The Journal of Systems & Software 168 (2020) 110661

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

HMER: A Hybrid Mutation Execution Reduction approach for Mutation-based Fault Localization

Zheng Li, Haifeng Wang, Yong Liu *

College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China

A R T I C L E I N F O

Article history:
Received 15 September 2019
Received in revised form 14 May 2020
Accepted 22 May 2020
Available online 25 May 2020

Keywords:
Spectrum-based Fault Localization
Mutation-based Fault Localization
Mutant sampling
Cost reduction

A B S T R A C T

Identifying the location of faults in programs has been recognized as one of the most manually and time cost activities during software debugging process. Fault localization techniques, which seek to identify faulty program statements as quickly as possible, can assist developers in alleviating the time and manual cost of software debugging. Mutation-based fault localization(MBFL) has a promising fault localization accuracy, but suffered from huge mutation execution cost. To reduce the cost of MBFL, we propose a Hybrid Mutation Execution Reduction(HMER) approach in this paper. HMER consists of two steps: Weighted Statement-Oriented Mutant Sampling(WSOME) and Dynamic Mutation Execution Strategy(DMES). In the first step, we employ Spectrum-Based Fault Localization(SBFL) techniques to calculate the suspiciousness value of statements, and guarantee that the mutants generated from statements with higher suspiciousness value will have more chance to be remained in the sampling process. Next, a dynamic mutation execution strategy is used to execute the reduced mutant set on test suite to avoid worthless execution. Empirical results on 130 versions from 9 subject programs show that HMER can reduce 74.5%-93.4% mutation execution cost while keeping almost the same fault localization accuracy with the original MBFL. A further *Wilcoxon signed — rank test* indicates that when employing HMER strategy in MBFL, the fault localization accuracy has no statistically significant difference in most cases compared with the original MBFL without any reduction techniques.

1. Introduction

Software debugging, which includes fault detecting, fault localization, and bug fixing (Howden William, 1978), is one of the essential parts of software development and maintenance. After faults are detected, developers should identify the locations of faults, where the process of defective statement location identification is usually referred to as fault localization. During software debugging, fault localization is one of the most complicated and time cost activity (Donghwan and Doo-Hwan, 2016). To reduce the time and manually cost, and guide developers to locate faults with minimal human intervention, researchers have developed a lot of automated fault localization techniques in the past decades (Howden William, 1978; Eric et al., 2016; Wes, 2010).

Among the automated fault localization techniques, one of the most commonly studied methods is called Spectrum-Based Fault Localization(SBFL). SBFL executes test cases to obtain the coverage information and execution results of test cases. Such information will be used for calculating the probability that program entities incur a fault, and a ranking list of them will be generated finally. Developers can check program entities in sequence according to the ranking list, in the hope that the exact faulty entity will be encountered near the top of the ranking list (Fabian et al., 2017; Bob et al., 2017; Nicholas and Jones, 2017). SBFL is simple to implement, and empirical results showed that it is useful and helpful to reduce the cost of the software debugging process. However, SBFL has shortcomings for its impractical localization accuracy since all the statements in one same block will have the same coverage information and then share the same ranking, which will often increases the number of program entities needed to be checked before encountering the faults (Seokhyeon et al., 2014). In this paper, we will employ SBFL techniques to calculate the weights of different program entities for further analysis.

Studies have shown that fault localization techniques that employ mutation analysis have better fault localization accuracy compared with the traditional SBFL techniques. Such techniques are defined as Mutation-Based Fault Localization(MBFL) (Maha et al., 2017; Mike and Yves, 2012, 2015). There are two main kinds of MBFL techniques. One is called Metallaxis-FL (Mike and Yves, 2015), which is proposed by M. Papadakis et al. Similar to SBFL, Metallaxis-FL calculates the suspiciousness value of each statement in the program under test and ranks them by the suspiciousness value. The critical point of Metallaxis-FL is that

* Corresponding author.
E-mail address: lyong@mail.buct.edu.cn (Y. Liu).

https://doi.org/10.1016/j.jss.2020.110661
0164-1212/© 2020 Elsevier Inc. All rights reserved.

it uses mutation analysis to mutate statements through mutation operators and generate mutation programs, that is, mutants. Metallaxis-FL uses test cases to execute mutants and get the execution results for calculating the suspiciousness value of mutants, where the maximum value of all mutants generated from the same statement will be assigned to the suspiciousness value of the corresponding statement. Metallaxis-FL has finer granularity than SBFL since quite a few mutants can be generated by injecting artificial faults into one single statement. The other is called MUSE (Seokhyeon et al., 2014; Shin et al., 2017), which is proposed by S. Moon et al. MUSE applies the idea of mutation analysis and calculates the suspiciousness value of statements by measuring the proportion of test cases which turned from passed to failed or from failed to passed due to mutating on the corresponding statements. S. Pearson et al. compared the two MBFL techniques and found that Metallaxis-FL has better fault localization accuracy than MUSE (Pearson et al., 2017). So, in this paper, we focus our study on Metallaxis-FL, which is indicated by MBFL for short in the rest of this paper.

MBFL technique has the advantage of higher fault localization accuracy but is suffered from an extremely mutants execution cost problem (Maha et al., 2017; Titcheu et al., 2016). This problem is due to that a large number of mutants can be generated from the program under test, and each mutant has to be executed on all test cases. And MBFL technique needs to employ such execution information to calculate the suspiciousness value of statements, to locate the faulty statements in the program under test finally. To reduce the execution cost, then improve the execution efficiency of the MBFL technique, many researchers have proposed several approaches (Wes et al., 2009; Bin and Hadi, 2017; Jingxuan et al., 2016; Xiao-Yi et al., 2018; Oliveira et al., 2015; Nan and Jeff, 2017; Barr Earl et al., 2015; Weyuker Elaine, 1982).

From different aspects, all different mutant execution reduction approaches can be divided into three groups. The first kind of approach aims to reduce the number of generated mutants. M. Papadakis et al. proposed SELECTIVE method (Mike and Yves, 2014), which tries to select a part of an efficient subset of mutation operators to replace the original set, then the number of generated mutants can be reduced. Another method proposed by M. Papadakis et al. is called SAMPLING (Mike and Yves, 2015), which randomly selects a certain proportion of mutants from the original mutant set. In our previous work, we proposed a statement-oriented mutant sampling strategy, SOME (Yong et al., 2017), to reduce the number of generated mutants in the statement level, which leads to a better distribution of mutants from a better sampling granularity. The second group is considered from the test suite aspect, Andre Assis et al. proposed an FTMES method (Assis Lobo de Oliveira et al., 2018), which only employs the failed test cases to localize faults but ignores all the passed test cases. The execution cost of MBFL can be reduced along with the reduction of the number of test cases. The third kind of methods is optimizing MBFL technique during the mutant execution process, we proposed a dynamic mutation execution strategy, DMES (Yong et al., 2018), to dynamically optimize the execution process of mutants and test cases. By employing the DMES strategy, the execution cost of MBFL can be reduced, and the fault localization accuracy has not been affected.

All the above methods can be used to reduce the execution cost of MBFL, but these methods still have a lot of potentials to improve the reduction rate further or alleviate the fault localization accuracy loss when employing such methods. In this paper, we present a hybrid mutation execution reduction strategy, HMER, which aims to reduce the execution cost of MBFL from two aspects, which are called as “do fewer” and “do smarter”. “Do fewer“ refers to the strategy of reducing the number of used mutants before executing them, and “do smarter” is optimizing the process of mutants execution during MBFL. As fewer mutants are used, combined with the optimized mutant execution strategy, the whole process of MBFL will become much faster than the original. At the same time, the fault localization accuracy loss has been fine controlled and can be accepted when considering the huge amount of reduced mutant execution time cost.

The main contributions of this paper are summarized as follows:

- From the mutant set aspect, this paper firstly propose a WSOME approach, which has considered three important factors during the mutant sampling process: statements, mutation operators, and the probability of statements incur faults. Guided by the three factors, the mutant subset after sampling has a better distribution, which guarantees that every statement and mutation operator must have at least one corresponding mutant, and the mutants generated from statements with higher suspiciousness value will have more chance to remain.
- Furthermore, a hybrid mutation reduction approach, HMER, is proposed. HMER combines WSOME and our previously proposed DMES method. The mutation reduction rate of MBFL with HMER can be enhanced compared with MBFL with only one single method, WSOME or DMES.
- This paper reports an empirical study on 130 faulty versions from 9 real-world programs, and the results indicated that the mutant execution cost of MBFL when employing our proposed HMER strategy could be remarkably reduced compared with the original MBFL. At the same time, the fault localization accuracy is kept almost the same as the original MBFL.

The present paper extends our prior conference publication (Yong et al., 2017) in four aspects: (1) Based on the SOME method proposed in Yong et al. (2017), an improved method, WSOME, is presented in this paper, which further considers the fault distribution of mutants from different statements with different probability to incur faults. (2) On the basis of WSOME, the DMES strategy is employed to reduce mutation execution cost further. (3) Three suspiciousness formulas, *Dstar**, *Jaccard*, and *Tarantula* are added into the empirical study. (4) Two larger-scale subject programs with real faults, *sed* and *grep*, are introduced in the experimental design.

The remainder of this paper is organized as follows. Section 2 provides a background of fault localization techniques, and the case to apply them. Section 3 describes the mutation reduction strategies. Section 4 presents our proposed strategy WSOME and its algorithm. Section 5 describes the experimental design we used to investigate our research question and presents the results of the experiment we carried out and statistical analysis. Section 6 discusses threats to the validity of our experiment. Section 7 finally concludes the paper and reports some possible future directions.

2. Fault localization techniques

During the software testing process, when erroneous software behavior is exposed by test cases, software developers still have to manually identify the root cause behind a failure, which requires significant time cost and human effort (Heiden et al., 2019). This motivates a significant effort within the research community to investigate automated fault localization techniques (Eric et al., 2016). Thus, a lot of algorithms were proposed, all of which are intended to determine the exact location of faults. Such as Sliced-Based methods (Zhang et al., 2005), Program Spectrum-based Methods (Jones and Harrold, 2005; Taha

$$\begin{array}{cc} & T:\begin{pmatrix} t_1 & t_2 & \cdots & t_m \end{pmatrix} \\ P:\begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} & M:\begin{pmatrix} 1/0 & 1/0 & \cdots & 1/0 \\ 1/0 & 1/0 & \cdots & 1/0 \\ \vdots & \vdots & \ddots & \vdots \\ 1/0 & 1/0 & \cdots & 1/0 \end{pmatrix} \\ & R:\begin{pmatrix} p/f & p/f & \cdots & p/f \end{pmatrix} \end{array}$$

Fig. 1. Basic information of SBFL.

et al., 1989), Program State-based Methods (Wang and Roychoudhury, 2005), statistics-based Methods (Chao et al., 2006), Machine Learning-based Methods (Brun and Ernst, 2004; Cellier et al., 2008; Wong et al., 2008), Model-based Methods (Mateis et al., 2000), Mutation-based Methods (Maha et al., 2017; Mike and Yves, 2012, 2015), and so on. In this paper, our study focused on two kinds of fault localization techniques, which are introduced below.

