrates that our counterfactual analysis significantly enhances the performance of fault localization when many statements are tied in the ranking. Furthermore, FLEX consistently improves the ranking quality in function-level fault localization for all 65 cases. Table 5 also shows the results in the function-level ranking. For all the target faults, FLEX outperforms the baselines by 28%, 28%, and 28% improvements on the ranking of the faulty function.

5.2.2 Java Benchmarks. This section evaluates the effectiveness of FLEX compared to the baselines on the Java benchmarks. While our primary goal is to improve the accuracy of fault localization with external oracles, FLEX is still effective when internal oracles are available.

For Java benchmarks, FLEX can employ a more precise approximated oracle for failing test cases by further utilizing error logs from JVM. Instead of just comparing the call sequences as in C, FLEX checks the exception type and, the call and return site of each method call on the call stack at the

Table 6. Performance of Baseline Tools and FLEX on the Java Benchmarks

	Ochiai			Tarantula			Dstar		
Program	**Vanilla**	**FLEX**	**Improv.**	**Vanilla**	**FLEX**	**Improv.**	**Vanilla**	**FLEX**	**Improv.**
Lang	1,947	1,909	2%	1,985	1,949	2%	1,947	1,909	2%
Math	10,880	10,188	6%	11,023	10,305	7%	10,923	10,244	6%
Chart	3,901	3,867	1%	4,121	4,081	1%	3,856	3,835	1%
Time	2,024	1,722	15%	2,103	1,814	14%	2,034	1,732	15%
Closure	71,350	57,510	19%	73,202	58,621	20%	71,861	57,842	20%
Mockito	2,996	2,860	5%	2,996	2,860	5%	2,996	2,860	5%
Total	93,098	78,056	16%	95,430	79,630	17%	93,617	78,422	16%

Each baseline is augmented with **FLEX**, and is compared to the original version, denoted as **Vanilla**. Both columns report the total number of statements that must be examined to locate all the faults in the project. **Improv.** reports the improvement ratio of FLEX over the vanilla version of the baselines. We report the rank of the faulty statement as the last among its ties.

failure point. This oracle is more effective for Java programs because of the different coding styles between C and Java programs. Each method in Java is usually short and involves more method calls compared to C programs. As a result, a condition change in a variant program has a higher chance of introducing a different call sequence even though the failing behavior is preserved in the variant. To avoid this noise, we utilize the exception type (e.g., NullPointerException) and call stack of each failure.

To show the effectiveness, we also applied FLEX to SMARTFL [29], which is a state-of-the-art fault localization technique that utilizes internal oracles. In this experiment, we report the number of statements that must be examined to locate all the faults in the project.

We integrated FLEX with SMARTFL in a slightly different way from the other techniques. Since the others utilize coverage profile, FLEX can directly compute the suspiciousness scores from the collaborative coverage. However, SMARTFL does not utilize coverage profile but is based on a probabilistic model. Thus, we first compute suspiciousness scores using SMARTFL and filter out the statements whose failing coverage count is 0 in our collaborative coverage profile.

Tables 6 and 7 show the experimental results. In total, FLEX improves the performance of the vanilla Ochiai, Tarantula, Dstar, and SMARTFL by 16%, 17%, 16%, and 22%. Note that FLEX significantly improves the accuracy for Closure compared to the other benchmarks. This is because of the highly complex nature of the project. Most of the unit tests in the other projects are designed to check the correctness of individual functions. Therefore, they simply compare the return values of target functions to their reference outputs. However, the test cases for Closure mainly check the correctness of the whole program. Thus, a single test executes a large number of statements and assertion checks as in the C cases. Moreover, since Closure is a compiler, the test cases involve complex custom checkings for the internal states of the compiler (e.g., parsing error). In such cases, FLEX can effectively improve the accuracy of fault localization compared to other techniques.

Notice that SMARTFL failed to run for Closure and Mockito in our experiment. SMARTFL is based on the dependency of the return values from the target functions under test. We conjecture the complex nature of the test cases makes it difficult for SMARTFL to derive accurate dependencies between the test outcomes and program statements.

