RESEARCH-ARTICLE

Revisit the Intuition of Mutation-Based Fault Localization in Real-world Programs

CHENLIANG XING, Wuhan University, Wuhan, Hubei, China

GONG CHEN, Wuhan University, Wuhan, Hubei, China

QI XIN, Wuhan University, Wuhan, Hubei, China

XIAOYUAN XIE, Wuhan University, Wuhan, Hubei, China

Published: 20 June 2025

Internetware 2025: the 16th International Conference on Internetware
June 20 - 22, 2025
Trondheim, Norway

Conference Sponsors:
SIGSOFT

Internetware '25: Proceedings of the 16th International Conference on Internetware (June 2025)
https://doi.org/10.1145/3755881.3755919
ISBN: 9798400719264

Revisit the Intuition of Mutation-Based Fault Localization in Real-world Programs

Chenliang Xing
School of Computer Science
Wuhan University
Wuhan, China
xingchenliang@whu.edu.cn

Gong Chen
School of Computer Science
Wuhan University
Wuhan, China
chengongcg@whu.edu.cn

Qi Xin
School of Computer Science
Wuhan University
Wuhan, China
qxin@whu.edu.cn

Xiaoyuan Xie*
School of Computer Science
Wuhan University
Wuhan, China
xxie@whu.edu.cn

Abstract

Mutation-based fault localization (MBFL) is an automated fault localization method that has been extensively studied in recent years. The intuition behind MBFL is based on the assumption that mutation operations can correct faults in a program. However, this assumption has only been experimented and validated on simulated datasets, and whether it truly holds in the real world has never been investigated. Fault types in simulated datasets are simple and differ significantly from the complex and diverse faults found in real-world programs. Therefore, to investigate whether MBFL works in the real world, it is necessary to validate its intuition in the real world. The goal of this study is to analyze whether the intuition of MBFL still holds in the real world. We quantified the MBFL intuition by establishing an algorithm, which eliminated the interference of factors unrelated to MBFL itself, allowing us to directly validate the intuition of MBFL. Based on this algorithm, we conducted extensive experiments on both real-world programs and programs in simulated datasets. The results revealed an interesting trend: due to the complexity of faults in real-world programs compared to those in simulated datasets, MBFL's intuition probably cannot hold in the real world. This indicates that MBFL's intuition is difficult to hold in the real world. Consequently, we focused on analyzing the real-world faulty versions and summarized a set of mutation operators that perform better in the real world by studying the types and effects of each mutant, providing guidance for the application of MBFL.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Software testing, Fault localization, Mutation testing, Debugging

ACM Reference Format:
Chenliang Xing, Gong Chen, Qi Xin, and Xiaoyuan Xie. 2025. Revisit the Intuition of Mutation-Based Fault Localization in Real-world Programs. In *the 16th International Conference on Internetware (Internetware 2025), June 20–22, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3755881.3755919

*Corresponding author.

Internetware 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1926-4/25/06
https://doi.org/10.1145/3755881.3755919

1 Introduction

Automated fault localization is an effective means of ensuring software quality, and mutation-based fault localization (MBFL) has been a widely studied direction in recent years [38]. The main idea of MBFL is to apply mutation operations to certain locations in the program, and then compare the execution paths or results of the program before and after the mutation. Based on the changes in the execution paths and results, the suspiciousness of each statement being faulty is estimated. MBFL is built upon the following intuition [27]: *mutating faulty statements is more likely to convert failing test cases into passing ones, whereas mutating correct statements tends to cause passing test cases to fail.*

Moon et al. [27] empirically evaluated the intuition of MBFL on SIR benchmark [9], a simulated dataset that contains artificially injected faults. We observed that the experiments were able to validate the intuition behind MBFL because the faults in the SIR dataset could be coincidentally corrected by the mutation operations. Therefore, we conducted a preliminary study using an experimental setup modeled after theirs to explore the reason why the MBFL's intuition holds in simulated datasets. The results have indicated that the intuition holds on simulated datasets is exactly because the mutation operator happens to change the faulty statements correctly (shown in Section 2.2). **However, the types of faults in real-world programs are significantly different from those in simulated datasets [14, 35].** The fact that these mutation operations can fix faults in simulated datasets does not guarantee that they can do the same in the real world. Consequently, we are highly interested in whether the intuition behind MBFL still holds in the real world. **Therefore, to assess whether the intuition behind MBFL holds across all programs, it is essential to validate it in the real world.**

Based on these considerations, we conducted extensive experiments in both simulated environments and the real world to validate the intuition behind MBFL, and further analyzed the impact of mutation operators on this intuition. It is worth noting that although some recently proposed MBFL methods have demonstrated their fault localization capabilities in the real world [23, 24, 35], **the ability to effectively localize faults in the real world does not directly imply that MBFL's intuition also works in real-world scenarios,** as MBFL's fault localization capability is influenced by various factors [36], such as the risk evaluation formula, the quality of test cases, and additional information introduced to support MBFL.

Since no study has directly validated this intuition in real-world programs, this paper will revisit the validity of MBFL's intuition across simple and complex datasets and explore the factors that influence its effectiveness, ultimately aiming to provide guidance for future research in the field. We designed an algorithm to directly quantify MBFL's intuition, which eliminates the interference of factors unrelated to MBFL itself, allowing us to directly validate its intuition. We used this algorithm for evaluation on both simulated datasets and real-world programs. **The results show that in the real world, the proportion of programs that align with MBFL's intuition is only 28.8%**, which is nearly half of that in the simulated environment (53.3%). This result indicates that **due to the complexity of faults in real-world programs compared to those in simulated datasets, MBFL's intuition probably cannot hold in the real world.** Therefore, we analyzed mutation operations and identified the characteristics of more effective mutation operations, providing recommendations for future work. The main contributions of this paper are as follows:

- We conducted the first comprehensive experiment on real-world programs to validate the intuition of MBFL.
- We conducted comprehensive experiments on various programs to assess the validity of MBFL's intuition in simulated datasets and real-world programs with a carefully designed algorithm to quantify intuition and eliminate unrelated factors. We were surprised to discover that only 28.8% of real-world programs conform to MBFL's intuition, suggesting it is potentially infeasible.
- We filtered out mutation operators that performed well in the real world and summarized their characteristics, providing guidance for selection or design of future mutation operators.

The remainder of this paper is organized as follows. Section 2.1 provides a brief introduction to MBFL. The motivation of our study is introduced in more detail in Section 2.2. Research design and result evaluation are described in Sections 3 and 4, respectively. We will discuss potential threats in Section 5. We discuss other problems encountered in the experiment in section 6. Section 7 summarizes the related work. In Chapter 8 we summarize the full text.