2.1. Spectrum-based fault localization

Spectrum-Based Fault Localization(SBFL) is a statistically dynamic technique based on two types of information collection of entities, called test results and program spectra. The test results are the execution results of all test cases, where the result of each test case is denoted as $Pass(P)$ or $Fail(F)$, which means the corresponding test case is a passed or failed test case. Program spectrum refers to the coverage information of each entity by the test case, which is a special angle to look at the dynamic behavior of the software (Thomas et al., 1997; Jean et al., 1998). Commonly used program entities include statements, branches, paths and basic blocks, call sequences (Dallmeier et al., 2005), du-pairs (Masri, 2010), statement frequency (Shu et al., 2016), and so on. There are many kinds of combinations in previous researches (Jean et al., 2000; Thomas et al., 1997), and the most popularly used is the combination of a statement and its binary coverage status (Hiralal et al., 1995; Eric and Yu, 2006).

Because of the simplicity and effectiveness of SBFL, a lot of researchers have paid a lot of attention to it. Many approaches have been proposed to improve the fault localization accuracy of SBFL. For example, some researchers identify the coincidental correct test cases from the perspective of the test cases (Wes and Abou, 2014; Yihan and Chao, 2012; Zheng et al., 2016). In addition, according to the performance characteristics of the program spectrum, the researchers also proposed a number of suspiciousness calculation formulas to enrich SBFL techniques, such as *Jaccard* (Chen Mike et al., 2002), *Ochiai* (Rui et al., 2006), *Op2* (Lee et al., 2011), *Tanraula* (Jones James et al., 2002), *Dstar** (Eric et al., 2014), and so on.

The following contents describe the specific implement principles of SBFL techniques:

Given a program $P = \{s_1, s_2, \ldots, s_n\}$ with n statements and executed by a test suite of m test cases $T = \{t_1, t_2, \ldots, t_m\}$. The program spectrum information and execution results of test cases can be gathered when running all test cases on the program under test. Fig. 1 is an illustrated example of showing how SBFL works. In Fig. 1, p and f indicate that the corresponding test cases are passed or failed test cases respectively. The element in the ith row and the jth column represents the coverage information of statement s_i, executed by test case t_j, where 1 and 0 mean that the statement s_i is executed or not executed by the corresponding test case t_j respectively.

Table 1
Suspiciousness formulas for SBFL.

Name	Formula
Jaccard	$Sus(s) = \frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Ochiai	$Sus(s) = \frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})*(a_{ef}+a_{ep})}}$
Op2	$Sus(s) = a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$
Tarantula	$Sus(s) = \frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$
Dstar*	$Sus(s) = \frac{a_{ef}^{*}}{a_{ep}+a_{nf}}$

After that, for each statement s, a vector with four parameters $P_s = \{a_{ep}, a_{ef}, a_{np}, a_{nf}\}$ can be calculated based on the above program spectrum information and execution results. Parameter a_{ep} and a_{ef} indicate the number of test cases that have executed the statement s and return the result as *pass* or *fail*, respectively. Similarly, the parameter a_{np} and a_{nf} denote the number of test cases which do not execute the statement s and return the result of *pass* or *fail*, respectively. Based on these four parameters, the suspiciousness of program statements can be calculated by employing a number of different formulas, where some popularly used formulas are shown in Table 1. A statement with higher suspiciousness value is considered to have a higher possibility of being faulty, which should be examined by developers with higher priority.

Fig. 1 shows an illustrative example of how SBFL techniques localize the fault from a program segment. The program segment under test in Fig. 1 is a function called $mid()$, which will return the middle value of three integers. Statement s_3 is a faulty statement, as it should be changed to $if(y < z)$. The test suite has six test cases, from tc_1 to tc_6, whose coverage information and execution results are shown in column 2 to 7 too. In the example, “1” means the corresponding statement is covered by the corresponding test case, while “0” means the opposite. The execution results are listed in the last row, where P and F denote the corresponding test case is a passed or failed test case, respectively. The suspiciousness value of all statements calculated by the five formulas listed in Table 1 is shown in the right part of Fig. 1.

Guided by the suspiciousness values, a ranking list of all statements can be generated by descending sorting them. Then developers can check each statement in the ranking list, from statements with higher suspiciousness value to those with lower suspiciousness value. If the exact faulty statement’s suspiciousness value is relatively bigger then other correct statements, then the position of the faulty statement in the generated ranking list will be near the top, which means developers can easily find the faulty statement and finally fix the bug. In other words, the fault localization techniques which can rank the exact faulty statement near the top of the ranking list have better fault localization accuracy. From Fig. 1, it is seen that the suspiciousness value of the faulty statement, line 3, is not the biggest among all statements. Besides, since the statements in line 1, 2, 3, and 13 share the same coverage information, then the suspiciousness values of them are the same. Such an example illustrates that SBFL techniques can alleviate the efforts of developers to find the exact fault of the program, but the fault localization accuracy still needs to be improved, which motivates researchers to promote other techniques to precisely locate bugs in the software.

Table 2
An illustrated example of SBFL.

Program (P)		Test suite						Suspiciousness value				
		t1	t2	t3	t4	t5	t6	Jaccard	Ochiai	Op2	Tarantula	Dstar[3]
	int mid(int x,int y,int z)	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,4					
1	int m;	1	1	1	1	1	1	0.33	0.58	1.20	0.50	2.00
2	m = z;	1	1	1	1	1	1	0.33	0.58	1.20	0.50	2.00
3	**if(y < z − 1)//fault**	**1**	**1**	**1**	**1**	**1**	**1**	**0.33**	**0.58**	**1.20**	**0.50**	**2.00**
4	if(x < y)	1					1	0.00	0.00	−0.40	0.00	0.00
5	m = y;							0.00	0.00	0.00	0.00	0.00
6	else if(x < z)	1					1	0.00	0.00	−0.40	0.00	0.00
7	m = x;	1					1	0.00	0.00	−0.40	0.00	0.00
8	else		1	1	1	1		0.50	0.71	1.60	0.67	4.00
9	if (x > y)		1	1	1	1		0.50	0.71	1.60	0.67	4.00
10	m = y;			1		1		0.33	0.50	0.80	0.67	0.50
11	else if(x > z)		1		1			0.33	0.50	0.80	0.67	0.50
12	m = x;							0.00	0.00	0.00	0.00	0.00
13	return m;}	1	1	1	1	1	1	0.33	0.58	1.20	0.50	2.00
	Results	**P**	**F**	**P**	**P**	**F**	**P**					

Table 3
Typical mutation operators.

Mutation operators	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

2.2. Mutation-based fault localization

MBFL technique is based on mutation analysis, which works by making syntactic changes on the program under test, for example, changing from $if(x > 1)$ to $if(x < 1)$, and then produced various program versions with artificially injected faults, which are called *mutants* (DeMillo Richard et al., 1978). The rule that generates a mutant from the original program is known as *mutation operator* (Jefferson and Untch, 2001). Table 3 lists ten typical C mutation operators proposed by Agrawal et al. (1989).

In the process of mutation analysis, a remarkable amount of mutants will be generated by using a large number of mutation operators, and the generated mutants will be executed on test cases to do further analysis. If the output of a mutant executed from a test case is different from the output of the original program executed on the same test case, then the mutant is said to be *killed* (*or distinguished*) be the corresponding test case. Otherwise, it is said to be *notkilled* (*or alive*) (Vidroha and Wong, 2014).

Mutation analysis is different from program spectrum information-based techniques because it forces candidate test cases not only to perform special programs to locate but also to find potential triggering faults at these locations based on the output. The requirement of mutations makes it particularly powerful in terms of testing and analysis. Mutants have proven to be very useful in simulating the behavior of real faults and can find more faults than other test criteria (Mike and Yves, 2012).

Mutation-Based Fault Localization(MBFL) technique is based on mutation analysis (Maha et al., 2017). It mutates all executable statements under the test and uses the suspiciousness value of mutants to measure the faulty probability of the statement. Papadakis Mike et al. firstly applied it to do fault localization, and they found that this proposed method can effectively locate "unknown" faults in the program (Mike and Yves, 2012). In their later work, they extended the proposed MBFL technique, and proposed Metallaxis-FL (Mike and Yves, 2015), and also discussed some optimization methods to reduce the execution cost. From the perspective of the performance of the mutant, Moos et al. found that mutating the statement is more likely to cause the correct statement to become an error and the wrong statement to be correct, inspired of such ideas, they presented a novel MBFL technique, which is called as MUSE (Seokhyeon et al., 2014). Their experimental results show that MUSE can be 25 times more precise than the state-of-the-art SBFL technique *Op*2 on average. To compare the fault localization accuracy of two different MBFL techniques, S. Pearson et al. conducted an empirical study and found that Metallaxis-FL has better fault localization accuracy than MUSE (Pearson et al., 2017). So, in this paper, we focus our study on Metallaxis-FL, which is indicated by MBFL for short in the rest of this paper.

Fig. 2 shows the framework of MBFL, and the detailed implementation of such technique is described as below:

1. Given a program $P = \{s_1, s_2, \ldots, s_n\}$ with n statements and a test suite T with m test cases $T = \{t_1, t_2, \ldots, t_m\}$.
2. Execute program P against test suite T, then failed test set T_f and the passed test set T_p can be obtained by comparing the execution results and expected output. The corresponding executed statement set covered by T_f and T_p can be recorded as Cov_f and Cov_p.
3. For each statement covered by failed test cases, $s_i \in Cov_f$, artificial defects are injected to generate mutants by employing mutation operators, $M(s_i) = \{m_1, m_2, \ldots, m_q\}$.
4. For each mutant, all test cases will execute on it, and the information on whether it is killed or not is recorded. T_k is defined to indicate the test set, which can kill m, and T_n is the test set that cannot kill m. T_p is the pass test set, and T_f is the failed test set. Based on this information, four parameters can be obtained:

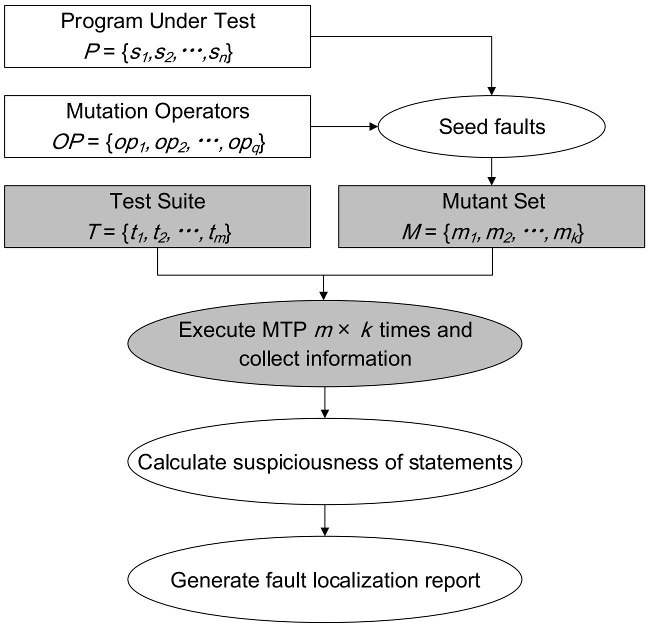

Program Under Test
$P = \{s_1, s_2, \cdots, s_n\}$
Mutation Operators
$OP = \{op_1, op_2, \cdots, op_q\}$
Seed faults
Test Suite
$T = \{t_1, t_2, \cdots, t_m\}$
Mutant Set
$M = \{m_1, m_2, \cdots, m_k\}$
Execute MTP $m \times k$ times and collect information
Calculate suspiciousness of statements
Generate fault localization report

Fig. 2. Framework of MBFL.