There exist few cases where FLEX failed to improve the accuracy of fault localization against the baselines. This mainly happens when the failing test cases are not well-designed. For example, a failing test case for Chart aims to check whether the program handles the null value properly. However, the test case checks for any exception and simply reports that an exception has occurred.

Table 7. Performance of SmartFL and Flex on the Java Benchmarks

	SmartFL		
Program	**Vanilla**	**Flex**	**Improv.**
Lang	304	261	14%
Math	1,685	1,621	4%
Chart	153	148	3%
Time	2,124	1,303	39%
Closure	-	-	-
Mockito	-	-	-
Total	4,266	3,333	22%

SmartFL is augmented with **Flex**, and is compared to the original version, denoted as **Vanilla**. Both columns report the total number of statements that must be examined to locate all the faults in the project. **Improv.** reports the improvement ratio of Flex over the vanilla version of the baselines. We report the rank of the faulty statement as the last among its ties. Note that the results of SmartFL are not available for Closure and Mockito because SmartFL failed to run on these projects.

Since the test case does not provide any information about the captured exception, Flex cannot distinguish different failing behaviors. In such cases, Flex may falsely determine that the failing behavior is preserved in the variant, thus ending up filtering out the faulty statements.

Overall, the result demonstrates that Flex is also effective when internal oracles are available. Even though existing techniques can improve the accuracy by exploiting internal oracles, it is often difficult to analyze the complex behavior of the program under test. Instead, Flex can be easily integrated with existing techniques to further improve the accuracy of fault localization.

Answer to RQ1. Flex significantly improves the ranking quality of SBFL techniques by 24%, 23%, and 24% on average over each baseline, respectively, for Ochiai, Tarantula, and Dstar. Flex is also effective when internal oracle is available, improving the performance of the vanilla Ochiai and SmartFL by 16% and 22% on average. Flex is easily integrated with existing techniques with the ability to filter out the irrelevant statements.

5.3 Impact of Approximated Oracles

In this section, we perform an analysis of the overall impact of the approximated oracles. Recall from Section 3.2 that Flex uses the approximated oracles for $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$ to examine whether the program behavior is preserved in the program variant. We investigate the impact of different approximated oracles for $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$ in the following sections.

5.3.1 Approximated Oracle for $\mathcal{P}_{pass}$. Flex uses the approximated oracle for $\mathcal{P}_{pass}$ to examine whether the behavior of the passing test cases is still preserved in the program variant. In our main experiment, we set the threshold of preserved passing test cases to 90%. In order to analyze the impact of this threshold, we perform an experiment with a different threshold. We instantiated a

Table 8. Comparative Analysis of FLEX's Approximated Oracle for $\mathcal{P}_{pass}$

	Vanilla	FLEX		FLEX$_{P80}$		FLEX$_{P100}$	
Project	**# Stmts**	**# Stmts**	**Improv.**	**# Stmts**	**Improv.**	**# Stmts**	**Improv.**
gzip	2,233	1,881	16%	1,892	15%	1,881	16%
libtiff	5,914	4,827	18%	4,856	18%	4,942	16%
php	1,396	1,135	19%	1,143	18%	1,172	16%
make	430	292	32%	292	32%	345	20%
grep	1,819	846	53%	846	53%	842	54%
Total	11,792	8,981	24%	9,029	23%	9,182	22%
Selected Var.		2,273		2,630		670	

Vanilla represents the vanilla Ochiai. **FLEX**, **FLEX$_{P100}$**, and **FLEX$_{P80}$** represent FLEX with the threshold set to 90%, 100%, and 80%, respectively. The column **# Stmts** is the total number of statements that must be examined to locate all the faulty statements in the project. The column **Improv.** is the reduction ratio of the number of statements to be examined. We also report the total number of selected variants in $\mathcal{P}_{pass}$ using the threshold in the row **Selected Var.**

variant of FLEX with a tighter threshold, FLEX$_{P100}$ with the threshold of 100%, and compared the performance with FLEX and vanilla Ochiai. Table 8 shows the results of the experiment.

