Improving the Performance of Mutation-based Fault Localization via Mutant Bias

Bin Du[†], Yuxiaoyang Cai[†], Haifeng Wang[†*], Yong Liu[†*], Xiang Chen[‡§]
[†]*College of Information Science and Technology, Beijing University of Chemical Technology,* Beijing 100029, China
[‡]*School of Information Science and Technology, Nantong University,* Nantong 226019, China
[§]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences,* Beijing 100093, China
Email: h.f.wang@hotmail.com

***Abstract*—Mutation-Based Fault Localization (MBFL) is one of the most widely studied techniques. MBFL adopts mutation analysis to generate mutants for revealing potential faults in the program. Previous studies proposed approaches to optimize MBFL in terms of efficiency and accuracy. However, these approaches ignored the difference of mutants on correct entities (such as statements) and faulty entities, which we refer this kind of difference as mutant bias. In this study, we identify and analyze the impact of mutant bias on MBFL. We find that the mutant bias may introduce effects to statement suspiciousness and negatively influence the fault localization accuracy of MBFL. To mitigate the mutant bias, we propose Delta4Ms, a model that captures the mutant bias from the mutants of the same statements. Then the real suspiciousness is obtained by removing the bias from the practical suspiciousness. To evaluate the performance of our proposed method, we conduct experimental studies on 320 real-world programs from Codeflaws. The experimental results show that Delta4Ms improves the fault localization accuracy of MBFL. Besides, Delta4Ms outperforms the state-of-the-art SBFL and three MBFL techniques significantly. Moreover, Delta4Ms ranks 94 and 161 of the target faults within the top-5 suspicious statements in single-fault and multiple-fault programs, respectively.**

***Index Terms*—Software testing, Fault localization, Mutation testing, Mutation-based fault location**

I. Introduction

Fault localization is an expensive and time-consuming phase in the software debugging activity [1]. Developers spend a large amount of effort to locate the root cause of the program failures. As a result, automated fault localization techniques (such as spectrum-based techniques [2]–[4], information retrieval-based techniques [5]–[7], machine learning-based techniques [8]–[10], and mutation-based techniques [11]–[15]) have been widely studied.

Among these automatic fault localization techniques, Spectrum Based Fault Localization (SBFL) is the most widely studied technique [2]–[4], [16]. SBFL obtains coverage information and execution results of program elements (In this study, we concern statement-level faults) and uses this information to calculate the probability of faulty program entities (i.e., suspiciousness) and returns a ranking list of the program entities [17]. Previous studies [2]–[4] have shown that SBFL is simple, lightweight, and efficient. However, such a technique cannot achieve a better fault localization accuracy since it only adopts program spectrum information [12], [18].

Mutation-Based Fault Localization (MBFL) is a kind of technique that utilize mutation analysis [19]. MBFL works by making simple changes to the programs, which can generate faulty programs (i.e., $mutants$ [11]). Two pioneer MBFL techniques (i.e., Metallaxis [13] and MUSE [11]) are proposed and this kind of technique performs better than the SBFL techniques [11], [13]. Since Metallaxis outperforms MUSE in the study of Pearson et al. [20], we mainly focus on Metallaxis and propose a method to improve its accuracy.

Recently, researchers have improved MBFL by combining different techniques, such as utilizing higher-order mutants to simulate complex faults to improve the location accuracy or addressing the practical issues related to the high cost of MBFL [21], [22]. Meanwhile, a theoretical framework has been proposed for understanding the MBFL [1], [23], which moves a step forward in revealing the reasons why MBFL techniques are effective.

However, there is no research investigating the intrinsic limitation of the traditional MBFL technique Metallaxis. Metallaxis takes the maximum suspiciousness of the mutants to the statements but cannot capture the difference of mutants on correct entities (i.e., statements or functions) and faulty entities, which is called as $mutant\ bias$. Especially, such bias may lead Metallaxis to assign an inflated (i.e., higher than the actual) suspiciousness on a correct entity or a deflated one on a faulty entity. We notice that the suspiciousness of the program entity can be assessed by the execution behavior of the mutants, and this analysis inspires our study.

In this paper, we propose a model Delta4Ms (**Delta for M**utant**s**) to address the intrinsic limitation of MBFL. We formulate the problem of mutant bias and present a theoretical model to analyze its influence on MBFL. Delta4Ms is built upon the practical observation (i.e., calculated suspiciousness of program entities) and signal theory that the information of a signal is often accompanied by noise (i.e., the effects come from mutants). Delta4Ms models two fundamental components of the actual suspiciousness of program entities and the effects brought by mutant bias, as the $desired\ signal$ and $false\ signal$. Then, Delta4Ms returns the desired signal by removing the false signal. Furthermore, to obtain a precise mutant bias, we introduce higher-order mutants (i.e., multiple changes in single statements) for Delta4Ms.

The experiments on 320 real-fault programs show that Delta4Ms outperform SBFL(i.e., GP13 [24], Ochiai [25], Op2 [26], and Tarantula [27]) and three MBFL techniques (i.e., MUSE [11], Metallaxis [13], and MCBFL-hybrid-avg [20] in terms of the metrics $EXAM$, MAP and $Top\text{-}N$. Also, Delta4Ms can significantly improve the accuracy of MBFL both in single-fault programs and multiple-fault programs by 134.9% and 100.9% respectively in terms of the metric MAP. Detailed statistical results indicate that Delta4Ms significantly improve the fault localization effectiveness of MBFL techniques. However, Delta4Ms also needs more costs than MBFL techniques, which leaves room for reducing the cost of Delta4Ms in the future.

In summary, the contributions of our study can be summarized as follows:

- We systematically investigate the influence of mutant bias on the performance of MBFL techniques. We formalize the model and model the mutant bias with signal theory and analyze its effects. Moreover, we provide a theoretical foundation of our model via proof and derivation.
- We propose Delta4Ms, which can mitigate the negative effect of mutant bias and improve the performance of MBFL.
- By conducting the empirical study on a real-world benchmark Codeflaws, We demonstrate the impact of mutant bias on the performance of MBFL in single-fault and multiple-fault programs and show the effectiveness of Delta4MS.
- We share both the dataset and source code of our study on the GitHub repository[1] for other researchers in replicating our study and evaluating their proposed new techniques.

The rest of this paper is organized as follows. Section II summarizes the background and related work. Section III shows our research motivation by a simple example. Section IV describes the details of our proposed method. Section V illustrates the experimental setup and Section VI analyzes our experimental results. Finally, Section VII summarizes our study with potential future work.

II. Background and Related Work

A. Mutation-based Fault Localization

Mutation-based fault localization (MBFL) is a well-studied technique that is based on mutation analysis [18]. Mutation analysis works by seeding artificial faults (i.e., $mutants$) in the program under test. The rules that define how mutants are generated are called $mutation\ operators$ (see Table I). Mutation analysis uses mutants to evaluate the quality of test cases based on how their behavior differs from that of the original program [28]. If the behavior of a mutant is different from the original program for a test case, we say that this mutant is $killed$ or $detected$. Otherwise, we say that the mutant is $notkilled$ or $live$ [29].

[1]The url of the GitHub repository will be available after this paper is accepted.

MBFL identifies suspicious mutants and uses them to identify faulty statements. Papadakis and Traon firstly applied MBFL to fault localization [12], [13] and they found MBFL can effectively locate "unknown" faults in the program. Later, they proposed Metallaxis [13] to further optimize MBFL. From another aspect, Moon et al. [11] observed that mutants are more likely to fail on the correct statements and pass on the faulty statements. Then they proposed MUSE that can significantly outperform the state-of-the-art SBFL technique Op2 [26]. Pearson et al. [20], [30] conducted an empirical study and indicated that Metallaxis can outperform MUSE on fault localization effectiveness. Therefore, we focus our study on Metallaxis in this study. The classical MBFL approach includes the following four steps:

*1) **Obtain statements covered by failed test cases** :* MBFL firstly executes the program under test P against the test suite T. Then the coverage information and the results (passed or failed) can be collected. Based on the execution results, T is split into the passed test set T_p and the failed test set T_f. The statements covered by the failed test cases are denoted as Cov_f.