Table 4
Suspiciousness formulas for MBFL.

Method	Formula
Jaccard	$Sus(m) = \frac{a_{kf}}{a_{kf}+a_{nf}+a_{kp}}$
Ochiai	$Sus(m) = \frac{a_{kf}}{\sqrt{(a_{kf}+a_{nf})*(a_{kf}+a_{kp})}}$
Op2	$Sus(m) = a_{kf} - \frac{a_{kp}}{a_{kp}+a_{np}+1}$
Tarantula	$Sus(m) = \frac{\frac{a_{kf}}{a_{kf}+a_{kp}}}{\frac{a_{kf}}{a_{kf}+a_{nf}} + \frac{a_{kp}}{a_{kp}+a_{np}}}$
Dstar*	$Sus(m) = \frac{a_{kf}^{*}}{a_{kp}+a_{nf}}$

$a_{np} = |T_n \bigcap T_p|$, $a_{kp} = |T_k \bigcap T_p|$, $a_{nf} = |T_n \bigcap T_f|$, $a_{kf} = |T_k \bigcap T_f|$.
where, a_{np} is the number of passed test cases which cannot kill m, a_{kp} means the number of passed test cases which can kill m, a_{nf} indicates the number of failed test cases which cannot kill m, and at last, a_{kf} presents the number of failed test cases which can kill m.

5. Based on the four parameters, MBFL technique will further calculate the suspiciousness value of mutants by employing some formulas. These formulas are transferred from SBFL, and Table 4 shows five popularly used formulas. It is obvious that a number of mutants can be generated by mutating on the same statement, and MBFL techniques assign the suspiciousness value of each statement with the maximum suspiciousness value of all mutants, which are generated from the corresponding statement. The relationship of the suspiciousness value between statement s_i and the corresponding generated mutants can be defined as $sus(s_i) = max\left\{sus(m_1), sus(m_2), \ldots, sus(m_q)\right\}$.
6. Similar to SBFL techniques, MBFL will also generate a ranking list by descending ordering all statements with suspiciousness value. The developers can check each statement one by one according to the ranking list, and finally find and then fix the faults in the program under test.

Table 5 shows an illustrative example of how MBFL localizes a fault, and it should be noted that the program under test used in this example is the same from the previously discussed SBFL example in Fig. 1. In Table 5, all the 35 mutants generated from mutating on five statements are shown in column two with the corresponding changing rules. The test suite with 6 test cases and the execution results of all mutant-test-pairs are shown from column three to column eight, where “1” means the corresponding mutant is killed by the corresponding test case. In the last five columns, all mutants’ suspiciousness values calculated by five formulas are shown, and the exact suspiciousness value of each statement is highlighted in boldface. Take the mutant M_1 generated from the third statement s_3 as an example, t_2 and t_5 are two failed test cases that kill M_1, and other four passed test cases cannot kill it. Then the four parameters for mutant M_1 is calculated as follows, $a_{np} = 4$, $a_{kp} = 0$, $a_{nf} = 0$, and $a_{kf} = 2$. The suspiciousness value calculated by using *Jaccard* is $Sus(M_1) = \frac{a_{kf}}{a_{kf}+a_{nf}+a_{kp}} = \frac{2}{2+0+0} = 1.00$. The other mutants generated from s_3 can be assigned with suspiciousness values in the same way. The suspiciousness value of statement s_3 is equal to the maximum value of the seven suspiciousness values of mutants generated from s_3, $Sus(s_3) = Sus(M_1) = 1.00$. From the above analysis, the ranking list can be obtained after all statements’ suspiciousness values have been calculated. From Table 5, we can see that no matter which formula is used, the exact faulty statement, s_3, still has the highest suspiciousness value, which means s_3 will be at the top in the ranking list.

From the two examples in Tables 2 and 5, it can be seen that MBFL has the advantage of better-analyzing granularity since mutating from one statement can generate a number of mutants, which is the reason that MBFL has better fault localization accuracy than SBFL. But the downside of better granularity is the high cost since such a large number of mutants can be generated, and each mutant has to be executed on all test cases.

3. Cost reduction strategies for MBFL

Benefited from fined analyzing granularity, MBFL has high fault localization accuracy but also suffers from a huge execution cost problem. To improve efficiency, two mutant reduction strategies, SELECTVIE (Mike and Yves, 2014), and SAMPLING (Mike and Yves, 2015) were proposed by researchers.

3.1. SELECTIVE

The SELECTIVE (Mike and Yves, 2014) strategy aims to reduce the number of generated mutants by only using a few mutation operators to replace the whole mutation operator set. By using this strategy, a sufficient criterion of mutation operators was firstly defined to guide the selection procedure of mutation operators. In the rule, different mutation operators are assigned with varying degrees of contribution, where some mutation operators with lower contribution degrees will be abandoned from the target mutation operation subset.

Fig. 3 shows a brief framework of SELECTIVE. After sampling a specific subset of mutation operators, a new operator set will be implemented to generate fewer mutants compared with using the whole mutation operator set. But in MBFL, each mutant is considered as a similar version of the real fault in the program under test, losing some mutation operators will lead to some kind of faults cannot be simulated, and then affect the fault localization ability of MBFL.

Table 5
An illustrated example of MBFL.

	Program (P)	Mutants	Test suite						Suspiciousness				
			t_1 3,3,5	t_2 1,2,3	t_3 3,2,1	t_4 5,5,5	t_5 5,3,4	t_6 2,1,4	Jaccard	Ochiai	Op2	Tarantula	Dstar^3
	int mid(int x,int y,int z){												
1	int m;												
2	m = z;												
		M_1: $< \to <=$		1			1		**1.00**	**1.00**	**2.00**	**1.00**	$+\infty$
		M_2: $< \to >$	1		1			1	0.00	0.00	0.00	0.00	0.00
		M_3: $< \to >=$	1	1	1		1		0.40	0.63	1.40	0.57	2.67
3	**if(y < z − 1)//fault**	M_4: $< \to ==$	1	1			1	1	0.50	0.71	1.60	0.67	4.00
		M_5: $< \to !=$			1				0.00	0.00	0.00	0.00	0.00
		M_6: $< \to$ true		1	1		1		0.67	0.82	1.80	0.80	8.00
		M_7: $< \to$ false	1					1	0.00	0.00	0.00	0.00	0.00
4	if(x < y)	M_8: $< \to <=$							**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
		M_9: $< \to >$						1	0.00	0.00	0.00	0.00	0.00
		M_{10}: $< \to >=$							0.00	0.00	0.00	0.00	0.00
		M_{11}: $< \to ==$							0.00	0.00	0.00	0.00	0.00
		M_{12}: $< \to !=$						1	0.00	0.00	0.00	0.00	0.00
		M_{13}: $< \to$ true						1	0.00	0.00	0.00	0.00	0.00
		M_{14}: $< \to$ false							0.00	0.00	0.00	0.00	0.00
5	m = y;												
		M_{15}: $< \to <=$							**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
		M_{16}: $< \to >$	1					1	0.00	0.00	0.00	0.00	0.00
		M_{17}: $< \to >=$	1					1	0.00	0.00	0.00	0.00	0.00
6	else if(x < z)	M_{18}: $< \to ==$	1					1	0.00	0.00	0.00	0.00	0.00
		M_{19}: $< \to !=$							0.00	0.00	0.00	0.00	0.00
		M_{20}: $< \to$ true							0.00	0.00	0.00	0.00	0.00
		M_{21}: $< \to$ false	1					1	0.00	0.00	0.00	0.00	0.00
7	m = x;												
8	else												
		M_{22}:$> \to !=$		1					0.50	0.71	1.00	1.00	1.00
		M_{23}:$> \to ==$			1		1		0.33	0.50	0.80	0.67	0.50
		M_{24}:$> \to >=$		1	1		1		0.67	0.82	1.80	0.80	8.00
9	if (x > y)	M_{25}:$> \to$ false			1		1		0.33	0.50	0.80	0.67	0.50
		M_{26}:$> \to >$		1	1		1		**0.67**	**0.82**	**1.80**	**0.80**	**8.00**
		M_{27}:$> \to$ true		1					0.50	0.71	1.00	1.00	1.00
		M_{28}:$> \to <=$							0.00	0.00	0.00	0.00	0.00
10	m = y;												
		M_{29}:$> \to <=$							0.00	0.00	0.00	0.00	0.00
		M_{30}:$> \to >$		1					**0.50**	**0.71**	**1.00**	**1.00**	**1.00**
		M_{31}:$> \to >=$		1					0.50	0.71	1.00	1.00	1.00
11	else if(x > z)	M_{32}:$> \to ==$							0.00	0.00	0.00	1.00	0.00
		M_{33}:$> \to !=$		1					0.50	0.71	1.00	0.00	1.00
		M_{34}:$> \to$ true		1					0.50	0.71	1.00	1.00	1.00
		M_{35}:$> \to$ false							0.00	0.00	0.00	0.00	0.00
12	m = x;												
13	return m;}												
	Results		**P**	**F**	**P**	**P**	**F**	**P**					

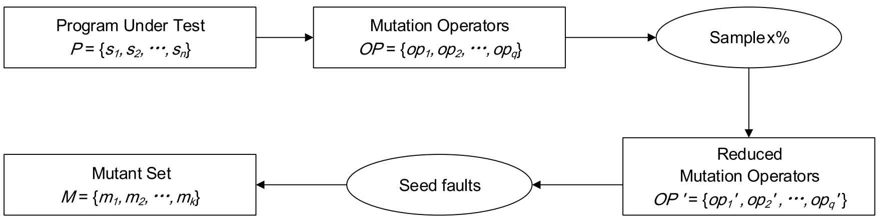

Program Under Test
$P = \{s_1, s_2, \cdots, s_n\}$
Mutation Operators
$OP = \{op_1, op_2, \cdots, op_q\}$
Samplex%
Reduced
Mutation Operators
$OP' = \{op_1', op_2', \cdots, op_q'\}$
Seed faults
Mutant Set
$M = \{m_1, m_2, \cdots, m_k\}$

Fig. 3. Framework of SELECTIVE strategy.