FLEX$_{P100}$ improves the performance of vanilla Ochiai by 22% and FLEX$_{P80}$ improves it by 23%, which are relatively small improvements compared to FLEX. This is because the choice of threshold affects the number of selected variants, hence the accuracy of fault localization. If a threshold is high, FLEX may not be able to select enough variants, thus resulting in a less performance improvement. When the threshold is 100%, FLEX only selected 670 passing variants. This in turn introduces less chance to filter out statements that are irrelevant to the faults. On the other hand, if a threshold is low, FLEX may contain unwanted program variants which the behavior of passing test cases are not preserved, thus resulting in a less performance improvement. Nevertheless, FLEX$_{P100}$ and FLEX$_{P80}$ still improve the performance by 22% and 23%. Thus, the overall performance of FLEX with different thresholds shows consistent improvement.

5.3.2 Approximated Oracle for $\mathcal{P}_{fail}$. FLEX uses the approximated oracle for $\mathcal{P}_{fail}$ to examine whether the behavior of the failing test cases is still preserved in the program variant. In our main experiment, we selected the program variants that resulted in the exact same call sequence of the failing test case as the original version. In other words, we set the threshold of the preserved call sequence to 100%. In order to investigate the impact of this threshold, we instantiated FLEX$_{F90}$ with a threshold of 90% and compared the performance with FLEX and vanilla Ochiai.

Table 9 shows the results of the experiment. With FLEX$_{F90}$, the number of selected variants increases, but the performance of fault localization decreases. This is because we take an aggressive approach when utilizing the variants in $\mathcal{P}_{fail}$. So unlike $\mathcal{P}_{pass}$, if we use variants that diverge slightly from the original failing test case, the side effects of such variants can be more severe. In summary, preserving the exact call sequence of the failing test case performs the best.

Answer to RQ2. The choice of different approximated oracle impacts the performance of FLEX. In our empirical study, we found that 90% threshold for $\mathcal{P}_{pass}$ and exact call sequence for $\mathcal{P}_{fail}$ performs the best.

Table 9. Comparative Analysis of Flex's Approximated Oracle for $\mathcal{P}_{fail}$

	Vanilla	Flex		Flex_{F90}	
Project	**# Stmts**	**# Stmts**	**Improv.**	**# Stmts**	**Improv.**
gzip	2,233	1,881	16%	1,455	35%
libtiff	5,914	4,827	18%	13,411	−127%
php	1,396	1,135	19%	2,419,416	−173,211%
make	430	292	32%	39,457	−9,076%
grep	1,819	846	53%	17,101	−840%
Total	11,792	8,981	24%	2,480,604	−20,936%
Selected Var.		46,966		65,393	

Flex and $\textbf{Flex}_{F90}$ represent Flex with the threshold set to 100% and 90%, respectively. Note that the row **Selected Var.** now reports on the total number of selected variants in $\mathcal{P}_{fail}$. The rest of the notations are the same as in Table 8.

Table 10. Comparative Analysis of Flex's Aggregation Schemes

	Vanilla	Flex		$\text{Flex}_{P_{max}}$		$\text{Flex}_{F_{avg}}$	
Project	**# Stmts**	**# Stmts**	**Improv.**	**# Stmts**	**Improv.**	**# Stmts**	**Improv.**
gzip	2,233	1,881	16%	1,886	16%	2,067	7%
libtiff	5,914	4,827	18%	4,837	18%	5,364	9%
php	1,396	1,135	19%	1,134	19%	1,162	17%
make	430	292	32%	1,886	−339%	348	19%
grep	1,819	846	53%	2,091	−15%	929	49%
Total	11,792	8,981	24%	11,834	0%	9,870	16%

$\textbf{Flex}_{P_{max}}$ and $\textbf{Flex}_{F_{avg}}$ each represent the variant of Flex with the aggregation scheme of passing coverage set to maximum and failing coverage set to average, respectively. The rest of the notations are the same as in Table 8.