*2) **Generate and execute mutants**:* MBFL uses mutation operators to insert faults into the statements from Cov_f. Then, the mutants of the statement s are generated, denoted as $\mathcal{M}(s)$. Each mutant m in $\mathcal{M}(s)$ are executed by the test cases in T. Later, the test cases can be classified into $T_n(m)$ and $T_k(m)$, where $T_n(m)$ is the set of test cases that cannot kill the mutant m and $T_k(m)$ is the set of test cases that can kill the mutant m.

TABLE I
MUTATION OPERATORS FOR MBFL

Mutation Operator	Description	Example
CRCR	Required constant replacement	a=b + *p → a=0 + *p
OAAN	Arithmetic operator mutation	a + b → a * b
OAAA	Arithmetic assignment mutation	a += b → a -= b
OCNG	Logical context negation	if(a) → if(!a)
OIDO	Increment/decrement mutation	++a → a++
OLLN	Logical operator mutation	a && b → a || b
OLNG	Logical negation	a && b → !(a && b)
ORRN	Relational operator mutation	a < b → a <= b
OBBA	Bitwise assignment mutation	a &= b → a |= b
OBBN	Bitwise operator mutation	a & b → a | b
OCOR	Cast operator replacement	int a → float a
SRSR	Return statement replacement	return 0 → return 1
VTWD	Twiddle Mutations	a = b → a = b + 1
VDTR	Domain Trap	c = a → c = a *0
SSDL	Statement deletion	a = 1 → <no-op>

*3) **Compute the suspiciousness**:* MBFL first computes the suspiciousness of mutants based the following four parameters: $a_{np}(m) = |T_n(m) \cap T_p|$, $a_{kp}(m) = |T_k(m) \cap T_p|$, $a_{nf}(m) = |T_n(m) \cap T_f|$, and $a_{kf}(m) = |T_k(m) \cap T_f|$, where $a_{np}(m)$ denotes the number of the passed test cases which cannot kill m, $a_{kp}(m)$ denotes the number of the passed test cases which can kill m, $a_{nf}(m)$ denotes the number of the failed test cases which cannot kill m, and $a_{kf}(m)$ denotes the number of the failed test cases which can kill m. For MUSE, the following two parameters will be used: (1) the parameter $f2p$ denotes the

number of cases in the whole program there a mutant caused any failing test to pass. (2) the parameter $p2f$ denotes the number of cases in the whole program there a mutant caused any passing test to fail.

Table II listed five MBFL formulas (i.e., GP13 [24], Ochiai [25], Op2 [26], Tarantula [27], MUSE [11]) used in our study and these formulas have been widely used in previous studies [24]–[27].

TABLE II
SUSPICIOUSNESS FORMULAS FOR MBFL

Name	Formula
GP13 [24]	$Sus(m) = a_{kf}\left(1 + \frac{1}{2a_{kp}+a_{kf}}\right)$
Ochiai [25]	$Sus(m) = \frac{a_{kf}}{\sqrt{(a_{kf}+a_{nf})(a_{kf}+a_{kp})}}$
Op2 [26]	$Sus(m) = a_{kf} - \frac{a_{kp}}{a_{kp}+a_{np}+1}$
Tarantula [27]	$Sus(m) = \frac{\frac{a_{kf}}{a_{kf}+a_{kp}}}{\frac{a_{kf}}{a_{kf}+a_{kp}}+\frac{a_{kp}}{a_{kp}+a_{np}}}$
MUSE [11]	$Sus(m) = (a_{kf} + a_{nf}) - \frac{f2p}{p2f}(a_{kp} + a_{np})$

Then, the suspiciousness of the statement s is assigned by the maximum suspiciousness of the mutants generated by s: $sus(s) = max\{sus(m_1), sus(m_2), \cdots, sus(m_q)\}$, where $m_1, \cdots, m_q$ are mutants in $\mathcal{M}(s)$ and $sus(s)$ is the suspiciousness of the statement s. For MUSE, max will be changed to $average$.

*4) **Generate fault localization report**:* Finally, MBFL arranges the statements in descending order based on their suspiciousness value and returns a ranking list. Later the developers use this ranking list to localize the faults in the program and then fix them.

Based on the description of the above process, we can find that MBFL works based on the assumption that mutants killed mostly by the failed test cases have a connection with the program faults. Recent studies [13], [18] also demonstrated that MBFL could significantly outperform other types of fault localization techniques (such as spectrum-based fault localization techniques [12], [18]).

B. Higher-order Mutation Testing

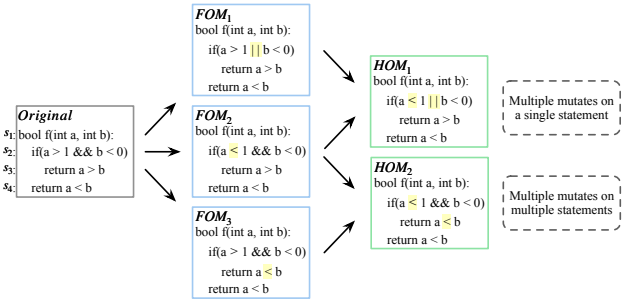


Fig. 1. An example of FOMs and HOMs

Higher-order Mutation Testing (HOM Testing) was first introduced by Jia and Harman [31]. According to their study, mutants can be classified into two types: First-Order-Mutants (FOMs) and Higher-Order-Mutants (HOMs). Both these two types of mutants were first proposed by Offutt et al. [32], and these two types were named simple mutants and complex mutants in their study.

Definition 1 (FOM). A First Order Mutant (FOM) of a program p is generated by making a single syntactic change to p. The rules of generating FOMs is called first-order mutation operators FOP.

Definition 2 (HOM). A Higher-Order Mutant (HOM) of a program p is generated from p by applying k operators from FOP. This HOM is said to be a k^{th} order mutant of p and is recorded as k-HOM.

Note that in most of the previous studies [33], the k operators are applied in k different statements, which HOMs mutate on multiple statements. Also, HOMs only mutating on single statements are another kind of HOMs. In this study, to reveal the mutant bias more accurately in MBFL, we adopt the first kind of HOMs to expand the mutant space in a single statement.

Figure 1 shows an example of the FOM and two types of the HOM. The original program is mutated into three FOMs (FOM_1, FOM_2, FOM_3). Specifically, FOM_1 and FOM_2 mutate the statement s_2, while FOM_3 mutates the statement s_3. Then, HOM_1 is formed by FOM_1 and FOM_2 that mutate only on s_2, while HOM_2 is formed by FOM_2 and FOM_3 that mutate both on s_2 and s_3.

In higher-order mutation testing, HOMs are considered more complex faults. Therefore they can solve the complex mutation testing problems. For example, firstly, HOMs can be used to reduce testing costs [22], [34], [35]. Secondly, HOMs can be applied to evaluate the quality of test suites [36] and test data generation [37]. Thirdly, some researchers employed HOMs for coupling effect analysis [19], [38] and fault localization [39], [40].

III. RESEARCH MOTIVATION

In this section, we first present a motivation example to illustrate the mutant bias in traditional mutation-based fault localization techniques. Based on it, we discuss the motivation of our work by demonstrating the problem of how the accuracy of MBFL techniques is affected by mutant bias. Then we analyze the root cause of this problem and introduce our solution.

A. Motivation Example

Table III shows an example of how the mutant bias influences the performance of MBFL. The program is the one that with real faults from Codeflaws [41]. The program contains 17 statements where the statement s_7 (the correct code should be "if(i%2)") and the statement s_{12} (the correct code should be "if(false)") are both faulty. In this example, we assume that MBFL has two FOMs per statement, resulting in a total of 20 FOMs. The test suite for this program contains 14 passed test cases and 37 failed test cases.