2 Preliminary

2.1 Mutation-based Fault Localization

Based on the intuition behind MBFL, the MBFL methods proposed by researchers can generally be divided into the following steps [3]:

1) Execute the program under test (PUT) with the complete set of test cases T, collect the coverage information, and divide T into two groups based on whether or not the test result passes. All test cases $t \in T$ that fail to execute on the PUT can be denoted as T_f, when the test case executes successfully it is denoted as T_p. The mutation m is generated from all statements that can be modified by mutation operations. All mutations generated from s are denoted as $\mathcal{M}(s)$. Then execute all $m \in \mathcal{M}(s)$ with the complete set of test cases T.
2) As Debroy et al. [7] proposed in their work, the mutant is said to be **killed** by a test case, if the output of executing the test case after the mutation is different from the output of the original program, the opposite is not killed (**survived**). The test cases are divided into two groups based on the execution results. All test cases $t \in T$ that can kill m are denoted as T_k, while the test cases that can not kill m are denoted as T_s. Then each mutant generates a vector $a =< a_{kp}, a_{sp}, a_{kf}, a_{sf} >$, where $a_{kp} = |T_k \cap T_p|, a_{sp} = |T_s \cap T_p|, a_{kf} = |T_k \cap T_f|, a_{sf} = |T_s \cap T_f|$.
3) Using the vector a obtained in 2), compute the suspiciousness of mutant m. Suspiciousness of each mutant can be calculated as (1). Then we can use $Sus(m)$ to calculate $Sus(s)$, the usual case takes the average value as the suspiciousness, as shown in (2) [27].

$$Sus(m) = (a_{kf} + a_{sf}) - \frac{\sum_{m \in \mathcal{M}(s)} a_{kf}}{\sum_{m \in \mathcal{M}(s)} a_{kp}} (a_{kp} + a_{sp}) \tag{1}$$

$$Sus(s) = \frac{\sum_{m \in \mathcal{M}(s)} Sus(m)}{|\mathcal{M}(s)|} \tag{2}$$

4) Finally, the MBFL method generates a report of fault localization by sorting all statements in order of suspiciousness $Sus(s)$ obtained above from highest to lowest. Developers can refer to the statement suspiciousness ranking to fix the program.

2.2 Motivation

To validate the intuition of MBFL, Moon et al. conducted an experiment, and the experimental results are presented in the Table 1 [27]. Let m_e denote the mutants generated from faulty statements, and m_n denote the mutants generated from correct statements. They respectively calculated the average number of test cases whose execution results are changed by each m_e and m_n. Column (B)/(A) represents the number of failed test cases that pass on m_e divided by the number of failed test cases that pass on m_n. Conversely, column (C)/(D) represents the average number of failed test cases that pass on m_n divided by the average number of failed test cases that pass on m_e. Finally, both ratios (B)/(A) and (C)/(D) are much greater than 1, and these ratios respectively validate the two statements in the intuition: *mutating faulty statements is more likely to convert failing test cases into passing ones, whereas mutating correct statements tends to cause passing test cases to fail.*

Here we give a simple program shown in Table 2 selected from SIR benchmark to help understand the intuition of MBFL. In this example, MBFL successfully placed the faulty statement s_6 at the top of the suspicious list. However, this result occurred because the

Table 1: The average numbers of the test cases whose results change on the mutants.

#	# of Failing Tests that Pass after Mutating:			# of Passing Tests that Fail after Mutating:		
	Correct Stmts. (A)	Faulty Stmts. (B)	(B)/(A)	Correct Stmts. (C)	Faulty Stmts. (D)	(C)/(D)
Average	0.6835	12.9271	1435.9	67.0660	12.2793	73.4

mutation generated from s_6 happened to fix the program, which raises the question of whether most programs exhibit a similar behavior. Therefore, following the experimental setup of Moon et al., we conducted a preliminary study using the same dataset and tools. Specifically, we selected five faulty versions of `tcas` in the SIR benchmark [9]. Then, we used the MBFL tool to localize faults in each faulty version. Next, we selected the faulty versions where the faulty statements were successfully identified and recorded the mutants generated from these faulty statements. Finally, we observed all the mutants with the highest suspiciousness, as they are the ones most likely to transform failed test cases into successful ones. Interestingly, **66.3%** of these mutants were modified in a way that exactly matches the fix applied in the program patch. This experimental result demonstrates that Moon et al.'s study validates the intuition behind MBFL precisely because the faulty statements in these programs can be corrected by the mutation operators.

However, can real-world faults also be modified correctly by the mutation operator? Figure 1 gives a patch for the real-world program `Joda-Time`, and it is easy to see that this modification is significantly different from the modification made by the mutation operator. It is clear that in real-world programs, faults are fixed in such an intricate way [36] that it's hard to change the result of test cases from failing to pass with these simple mutation operators. So in this case, it is doubtful whether the intuition of MBFL will still hold in the real world.

```
diff --git a/src/main/java/org/joda/time/DateTimeZone.java
b/src/main/java/org/joda/time/DateTimeZone.java
index a127604..7d1719b 100644
--- a/src/main/java/org/joda/time/DateTimeZone.java
+++ b/src/main/java/org/joda/time/DateTimeZone.java
@@ -276,17 +276,14 @@ public abstract class DateTimeZone implements Serializable {
        if (hoursOffset < -23 || hoursOffset > 23) {
                throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
        }
-       if (minutesOffset < -59 || minutesOffset > 59) {
+       if (minutesOffset < 0 || minutesOffset > 59) {
                throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
        }
-       if (hoursOffset > 0 && minutesOffset < 0) {
-               throw new IllegalArgumentException("Positive hours must not have negative
minutes: " + minutesOffset);
-       }
        int offset = 0;
        try {
                int hoursInMinutes = hoursOffset * 60;
                if (hoursInMinutes < 0) {
-                       minutesOffset = hoursInMinutes - Math.abs(minutesOffset);
+                       minutesOffset = hoursInMinutes - minutesOffset;
        } else {
                minutesOffset = hoursInMinutes + minutesOffset;
        }
```

Figure 1: A patch of `Joda-Time`.

To mitigate the difficulty of changing test case results in real-world programs, Papadakis et al. [30] modified the condition in the intuition by changing the criterion from altering the test case results (pass or fail) to altering the output of the test cases. This modification alleviated the problem of it being too difficult to change the test case result from failure to pass. As a result, in recent mainstream works [23, 24, 34, 38, 39], researchers generally follow the modified intuition rather than the original intuition. The modified intuition can be generalized as below:

> Mutants that are killed by more failed test cases and fewer passed test cases are more indicative of real faults. In contrast, mutants that are killed by more passed test cases and fewer failed test cases are less indicative of real faults.