5.4 Impact of Aggregation Scheme for the Collaborative Coverage

In this section, we analyze the impact of the aggregation scheme for collaborative coverage. Note that Flex uses the *average* of the passing coverage from the program variants in $\mathcal{P}_{pass}$, and the *minimum* of the failing coverage from the program variants in $\mathcal{P}_{fail}$ and the original version to obtain the collaborative coverage.

We perform an experiment with different aggregation schemes to analyze their impact. We instantiated Flex with the following alternative aggregation schemes: maximum for the passing coverage and average for the failing coverage. The variants of Flex with alternative aggregation schemes are denoted as $\text{Flex}_{P_{max}}$ and $\text{Flex}_{F_{avg}}$, respectively. Note that we have set the aggregation scheme for passing coverage to average and failing coverage to minimum when altering the aggregation scheme for the other. Table 10 shows the results of the experiment.

The results are consistent with the insight we elaborated in Section 3.3. For the passing coverage, the average aggregation scheme is more effective and for the failing coverage, the minimum aggregation scheme is more effective. The maximum aggregation scheme for passing coverage is less effective because some variants may have high passing coverage for faulty statements. Thus

Table 11. Impact of Variants from $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$ on the Effectiveness of FLEX

	Vanilla	FLEX		FLEX$_{pass}$		FLEX$_{fail}$	
Project	**# Stmts**	**# Stmts**	**Improv.**	**# Stmts**	**Improv.**	**# Stmts**	**Improv.**
gzip	2,233	1,881	16%	2,170	3%	1,874	16%
libtiff	5,914	4,827	18%	5,179	12%	4,936	17%
php	1,396	1,135	19%	1,307	6%	1,172	16%
make	430	292	32%	363	16%	345	20%
grep	1,819	846	53%	970	47%	1,521	16%
Total	11,792	8,981	24%	9,989	15%	9,848	16%
Selected Var.		49,085		2,119		46,966	

FLEX$_{pass}$ and **FLEX$_{fail}$** represent FLEX utilizing only the variants from $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$, respectively. Note that the row **Selected Var.** now reports the number of selected variants in $\mathcal{P}_{pass} \cup \mathcal{P}_{fail}$. The rest of the notations are the same as in Table 8.

taking such an aggressive aggregation scheme results in the degradation of the faulty statement's rank. The average aggregation schemes for failing coverage are less effective than the minimum aggregation scheme because they do not effectively utilize the information from the program variants in $\mathcal{P}_{fail}$. Thus, we lose the chance to filter out more of the irrelevant statements that are ranked higher than, or tied with the faulty statement.

In summary, our insight is validated by the experiment. It is better to take a conservative approach when aggregating information from the program variants in $\mathcal{P}_{pass}$, and aggressive approach when aggregating information from the program variants in $\mathcal{P}_{fail}$.

> *Answer to RQ3.* Consistent with our insight, the average aggregation scheme for passing coverage and the minimum aggregation scheme for failing coverage are more effective.

5.5 Impact of Variants from $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$

In this section, we analyze the impact of the variants from $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$. In order to identify their impact separately, we have instantiated FLEX$_{pass}$ and FLEX$_{fail}$ to compare the performance with FLEX and vanilla Ochiai. FLEX$_{pass}$ is an instantiation of FLEX which utilizes only the variants from $\mathcal{P}_{pass}$, and FLEX$_{fail}$ is an instantiation of FLEX which utilizes only the variants from $\mathcal{P}_{fail}$. Table 11 shows the results of the experiment.

The results show that both FLEX$_{pass}$ and FLEX$_{fail}$ outperform the vanilla Ochiai by 15% and 16%, respectively. This is because the use of variants from $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$ both play an effective role in filtering out the irrelevant statements. Moreover, FLEX outperforms both FLEX$_{pass}$ and FLEX$_{fail}$, demonstrating that the variants from $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$ are complementary to each other.