3.2. SAMPLING

The main idea of SAMPLING (Mike and Yves, 2015) is extracting a certain amount of mutants after generating the total mutants by using the whole mutation operator set on statements covered by failed test cases. To alleviate the bias of the sampling process, such a strategy will sample ten independent subsets of mutants, and using the average results to do analysis.

Fig. 4 shows the framework of the SAMPLING strategy. It can be seen that the number of mutants is reduced by randomly choosing a certain ratio of mutants from the whole mutant set. In Mike et al.'s study, they conduct a series of experiments by using different sampling ratios, and the results showed that MBFL with SAMPLING-10% strategy still has better fault localization accuracy than SBFL techniques. The fault localization accuracy of MBFL with SAMPLING-30% strategy is similar to the original MBFL with

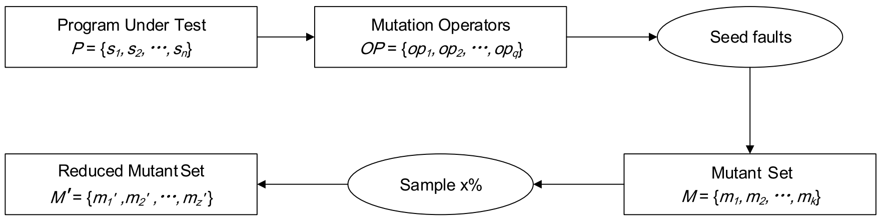

Program Under Test
$P = \{s_1, s_2, \cdots, s_n\}$
Mutation Operators
$OP = \{op_1, op_2, \cdots, op_q\}$
Seed faults
Mutant Set
$M = \{m_1, m_2, \cdots, m_k\}$
Sample x%
Reduced Mutant Set
$M' = \{m_1', m_2', \cdots, m_z'\}$

Fig. 4. Framework of SAMPLING strategy.

a total set of mutants. Their study only uses small programs from the Siemens suite, so the experiments need to be validated by using some other real-world big programs.

4. The hybrid mutation execution reduction strategy for MBFL

Based on the relationship of suspiciousness value between statements and mutants, it can be observed that only the mutants with the maximum suspiciousness value can affect the fault localization accuracy of the MBFL technique. Thus, the mutants with lower suspiciousness are worthless and can be omitted theoretically.

To reduce the execution cost of MBFL, in this paper, we present a hybrid mutation execution reduction strategy, HMER, which aims to reduce the execution cost of MBFL from two aspects, which are called as “do fewer” and “do smarter”. “Do fewer” refers to the strategy of reducing the number of used mutants before executing them, and “do smarter” is optimizing the process of mutants execution during MBFL.

4.1. Do fewer

In the first phrase of HMER, a Weighted Statement-Oriented Mutant Sampling strategy, WSOME, is proposed to reduce the number of used mutants in MBFL. Since fewer mutants are executed, the execution cost of MBFL can be reduced. Therefore, in this paper, “do fewer” refers to using WSOME strategy before executing mutants on test cases. In WSOME, we consider the importance of program statements differently and use SBFL formulas to calculate the weights of program statements. During the mutant sampling process, the mutants generated from statements with higher weights will have more chance to be kept to do further analysis.

Algorithm 1 provides the pseudo-code of using WSOME strategy for mutant sampling. The inputs of Algorithm 1 include the program under test P, mutation operators OP, test suite T, test cases' execution results R, and finally, the mutant sampling ratio $X\%$. The output of this algorithm is a subset of mutants.

In what follows, we will explain the algorithm in detail. Line (1) obtains the failed test cases set T_f, and coverage information Cov. Line (2) returns the statements covered by failed test cases. Initialize a set for generated mutants in line (3). The loop in line (4) calculates the weight of statements and generate corresponding mutants. Lines from (5) to (6) calculate the suspiciousness values of all statements in S_f as $Sus(S_i)$ by $Dstar^3$. The following formula obtains the sampling weight of different statements:

$$Weight(S_i) = \frac{Sus(S_i)}{\sum_{i=1}^{n} Sus(S_i)},$$

where n is the number of elements in S_f. (1)

From the above formula, it is obvious that statements with higher suspiciousness value will be assigned with bigger weights. Line (7) produces mutants by seeding fault on statements from S_f and adds them to *MutantSet* in line (8). Line (10) and (11)

Algorithm 1: WSOME for mutant sampling

```
Input: program under test $P$, mutation operators $OP$, test suite
$T$, test cases' execution results $R$ , sampling ratio $X\%$
Output: Reduced mutant set $ReducedMutantSet$

 1. Obtain failed test cases set $T_f$ and coverage information
    $Cov$ after executing $T$ on $P$.
 2. Get statements set $S_f$ covered by $T_f$ using $Cov$ and $P$.
 3. $MutantSet$: a set for generated mutants, initialized as
    empty.
 4. for statement $S_i$ in $S_f$ do
 5.     Compute the suspiciousness of $S_i$ using SBFL formula
        $Dstar^3$.
 6.     Compute the weight of $S_i$ as $Weight(S_i)$ (equation (1)).
 7.     Mutanting $S_i$ using $OP$, obtain the mutant set
        $Mutant(S_i)$ for $S_i$.
 8.     Add $Mutant(S_i)$ to $MutantSet$.
 9. end for
10. Count the number of all mutants $MutantNum$.
11. Count the number of reduced mutants
    $ReducedMutantNum$, where $ReducedMutantNum$ =
    $MutantNum * X\%$.
12. $ReducedMutantSet$: a set for reduced mutants, initialized
    as empty.
13. for statement $S_i$ in $S_f$ do
14.     Compute the number of mutant sample
        $SampleMutantNum(S_i)$ for $S_i$, where $SampleMutantNum(S_i)$
        = $Weight(S_i) * ReducedMutantNum$.
15.     Sample $SampleMutantNum(S_i)$ mutants from $Mutant(S_i)$
        and add to $ReducedMutantSet$.
16. end for
17. Return $ReducedMutantSet$.
```

calculate the number of all mutants and reduced mutants. Initialize a new set for reduced mutants in line (12). After mutant generation, the loop in line (13) returns a reduced mutant set for each statements. Lines (14) computes the mutant sampling number of each statements. The ratio of mutant sampling is $X\%$, and the total number of mutants is *MutantNum*. Line (15) samples mutants randomly from each $Mutant(S_i)$ and add them to *ReducedMutantSet*. In line (17), a new mutant reduction set can be generated for further analysis. The easier understand framework of WSOME strategy can be seen in Fig. 5.

4.2. Do smarter

In this paper, “do smarter” refers to the strategies employed to avoid executing the worthless mutant-test-pairs, which does not affect the suspiciousness value of statements. Thus eliminating such executions has no harmful effect on the fault localization accuracy of MBFL. More specifically, in the second step, after the used mutant set is determined, a Dynamic Mutation Execution

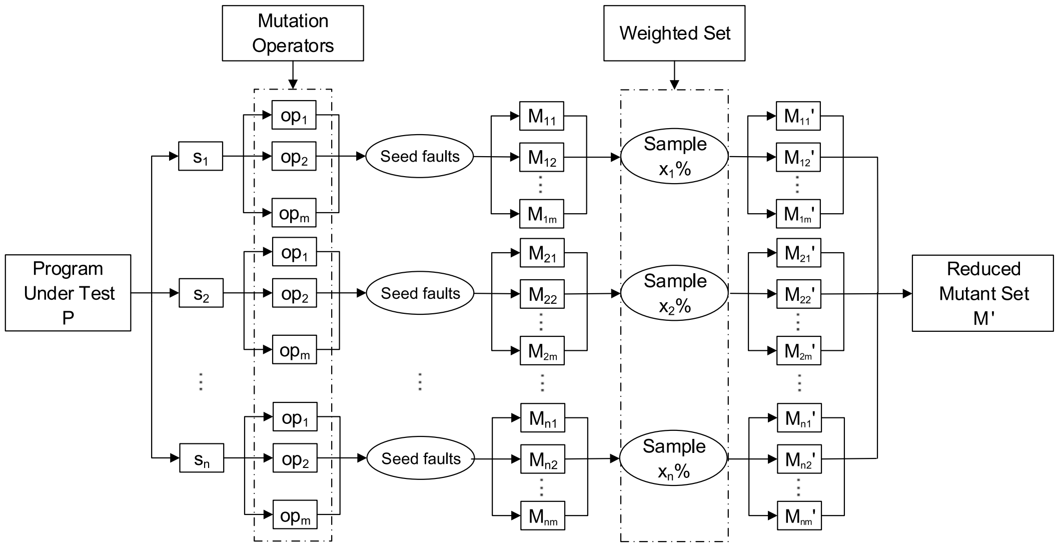


Fig. 5. Framework of WSOME strategy.

Strategy proposed in our previous work (Yong et al., 2018), DMES, is employed to reduce the worthless executions of mutants and test cases further.

DMES strategy is mainly based on the execution of mutants and the relationship of suspiciousness value between mutants and statements. The suspiciousness value of each statement is determined by only one mutant, so the other mutants' execution is worthless in theoretically. From the MBFL formulas listed in Table 4, it can be seen that the suspiciousness of mutant M(denoted as $Sus(M)$) is determined by four parameters, a_{kf}, a_{kp}, a_{nf}, and a_{np}. It should be noted that the number of failed test cases, *totalfailed*, is a constant value, $totalfailed = a_{kf} + a_{nf}$. Similarly, the number of passed test cases, *totalpassed* is a constant value too, $totalpassed = a_{kp} + a_{np}$. Therefore, the suspiciousness value of each mutant is determined by only two parameters, which are a_{kf} and a_{kp}, and there is a positive correlation between $Sus(M)$ and a_{kf}, while a_{kp} has a negative correlation with $Sus(M)$.