TABLE III
A MOTIVATION EXAMPLE

Program		Mutant						Metallaxis		Delta4Ms		
			a_{kf}	a_{kp}	a_{nf}	a_{np}	Sus	Stmt. Sus	Rank	$\mathcal{M}$	Stmt. Sus	Rank
s_1	int main(int argc, char *argv[]){											
s_2	int n,a,b,c,sum;											
s_3	scanf("%d%d%d%d", &n,&a,&b,&c);											
s_4	int count =0;	FOM_1: int → short int	27	0	10	14	0.85	0.85	5	0.84	0.01	8
		FOM_2: int → char	25	0	12	14	0.82					
s_5	int i,j;											
s_6	for(i=0;i<=a;i++){	FOM_3: ++ → +=2	37	0	0	14	1.00	1.00	1	0.62	0.38	2
		FOM_4: <= → <	11	0	26	14	0.55					
		HOM_1: ++ → +=2, <= → <	11	0	26	14	0.55					
		HOM_2: ++ → +=2, <= → ==	12	12	25	2	0.40					
s_7	**if(i/2) //fault1. Correct: if(i%2)**	FOM_5: if(()) → if(!())	6	0	31	14	0.40	0.40	9	0.23	0.29	5
		FOM_6: / → +	0	4	37	14	0.00					
		HOM_3: if(()) → if(!()), / → +	0	4	37	14	0.00					
		HOM_4: if(()) → if(!()), / → *	10	0	27	14	0.52					
s_8	continue;											
s_9	for(j=0;j<=b;j++) {	FOM_7: ++ → +=2	9	0	28	14	0.49	0.49	7	0.46	0.36	4
		FOM_8: <= → ==	5	0	32	14	0.37					
		HOM_5: ++ → +=2, <= → ==	3	6	24	8	0.16					
		HOM_6: ++ → −−, <= → ==	30	6	7	8	0.82					
s_{10}	sum=0;											
s_{11}	sum=(n)-(int)(i*0.5)-(j*1);	FOM_9: 1 → -1	9	0	28	14	0.49	1.00	1	0.63	0.37	3
		FOM_{10}: * → /	37	0	0	14	1.00					
		HOM_7: 1 → -1, * → /	10	0	27	14	0.52					
		HOM_8: - → *, * → /	9	0	28	14	0.49					
s_{12}	**if(sum<0) //fault2. Correct: if(false)**	FOM_{11}: if(() → if(!())	28	0	9	14	0.87	0.87	4	0.42	0.45	1
		FOM_{12}: <→ !=	6	0	31	14	0.40					
		HOM_9: if(()) → if(!()), <→ >=	0	0	37	14	0.00					
		HOM_{10}: if(()) → if(!()), <→ <=	6	0	31	14	0.40					
s_{13}	continue;											
s_{14}	if((sum%2 ==0)&&((sum/2)<=c))	FOM_{13}: / → %	25	0	12	14	0.82	0.82	7	0.66	0.16	6
		FOM_{14}: <= → !=	15	6	22	8	0.54					
		HOM_{11}: / → *, if(()) → if(!())	22	6	15	8	0.68					
		HOM_{12}: && → ||, <= → >=	20	10	17	4	0.60					
s_{15}	count−−;}}	FOM_{15}: −− → +=2	9	0	28	14	0.49	0.49	7	0.43	0.06	7
		FOM_{16}: −− → -=2	5	0	32	14	0.37					
s_{16}	printf("%d",count);	FOM_{17}: count → count*-1	0	14	37	0	0.00	0.00	10	0.00	0.00	9
		FOM_{18}: count → count*2	0	14	37	0	0.00					
s_{17}	return 0;}	FOM_{19}: return 0 → return NULL	37	0	0	14	1.00	1.00	1	1.00	0.00	9
		FOM_{20}: return 0 → return -1	37	0	0	14	1.00					

The traditional MBFL Metallaxis executes the program against the test suite and we display the four parameters $\langle a_{kf}, a_{kp}, a_{nf}, a_{np} \rangle$ in the section of "Mutant". In this example, we use Ochiai [25] as the MBFL formula and the mutants' suspiciousness is in the column Sus of "Mutant" section.

For Metallaxis, it assigns the maximum value of mutant suspiciousness to the statements. For example, statement s_4 takes 0.85 as the the suspiciousness of s_4. Then, Metallaxis ranks the statements based on the suspiciousness and the faulty statements s_7 and s_{12} in 9-th and 4-th places, respectively.

For the traditional MBFL technique Metallaxis, the suspiciousness of statements is assigned by the maximum value of the mutants' suspiciousness. This kind of calculation method hinders the difference between mutants from the correct statements and faulty statements. In the study of Moon et al. [11], they observed that the failed test cases are more likely to pass on the mutants from the faulty statements and the passed test cases are more likely to fail on the mutants from the correct statements. This means that the mutants on the faulty statements are more "value" than the correct statements. For

example, FOM_5 and FOM_6 have large different value of suspiciousness for s_6, while FOM_1 and FOM_2 have closer suspiciousness for s_1. The Metallaxis with fewer mutants cannot perceive this kind of difference and we introduce HOMs to discover it.

B. How Delta4Ms alleviate the impact

In this section, we show an idea to inspire Delta4Ms. Our idea uses the mutant set of one statement as the reference to adjust the suspiciousness scores of the statement, which aims to alleviate the impacts of mutant bias.

We consider the calculated suspiciousness of the statement as a superposition of the $desired$ suspiciousness component (that reflects the real probability of the statement being faulty) and the $false$ suspiciousness component (that comes from the impact of mutant bias). We capture the latter by taking the average of the mutants' suspiciousness from the same statement. We remove it from the calculated suspiciousness of the statement to restore the $desired$ suspiciousness. In the "Delta4Ms" section of Table III, we present the detailed result of our solution.

In Table III, Delta4Ms first generates HOMs on single statements. For example, the statement s_6 generates two HOMs (HOM_1 and HOM_2), where HOM_1 combined FOM_3 and FOM_4, HOM_2 combined FOM_4 and the mutant (<= $\rightarrow$ ==). The statements s_4, s_{15}, s_{16} and s_{17} only have FOMs due to the limits of the mutation positions.

Next, Delta4Ms calculates the average mutants' suspiciousness as the mutant bias for the statements. For example, the statement s_6 has four mutants (FOM_3, FOM_4, HOM_1, HOM_2) and the suspiciousness of them are 1.00, 0.55, 0.55, and 0.40, respectively. The average of the suspiciousness is calculated as $\frac{1.00+0.55+0.55+0.40}{4} = 0.62$. Similarly, the column of "$\mathcal{M}$" also lists out the mutant bias of other statements. We can find that the mutant bias for each statement is different.

Using the mutant bias as the references, we contrast the suspiciousness scores calculated in Metallaxis with them to restore the desired suspiciousness scores. In particular, we subtract them from the corresponding calculated suspiciousness scores in Metallaxis to compute the differences, which are shown in the column "Stmt." of the "Delta4Ms" section. We can find that Delta4Ms assigns the faulty statements s_7 and s_{12} with the suspiciousness of 0.29 and 0.45. Therefore, Delta4Ms improves Metallaxis by raising the rank of s_7 from 9-th to 5-th, and raising the rank of s_{12} from 4-th to 1-st, respectively.

Delta4Ms captures the mutant bias by considering more mutants. For the statements s_6 and s_{11}, they both rank at the top 1 in Metallaxis and with lower ranks in Delta4Ms after removing the mutation bias $\mathcal{M}$. The reason is that the mutants generated from correct statements with similar behavior leads to similar suspiciousness for the mutants. As shown in Table III, s_6 has three mutants (FOM_4, HOM_1, HOM_2) with similar suspiciousness (i.e., 0.55, 0.55, and 0.40), and with a mutant FOM_3 different from them. Therefore, the bias $\mathcal{M}$ (0.62) should be closer to the maximum value of the mutant suspiciousness (1.00).