> *Answer to RQ4.* The variants from $\mathcal{P}_{pass}$ and $\mathcal{P}_{fail}$ both contribute to the overall performance of FLEX, complementary to each other.

Table 12. Overhead of Flex

Program	T_p	T_f	$Pred_f$	$Pred_{pf}$	$O.H.$
gzip	6	1	181	3	29×
libtiff	59	2	247	30	39×
php	7,715	1	1,255	3	3×
make	108	1	401	153	155×
grep	32	1	181	60	64×

T_p and T_f denote the average number of passing and failing tests, respectively. $Pred_f$ denotes the average number of predicates covered by the failing tests. $Pred_{pf}$ denotes the average number of predicates covered by both the passing and failing tests. $O.H.$ denotes the overhead of Flex.

6 Discussion

6.1 Overhead

In this section, we discuss the overhead of Flex. Since Flex is also based on the spectrum of given test cases, it is basically expected to have comparable running cost as the traditional SBFL techniques.

The necessary time cost can be estimated as follows. The dominant cost of Flex and the traditional SBFL techniques is the execution time of the test cases given by the program. Let T be the number of test cases. The necessary number of program executions by the traditional techniques is equal to T. Let T_p and T_f be the passing and failing test cases, respectively. Also, let $Pred_f$ and $Pred_{pf}$ be the number of predicates covered by the failing test cases and both the passing and failing test cases, respectively. Then, the required number of executions by Flex is the sum of the following:

— # executions of the original program to collect the coverage information: T.
— # executions to select the variants for $\mathcal{P}_{fail}$: $T_f \times Pred_f$.
— # executions to select the variants for $\mathcal{P}_{pass}$: $T_p \times Pred_{pf}$.

In summary, Flex additionally executes the program K times more than the traditional approaches where $K = T_f \times Pred_f + T_p \times Pred_{pf}$. Then, the overhead of Flex (denoted as $O.H.$ in Table 12), as a constant factor multiplied by the cost of the traditional SBFL techniques, is calculated as $\frac{T+K}{T}$.

Our experimental results demonstrate that Flex has a reasonable overhead. Table 12 shows the actual number of test cases and predicates. Note that T_f is mostly 1, and $Pred_{pf}$ is much smaller than $Pred_f$, approximately 10% on average.

For the projects with a large number of test cases, such as php, the overhead of Flex is only 3× compared to the traditional SBFL techniques. For the other projects with bigger overheads, the number of test cases is much smaller, thus still scalable. Furthermore, all the executions of the program variants can be fully parallelized. Overall, we believe that this is a reasonable overhead compared to other fault localization techniques such as **mutation-based fault localization (MBFL)**.

6.2 Threats to Validity

6.2.1 Threats to External Validity: Generality. The choice of benchmarks used in our study can have an impact on the generalizability of our results. We used real-world open source programs written in C and Java from widely used benchmarks. However, our findings might not apply to

other programs with different characteristics, such as close-sourced software systems or programs written in different programming languages.

The approach of FLEX designs to target general programs for improved fault localization. However, it may not be effective for programs that have few branches. Also, obfuscated programs which can have meaningless branches can hinder FLEX from working properly.

6.2.2 Threats to Internal Validity: Sensitivity. The performance of FLEX is dependent on the choice of parameters, such as the preserving ratio used in the criteria of variant selection mentioned in Section 3.2. While we used reasonable parameters based on our insight in our experiments, there may be other parameter combinations that could improve the performance of our approach.

FLEX adopts the minimum and average value policy to integrate the execution profile from program variants selected by approximated oracles. However, other methods can be applied to them, and it may affect the performance of FLEX.

6.2.3 Threats to Construct Validity. Our tool assumes that a target program contains predicates and it can be tested by passing and failing test suites, so that it is able to generate program variants. In addition, this study makes use of counterfactual assumptions, which anticipate the target program's behavior changes when negating its predicates and running the test suites.