Motivated by the above observations, DMES strategy consists of two parts: mutant execution optimization and test case execution optimization, which are named as MEO and TEO, respectively. Guided by the positive correlation between $Sus(M)$ and a_{kf}, MEO calculating the upper suspiciousness value of mutants by running them towards failed test cases, and then order these mutants. Mutants with higher upper suspiciousness value will be executed on other passed test cases first to calculate the exact suspiciousness value. TEO strategy is based on the negative correlation between $Sus(M)$ and a_{kp}. When executing mutants on passed test cases, $Sus(M)$ will be decreased along with the increasing of a_{kp}, and there exists a threshold value when a_{kp} is greater than it, the corresponding mutant's suspiciousness value will have no chance to be qualified as the mutant with the maximum suspiciousness value, then the rest executions can be terminated, where the suspiciousness value of the corresponding statement will keep the same.

4.2.1. Example

A working example of MBFL with DMES strategy is shown in Fig. 6. In this example, MBFL with *Ochiai* formula is used for suspiciousness calculation since the previous study showed that it performs better than others (Yong et al., 2018). The mutants in Fig. 6 are coming from Table 5. In Fig. 6, the information of test cases, coverage, mutants, and the execution results of mutants and test cases are listed. When one mutant is executed on one test case, the result is marked as k or n, which indicates the corresponding mutant is killed or not by the test case. It can be seen that the seven mutants are ordered by $\overline{Sus}(M)$, which is the upper bound value of each mutant. $\overline{Sus}(M)$ computed by setting a_{kp} to 0 and a_{np} to the number of passed test cases set T_p. *Ochiai* used in this example, $\overline{Sus}(M)$ is calculated as follows:

$$\overline{Sus}(M) = \frac{a_{kf}}{\sqrt{(a_{kf} + a_{nf}) * a_{kf}}} \tag{2}$$

If $\overline{Sus}(M)$ is lower than the current maximal suspiciousness (denoted by *SusMax*) for the mutants already executed, mutant M can be skipped in the follow-up executions.

The *threshold* value shown in the second last column is guided to terminate the worthless execution of passed test cases. In our definition (Yong et al., 2018), if *SusMax* equals to 0, *threshold* is maximum and equals to $|T_p| = a_{kp} + a_{np}$, which means that all test cases in T_p covering statement should be checked. Hence, *threshold* can be calculated as follows:

$$threshold = \begin{cases} a_{kp} + a_{np}, & SusMax = 0, \\ \left\lceil \dfrac{a_{kf}^2}{SusMax^2 * (a_{kp} + a_{np})} - a_{kf} \right\rceil, & SusMax \neq 0. \end{cases} \tag{3}$$

It is helpful for skipping test cases execution with a suitable execution order of test cases in T_p. The reason is that, if the test cases killing mutant are executed earlier, the current number of executed test cases(denoted as *KillNum*) in T_p killing mutant, will reach *threshold* more quickly.

Previous work showed that if a test case can kill one mutant in mutant set of statement s(denoted as $MutantSet(s)$), it possibly can kill other mutants in $MutantSet(s)$ (Zhang et al., 2013). Therefore, to run test cases in T_p kill mutant earlier, the execution order of T_p can be determined based on the history that test cases killed mutants. More specifically, if test cases $t_i, t_j \in T_p$ and t_i already killed more mutants in $MutantSet(s)$ than t_j before mutant M is checked, t_i should run on M before t_j.

In Fig. 6, TEO first initialized *SusMax* as 0 and *history* is set to $\langle 0, 0 \rangle$, since only t_3 and t_4 in T_p covering s. Then, MEO ran t_2 and t_5 on each mutant with $2 \times 7 = 14$ executions. The upper boundary from $\overline{Sus}(m_1)$ to $\overline{Sus}(m_7)$ computed by Formula (2). Next, new order of mutants executed by T_p is decided by upper boundary, that is, $m_3, m_5, m_1, m_2, m_4, m_6, m_7$, showed in leftmost column.

	t_1	t_2	t_3	t_4	t_5	t_6							
Coverage of s and test results of mid													
C	0	1	1	1	1	0							
R	P	F	P	P	F	P	Suspiciousness of s						
Results of executing T on M(s)							a_{np}	a_{nf}	a_{kp}	a_{kf}	$\overline{Sus}$	*threshold*	*Sus*
m_3	-	*k*	*k*	*n*	*k*	-	3	0	1	2	**1**	2	0.82
m_5	-	*k*	*k*	/	*k*	-	3	0	1	2	**1**	1	0.82
m_1	-	*n*	/	/	*k*	-		1		1	0.71		
m_2	-	*n*	/	/	*k*	-		1		1	0.71		
m_4	-	*n*	/	/	*k*	-		1		1	0.71		
m_6	-	*k*	/	/	*n*	-		1		1	0.71		
m_7	-	*n*	/	/	*n*	-		2		0	0.00		

Fig. 6. A working example of MBFL with DMES strategy.

In the new execution sequence, m_3 should be executing T_p first. Here the *threshold* of m_3 is equal to *totalpassed*, i.e. 2, because $SusMax = 0$. After executing T_p on m_3, $Sus(m_3) = 0.82$, *SusMax* is updated to 0.82 and *history* is modified as $\langle 1, 0\rangle$(t_3 killed m_3 and t_4 did not). Then the *threshold* of m_5 computes as $\left\lceil \frac{2^2}{0.82^2 * 2} - 2 \right\rceil = 1$. According to $history = \langle 1, 0\rangle$, t_3 run m_5 firstly and t_3 killed m_5. At the moment, the current number of passed test cases killed m_5, $KillNum = 1$ and the condition $Kill \geq threshold$ is satisfied. As a result, t_4 is do not need to be executed on m_5. At the same time, $\overline{Sus(m_5)} = 1$ is larger than *SusMax* and $Sus(m_5)$ equals to *SusMax*. But the upper boundary of m_3, $\overline{Sus(m_3)} = 0.71$ is less than *SusMax*. For this reason, the rest of executions could be terminated and the suspiciousness of s is set with *SusMax*, i.e., $Sus(s) = SusMax = 0.82$. Hence, 5 mutants do not need to be executed by pass test cases.

Obviously, when applying both MEO and TEO, it requires $2 \times 7 + 1 \times 2 + 1 = 17$ executions to get $Sus(s) = 0.82$. Fig. 6 cells highlighted with red "/" means the reduced mutant executions. Thus in this working example, 11 mutant executions can be reduced, and the efficiency of MBFL can be improved under the promise that the suspiciousness value of the statement remains 0.82, as the same with the original MBFL technique. From Fig. 6, it can be seen that the execution cost can be reduced a lot while the fault localization accuracy is the same with the original MBFL.

4.2.2. Algorithm

Algorithm 2 shows the pseudo-code of using MBFL with DMES strategy to calculate the suspiciousness value of one statement with some generated mutants. The inputs of the algorithm include statement s, mutant set *MutantSet* generated from s, test cases' coverage information *Cov*, failed test cases set T_f, and passed test cases set T_p. To more specific, test cases failed or passed denote the results of them when executed on the original faulty program. The output of the algorithm is the suspiciousness value of the statement $Sus(s)$.

In what follows, we explain the algorithms in detail. Line (1)–(2) initialize *SusMax* and *history*, separately. Lines (3)–(8) iterate over the mutants in *MutantSet* to get parameters a_{kf} and a_{nf} for each mutant. In particular, lines (4)–(5) execute failed test cases on mutant if the test case covered statement s. Line (7) counts a_{kf} and a_{nf}. Line (8) employs equation (2) to calculate the upper boundary suspiciousness value of mutant $Mutant_i$, $\overline{Sus(Mutant_i)}$. Line (10) descending sorts all mutants base on the upper boundary suspiciousness value of each mutant calculated in line (8). Then according to *ReorderedMutantSet*, lines (11)–(22) iterate over M to obtain the suspiciousness value of statement $Sus(s)$. During this process, TEO strategy is employed to avoid worthless test cases' executions. Lines (11)–(12) reorder T_p as $ReorderedT_p$ based on *history* and calculate the *threshold* with Eq. (3). Lines (14)–(18) iterate over $ReorderedT_p$ to run the passed test cases on $Mutant_i$. Specifically, lines (15)–(17) execute t_j on $Mutant_i$, update *history* and record the number of passed test cases that already killed $Mutant_i$, named $KilledNum_i$. Line (18) judges whether $KilledNum_i$ is greater than or equal to *threshold*. If it is true, the execution of $Mutant_i$ is terminated and goto line (11). Otherwise, $Mutant_i$ is checked by the following test cases in $ReorderedT_p$. Then line (20) computes the suspiciousness value of $Mutant_i$, named $Sus(Mutant_i)$ and line (22) updates *SusMax* by comparing *SusMax* and $Sus(Mutant_i)$. If *SusMax* is larger than or equal to $\overline{Sus(NEXT(Mutant_i))}$, line (24) returns *SusMax* as the suspiciousness value of statement s, where $NEXT(Mutant_i)$ means the mutant next to $Mutant_i$. Otherwise, the iterations continue until all mutants in *ReorderedMutantSet* are checked. In this case, current maximum suspiciousness value of mutants *SusMax* is $Sus(s)$ returned in line (24).

4.3. Framework of MBFL with HMER strategy

From the above analysis, it can be seen that MBFL has the benefit of high fault localization accuracy but suffers from a huge mutation execution cost problem. To reduce the cost, we introduce a Hybrid Mutation Execution Reduction strategy, HMER. Where "Hybrid" indicates that a combination of two strategies, "do fewer" and "do smarter", are employed to reduce the number of used mutants and optimize the execution of mutants and test cases, then finally, the execution cost of MBFL can be reduced a lot by using HMER approach. More specifically, "do fewer" refers to using WSOME strategy, which is described in Section 4.1, and "do smarter" is applying the DMES method, which is presented in Section 4.2.

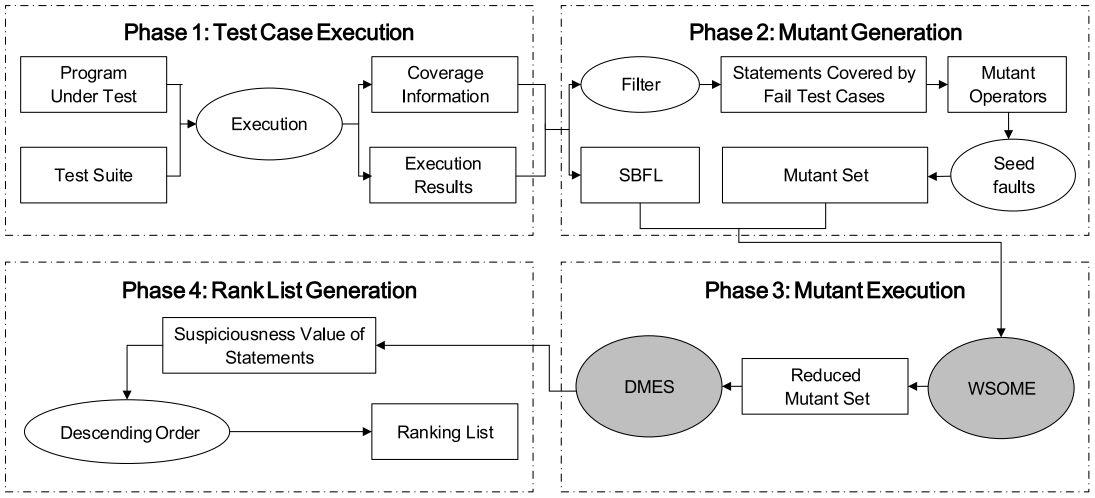

Phase 1: Test Case Execution
Program Under Test
Test Suite
Execution
Coverage Information
Execution Results
Phase 2: Mutant Generation
Filter
Statements Covered by Fail Test Cases
Mutant Operators
SBFL
Mutant Set
Seed faults
Phase 4: Rank List Generation
Suspiciousness Value of Statements
Descending Order
Ranking List
Phase 3: Mutant Execution
DMES
Reduced Mutant Set
WSOME

Fig. 7. Framework of MBFL with HMER strategy.