From another aspect, for the faulty statements s_7 and s_{12}, Delta4Ms improves their ranking of them. It is matched to the idea that "mutants are a potential fix of the program [1]", which leads to more differences in the mutants from the same statement. As shown in Table III, s_7 has four mutants (FOM_5, FOM_6, HOM_3, and HOM_4) with different suspiciousness (0.40, 0.00, 0.00, and 0.52). Among them, the suspiciousness of FOM_6 and HOM_3 are significantly different from FOM_5 and HOM_4, which can result the bias $\mathcal{M}$ (0.23) far from the suspiciousness of HOM_4 (0.52). Therefore, the relative rank of the faulty statements can be improved by removing the mutant bias.

IV. Our Method

In this section, we present the model of Delta4Ms, which can capture and remove the impact of mutant bias on the performance of MBFL.

A. Problem Settings

As mentioned in Section III, the mutant difference influences the performance of traditional MBFL techniques. While traditional MBFL techniques generate a few mutants (i.e., FOMs) that cannot reveal the real bias caused by mutants. Therefore, we introduce higher-order mutants (multiple changes in a single statement) to jointly calculate the mutant bias.

Assume that a program $P = \langle s_1, s_2, \cdots, s_n \rangle$ with n statements. We further denote the corresponding mutants (i.e., FOMs and HOMs) generated from each statement s_i as $\mathcal{M}(s_i) = \langle m_{i1}, m_{i2}, \cdots, m_{ik} \rangle$.

Suppose that each fault in the program has been assessed by the MBFL technique and we denote the suspiciousness score computed by the technique for s_i as $\mathcal{S}_i$, where $i \in [1, n]$ is the index of the program statements.

As the settings mentioned previously, we model the suspiciousness $\mathcal{S}_i$ as a superposition of two kinds of signals: $\mathcal{D}_i$ and $\mathcal{M}_i$.

$$\mathcal{S}_i = \mathcal{D}_i + \mathcal{M}_i \tag{1}$$

We present Equation (1) to detail the modeling procedure. In Equation (1), $\mathcal{S}_i$ denotes the suspiciousness of the statement s_i computed by the MBFL techniques with both FOMs and HOMs. That is $\mathcal{S}_i = f(\mathcal{S}_{m_1}, \mathcal{S}_{m_2}, \cdots, \mathcal{S}_{m_k})$, where the function f is the method of calculating the values from mutants' suspiciousness $\mathcal{S}_{m_k}$. Traditional MBFL techniques takes the maximum value of the mutants' suspiciousness (Function f is $Maxing$). In the model, $\mathcal{S}_i$ is computed as the sum of $\mathcal{D}_i$ and $\mathcal{M}_i$. $\mathcal{D}_i$ denotes the observable suspiciousness signal beyond mutant bias, and $\mathcal{M}_i$ indicates the impact of mutant bias. We use all mutants generated from the statement s_i to approximate $\mathcal{M}_i$. In single processing, $\mathcal{M}_i$ is referred to as a false signal, and $\mathcal{D}_i$ is the desired signal. To assess the statement s_i, we compute $\mathcal{D}_i$ from $\mathcal{S}_i - \mathcal{M}_i$.

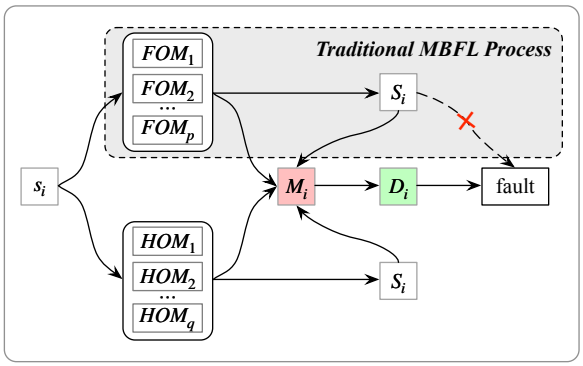


Fig. 2. Workflow of Delta4Ms

B. Delta for mutants (Delta4Ms)

Figure 2 shows the workflow of our proposed Delta4Ms, where the upper part (with background color) is the conventional solution that only adopts FOMs to compute the suspiciousness $\mathcal{S}_i$. The traditional MBFL technique locates the fault according to the ranking of the suspiciousness.

In our solution, we do not rank the suspicious program entities by $\mathcal{S}_i$ (i.e., "×" is marked on the dashed arrow in Figure 2). Rather, Delta4Ms first captures the mutant bias $\mathcal{M}$ from two kinds of mutants (FOMs and HOMs). Then we calibrate $\mathcal{S}_i$ (both obtained from FOMs and HOMs) to $\mathcal{D}_i$ from removing the $\mathcal{M}_i$ and use $\mathcal{D}_i$ to rank program entities. Each step of the Delta4Ms is illustrated in the following sections.

Definition 1 (Impact of Mutant Bias): For a statement s, we parameterize Equation(1) as $\mathcal{S}_i = \mathcal{D}_i + \mathcal{M}_i(ms_i)$ to express that $\mathcal{S}_i$ consists of two components: $\mathcal{D}_i$ is the observed failures on s_i, and $\mathcal{M}_i(ms_i)$ is the magnitude brought by the mutant bias from the mutant set ms_i (all mutants in ms_i are generated from s_i), which is also defined the impact of mutant bias in this study.

Our goal is to rank all the statements to reflect the extent of their suspiciousness, which is related to the fault. For this purpose, we compare two statements s_i and s_j by their suspiciousness $\mathcal{D}_i$ and $\mathcal{D}_j$. Thus, we define the following term:

$$\begin{aligned}\mathcal{M}_i(ms_i) &= \mathcal{S}_i(ms_i) - \mathcal{D}_i \\ &= f\left(\mathcal{S}_{m_x} | m_x \in ms_i\right) - \mathcal{D}_i\end{aligned} \tag{2}$$

where the function f is the method of obtaining the statement's suspiciousness from mutants' suspiciousness. If $\mathcal{C}$ is a constant, then $f(\mathcal{C}) = \mathcal{C}$.

To denote the difference between individual mutant m_x and the real suspiciousness $\mathcal{D}_i$, we define δ_{m_x} as:

$$\delta_{m_x} = \mathcal{S}_{m_x} - \mathcal{D}_i \tag{3}$$

Based on Equation (2) and Equation (3), the mutant bias $\mathcal{M}_i(ms_i)$ is represented by:

$$\begin{aligned}\mathcal{M}_i(ms_i) &= f\left(\mathcal{S}_{m_x} | m_x \in ms_i\right) - \mathcal{D}_i \\ &= f\left(\delta_{m_x} + \mathcal{D}_i | m_x \in ms_i\right) - \mathcal{D}_i \\ &= f\left(\delta_{m_x} | m_x \in ms_i\right) + f\left(\mathcal{D}_i\right) - \mathcal{D}_i \\ &= f\left(\delta_{m_x} | m_x \in ms_i\right) + \mathcal{D}_i - \mathcal{D}_i \\ &= f\left(\delta_{m_x} | m_x \in ms_i\right)\end{aligned} \tag{4}$$

Then our goal is to calculate the mutant difference δ_{m_j} in the mutant set ms_i. For this purpose, we define the following term to describe the average difference between any two mutants in ms_i:

$$\overline{\delta^{ms_i}_{m_x,m_y}} = \frac{\sum\limits_{m_x,m_y \in ms_i} \delta_{m_x} - \delta_{m_y}}{k^2} \tag{5}$$

where $k = |ms_i|$ is the number of mutants in ms_i.