7 Related Work

Our work is applicable to a wide range of fault localization techniques that are based on the spectrum of given test cases [1, 8, 9, 24]. Recently, researchers have proposed several techniques to improve the accuracy of SBFL techniques by combining various techniques including dynamic analysis or probabilistic models [13, 23, 29, 31]. However, they rely on *internal* test oracles in the form of assertions or unit tests. Therefore, these techniques are not directly applicable for fault localization without internal oracles. Furthermore, we observed that it is still challenging to achieve accurate fault localization for large and complex programs even with internal oracles. FLEX tackles this problem to improve by using counterfactual execution and approximated oracle.

The line of work in MBFL has commonality with FLEX in that they change the program to observe the behavioral difference. MBFL is another fault localization approach based on the insight that there is more chance for a program to pass the failing test when the faulty statement is mutated [16, 18, 30]. Therefore, they iteratively and randomly mutate program locations and observe the behavior. While this technique is accurate, it is also well known to be heavily expensive [33]. In contrast, FLEX generates *variant* programs by setting a branch condition executed by the failing test case to a constant Boolean value. This is different from the mutation operators used in MBFL techniques due to the following reasons: (1) Each variant is not randomly generated but is dedicated to a specific purpose, which is to force the execution flow to a desired path. Thus, FLEX generates much fewer variants than MBFL and every information from the variant is used to adjust the suspiciousness score of SBFL. (2) There is no randomness in the generation of the variant program. The Boolean constant is decided by the branch condition during the failing execution. In line with this point, we have intentionally used the term *variants* instead of *mutants* to emphasize that the program is not randomly mutated but modified to force the execution flow to a specific path.

Value-based fault localization techniques iteratively execute failing runs with changed values or use learning methods to predict the faulty locations [4, 7, 12, 21, 25]. For example, Unival [12] learns from the concrete execution traces to estimate the impact of each variable to the passing and failing behavior, thus identifying the most suspicious variable. FLEX also iteratively executes the program with changed predicate values. However, FLEX rather focuses on the direct and concrete impact of changed predicates values to the program behavior (i.e., control flow and execution results).

Recently, researchers have proposed several learning-based techniques to combine information from multiple sources including different SBFL techniques, source code metrics, and text similarity [14, 22, 28, 33]. Flex is orthogonal to these approaches because it can be used as a component in these learning-based approaches to improve their performance. Thus, the improvement of both sides will be beneficial to the community.

There are several fault localization techniques that manipulate predicates to improve the accuracy of fault localization [27, 32]. Zhang et al. [32] were the first to apply predicate switching to locating faults. However, they target a specific class of faults where the failing test cases pass when a certain predicate is flipped. On the other hand, Flex is generally applicable to any fault because we utilize predicates to identify non-relevant statement with counterfactual execution and approximated oracles.

Flip [27] also inverts the predicate condition and observe the behavior to improve the accuracy of SBFL. After flipping a predicate, Flip first checks whether the failing behavior is removed or not. If so, the predicate is considered as a critical predicate with respect to the fault. Then, Flip increases the suspiciousness score of the statements on which the predicate is data dependent. Otherwise, Flip decreases the suspiciousness score of the dependent statements.

Unfortunately, Flip is not publicly available, so, we could not directly compare Flex to Flip. However, we believe that Flex will be more effective than Flip in most cases for the following reasons: (1) Flex targets a wider range of predicates, specifically all predicates executed by the failing test case. In contrast, Flip focuses on critical predicates, which can change the failing behavior when flipped. However, we observed that such critical predicates are scarce in our benchmark. For example, among the 64 faults in Lang project, only two cases have critical predicates, 1 and 6, respectively. Thus, Flip may not be effective in these cases. (2) Flex can adjust the scores of more statements than Flip. While Flip adjusts the suspiciousness scores of statements dependent on critical predicates, Flex adjusts the scores of all statements executed with counterfactual execution. Nonetheless, the Flex and Flip are complementary to each other because Flex augments the coverage profile of the statements with the counterfactual execution, while Flip directly adjusts the suspiciousness scores of statements to improve the accuracy of fault localization. In summary, Flex and Flip are complementary to each other and can be used together to improve the accuracy of fault localization.