Even though the condition in the modified intuition no longer requires changing the test case results, we still believe that altering the output of test cases in real-world programs through mutation remains very difficult. To prove our conjecture, we conducted a preliminary experiment on the Defects4J dataset, where we generated 98,492 mutants, 1,321 of which were generated from faulty statements. We then recorded the number of mutants that have been killed. The results showed that only **11.4%** of the mutants were killed by test cases, with many faulty statements having no mutants killed at all. This suggests that altering the output of test cases by mutating faulty statements in real-world programs is a challenging task. Therefore, we still have significant doubts about whether the modified intuition can work effectively in real-world scenarios.

In summary, faults in real-world programs differ significantly from those in simulated scenarios. Validating MBFL's intuition on simulated datasets does not guarantee that it will hold true in real-world programs. Therefore, it is essential to revisit the intuition behind MBFL in the real world.

3 Experiment Design

3.1 Research Questions

The goal of this paper is to revisit the intuition of MBFL in the real world. We need to analyze whether MBFL's intuition still holds in the real world as it does in simulated datasets, and if it does not, identify the reasons behind its failure. To achieve this goal, we formulated the following Research Questions (RQs):

RQ1: *Does the intuition of MBFL still hold for MBFL methods considering execution results in the real world?*

The core idea of this paper is to revisit whether the intuition of MBFL still holds in the real world. So in RQ1, we will produce our experiment with the MBFL method considering execution results on large-scale data in the real world. For comparison, we also included a simple dataset. Section 3.2 introduces three datasets of varying sizes. Specifically, in RQ1, we use Defects4J and Bears to represent real-world programs and Codeflaws to represent the simple dataset.

According to the analysis in Section 3.3, satisfying both Assertion 1 and Assertion 2 indicates that the MBFL's intuition is valid. Therefore, we need to calculate the proportion of faulty versions that satisfy both of these assertions out of the total number of faulty versions. A significantly lower proportion in real-world programs compared to simulated ones would indicate that MBFL's intuition does not hold in the real world.

RQ2: *Does the intuition of MBFL still hold for MBFL methods considering execution paths in the real world?*

In real-world programs, mutations may be less effective at altering the execution results of test cases. Therefore, in addition to

Table 2: Example of how MBFL localizes fault statements.

Statements		Mutants	Test Case Result			Suspiciousness	Rank
			t_1	t_2	t_3		
s_1	`int main(int argc, char *argv[])`	/	f	f	p	0	2
s_2	`double n, x, y , temp;`	double → int	f	f	p→f	0	2
s_3	`scanf("%lf %lf %lf", &n, &x, &y);`	/	f	f	p	0	2
s_4	`temp =ceil( n * (y/100));`	/ → *	f	f	p→f	0	2
s_5	`double ans = temp - x;`	- → +	f	f	p→f	0	2
s_6	`if(x > ans) printf("0");//correct:x<ans`	>→<	f→p	f→p	p	1	1
s_7	`else printf("%d", (int)ans);`	/	f	f	p	0	2
s_8	`return 0;`	return 0; → return;	f	f	p→f	0	2

MBFL methods that consider execution results, some researchers have proposed using changes in execution paths as a condition for determining whether a mutant is killed [15, 35]. MBFL methods that consider execution results can only detect faults when the test case outcomes change, whereas this method can detect faults as long as there is a change in the execution path.

MBFL method considering execution paths first splits the PUT into n program blocks b_i so that each test case t gets an execution vector $V_t =< v_1, v_2, ..., v_n >$, where v_i is 1 when the test case t covers program block b_i and 0 when it does not. Then we can calculate the similarity between the two program execution paths before and after the mutation, the lower the similarity, the greater the influence of the mutant on the program. In this case, intuition can be interpreted to mean that if a mutant has a greater impact on the execution path of a failed test case than it does on a passed test case, there is likely to be a fault in the location of that mutant.

We can use Equation (3) to calculate the suspiciousness of each statement.

$$Sus(s) = \overline{I_f}(m) - \beta * \overline{I_p}(m), m \in \mathcal{M}(s) \tag{3}$$

$$\overline{I_f}(m) = \frac{\sum_{i=1 \to |T_f|} 1 - Cos(V_{f_i}, V_{f_i'})}{|T_f|} \tag{4}$$

$$\overline{I_p}(m) = \frac{\sum_{i=1 \to |T_p|} 1 - Cos(V_{p_i}, V_{p_i'})}{|T_p|} \tag{5}$$

$$Cos(V_a, V_b) = \frac{\sum_{i=1 \to n} v_i * u_i}{\sqrt{\sum_{i=1 \to n} v_i^2} * \sqrt{\sum_{i=1 \to n} u_i^2}} \tag{6}$$

V_a and V_b are two execution paths, v_i and u_i are the ith elements of the vectors V_a and V_b respectively. $\overline{I_f}(m)$ and $\overline{I_p}(m)$ represent the average influence of mutants on passing and failing test cases, respectively. Equation (4) and (5) show their specific formulas, where $V_{p_i'}$ and $V_{f_i'}$ represent the execution traces after mutation. β is the parameter that controls the magnitude of the influence on the result between two parts. Finally, the above obtained $Sus(s)$ is utilized to rank the statements to obtain the suspiciousness ranking.

In RQ2, we will observe if the intuition holds with MBFL method considering execution paths in the real world. Since MBFL method considering execution paths requires statistical analysis of execution paths at the granularity of program basic blocks, and in small-scale simulated datasets, the number of basic blocks in a faulty version is limited, it is difficult to gather execution path statistics in such datasets. Therefore, in RQ2, we will perform the statistical analysis only on real-world programs. To reflect the real-world program, we chose Defects4J and Bears to represent real-world programs for RQ2.

To validate whether the intuition holds in MBFL method considering execution paths, we calculated the proportion of faulty versions that simultaneously satisfy both Assertion 1 and Assertion 2 out of the total number of faulty versions. The higher the percentage of faulty versions that satisfy both assertions, the more consistent the method is with intuition. At the same time, we can compare this proportion to determine which method, the method considering execution paths or the method considering execution results, aligns better with the intuition in the real world.

RQ3: *Which mutation operators are effective in real-world projects?*

In RQ1 and RQ2, we found that some of the faulty versions in the real-world programs gave results that clearly conformed to the two assertions. However, not all faulty versions satisfy both of these assertions, so we will explore the reasons behind this phenomenon in RQ3.

Based on our observations, the inconsistency in results across different faulty versions may be due to the type of faulty statements in those versions, as the type of the faulty statement directly determines whether the mutation operator has an effect. In other words, by selecting or designing mutation operators that better match the types of faulty statements, we can enable MBFL's intuition work across a broader range of faulty versions. Therefore, we attempt to identify the more effective mutation operators from all the mutants we have generated. We believe that an effective mutation operator should satisfy the following two conditions:

1) The conditions for the mutation operator should be met more frequently, which would result in the generation of more mutants;
2) The mutation operator should be able to more easily alter the output results of the test cases after mutating the statement.