Then, we take Equation (3) into Equation (5), introducing to:

$$\begin{aligned}\overline{\delta^{ms_i}_{m_x,m_x}} &= \frac{\sum\limits_{m_x,m_y \in ms_i} \left[\left(\mathcal{S}_{m_x} - \mathcal{D}_i\right) - \left(\mathcal{S}_{m_y} - \mathcal{D}_i\right)\right]}{k^2} \\ &= \frac{1}{k^2} \cdot \sum_{m_x,m_y \in ms_i} \left(\mathcal{S}_{m_x} - \mathcal{S}_{m_y}\right) \\ &= \frac{1}{k^2} \cdot \sum_{m_x \in ms_i} \left(k \cdot \mathcal{S}_{m_x} - \sum_{m_y \in ms_i} \mathcal{S}_{m_y}\right) \\ &= \frac{1}{k^2} \cdot \left(\sum_{m_x \in ms_i} k \cdot \mathcal{S}_{m_x} - \sum_{m_y \in ms_i} k \cdot \mathcal{S}_{m_y}\right) \\ &= \frac{\sum\limits_{m_x \in ms_i} \mathcal{S}_{m_x}}{k} - \frac{\sum\limits_{m_y \in ms_i} \mathcal{S}_{m_y}}{k}\end{aligned} \tag{6}$$

The law of large numbers states that as the size of samples grows, their mean gets closer to the average of the whole population. Therefore, the mutant difference can be approximated by the mean value of the mutant difference in ms_i, $\delta_{m_x} \approx \frac{\sum\limits_{m_y \in ms_i} \mathcal{S}_{m_y}}{k}$. As a result, we deduce Equation (4) as follows:

$$\begin{aligned}\mathcal{M}_i(ms_x) &= f\left(\delta_{m_y} | m_y \in ms_i\right) \\ &\approx f\left(\frac{\sum\limits_{m_y \in ms_i} \mathcal{S}_{m_y}}{k} | m_y \in ms_i\right) \\ &= \frac{\sum\limits_{m_y \in ms_i} \mathcal{S}_{m_y}}{k}\end{aligned} \tag{7}$$

Therefore, we have shown how to generate a ranking list of suspicious program elements in our model.

C. Why Using Higher-Order Mutants

As defined in Section IV-B, the mutant bias $\mathcal{M}_i$ is captured from the mutants from the same statement s_i. According to the law of large numbers, the greater the number of mutants used, the higher the probability that the mean of the suspiciousness will be close to the expected value of mutant bias. We introduce HOMs on the single statements to extend the mutant space of traditional MBFL techniques for calculating mutant bias precisely.

V. Experimental Setup

A. Research Questions

To evaluate the effectiveness of Delta4MS, we investigate the following three research questions:

- **RQ1**: How does Delta4MS perform in the single-fault programs in terms of fault localization effectiveness?
- **RQ2**: How does Delta4MS perform in the multiple-fault programs in terms of fault localization effectiveness?
- **RQ3**: How much execution cost does Delta4MS need when compare to SBFL and MBFL techniques?

RQ1 and RQ2 are designed to evaluate the fault localization effectiveness of Delta4MS in terms of the $EXAM$, $Top\text{-}N$, and MAP metrics. We compare Delta4MS with four SBFL techniques (i.e., GP13 [24], Ochiai [25], Op2 [26], and Tarantula [27]) and three MBFL techniques (i.e., MUSE [11], Metallaxis [13], and MCBFL-hybrid-avg [20],). RQ3 is designed to evaluate the efficiency of Delta4M when compared to SBFL and MBFL techniques. We measure the mutation execution cost in terms of the MTP metric to compare the efficiency of these techniques.

In our experiments, we use four suspiciousness formulas (i.e., GP13 [24], Ochiai [25], Op2 [26], and Tarantula [27]) as SBFL techniques and MBFL formulas. Of these formulas, GP13 and Op2 are proven to be maximal in theory [42]. Besides, Perez et al. [43] empirically showed that Op2 is optimal to localize a single fault. Note that we additionally add Ochiai and Tarantula, since they have also been widely studied in previous studies on fault localization [25], [27].

B. Subject Programs

We conduct the experiments on Codeflaws benchmark [41]. Codeflaws is a large-scale benchmark of real faults on C programs. These faults are diverse and relatively hard to expose [44], [45]. Codeflaws consists of 3,902 real fault programs in 7,436 programs selected from the Codeforces online database[2]. Note that each fault in this benchmark has a unique rejected 'faulty' submission and the accepted 'corrected' submission. This means each program is different from the others. We excluded the programs where the failures cannot be detected and the programs that suffered from running-time errors. Finally, we considered 317 different programs. Of these programs, 160 programs are single-fault programs and 157 programs are multiple-fault programs.

[2]https://codeforces.com/

C. Configuration

In our study, we use the GNU gcov tool [46] to collect coverage information. Then we develop a tool to generate FOMs and HOMs, which are publicly available in Github repositories[3]. In our tool, we employ mutation operators suggested by Agrawal et al. [47]. Table I lists ten typical mutation operators. We choose to generate HOMs with 2-order (2-HOMs for short) since the study of Nguyen et al. [48] and Wong et al. [49] indicated that the lower order of mutants has more effective in mutation testing. Moreover, 2-HOMs have been studied in previous studies [50], [51].

Due to the huge space of HOMs, we have tested the number of HOMs from one to five times that of FOMs in our pre-experiments. The results showed that HOMs has similar fault localization effectiveness for different number of HOMs. Considering the huge computation cost of MBFL, we generated the same number of HOMs as FOMs. In summary, we generate 42,861 FOMs and 42,975 HOMs for all of the programs.

All experiments were performed on four machines (Intel i5 Xeon e7-480, 2GHz CPU, and 32GB of memory) with Ubuntu Linux 4 64 bits.

D. Evaluation Metrics

We adopt four performance metrics (i.e., $EXAM$, $Top\text{-}N$, MAP and MTP) to evaluate the fault localization effectiveness of our proposed approach, since these metrics have been widely used in previous fault localization studies [52], [53].

1) EXAM: This metric measures the percentage of program elements (i.e., statements in our study) that need to be inspected by developers until finding the exact faulty element. $EXAM$ is a commonly used metric for fault localization techniques, and a lower $EXAM$ value indicates a better fault localization technique [52].

The $EXAM$ metric is formulated as:

$$EXAM = \frac{rank}{\text{Number of executable statements}} \tag{8}$$

The numerator in Equation (8) is the ranking of the faulty statement. The denominator is the total number of statements that need to be checked. Specifically, $rank$ is defined as:

$$rank = \frac{(i+1)+(i+j)}{2} \tag{9}$$

In Equation (9), i is the number of non-faulty statements whose suspiciousness value is higher than the faulty statement, and j is the number of statements that share the same suspiciousness value with the faulty statement. To break the tie, we take the average of the first $(i+1)$ and last $(i+j)$ ranks to determine the rank of the faulty statement.

[3]https://github.com/759031482/DNMBFL

2) Top-N: This metric counts the number of faults localized within the top N program elements among all candidates [54]. In the survey of Kochhar et al. [55], 73.58 % of developers only inspect *Top-5* program elements and almost all developers agree that *Top-10* elements are the upper bound for inspection within their acceptability level. Therefore, following the previous study [54], [56], [57], we use 1, 3, 5 for the value of N. Note that if two statements share the same suspiciousness score, we break the tie by computing the mean value of their ranks (as Equation (9)). A fault localization technique with higher *Top-N* is better than others.

3) Mean Average Precision (MAP): This metric evaluates ranking statements in information retrieval [58]. It is the mean of the average precision of all faults. AP (Average precision) is defined as follows:

$$AP = \sum_{i=1}^{M} \frac{P(i) \times pos(i)}{\text{Number of faulty statements}} \tag{10}$$

In Equation (10), i is a rank of the method, M is total number of statements in the ranked list, and $pos(i)$ is a Boolean function, while $pos(i) = 1$ indicates the i^{th} statement is faulty, otherwise $pos(i) = 0$. $P(i)$ is the precision of localization at each rank i, which is defined as follows.

$$P(i) = \frac{\text{Number of faulty statements in top } i \text{ ranks}}{i} \tag{11}$$

MAP is the mean of AP (Average Precision) values computed for a set of faults. We calculate MAP for faults belonging to the same project. A higher MAP value demonstrates a better technique.