8 Conclusion

In this article, we presented a new approach for SBFL with external oracles. While there has been a large body of research for improving the accuracy of SBFL, they require precise internal oracles. In contrast, our approach suppresses program components that are unrelated to faults by using counterfactual executions and approximated oracles. We evaluated our approach on a large number of real faults from widely used open source projects. The results show that our approach can significantly improve the accuracy of SBFL with external oracles.

Data Availability Statement

We make all the detailed data and replication packages publicly available at https://github.com/prosyslab/flex-artifact.

References

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

[2] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying complexity of regression errors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '14)*, 105–115.
[3] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2016. Assessing and comparing mutation-based fault localization techniques. arXiv:1607.05512v1. Retrieved from https://doi.org/10.48550/arXiv.1607.05512
[4] Ross Gore, Paul F. Reynolds Jr., and David Kamensky. 2011. Statistical debugging with elastic predicates. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, 492–495.
[5] Claire Le Goues, Neal J. Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar T. Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)* 41, 12 (2015), 1236–1256.
[6] Jong-In Jang, Duksan Ryu, and Jongmoon Baik. 2022. HOTFUZ: Cost-effective higher-order mutation-based fault localization. *Software Testing, Verification and Reliability* 32, 8 (2022), e1802.
[7] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2009. Effective and efficient localization of multiple faults using value replacement. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09)*.
[8] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*.
[9] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE '02)*.
[10] 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 International Symposium on Software Testing and Analysis (ISSTA '14)*, 437–440.
[11] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. 2019. Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 23:1–23:34.
[12] Yigit Küçük, Tim A. D. Henderson, and Andy Podgurski. 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE '21)*, 649–660.
[13] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*.
[14] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.
[15] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault localization with code coverage representation learning. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE '21)*, 661–673.
[16] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST '14)*, 153–162.
[17] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS '02)*. R. Nigel Horspool (Ed.), Vol. 2304. Springer, 213–228.
[18] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based fault localization. *Software Testing, Verification and Reliability (STVR)* 25 (2015), 605–628.
[19] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.
[20] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE / ACM, 609–620.
[21] Andy Podgurski and Yigit Küçük. 2020. CounterFault: Value-based fault localization by modeling and predicting counterfactual outcomes. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME '20)*, 382–393.
[22] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using code and change metrics to improve fault localization. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*.
[23] Ezekiel O. Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. 2021. Locating faults with program slicing: An empirical analysis. *Empirical Software Engineering* 26, 3 (2021), 51.
[24] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar method for effective software fault localization. *IEEE Transactions on Reliability (TRel)* (2014).

[25] Tao Xie and David Notkin. 2005. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering (TSE)* 31, 10 (2005), 869–883.

[26] Xiaofeng Xu, Vidroha Debroy, W. Eric Wong, and Donghui Guo. 2011. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* 21, 6 (2011), 803–827.

[27] Xuezheng Xu, Changwei Zou, and Jingling Xue. 2020. Every mutation should be rewarded: Boosting fault localization with mutated predicates. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME '20)*, 196–207.

[28] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*, 196–207.

[29] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault localization via efficient probabilistic modeling of program semantics. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, 958–969.

[30] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '13)*, 765–784.

[31] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. 2021. An empirical study of boosting spectrum-based fault localization via PageRank. *IEEE Transactions on Software Engineering* 47, 6 (2021), 1089–1113.

[32] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the International Conference on Software Engineering (ICSE '06)*, 272–281.

[33] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering (TSE)* 47, 2 (2021), 332–347.

Received 8 February 2024; revised 11 June 2024; accepted 10 August 2024