Algorithm 2: Statement suspiciousness value calculation with DMES strategy

Input: statement s, mutant set $MutantSet$ generated from s, coverage information Cov, failed test cases set T_f, passed test cases set T_p
Output: Suspiciousness value of statement s $Sus(s)$

```
1.  SusMax: the maximum suspiciousness value of all
    mutants generated from statement s, initialized to 0.
2.  history: an array for recording the number of mutants
    killed by passed test cases, all elements initialized to 0.
3.  for $Mutant_i$ in MutantSet do
4.    for test case $t_j$ in $T_f$ do
5.      if $t_j$ covered the statement s, then Execute $t_j$ on
        $Mutant_i$
6.    end for
7.    Count MBFL parameters $a_{kf}$ and $a_{nf}$.
8.    Calculate the upper boundary suspiciousness
      $\overline{Sus(Mutant_i)}$ with equation (2).
9.  end for
10. ReorderedMutantSet: new order of MutantSet based on the
    upper boundary suspiciousness.
11. for $Mutant_i$ in MutantSet do
12.   Reorder $T_p$ based on the history as $ReorderedT_p$.
13.   Calculate threshold with equation (3).
14.   for test case $t_j$ in $ReorderedT_p$ do
15.     Execute $t_j$ on $Mutant_i$.
16.     Update history.
17.     Count the number of test cases killed $Mutant_i$ as
        $KilledNum_i$.
18.     if $KilledNum_i \geq threshold$ then goto line 11.
19.   end for
20.   Calculate the suspiciousness of $Mutant_i$ as $Sus(Mutant_i)$.
21.   Assign $Sus(Mutant_i)$ to SusMax if $Sus(Mutant_i) \geq$
      SusMax.
22.   if $SusMax \geq \overline{Sus(NEXT(Mutant_i))}$ then assign SusMax to
      the suspiciousness of statement s to as $Sus(s)$ and return
      $Sus(s)$.
23. end for
24. return $Sus(s)$.
```

Since the “do fewer” and “do smarter” in HMER are strategies that aim to reduce the execution cost in different steps of MBFL, so the two strategies can be combined together to enhance the reduction effect. As shown in Fig. 7, MBFL with HMER strategy contains four phases. At first, test cases are executed on the program under test to get coverage information and execution results. Second, statements covered by failed test cases are filtered and also used to generate mutants by seeding faults in them. Besides, SBFL techniques are employed to calculate the suspiciousness value of these statements. Next, WSOME employs a weighted set calculated by SBFL formulas to sampling mutants and obtain a reduced mutant set. The mutants in the reduced mutant set will be executed on test cases with the guidance of the DMES strategy. Finally, the suspiciousness value of statements is recorded, and a ranking list will be generated to assist developers in finding faults.

5. Experimental study

5.1. Research questions

To evaluate the effectiveness of our proposed HMER strategy, we compared it with some existing techniques in two aspects: mutation execution cost reduction and fault localization accuracy. The following research questions are investigated:

RQ1: Compared with the original MBFL and MBFL with SAMPLING or SOME strategies, how does MBFL with our proposed HMER method perform when considering the mutation execution reduction rate?

RQ2: When employing our proposed HMER approach in MBFL, is the fault localization accuracy better than SBFL and MBFL with SAMPLING or SOME methods?

RQ3: Compared with the original MBFL, how does MBFL with the HMER method perform about the accuracy of fault localization?

RQ4: There are lots of different formulas to calculate the mutant sampling weights of statements, and which one is the best to make sure that MBFL with HMER has the highest fault localization accuracy?

RQ1 compares the effects of different mutation reduction strategies under various program versions. **RQ2** examines the performance of HMER on the accuracy of fault localization. And **RQ3** focuses on studying if WSOME has an impact on the accuracy of fault localization. **RQ4** finally compares the impact of different weighted formulas on HMER.

Table 6
Subject programs.

Programs	#Versions (used)	#LOC	#Mutants	#Test cases
printtokens	7(3)	342	4235	4130
printtokens2	10(6)	355	10 145	4115
schedule	9(5)	296	2223	2650
schedule2	10(8)	263	2889	2710
totinfo	23(23)	273	6314	1052
replace	32(26)	513	10 661	5542
tcas	41(41)	139	5115	1608
sed	8(8)	11 470	78 543	360
grep	10(10)	13 826	86 151	668

5.2. Experiment set-up

In order to address the research questions above, we select nine subject programs from SIR (Do et al., 2005) as benchmarks. The first seven programs come from Siemens Suite and relatively small programs with hundreds of lines of code. Besides, two are real-world, and large-sized programs, sed and grep are included to conduct our empirical study. All of them are open-source C programs with faulty versions and test suites. Table 6 lists the information of all subject programs. There are 150 faulty versions from nine programs; however, some faulty versions are excluded for reasons:

1. the related test suite cannot detect failures on the faulty versions;
2. the failures lead to segment faults and it is hard to collect full coverage information.

Therefore, a total of 130 faulty-version programs are used in our experiments. The GNU gcov tool (Yang et al., 2009) is employed to collect the coverage information on the Linux operating system. For every statement covered by failed test cases, the mutation analysis tool *Proteum*/*IM*2.0 (Delamaro et al., 2001) is utilized to generate mutants. *Proteum*/*IM*2.0 is a well-known tool widely used in many mutation testing related studies (Mike and Yves, 2015, 2014; Delamaro et al., 1996, 2001). The mutation operators we used are proposed by Agrawal et al. (1989), which are included in the *Proteum*/*IM*2.0 tool. The MBFL suspiciousness formulas used in this experiment are shown in Table 4.

In this paper, we implement the prototype of MBFL with HMER strategy. During the process of implementation, different SBFL and MBFL formulas, and different sampling rates can be selected and have different results. The experimentation presents results for the SBFL techniques as a baseline for MBFL techniques. In our experiments, the sampling rate of our proposed HMER strategy is 10%, which is the same with SOME strategy. Besides, there are three sampling rates in the SAMPLING strategy, which are 10%, 20%, and 30%, respectively. In the experiments of answering RQ1, RQ2, and RQ3, we use *Dstar*[3] as the SBFL formula to calculate the suspiciousness value of statements in HMER strategy, because it is proved as the state-of-art SBFL formula (Eric et al., 2014). In RQ4, all the five SBFL formulas list in Table 1 are selected to compare the effectiveness. In the experiments of answering RQ1 and RQ4, *Ochiai* is used to be the MBFL formula to calculate the suspiciousness value of mutants and statements. In RQ2 and RQ3, all MBFL formulas listed in Table 4 to do results analysis.

All experiments were performed on HP Server with 24 Intel Xeon(R) cores. In addition, considering the random process in sampling mutants and the initial sequence of mutants in the experiment, we have performed our experiments ten times and analyze the average results in all strategies.

5.3. Evaluation metrics

5.3.1. EXAM score

In this paper, we use *EXAM Score* to evaluate the fault location accuracy of MBFL techniques. *EXAM Score* metric uses the percentage of located faults by examining certain percentage statements (Renieres and Reiss, 2003), and lower *EXAM Score* indicates more advanced the faulty statement is in the ranking list of suspiciousness so that the developer can check fewer statements when the program is debugged, and the corresponding fault localization accuracy is higher, and the method is better. The *EXAM Score* is defined in the following formula:

$$EXAM\ Score = \frac{Rank}{Number\ of\ executable\ statements} \tag{4}$$

The numerator in the above formula represents the rank of the faulty statement in the ranking list generated by a fault localization techniques, and the denominator is the total number of executable statements that need to be checked by programmers.

In Formula (4), the value of *Rank* indicates the specific number of statements that need to be checked before finding out the faulty statement of PUT. *Rank* can be calculated by the following formula:

$$Rank = \Sigma_{i=1}^{N} 1[Sus(i) > Sus(f)] + \frac{\Sigma_{i=1}^{N} 1[Sus(i) = Sus(f)]}{2} \tag{5}$$

In Formula (5), N represents the total number of executed statements, $Sus(i)$ represents the suspicious value of statement i, and $Sus(f)$ represents the suspicious value of the faulty statement. The *Rank* value represents a wasteful effort to find the faulty statement. In general, we must check all statements that have a greater suspicious value than the faulty statement, and the number is represented by $\Sigma_{i=1}^{N} 1[Sus(i) > Sus(f)]$. On average, we must check half of the statements with suspiciousness values equal to the suspicious value of the faulty statement, and the number is represented by $\Sigma_{i=1}^{N} 1[Sus(i) = Sus(f)]$.