4) Mutant-Test-Pair (MTP): This metric measures the mutant execution cost of MBFL and has been used in previous studies [14], [59], [60]. The idea of MTP is that the number of mutation execution is linked to the computational cost required to obtain the rank of statements [61]. Compared with the actual run-time cost, the MTP metric has the advantage of avoiding the influence of the run-time environment.

A MTP counts the number of mutant executions on the test cases and the technique with lower MTP indicates it has better efficiency.

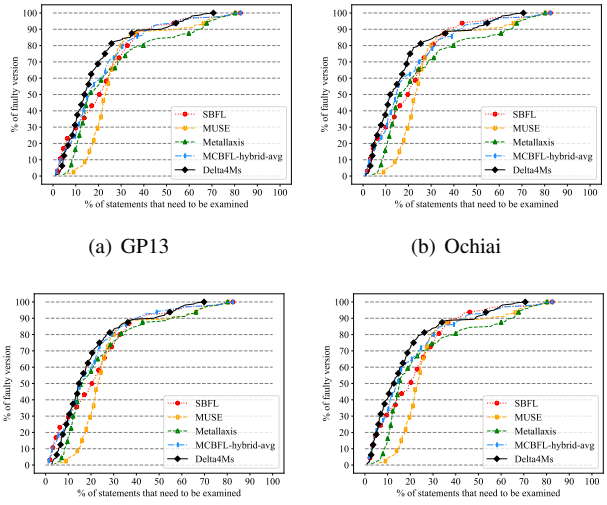
VI. Experimental Analysis

A. Answer for RQ1

RQ1: How do Delta4Ms perform in the single-fault programs in terms of fault localization effectiveness?

To answer RQ1, we collect the $EXAM$, *Top-N*, and MAP of single-fault programs for different techniques. We compare Delta4Ms with four SBFL techniques and three MBFL techniques, In addition, we display the results with four MBFL formulas.

In terms of the $EXAM$ metric, Delta4Ms localizes more faults with fewer inspected program statements than SBFL and MBFL. Figure 3 shows the effectiveness of Delta4Ms and other techniques. The X-$axis$ represents the cumulative percentage of code to be examined and the Y-$axis$ represents the total number of faults for which fault can be detected by examining this percentage of code. From Figure 3(a), Delta4Ms can discover 85.4% of the faults by examining only 30% of statements with the GP13 formula, while Metallaxis and MCBFL-hybrid-avg can discover only no more than 72.1% faults from the same percentage of code. By examining 30% of statements with the Ochiai formula, Delta4Ms technology can discover 85.3% of the faults, while Metallaxis and MCBFL-hybrid-avg can only discover 71.9% of the faults. Also, Delta4Ms localize more faults than SBFL technique and MUSE with 30% examined statements. Moreover, in most cases, we can find from the results on Op2, and Tarantula, Delta4Ms are more effective with less percentage of the examined statements (see Figure 3(c) and Figure 3(d)).



(a) GP13 (b) Ochiai

(c) Op2 (d) Tarantula

Fig. 3. Number of statements need to be examined to locate the faults on four formulas in single-fault programs

In terms of the metrics of *Top-N* and MAP, Delta4Ms again outperform SBFL and three MBFL techniques. Table IV shows Delta4Ms locate a fault statement at the *Top-1*, *Top-3*, *Top-5* ranks for 26, 65, 89 of the target faults on GP13 formula, separately. Also, Delta4Ms has the highest MAP of 0.350 at the GP13 formula. Moreover, Delta4Ms places more faults at the top ranks in Ochiai, Op2, and Tarantula formulas than SBFL and MBFL techniques.

We further analyze the fault localization accuracy improvement by Delta4Ms on single-fault programs. As shown in Table V, Delta4Ms improves the accuracy of SBFL technique by 122.9% at the metric of MAP with GP13 formula. Moreover, by comparing Delta4Ms with all techniques under a specific formula, the average improvement for MAP ranges from 91.6% to 178.2%. Later, by comparing all formulas across all Delta4Ms with a specific technique, the average improvement ranges from 70.5% to 225.4%. On average, Delta4Ms can improve the fault localization accuracy by 134.9%.

To further determine the statistical significance between Delta4Ms and SBFL, MBFL techniques, and mutant gen-

TABLE IV
Top-N AND *MAP* OF DELTA4MS AND OTHER TECHNIQUES ON FOUR FORMULAS IN SINGLE-FAULT PROGRAMS

Formula	Technique	Top 1	Top 3	Top 5	MAP
GP13	SBFL	6	18	26	0.157
	MUSE	0	1	12	0.119
	Metallaxis	0	15	59	0.166
	MCBFL-hybrid-avg	9	21	44	0.189
	Delta4Ms	**26**	**65**	**89**	**0.350**
Ochiai	SBFL	5	19	26	0.155
	MUSE	0	1	12	0.119
	Metallaxis	0	15	59	0.166
	MCBFL-hybrid-avg	11	26	53	0.217
	Delta4Ms	**45**	**78**	**94**	**0.436**
Op2	SBFL	6	18	26	0.157
	MUSE	0	1	12	0.119
	Metallaxis	0	15	59	0.169
	MCBFL-hybrid-avg	10	22	45	0.197
	Delta4Ms	**15**	**57**	**82**	**0.297**
Tarantula	SBFL	13	42	60	0.271
	MUSE	0	1	12	0.119
	Metallaxis	1	18	61	0.176
	MCBFL-hybrid-avg	22	50	82	0.317
	Delta4Ms	**51**	**81**	**94**	**0.460**

TABLE V
ACCURACY IMPROVEMENT(IN SINGLE FAULT) FOR EACH TECHNIQUES ON EACH FORMULAS

	GP13	Ochiai	OP2	Tarantula	**Avg.**
SBFL	122.9%	181.3%	89.2%	69.6%	115.7%
MUSE	195.3%	267.8%	150.6%	288.1%	225.4%
Metalalaxis	111.4%	162.7%	75.7%	161.7%	127.9%
MCBFL-hybrid-avg	85.2%	100.9%	50.8%	45.2%	70.5%
Avg.	128.7%	178.2%	91.6%	141.1%	134.9%

eration techniques, we collect the $EXAM$ of all program versions for different techniques and then employ the one-tailed *wilcoxon signed-rank test* [62] at a confidence level of 95%. The *p-value* less than 0.05 indicates Delta4Ms is significantly better than other techniques. Table VI shows the testing results on $EXAM$ of Delta4Ms with SBFL and MBFL techniques. We can find that the *p-value* of Delta4Ms are all less than 0.05 compared with SBFL, Metallaxis, MCBFL-hybrid-avg, and four mutant generation techniques on GP13, Ochiai, Op2, and Tarantula formulas.

In summary, Delta4Ms can localize more faults than SBFL, MBFL, and mutant generation techniques in terms of $EXAM$, $Top\text{-}N$ and MAP metrics. The statistical testing can also shown that Delta4Ms significantly improve the other techniques.