Specifically, we counted all the m_e in all faulty versions and counted all the m_e generated with the same mutation operator denoted as op, recording the number of op killed and survived. If a mutation operator satisfies both of the above conditions, it indicates that the operator is more effective in real-world scenarios. It should

be noted that in this RQ, in order to find more efficient mutation operators, we have introduced other advanced operators in addition to those listed in Table 4.

3.2 Datasets and Mutation Operators

The purpose of this paper is to revisit the intuition of MBFL in the real world, so the experimental dataset in this paper must be selected to include enough real-world programs. At the same time, in order to compare the difference between the intuition of MBFL in the real world and the simulation environment, we also need to select simple datasets for comparison.

Therefore, we introduced the datasets Defects4J [19] and Bears [26] in our study to represent faults in the real world. These two datasets comprise large, real-world open-source projects with numerous faulty versions and patches, accurately reflecting practical development scenarios and enjoying widespread use in fault localization and program repair [20, 21, 35, 40, 41]. We also introduced dataset Codeflaws [32] in our experiments as simple datasets. Specific details of the dataset are given below:

1) Codeflaws is a large-scale C benchmark for automated repair, derived from real-world programs of 20–200 LOC in the Codeforces database and containing 3,902 faults across 7,436 submissions [32]. We selected 1,762 faulty versions, each exhibiting one to three faults.
2) Defects4J is a Java program benchmark that has been widely used in auto program repair and fault localization in recent works [19]. The version we used of Defects4J is version 2.0.0. We selected 4 different projects with 190 real bugs and 10,936 test cases.
3) Bears is a new Java program benchmark for auto program repair, which contains 251 bugs from 72 projects (the first version of the Bears Benchmark) [26]. The projects contained therein differ significantly from those in Defects4J, and the bug types are more complex. We selected 32 faulty versions from 2 projects for validation.

Note that in order to run the mutation tool properly, we picked only faulty versions on the Bears and Defects4J datasets that restricted the Java version to jdk-1.8 and above. In the experiments section, we will use the three benchmarks mentioned above for validation, and Table 3 lists the specific information for all datasets.

Table 3: Details of subjects in datasets.

	Subject	Bugs	SLOC	Test Cases
Codeflaws	v1-v150(1 fault)	150	39.2(avg)	56.6(avg)
	v1545-v1695(2 faults)	150	37.2(avg)	56.3(avg)
	v1753-v1762(3 faults)	10	27.8(avg)	41.7(avg)
Defects4J	commons-math	106	85k	3602
	Joda-Time	27	28k	4130
	commons-Lang	41	27.3k	2908
	commons-csv	16	1615	296
Bears	FasterXML-jackson	20	54.6k	1570
	traccar-traccar	12	34.6k	255

The mutation operator is a very important part of MBFL. Here we give mutation operators we used in experiments in Table 4, which contains the most widely used mutation operators that are introduced by [2]. These mutation operators are representative and are widely used in MBFL work[5, 13, 24, 35].

Table 4: Common mutation operators.

Rule	Description
$a\ op_1\ b \rightarrow a\ op_2\ b, op \in \{>, <, \geq, \leq\}$	Replace conditionals.
$a\ op_1\ b \rightarrow a\ op_2\ b, op \in \{+, -, *, /\}$	Replaces binary arithmetic operation.
$a\ op_1 \rightarrow a\ op_2, op \in \{++, --\}$	Replace increments and decrements.
$a\ op_1\ b \rightarrow a\ op_2\ b, op \in \{\|\|, \&\&, \|, \&\}$	Replace logical operations.
$exp \rightarrow !exp$	Negate logical expressions.
$exp \rightarrow true\ or\ false$	Remove all conditionals statements.
$byte\|char\|short\|int \vdash v \rightarrow v + c$	Add a random constant after variables.
return $exp \rightarrow$ return exp' $exp \in \{true, false, 0, null\}$	Replace return values.

3.3 Evaluation Method

Algorithm 1 Evaluating MBFL Intuition by Computing Influence Factors

Require: Set of faulty statements S_e, set of correct statements S_n, mutant set $M(s)$ for each statement s, test case outcomes a_k for each mutant.

Ensure: Influence vector $influ$ for each faulty version.

for each faulty version in the program under test (PUT) **do**
  Initialize $influ = \{influ^p_{m_e}, influ^f_{m_e}, influ^p_{m_n}, influ^f_{m_n}\}$
  for each statement $s \in S_e$ **do** ▷ Process faulty statements
    for each mutant $m \in M(s)$ **do**
      Compute $influ^p_{m_e} = \frac{\sum a_{kp}}{\sum a_k}$
      Compute $influ^f_{m_e} = \frac{\sum a_{kf}}{\sum a_k}$
    end for
  end for
  for each statement $s \in S_n$ **do** ▷ Process correct statements
    for each mutant $m \in M(s)$ **do**
      Compute $influ^p_{m_n} = \frac{\sum a_{kp}}{\sum a_k}$
      Compute $influ^f_{m_n} = \frac{\sum a_{kf}}{\sum a_k}$
    end for
  end for
  Check Assertion 1: $influ^f_{m_e} > influ^f_{m_n}$
  Check Assertion 2: $influ^p_{m_n} > influ^p_{m_e}$
  if both assertions hold **then**
    The MBFL intuition is validated for this faulty version.
  else
    The MBFL intuition is not validated for this faulty version.
  end if
end for

As we mentioned in Section 1, the ability of MBFL to localize faults depends on various factors [36], such as the risk evaluation formula, the quality of test cases, and additional information introduced to support MBFL [25]. In other words, the ability to effectively localize faults does not directly imply that MBFL's intuition is valid. Therefore, we need to isolate other influencing factors and directly analyze MBFL's intuition.

Algorithm 1 shown below evaluates the intuition behind Mutation-Based Fault Localization (MBFL) by computing the proportion of failed and passed test cases that kill mutants from faulty and correct statements.

The algorithm initializes an influence vector, where each component represents the proportion of passed or failed test cases that kill mutants of faulty or correct statements (lines 1-2). The algorithm iterates through all faulty statements in the program and, for each mutant of a faulty statement, calculates $influ^{p}_{m_e}$, the proportion of passed test cases that kill the mutant, and $influ^{f}_{m_e}$, the proportion of failed test cases that kill the mutant (lines 3-8). A similar process is repeated for correct statements, where the algorithm computes $influ^{p}_{m_n}$, the proportion of passed test cases that kill the correct statement's mutants, and $influ^{f}_{m_n}$, the proportion of failed test cases that kill them (lines 9-14). Based on the $influ$ vector and the intuition of MBFL, we can obtain the following two assertions:

Assertion 1: $influ^{f}_{m_e}$ **should be greater than** $influ^{f}_{m_n}$**.** According to the intuition, mutants that are killed by more failed test cases and fewer passed test cases are more indicative of real faults. So the proportion of failed test cases among all test cases that kill m_e should be greater than for m_n. That is to say, $influ^{f}_{m_e}$ should be greater than $influ^{f}_{m_n}$.