5.3.2. Mutant-test-pair

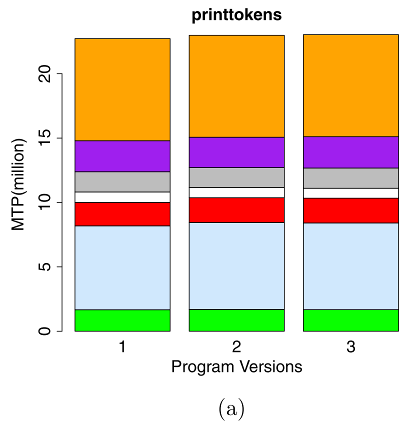
In this experiment, we note that each test case executed by a mutant is recorded as a Mutant-Test-Pair(MTP) (Yong et al., 2017). MTP is used to measure the mutant execution cost of calculating the suspiciousness value of mutants and statements; an MBFL technique with less MTP execution will have better efficiency. Assume that we have a mutant set with n mutants and executed by a test suite contains m test cases, So the number of MTPs can be calculated by the following formula:

$$MTP = n * m \tag{6}$$

5.4. Experimental results and analysis

5.4.1. Answer for RQ1

In order to answer RQ1, we calculate the executed number of MTPs from different MBFL techniques. In particular, to reduce the bias caused by the random process, we repeat ten times experiments and use the average value to do results analysis. The results are represented by histograms, where Fig. 8 is an example of three versions of the printtokens. In the figure, each bar on X-axis is a PUT corresponding to a faulty version, and Y-axis is the number of executed MTP measured by millions. As detailed in Fig. 8(a), each column is divided into six parts: HMER 10%, DMES, SOME 10%, SAMP 10%, SAMP 20%, SAMP 30% and TOTAL, which are related to the MTP executions of MBFL with HMER 10%, DMES, SOME 10%, SAMP 10%, SAMP 20%, SAMP 30% strategies and the original MBFL respectively. From the bottom up, HMER 10%, colored by green, is corresponding to the average MTP executions reduced by using HMER in MBFL, and the sampling ratio



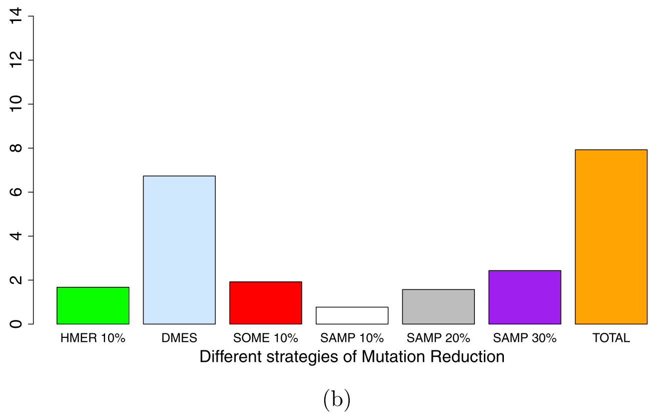


Fig. 8. *Printtokens* as the illustrated example of cost reduction comparison of different strategies. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 7
Average MTP reduction rates of HMER, SOME and SAMPLING strategies.

Programs	HMER 10%	DMES	SOME 10%	SAMP 10%	SAMP 20%	SAMP 30%
printtokens	78.8%	15.8%	76.2%	90.0%	80.3%	69.7%
printtokens2	78.2%	31.6%	74.5%	86.6%	79.1%	67.5%
schedule	74.5%	19.5%	72.6%	90.0%	80.0%	70.0%
schedule2	93.4%	91.1%	78.9%	89.3%	79.3%	73.1%
totinfo	85.0%	29.0%	79.8%	86.5%	73.7%	65.2%
replace	87.7%	40.0%	82.2%	90.5%	80.7%	70.5%
tcas	90.8%	60.2%	80.8%	90.1%	80.3%	70.4%
sed	91.1%	49.9%	79.1%	89.6%	79.7%	69.9%
grep	92.2%	62.1%	77.5%	91.1%	82.4%	74.4%
Adverage	87.9%	44.4%	79.7%	89.4%	79.2%	69.8%

is 10%. The light blue one is the average MTP of DMES. Third to last, denoted by the red box, is mapped to the average MTP executions further reduced by using SOME with 10% sampling ratio. The rest part of the bar are SAMP 10%(colored by white), SAMP 20%(colored by gray), SAMP 30%(colored by purple), and the original MBFL(colored by orange). It should be noted that the color representation in Fig. 9 is the same as in Fig. 8.

Fig. 9 shows that, in most cases, HMER, SOME, and SAMPLING can significantly reduce the MTP execution cost of MBFL, but DMES has no obvious execution reduction effect. As discussed in Section 4, HMER strategy has two steps, WSOME and DMES. The WSOME strategy has almost the same MTP reduction compared to SOME with the same sampling ratio of 10%, for the reason that WSOME changed the mutant distribution but not the number of mutants. To further investigate the performance of mutant execution reduction strategies, we depict the average MTP reduction ratio of every strategy under all 130 program versions in Table 7. As the table shows, the average reduction ratios of HMER 10% are varied from 74.5% to 93.4%. Further comparing HMER with SOME and three SAMPLING methods, the average reduction rate of HMER(87.9%) is greater than that of SOME 10%(79.7%), SAMP 20%(79.2%) and SAMP 30%(69.8%), and light less than that of SAMP 10%(89.4%) for all program versions. Further investigations reveal that HMER 10% requires at least one mutant for each type to be kept for each statement covered by failed test cases. Thus the total sampling rate for the whole program under test is larger than 10%, with SAMP 10% has no this limitation.

Now the RQ1 is answered that HMER 10% has a higher average mutation execution reduction rate than DMES, SOME 10%, SAMP 20%, and SAMP 30%, but light less than SAMP 10%.

5.4.2. Answer for RQ2 and RQ3

RQ2 and RQ3 concern the fault localization accuracy comparison among SBFL, MBFL with different strategies and the original MBFL. To answer the two research questions, *EXAM score* is calculated for SBFL, MBFL with each strategy. The results of SBFL and MBFL with each strategy are calculated using the corresponding five SBFL and MBFL suspiciousness formulas, which are visually compared through the violin plots shown in Fig. 10. In the violin plot, the X-axis represents SBFL and different mutation execution strategies, while the Y-axis indicates the fault localization accuracy. Each block in the violin plot indicates the distribution of one evaluation metric and the corresponding formula. The breadth of the block represents the data density of the corresponding value of the Y-axis for all subject program versions. So the wider in the bottom of the block and thinner in the up of the block indicates that the corresponding technique has a better fault localization accuracy.

From Fig. 10, we can see that no matter which MBFL formula is used to calculate the suspiciousness value of mutants, the first block (HMER 10%) has almost the same block shape with the second block, which represents the original MBFL technique. More specifically, the EXAM score distribution shows that HMER 10% of the data is almost gathered around 0, more concentrated than SOME 10% and far less than the other four blocks (SAMP 10%, SAMP 20%, SAMP 30%, SBFL), indicating that the fault localization accuracy of MBFL with HMER 10%, is much higher than SBFL, MBFL with SOME 10% and the other three SAMPLING strategies. Besides, the similar performance of DMES (the third block) can be seen in the plots.

We further conduct the empirical study of comparing the EXAM score of HMER and SOME on different MBFL suspiciousness formulas. Table 8 lists the various considered values of Exam score in the first column, and the other columns are the percentages of 130 faulty versions whose EXAM score are smaller than the corresponding value. Five formulas we used are same listed in Table 4, *Jaccard*(JA), *Ochiai*(OC), *Op*2(OP), *Tarantula*(TA) and *Dstar*^{3} (DS). The column named *TOTAL* means the original

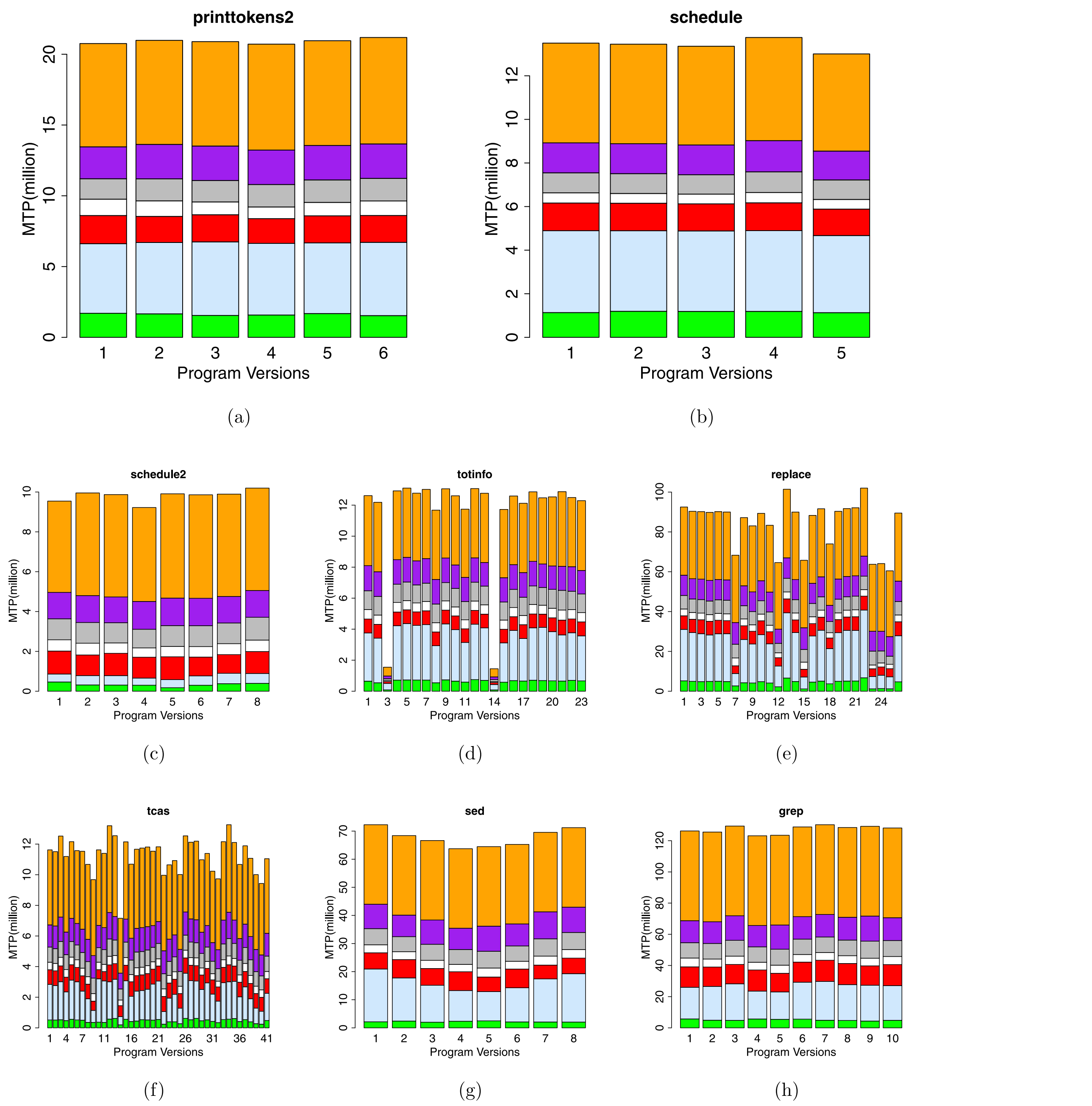
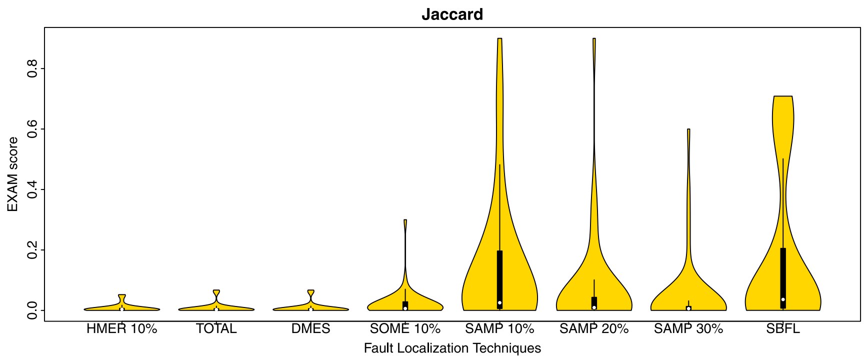


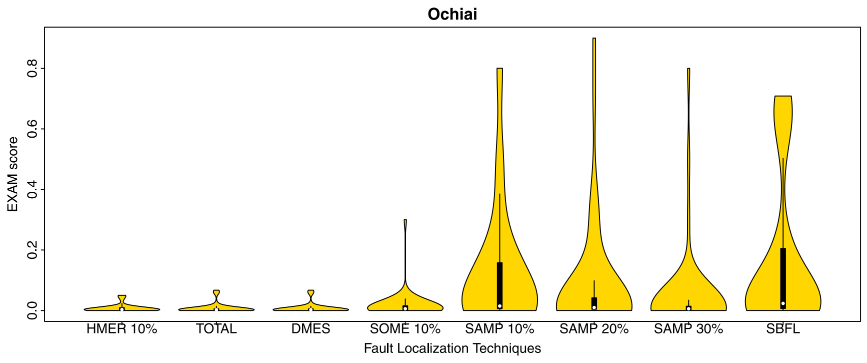
Fig. 9. Cost reduction performances of different strategies. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 8
Comparison of fault localization accuracy between MBFL with HMER and MBFL with SOME.