TABLE VI
THE *p-value* OF DELTA4MS AND OTHER TECHNIQUES ON FOUR FORMULAS IN SINGLE-FAULT PROGRAMS

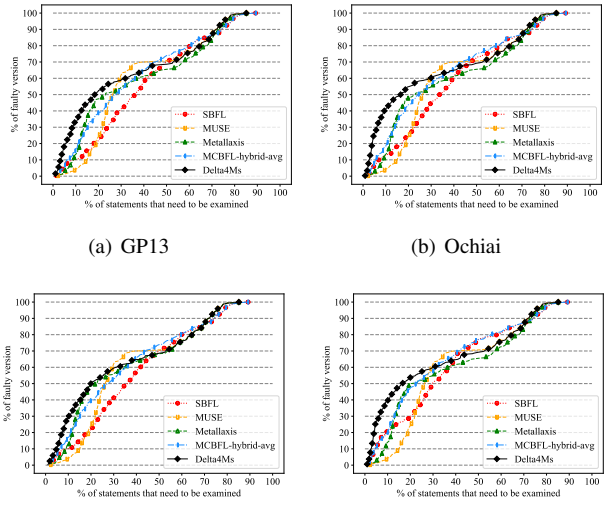
Technique	GP13	Ochiai	Op2	Tarantula
SBFL	2.0E-11	1.4E-14	1.3E-10	5.1E-09
MUSE	3.6E-08	3.0E-09	9.4E-07	2.0E-09
Metallaxis	6.7E-09	1.2E-11	3.8E-07	1.1E-18
MCBFL-hybrid-avg	3.5E-07	5.1E-08	2.8E-05	2.7E-06

B. Answer for RQ2

RQ2: How does Delta4Ms perform in the multiple-fault programs in terms of fault localization effectiveness?

To answer RQ2, we also use the $EXAM$, $Top\text{-}N$, and MAP to evaluate the fault localization effectiveness of Delta4Ms, SBFL, and MBFL techniques on multiple-fault programs.

In terms of $EXAM$, Delta4Ms localizes more faults within fewer program statements inspected. Figure 4 shows the effectiveness of Delta4Ms and other techniques on multiple-fault programs. In Figure 4(a), in GP13 formula, Delta4Ms can discover 52.5% of the faults by examining 20% of statements, while Metallaxis and MCBFL-hybrid-avg can discover 47.4% and 38.9% of all faults. Also, in Figure 4(b), Delta4Ms can discover 54.9% of the faults by examining 20% of statements, and the other techniques can discover faults no more than 47.9% at the same percentage of code. Similarly, Delta4Ms are more effective in the formula of Op2 and Tarantula (see Figure 4(c) and Figure 4(d)).



(a) GP13 (b) Ochiai

(c) Op2 (d) Tarantula

Fig. 4. Number of statements need to be examined to locate the faults on four formulas in multiple-fault programs

In terms of the metrics $Top\text{-}N$ and MAP, Delta4Ms again localizes faults more precisely than SBFL and three MBFL techniques. Table VI-B shows the fault localization results of these techniques. In GP13 formula, Delta4Ms locates 64, 117, and 149 faults at $Top\text{-}1$, $Top\text{-}3$, $Top\text{-}5$ rank, which is significantly better than other techniques. Moreover, Delta4Ms localizes faulty statements more precisely with a higher MAP of 0.697. Additionally, in the rest formulas (i.e., Ochiai, Op2, Tarantula), Delta4Ms also perform better than SBFL, MBFL techniques.

Table VIII also shows the accuracy improvement for each technique on each formula compared with Delta4Ms at MAP on multiple-fault programs. The last column shows that, by comparing Delta4Ms with all formulas under a specific technique, the average improvement ranges from 57.0% to 161.3%. At the last row, by comparing Delta4Ms with all techniques

with a specific formula, the average improvement ranges from 65.2% to 129.2%. Delta4Ms can improve fault localization accuracy by 100.9% on average on multiple-fault programs.

Table IX shows the result of one-tailed $wilcoxon\ signed\text{-}rank\ test$ results on the $EXAM$ between Delta4Ms and SBFL, MBFL techniques in multiple-fault programs. We can find that, in most cases, the $p\text{-}values$ are less than 0.05 and the $EXAM$ of Delta4Ms is significant better than the SBFL and MBFL techniques on four formulas.

In summary, Delta4Ms can performs better in terms of $EXAM$, $Top\text{-}N$, and MAP on multiple-fault programs. Statistical testing indicates that there is a significant difference between Delta4Ms and SBFL, MBFL techniques.

TABLE VII
$Top\text{-}N$ AND MAP OF DELTA4MS AND OTHER TECHNIQUES ON FOUR FORMULAS IN MULTIPLE-FAULT PROGRAMS

Formula	Technique	Top 1	Top 3	Top 5	MAP
GP13	SBFL	13	40	72	0.336
	MUSE	1	23	57	0.281
	Metallaxis	5	58	108	0.381
	MCBFL-hybrid-avg	15	67	106	0.436
	Delta4Ms	**64**	**117**	**149**	**0.697**
Ochiai	SBFL	16	48	81	0.367
	MUSE	1	23	57	0.281
	Metallaxis	5	59	109	0.382
	MCBFL-hybrid-avg	17	74	120	0.468
	Delta4Ms	**92**	**136**	**161**	**0.831**
Op2	SBFL	13	40	72	0.337
	MUSE	1	23	57	0.281
	Metallaxis	5	57	111	0.388
	MCBFL-hybrid-avg	16	69	109	0.445
	Delta4Ms	**40**	**104**	**135**	**0.582**
Tarantula	SBFL	31	72	99	0.478
	MUSE	1	23	57	0.281
	Metallaxis	5	60	114	0.390
	MCBFL-hybrid-avg	24	83	132	0.518
	Delta4Ms	**92**	**135**	**158**	**0.827**

TABLE VIII
ACCURACY IMPROVEMENT(IN MULTIPLE FAULT) FOR EACH TECHNIQUES ON EACH FORMULAS

Technique	GP13	Ochiai	OP2	Tarantula	Avg.
SBFL	107.1%	126.2%	72.9%	73.0%	94.8%
MUSE	148.0%	195.7%	107.1%	194.3%	161.3%
Metalalaxis	83.0%	117.3%	50.0%	112.3%	90.6%
MCBFL-hybrid-avg	60.0%	77.6%	30.7%	59.6%	57.0%
Avg.	99.5%	129.2%	65.2%	109.8%	100.9%

TABLE IX
THE $p\text{-}value$ OF DELTA4MS AND OTHER TECHNIQUES ON FOUR FORMULAS IN MULTIPLE-FAULT PROGRAMS

Technique	GP13	Ochiai	Op2	Tarantula
SBFL	5.6E-18	8.6E-18	1.5E-14	1.5E-10
MUSE	2.7E-03	5.5E-04	7.4E-03	8.4E-04
Metallaxis	1.1E-12	8.3E-30	5.0E-11	2.3E-25
MCBFL-hybrid-avg	3.6E-13	7.4E-19	8.2E-06	1.5E-11

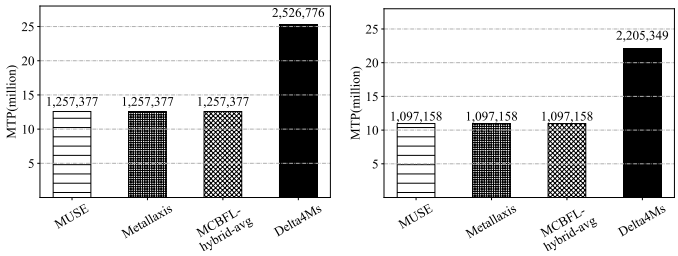
C. Answer for RQ3

RQ3: How much execution cost does Delta4Ms incur, compare to MBFL techniques?

To answer RQ3, we use the MTP metric to measure the mutation execution cost of each MBFL and mutant generation technique. Figure 5 shows the cost of Delta4Ms and MBFL techniques in terms of MTP values in single-fault programs and multiple-fault programs. The $X\text{-}axis$ shows different techniques and the $Y\text{-}axis$ is the sum of all versions' MTP measured by millions. From Figure 5(a), Delta4Ms costs the most since it generates both FOMs and HOMs for calculating the mutant bias of the program, which has twice the cost of the MBFL techniques. Also, the three MBFL techniques (i.e., MUSE, Metallaxis, and MCBFL-hybrid-avg) only generate FOMs.

Table X further presents the detailed MTP of each techniques. From Table X, We can see that Delta4Ms is 101% less efficient than MBFL techniques (MUSE, Metallaxis, and MCBFL-hybrid-avg) both in single-fault and multiple-fault programs, but Delta4Ms improve the fault localization effectiveness of these techniques. Therefore, Delta4Ms is a trade-off technique that obtains better fault localization effectiveness by losing some efficiency of running this technique. Therefore, there still exists room for boosting the efficiency of Delta4Ms.