Assertion 2: $influ^{p}_{m_n}$ **should be greater than** $influ^{p}_{m_e}$**.** The second assertion is based on the intuition that mutants that are killed by more passed test cases and fewer failed test cases are less indicative of real faults. So the proportion of passed test cases among all test cases that kill m_n should be greater than for m_e.

That is to say, $influ^{p}_{m_n}$ should be greater than $influ^{p}_{m_e}$. If a faulty version satisfies both Assertion 1 and Assertion 2, then it indicates that the intuition is working for this faulty version. The algorithm validates the first assertion by checking whether $influ^{f}_{m_e}$ is greater than $influ^{f}_{m_n}$, which would indicate that mutants of faulty statements are killed by more failed test cases than those of correct statements (lines 15-16). The second assertion is then validated by checking whether $influ^{p}_{m_n}$ is greater than $influ^{p}_{m_e}$, meaning that correct statement mutants are killed by more passed test cases than faulty statement mutants (lines 17-18). Finally, the algorithm evaluates both assertions and determines whether the MBFL intuition holds for the faulty version. If both conditions are met, the intuition is considered valid; otherwise, it is not supported (lines 19-21).

4 Evaluation Result

In this section, we will analyze each RQ based on the experimental results. To investigate whether the experimental results are generalizable, we used a total of 306 faulty versions of simple programs, which produced 41,324 mutants in total. And 222 faulty versions of real-world programs, which produced a total of 4,544,190 mutants. Specifically for each real-world faulty version, we generated 20,469 mutants on average.

4.1 RQ1: Result of Validation of the Intuition for MBFL Methods Considering Execution Results

We evaluated the MBFL methods considering execution results on three different datasets and compared them using the evaluation methods presented in Section 3.3. Table 5 lists the proportion of faulty versions in each project that satisfy Assertion 1, Assertion 2, and both Assertion 1 and Assertion 2, respectively. Additionally, Table 5 presents the overall proportions of these three categories across all projects in each dataset.

Specifically, in the simulated dataset CodeFlaw, the proportions of faulty versions that satisfy Assertion 1, Assertion 2, and both Assertions 1 and 2 are 53.3%, 87.3%, and 53.3%, respectively. In the dataset Defects4J, these proportions are 30.9%, 91.0%, and 30.9%, respectively. In the dataset Bears, these proportions are 17.6%, 79.4%, and 17.6%. For all programs in the real world, these three proportions are **28.8%**, **89.2%**, and **28.8%**, respectively. The proportion of programs that satisfy both assertions in the real world (28.8%) is less than half of that in the simulated dataset (53.3%). Based on the above experimental results, it is evident that compared to the simulated dataset, in real-world programs, the MBFL method considering execution results is less likely to satisfy both assertions simultaneously. This result suggests that the intuition of MBFL often does not hold in the real world.

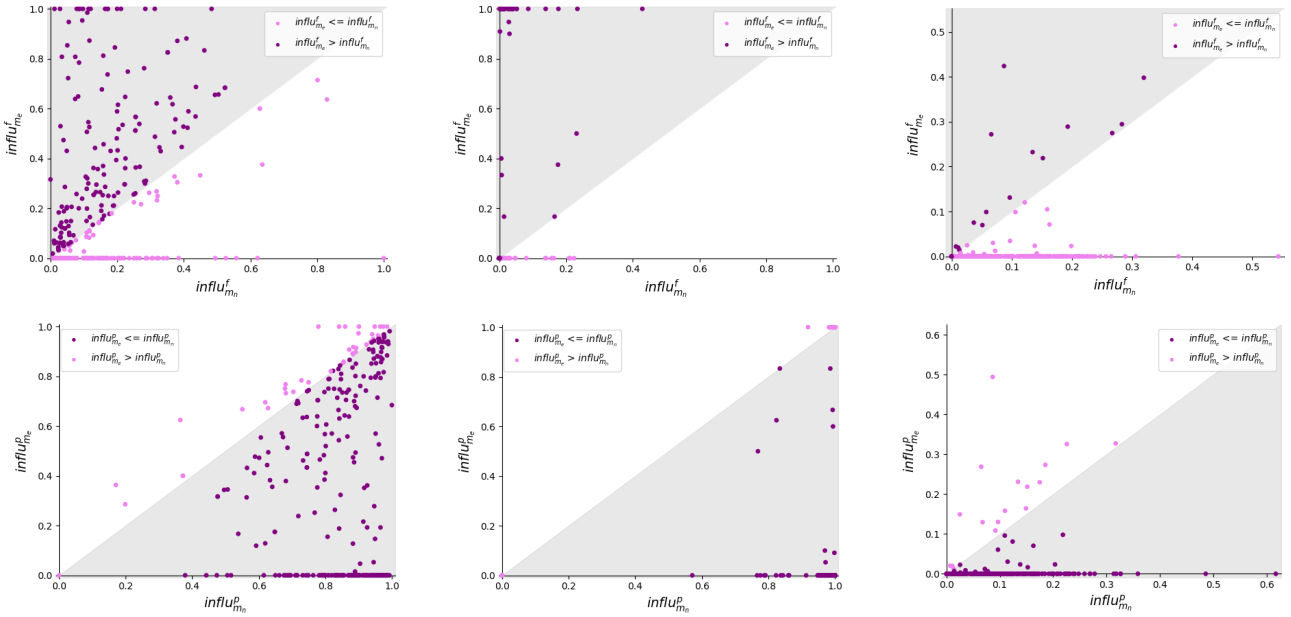
To show the distribution of data more clearly, we examined the raw data for each faulty version individually. The raw data of mutation in simple programs and mutation in real-world programs are presented in Figure 2(a) and Figure 2(b), respectively. In the first row of pictures in Figure 2, the x-axis and y-axis axes represent $influ^{f}_{m_n}$ and $influ^{f}_{m_e}$, respectively and the gray areas in the figure represent data points where $influ^{f}_{m_e}$ is greater than $influ^{f}_{m_n}$ (Assertion 1). In the second row of pictures in Figure 2, the x-axis and y-axis axes represent $influ^{p}_{m_n}$ and $influ^{p}_{m_e}$, respectively and the gray areas in the figure represent data points where $influ^{p}_{m_n}$ is greater than $influ^{p}_{m_e}$ (Assertion 2). Therefore, the points in the figures should be clustered on the gray side, where the points are also darker than the rest of the area.