In summary, Delta4Ms costs more computational cost than MBFL but with a significant improvement in these techniques. Also, there still exists room for reducing the computational cost of Delta4Ms.



(a) Single-fault programs (b) Multiple-fault programs

Fig. 5. MTP of Delta4Ms and other MBFL techniques on four formulas

TABLE X
THE EFFICIENCY AND EFFECTIVENESS OF DELTA4MS AND OTHER MBFL TECHNIQUES

Programs	Technique	MTP	Efficiency%
Single Fault	MUSE	1,257,377	-101%
	Metallaxis	1,257,377	-101%
	MCBFL-hybrid-avg	1,257,377	-101%
	Delta4Ms	2,526,776	-
Multi Faults	MUSE	1,097,158	-101%
	Metallaxis	1,097,158	-101%
	MCBFL-hybrid-avg	1,097,158	-101%
	Delta4Ms	2,205,349	-

VII. Conclusion

In this paper, we investigate the impacts of mutant characteristics of a single statement on the accuracy of MBFL techniques, and we formally formulate the problem of mutant bias. We present a theoretical model Delta4Ms to capture the impacts of the MBFL technique and remove them. We evaluate Delta4Ms by conducting an empirical experiment on 320 versions of programs from Codeflaws. The experimental results demonstrate that Delta4Ms can significantly improve the accuracy of MBFL techniques by 134.9% on single-fault programs and 100.9% on multiple-fault programs in terms of the metric MAP. In addition, Delta4Ms perform better in terms of the metrics $EXAM$, $mathitTop\text{-}N$ than SBFL and MBFL techniques (i.e., MUSE, Metallaxis, and MCBFL-hybrid-avg).

In the future, we want to evaluate our method on other realistic subject programs and we want to investigate more mutant reduction techniques to further improve our model.

References

[1] D. Shin and D.-H. Bae, "A theoretical framework for understanding mutation-based testing methods," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 299–308.

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

[3] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 114–125.

[4] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, "Spectrum-based fault localization in software product lines," *Information and Software Technology*, vol. 100, pp. 18–31, 2018.

[5] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[6] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 43–52.

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

[8] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an rbf neural network," *IEEE Transactions on Reliability*, vol. 61, no. 1, pp. 149–169, 2011.

[9] R. Gao, W. E. Wong, Z. Chen, and Y. Wang, "Effective software fault localization using predicted execution results," *Software Quality Journal*, vol. 25, no. 1, pp. 131–169, 2017.

[10] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 445–455.

[11] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 153–162.

[12] M. Papadakis and Y. Le Traon, "Using mutants to locate" unknown" faults," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 691–700.

[13] ——, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[14] Z. Li, H. Wang, and Y. Liu, "Hmer: A hybrid mutation execution reduction approach for mutation-based fault localization," *Journal of Systems and Software*, p. 110661, 2020.

[15] H. Wang, B. Du, J. He, Y. Liu, and X. Chen, "Ietcr: An information entropy based test case reduction strategy for mutation-based fault localization," *IEEE Access*, vol. 8, pp. 124 297–124 310, 2020.

[16] L. Zhang, Z. Li, Y. Feng, Z. Zhang, W. K. Chan, J. Zhang, and Y. Zhou, "Improving fault-localization accuracy by referencing debugging history to alleviate structure bias in code suspiciousness," *IEEE Transactions on Reliability*, vol. 69, no. 3, pp. 1021–1049, 2020.

[17] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Software Engineering—Esec/Fse'97*. Springer, 1997, pp. 432–449.

[18] M. Kooli, F. Kaddachi, G. Di Natale, A. Bosio, P. Benoit, and L. Torres, "Computing reliability: On the differences between software testing and software fault injection techniques," *Microprocessors and Microsystems*, vol. 50, pp. 102–112, 2017.

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[20] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 609–620.

[21] Q. V. Nguyen and L. Madeyski, "On the relationship between the order of mutation testing and the properties of generated higher order mutants,"

in *Asian Conference on Intelligent Information and Database Systems*. Springer, 2016, pp. 245–254.

[22] J. A. P. Lima, G. Guizzo, S. R. Vergilio, A. P. Silva, H. L. J. Filho, and H. V. Ehrenfried, "Evaluating different strategies for reduction of mutation testing costs," in *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*, 2016, pp. 1–10.

[23] D. Shin, S. Yoo, and D.-H. Bae, "A theoretical and empirical study of diversity-aware mutation adequacy criterion," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 914–931, 2017.

[24] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 244–258.

[25] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[26] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[27] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 467–477.

[28] M. Jimenez, T. T. Checkam, M. Cordy, M. Papadakis, M. Kintis, Y. L. Traon, and M. Harman, "Are mutants really natural? a study on how" naturalness" helps mutant selection," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[29] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[30] M. Papadakis and Y. Le Traon, "Effective fault localization via mutation analysis: a selective mutation approach," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1293–1300.

[31] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[32] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992.

[33] M. Harman, Y. Jia, and W. B. Langdon, "A manifesto for higher order mutation testing," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2010, pp. 80–89.

[34] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *2010 Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 300–309.

[35] Q. V. Nguyen and L. Madeyski, "Addressing mutation testing problems by applying multi-objective optimization algorithms and higher order mutation," *Journal of Intelligent & Fuzzy Systems*, vol. 32, no. 2, pp. 1173–1182, 2017.

[36] ——, "Higher order mutation testing to drive development of new test cases: An empirical comparison of three strategies," in *Asian Conference on Intelligent Information and Database Systems*. Springer, 2016, pp. 235–244.

[37] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 212–222.

[38] R. Gopinath, C. Jensen, and A. Groce, "The theory of composite faults," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 47–57.

[39] H. Wang, Z. Li, Y. Liu, X. Chen, D. Paul, Y. Cai, and L. Fan, "Can higher-order mutants improve the performance of mutation-based fault localization?" *IEEE Transactions on Reliability*, 2022.

[40] Z. Li, B. Shi, H. Wang, Y. Liu, and X. Chen, "Hmbfl: Higher-order mutation-based fault localization," in *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2021, pp. 66–77.

[41] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "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, 2017, pp. 180–182.

[42] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 1, pp. 1–30, 2017.

[43] A. Perez, A. Riboira, and R. Abreu, "A topology-based model for estimating the diagnostic efficiency of statistics-based approaches," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*. IEEE, 2012, pp. 171–176.

[44] M. Papadakis, T. T. Chekam, and Y. Le Traon, "Mutant quality indicators," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 32–39.

[45] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, no. 1, pp. 434–487, 2020.

[46] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[47] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the c programming language," Citeseer, Tech. Rep., 1989.

[48] Q.-V. Nguyen *et al.*, "Is higher order mutant harder to kill than first order mutant? an experimental study," in *Asian Conference on Intelligent Information and Database Systems*. Springer, 2018, pp. 664–673.

[49] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[50] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[51] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2010, pp. 90–99.

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

[53] Y. Liu, M. Li, Y. Wu, and Z. Li, "A weighted fuzzy classification approach to identify and manipulate coincidental correct test cases for fault localization," *Journal of Systems and Software*, vol. 151, pp. 20–37, 2019.

[54] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 177–188.

[55] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

[56] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

[57] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[58] A. Moffat and J. Zobel, "Rank-biased precision for measurement of retrieval effectiveness," *ACM Transactions on Information Systems (TOIS)*, vol. 27, no. 1, p. 2, 2008.

[59] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 235–245.

[60] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 315–326.

[61] Y. Liu, Z. Li, L. Wang, Z. Hu, and R. Zhao, "Statement-oriented mutant reduction strategy for mutation based fault localization," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 126–137.

[62] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, pp. 80–83, 1945.