In the first row of Figure 2(a), we can find that most of the points are concentrated in the correct region. But in the first row of Figure 2(b), we find that only a small fraction of the points are located in the correct region compared to Figure 2(a), which proves that there are actually only a few faulty versions that match the Assertion 1. Both in the last row of Figure 2(a) and Figure 2(b), we find that most of the points are located in the correct regions which means the MBFL method considering execution results can obey Assertion 2 in simulated datasets and real-world programs. The conclusions drawn from the figures are consistent with the proportional data presented in Table 5.

In addition, we also observed that in both Figure 2(a) and Figure 2(b), a large number of points lie on the $influ^{f}_{m_e} = 0$ or $influ^{p}_{m_e} = 0$ axis, indicating that many m_e in faulty versions were not killed

Table 5: MBFL methods considering execution results performance in simulated datasets and real-world programs.

Type	Datasets	Projects	Assertion 1 ($influ^f_{m_e} > influ^f_{m_n}$)		Assertion 2 ($influ^p_{m_n} > influ^p_{m_e}$)		Ass.1 & Ass.2	
Simulated	Codeflaws	v1-v150(1 fault)	39.0%	53.3%	86.3%	87.3%	39.0%	**53.3%**
		v1545-v1695(2 faults)	65.3%		88.7%		65.3%	
		v1753-v1762(3 faults)	80.0%		80.0%		80.0%	
Real World	Defects4J	commons-math	18.9%	30.9%	92.5%	91.0%	18.9%	**30.9%**
		Joda-Time	30.8%		84.6%		30.8%	
		commons-lang	55.0%		87.5%		55.0%	
		commons-csv	50.0%		100.0%		50.0%	
	Bears	FastXML-jackson	18.2%	17.6%	68.2%	79.4%	18.2%	**17.6%**
		traccar-traccar	16.7%		100.0%		16.7%	



(a) Simple datasets in MBFL methods considering execution results.

(b) Real-world programs in MBFL methods considering execution results.

(c) Real-world programs in MBFL method considering execution paths.

Figure 2: Visualization of raw data in different scenarios.

by any test cases. This phenomenon is more pronounced in the real world. This suggests that the mutants generated by existing mutation operators often fail to alter the output results of test cases, particularly those mutants generated from faulty statements. We will further analyze this phenomenon in RQ3.

✍ **Finding.** Due to the complexity of faults in real-world programs compared to those in simulated datasets, MBFL's intuition probably cannot hold for methods considering execution results in the real world.

4.2 RQ2: Result of Validation of the Intuition for MBFL Method Considering Execution Paths

We evaluated MBFL method considering execution paths in real-world programs and compared it using the evaluation methods introduced in Section 3.3. Table 6 presents the proportions of faulty versions in each project for MBFL method considering execution paths that satisfy Assertion 1, Assertion 2, and both Assertion 1 and Assertion 2. In addition, Table 6 lists the proportions of faulty versions in the real-world programs for MBFL methods considering execution results as obtained in RQ1.

Specifically, under MBFL method considering execution paths, in the dataset Defects4J, the proportions of faulty versions that satisfy

Table 6: Comparison of the MBFL method considering execution results and method considering execution paths in real-world programs.

Datasets	Projects	MBFL considering execution results			MBFL considering execution paths					
		Assertion 1	Assertion 2	Ass.1&2	Assertion 1 ($influ^{f}_{m_e} > influ^{f}_{m_n}$)		Assertion 2 ($influ^{p}_{m_n} > influ^{p}_{m_e}$)		Ass.1 & Ass.2	
Defects4J	common-math	28.8%	89.2%	**28.8%**	5.5%	7.9%	93.6%	90.9%	0.9%	**0.6%**
	Joda-Time				8.0%		88.0%		4.0%	
	commons-lang				15.4%		82.1%		0.0%	
	commons-csv				6.7%		93.3%		0.0%	
Bears	FastXML-jackson				0.0%	0.0%	100.0%	100.0%	0.0%	**0.0%**
	traccar-traccar				0.0%		100.0%		0.0%	

Assertion 1, Assertion 2, and both Assertion 1 and Assertion 2 are 7.9%, 90.9%, and 0.6%, respectively. In the dataset Bears, these proportions are 0.0%, 100.0%, and 0.0%, respectively. For all real-world programs, these proportions are: 6.1%, 91.5%, and 0.8%, respectively. By comparing the proportion of faulty versions that simultaneously satisfy Assertion 1 and Assertion 2 for MBFL method considering execution results (28.8%) and MBFL method considering execution paths (0.6%) in real-world programs, it is clear that MBFL method considering execution paths does not improve on the issue where MBFL method considering execution results struggles to follow the intuition in the real world. In fact, MBFL method considering execution paths is even more challenging in terms of aligning with the intuition.

The difficulty in MBFL method considering execution paths to achieve better results than the MBFL method considering execution results may lie in the coarser granularity of its execution path statistics. MBFL method considering execution paths operates at the program block level, whereas the MBFL method considering execution results works at the finer statement level. Due to the coarser granularity of execution path statistics, MBFL method considering execution paths fails to capture subtle differences in execution paths before and after mutation. It is possible that the execution path changes for successful and failing test cases before and after mutation differ, but at the coarser granularity, they are classified as the same.

We also analyze based on the raw data presented in Figure 2(c). In the first row of Figure 2(c), we find that most of the points are concentrated on the wrong region, which means most of the faulty versions do not conform to Assertion 1. In the second row of Figure 2(c), most of the points are located in the correct region, which means most of the faulty versions conform to Assertion 2. Additionally, the situation in Figure 2(c) is similar to the situation in Figures 2(a) and 2(b) in RQ1, a large number of points are located on the $influ^{f}_{m_e} = 0$ or $influ^{p}_{m_e} = 0$ axes. This indicates that in MBFL method considering execution paths, many of the m_e mutations in faulty versions are not killed by any test cases. This further suggests that the current mutation operators often fail to alter the output results of test cases, particularly for mutations generated from faulty statements.

✍ **Finding.** Similar to the challenges faced by MBFL methods considering execution results, MBFL's intuition also struggles to hold for methods considering execution paths in real-world programs.

4.3 RQ3: Analysis of Mutation Operators

In the experiments of RQ1 and RQ2, we found that current mutation operators often fail to change the output results of test cases, especially for mutants generated from faulty statements. Therefore, in RQ3, we introduced more advanced mutation operators, analyzed which ones are more effective, and ultimately summarized the characteristics of effective mutation operators and made recommendations.

Table 7 lists all the mutation operators that have acted on faulty statements. In order to satisfy condition 1) mentioned in RQ3, the mutation operator should occur as often as possible, with more occurrences indicating that the mutation operator's action condition is more likely to be satisfied in a real-world program. Therefore, we have highlighted the top 5 most frequent mutation operators in bold in Table 7. The results show that less common mutation operator types occupy 3 of the top 5 most frequent operator types, with only InlineConstant and NegateConditionals being the only operator types appearing in Table 4.

At the same time, to meet the condition 2) mentioned in RQ3, in addition to appearing sufficiently often, the mutation operators need to have a high kill ratio among the total occurrences of the mutants they generate. Therefore, we have bolded the top 5 mutation operators that have the highest kill ratio and appear more than 5 times. As can be seen from the results, the top-ranked mutation operators can all have a probability of being killed that is higher than 70% in absolute terms, which means that these mutation operators are more likely to be killed in real-world projects. As mentioned earlier for several of the most frequent occurrences of mutation operators, these mutation operators are similarly uncommon, and only NegateConditionals is present in Table 4.

Taken together, the mutation operators that rank high in both frequencies of occurrence and percentage of kills include: negate conditionals mutator, remove conditional mutator, math mutator, and so on. This result indicates that most naive mutation operators perform poorly in real-world programs, and therefore, more complex mutation operators need to be purposefully designed, such as

Table 7: The performance of all mutation operators in real-world programs.

Name of Mutator	Execution Result			Killed%	Description
	Killed	Survived	Sum		
InlineConstant	60	55	**115**	52.2	The inline constant mutator mutates inline constants that is a literal value assigned to a non-final variable.
NegateConditionals	66	23	**89**	**74.2**	The negate conditionals mutator will mutate all conditionals found.
RemoveConditional	92	85	**177**	52.0	The remove conditionals mutator will remove all conditionals statements such that the guarded statements always execute.
NonVoidMethodCall	97	52	**149**	65.1	The non void method call mutator removes method calls to non void methods.
EmptyObjectReturnVals	4	0	4	100.0	The return values mutator mutates the return values of method calls.
ArgumentPropagation	20	21	**41**	48.8	The mutator that replaces method call with one of its parameters of matching type.
PrimitiveReturns	11	3	14	**78.6**	Replaces int, short, long, char, float and double return values with 0.
ConstructorCall	14	3	17	**82.4**	The mutator that replaces constructor calls with null values.
NakedReceiver	7	12	19	36.8	The mutator that replaces method call with a naked receiver.
ConditionalsBoundary	12	20	32	37.5	The conditionals boundary mutator replaces the relational operators <, <=, >, >=with their boundary counterpart.
MemberVariable	7	1	8	**87.5**	The mutator mutates classes by removing assignments to member variables.
Math	22	9	31	**71.0**	The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation.
NullReturnVals	9	3	12	75.0	The mutator replaces return values with null.
Increments	4	1	5	80.0	The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables.
RemoveIncrements	2	3	5	40.0	The mutator that removes local variable increments.
VoidMethodCall	3	3	6	50.0	The void method call mutator removes method calls to void methods.
InvertNegs	1	0	1	100.0	The invert negatives mutator inverts negation of integer and floating point variables.
BooleanTrueReturnVals	9	7	16	56.3	The mutator replaces the unmutated return value true with false and replaces the unmutated return value false with true.

dynamically calling functions or modifying program branch conditions. The mutation operators that performed well and showed statistical significance in RQ3 can provide guidance for the practical application of MBFL and the design of mutation operators.

✍ **Finding.** Simple mutation operators are often ineffective on real-world programs, necessitating the design of more sophisticated mutation operators.

5 Threats to Validity

Internal Validity reflects on the definition of Faulty statements. First, in real-world programs, faulty statements are often not as easy to define as they are in simulated environments. In our work, the data for faulty statements in Defects4J and Bears come from patches provided by the dataset itself, and we define statements with changes in the patches as faulty statements. However, this strategy may lead to inaccurate definitions of faulty statements, in some faulty versions, the patch did not modify any statements in the program. In fact, there are many other ways of modifying programs, such as adding a totally new function or removing parts of the old code. Unfortunately, most of the MBFL works that we have investigated conform to this strategy without exception [23, 27], so we modified this strategy in some of the scenarios. In some of the patches, the insertion of statements both before and after a

statement led to the patch recognizing that the statement had also been changed, so we rechecked the patches and weeded out this situation.

Second, we apply MBFL tools to generate mutants and collect execution data across our benchmarks. Because real-world systems are typically Java and smaller programs often C, we use a customized tool modified from [23] for Codeflaws and PITEST [4] for Defects4J and Bears, filtering operators to ensure consistency. We also integrate IsoVar for MBFL method considering execution paths [35]. Notably, tool heterogeneity and language support discrepancies may threaten result validity.

External Validity refers to the degree to which our study results and findings can be generalized in other cases (e.g., other datasets of different sizes or languages). We consider two sets of real-world programs that are most commonly used in fault localization, which contain a large number of different types of real-world projects. Therefore, we argue that the communities and projects are representative and can increase the generalizability of our study results.

6 Disscussion

In our experiments, we found a significant weakening of the trend in intuition when reducing the total number of generated mutants per version. It is obvious that when the number of mutants decreases, the results will be more affected by special circumstances. As an example, when we set the number of mutants in Bears-2 to infinite generation, 52,977 mutants were generated, and Assertion 1 is clearly validated where the values of $influ^f_{m_e}$ and $influ^f_{m_n}$ are 0.16667 and 0.01313 respectively, that is, $influ^f_{m_e}$ is greater than $influ^f_{m_n}$. But when the upper limit of the number of mutants is set to 5,000, the trend simply disappears where $influ^f_{m_e}$ and $influ^f_{m_n}$ are 0 and 0.00628. This result does not conform to Assertion 1. So the number of mutants can only be increased as much as possible for the sake of the two assertions, but this results in a significant increase in the analysis time, and the longest time we observed for strong mutation testing in the real-world faulty version can be up to 21.2 hours, and the MBFL method considering execution paths is even longer, up to 65.1 hours, and that's only the time consuming for one version.

Therefore, MBFL for localization in the real world entails a trade-off between a long analysis phase and imprecise localization outcomes, which will significantly affect the practical usefulness of MBFL.

7 Related Work

Fault localization is an important research direction in the field of software testing, and there are many related me thods that have been proposed [22]. One of the most representative localization methods is spectrum-based fault localization (SBFL) [1, 6, 17, 18, 28, 37]. Its core idea is to rank code entities by suspiciousness scores calculated from execution spectra (e.g., statement coverage). However, studies [27, 31] reveal its practical limitations due to insufficient spectrum information. In contrast, mutation-based fault localization (MBFL) [16] enhances diagnostic capability by generating mutated program variants that provide richer execution behavior data for analysis.

Current mainstream MBFL methods primarily focus on enhancing fault localization through additional information integration and leveraging deep learning for mutant generation.

Introducing other sources of information into the MBFL to help with localization is also a valid direction for improvement, such as program spectrum information [5, 12, 35], program coverage information [29], and information on all mutants [10]. However, this information is not directly related to the intuition behind MBFL; it merely serves as an aid in the fault localization process.

Recent studies have replaced rule-based mutant generation with deep learning approaches, resulting in higher-quality mutants. Neural-MBFL leverages pretrained models like CodeBERT to generate context-aware, semantically valid mutants in place of rule-based mutations, yielding faults that better reflect real faults [11]. μBERT uses mask-prediction to produce natural mutants that adhere to coding conventions, eliminating the need for handcrafted mutation rules [8]. LEAM employs a two-stage, AST-driven deep learning pipeline: it first pinpoints mutation locations within the AST, then leverages contextual and structural cues to predict grammar-rule transformations, producing high-confidence, syntactically valid mutants [33].

8 Conclusion

Mutation-based fault localization (MBFL) has demonstrated impressive accuracy on synthetic benchmarks, and Moon et al. [27] even confirmed its underlying intuition in those controlled conditions. However, our findings suggest that this success is largely due to the fact that simulated faults align neatly with simple mutation operators. In contrast, real-world defects exhibit far greater complexity and diversity, so basic mutations rarely capture or resolve genuine bugs. This discrepancy raises serious doubts about MBFL's intuition.

To answer this question, this paper revisits the intuition of MBFL under various datasets, including a simulated dataset like Codeflaws, as well as real-world programs like Defects4J and Bears. In order to remove factors unrelated to intuition, we built an algorithm for directly responding to the effectiveness of intuition. Based on this algorithm, this paper poses an in-depth question: Does the intuition of MBFL hold in the real world? Our experimental results reveal that MBFL behaves significantly differently in real-world scenarios compared to simulated datasets. In real-world programs, the proportion of fault versions that align with MBFL's intuition is less than half of that observed in simulated datasets, indicating that MBFL's naive premise is difficult to uphold in practice. Therefore, we analyzed the reasons why MBFL doesn't work in the real world and found that the mismatch between the mutation operator and the fault in the real world is one of the important factors affecting its effectiveness, so we counted the effectiveness of the mutation operator based on the experimental results and gave the recommended mutation operators.

In the future, we will continue to investigate the factors that affect the robustness of MBFL in the real world, as well as improve the MBFL method based on these factors.

Acknowledgments

This work was partially supported by National Key R&D Plan of China (No.2024YFF0908003), National Natural Science Foundation of China (No.62472326 and No.62202344), and CCF-Zhipu Large Model Innovation Fund (No. CCF-Zhipu202408).

References

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

[2] Hiralal Agrawal, Richard A DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur, and Eugene Spafford. 1989. *Design of mutant operators for the C programming language*. Technical Report. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue ....

[3] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2016. Assessing and comparing mutation-based fault localization techniques. *arXiv preprint arXiv:1607.05512* (2016).

[4] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*. 449–452.

[5] Zhanqi Cui, Minghua Jia, Xiang Chen, Liwei Zheng, and Xiulei Liu. 2020. Improving software fault localization by combining spectrum and mutation. *IEEE Access* 8 (2020), 172296–172307.

[6] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight bug localization with AMPLE. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. 99–104.

[7] Vidroha Debroy and W Eric Wong. 2014. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software* 90 (2014), 45–60.

[8] Renzo Degiovanni and Mike Papadakis. 2022. μbert: Mutation testing using pre-trained language models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 160–169.

[9] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10 (2005), 405–435.

[10] Bin Du, Yuxiaoyang Cai, Haifeng Wang, Yong Liu, and Xiang Chen. 2022. Improving the Performance of Mutation-based Fault Localization via Mutant Bias Practical Experience Report. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 309–320.

[11] Bin Du, Baolong Han, Hengyuan Liu, Zexing Chang, Yong Liu, and Xiang Chen. 2024. Neural-MBFL: Improving mutation-based fault localization by neural mutation. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1274–1283.

[12] Arpita Dutta and Sangharatna Godboley. 2021. Msfl: A model for fault localization using mutation-spectra technique. In *Lean and Agile Software Development: 5th International Conference, LASD 2021, Virtual Event, January 23, 2021, Proceedings 5*. Springer, 156–173.

[13] Luxi Fan, Zheng Li, Hengyuan Liu, Doyle Paul, Haifeng Wang, Xiang Chen, and Yong Liu. 2023. SGS: Mutant Reduction for Higher-order Mutation-based Fault Localization. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 870–875.

[14] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 464–475.

[15] William E. Howden. 1982. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering* 4 (1982), 371–379.

[16] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[17] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[18] James A Jones, Mary Jean Harrold, and John T Stasko. 2001. Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer.

[19] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[20] Jeongho Kim, Jindae Kim, and Eunseok Lee. 2019. VFL: Variable-based fault localization. *Information and software technology* 107 (2019), 179–191.

[21] Jisung Kim and Byungjeong Lee. 2023. MCRepair: multi-chunk program repair via patch optimization with buggy block. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. 1508–1515.

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

[23] Zheng Li, Butian Shi, Haifeng Wang, Yong Liu, and Xiang Chen. 2021. Hmbfl: Higher-order mutation-based fault localization. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 66–77.

[24] Zheng Li, Haifeng Wang, and Yong Liu. 2020. Hmer: A hybrid mutation execution reduction approach for mutation-based fault localization. *Journal of Systems and Software* 168 (2020), 110661.

[25] Hengyuan Liu, Zheng Li, Baolong Han, Yangtao Liu, Xiang Chen, and Yong Liu. 2024. Delta4Ms: Improving mutation-based fault localization by eliminating mutant bias. *Software Testing, Verification and Reliability* (2024), e1872.

[26] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 468–478.

[27] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

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

[29] Mike Papadakis and Yves Le Traon. 2014. Effective fault localization via mutation analysis: A selective mutation approach. In *Proceedings of the 29th annual ACM symposium on applied computing*. 1293–1300.

[30] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

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

[32] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 180–182.

[33] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[34] Haifeng Wang, Zheng Li, Yong Liu, Xiang Chen, Doyle Paul, Yuxiaoyang Cai, and Luxi Fan. 2022. Can higher-order mutants improve the performance of mutation-based fault localization? *IEEE Transactions on Reliability* 71, 2 (2022), 1157–1173.

[35] Ming Wen, Zifan Xie, Kaixuan Luo, Xiao Chen, Yibiao Yang, and Hai Jin. 2022. Effective Isolation of Fault-Correlated Variables via Statistical and Mutation Analysis. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2053–2068.

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

[37] W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. IEEE, 449–456.

[38] Shumei Wu, Zheng Li, Yong Liu, Xiang Chen, and Mingyu Li. 2023. GMBFL: Optimizing Mutation-Based Fault Localization via Graph Representation. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 245–257.

[39] Yue Yan, Shujuan Jiang, Yanmei Zhang, and Cheng Zhang. 2023. An effective fault localization approach based on PageRank and mutation analysis. *Journal of Systems and Software* 204 (2023), 111799.

[40] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[41] 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*. 958–969.