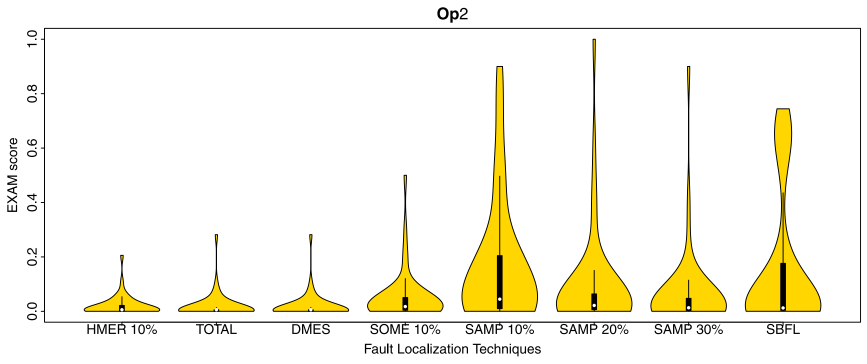
Score	TOTAL					HMER 10%					SOME 10%				
(≤)	JA	OC	OP	TA	DS	JA	OC	OP	TA	DS	JA	OC	OP	TA	DS
1%	0.90	0.90	0.83	0.58	0.90	0.70	0.68	0.53	0.53	0.80	0.60	0.58	0.38	0.45	0.55
5%	0.93	0.93	0.88	0.85	0.93	0.95	0.95	0.85	0.80	0.98	0.90	0.88	0.75	0.73	0.88
10%	1.00	1.00	0.95	0.93	1.00	0.98	0.95	0.95	0.90	1.00	0.93	0.90	0.90	0.80	0.90
15%	1.00	1.00	0.98	0.95	1.00	0.98	0.98	0.95	0.93	1.00	0.93	0.95	0.95	0.93	0.93
20%	1.00	1.00	0.98	1.00	1.00	0.98	0.98	0.98	0.95	1.00	0.93	0.95	0.98	0.93	0.93
30%	1.00	1.00	1.00	1.00	1.00	0.98	0.98	0.98	0.98	1.00	0.98	0.95	0.98	0.95	0.98
40%	1.00	1.00	1.00	1.00	1.00	0.98	0.98	0.98	0.98	1.00	0.98	0.98	0.98	0.95	0.98
50%	1.00	1.00	1.00	1.00	1.00	0.98	0.98	0.98	0.98	1.00	0.98	1.00	0.98	1.00	0.98
60%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.98	1.00	0.98	1.00	0.98	1.00	0.98
70%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	1.00	1.00
80%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
90%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
100%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00



(a)



(b)



(c)

Fig. 10. Fault localization accuracy comparison of different techniques.

MBFL technique without any mutant reduction strategy. The column HMER 10% and SOME 10% represent MBFL with a mutant sampling strategy in the case of sampling ratios 10%. For example, when we implement the original MBFL (TOTAL) with the formula *Jaccard*, there are 90% faulty versions of all 130 whose EXAM score are smaller than 1%.

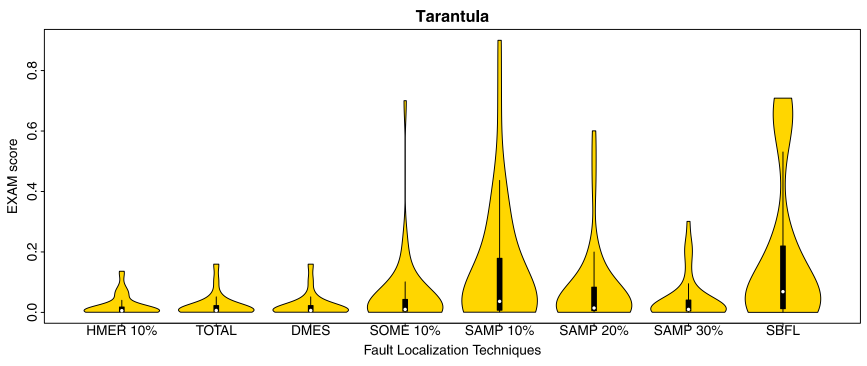
For the different formulas with the same reduction technique, *Jaccard*, *Ochiai*, and *Dstar*^{3} have the similar best performance in original MBFL and *Dstar*^{3} best in HMER 10%, with *Ochiai* and *Tarantula* best in SOME 10%. Besides, for the same formulas, HMER 10% exhibits better localization efficacy in *Jaccard*, *Op*2, *Dstar*^{3} than SOME 10%. It is worth highlighting that HMER 10% has a better performance than the original MBFL in *Dstar*^{3}.

To determine the statistical significance between proposed HMER and other techniques, we first collect the accuracy data of SBFL, and different MBFL techniques by recording the *EXAM Scores* (Eq. (4)) of all program versions, and then employ the *Wilcoxon signed − rank test* at the confidence level of 95%. Table 9 summarizes the testing results on the fault localization accuracy of HMER 10% with original MBFL, MBFL with four reduction strategies, SOME 10%, SAMP 10%, SAMP 20%, and SAMP 30%, and SBFL techniques. These results are presented in five formulas. In Table 9, the bold parts where P-Values greater than 0.05 represents that there is no statistically significant difference between HMER 10% and the corresponding reduction strategies.

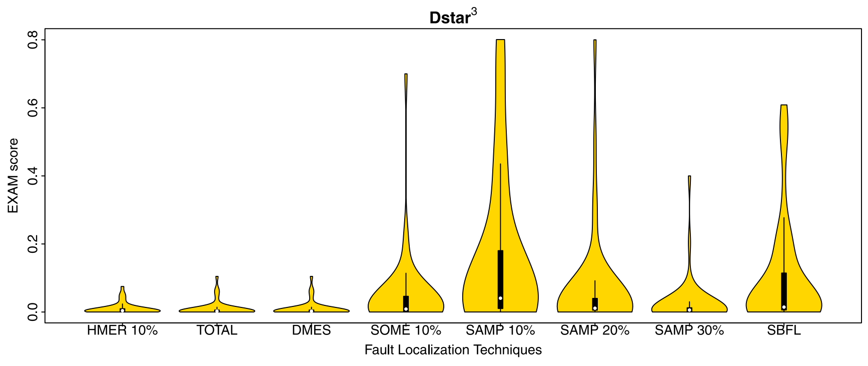
Table 9
P-values between HMER and other techniques on the fault localization accuracy.

Method	MBFL techniques					SBFL
	TOTAL	Reduction strategies				
		SOME 10%	SAMP 10%	SAMP 20%	SAMP 30%	
Jaccard	**0.872**	0.001	0.000	0.004	0.119	0.002
Ochiai	**0.465**	0.007	0.000	0.005	0.018	0.003
Op2	**0.429**	0.007	0.000	0.005	0.018	0.037
Tarantula	0.003	0.005	0.002	0.039	0.003	0.001
Dstar^3	**0.639**	0.001	0.000	0.003	**0.063**	0.048

From Table 9, we can observe that there is no statistically significant difference between HMER 10% and the original MBFL



(d)



(e)

Fig. 10. (*continued*).

(TOTAL) on the fault localization accuracy when using *Jaccard*, *Ochiai*, *Op*2 and *Dstar*^{3}. But in most cases, the fault localization accuracy of HMER 10% has a significant difference with SOME 10%, SAMPLING 10%, 20%, 30% strategies, and SBFL. According to our analysis, first of all, HMER is a weighted statement-oriented mutant sampling strategy that based on SOME, it samples mutants under the guidance of weights calculated by SBFL formulas, which can keep more "effective" mutants for further fault localization; therefore, HMER has a better performance than SOME. In addition, the previous study shows SOME outperforms SAMP 10%, SAMP 20%, and SAMP 30% in fault localization accuracy (Yong et al., 2017) and HMER is a non-random mutant sampling strategy, rather than the random sampling strategies SAMP 10%, SAMP 20%, and SAMP 30%, HMER samples mutants in a more "smarter" way, therefore, HMER has a better performance than these strategies. Secondly, after selecting more effective mutants, HMER combined with DMES, which has been proved for keeping fault localization accuracy theoretically (Yong et al., 2018), thereby the fault localization accuracy of HMER has not significantly different from the original MBFL.

From the above analysis, we can safely conclude that: for RQ2, MBFL with HMER 10% strategy has better fault localization accuracy than SBFL and MBFL with other four strategies; and for RQ3, in most cases, the fault localization accuracy of MBFL with HMER 10% strategy has no significant difference with the original MBFL technique.

5.4.3. Answer for RQ4

RQ4 discusses the effect of the SBFL formulas has on the accuracy of fault localization. To answer this question, we use five SBFL formulas (*Jaccard*, *Ochiai*, *Op*2, *Tarantula*, *Dstar*^{3}) to compare the effect. The experimental results are shown in Fig. 11. In Fig. 11, X-axis represents different weighted SBFL formulas while the Y-axis indicates the EXAM score. As introduced previously in RQ2, each block in Fig. 11 represents the distribution of the EXAM score calculated by one SBFL formula. The breadth of the block means the density of the EXAM score for all subject program versions. If one interval is wider, it means that the density of the EXAM score in this interval is higher and has a more concentrated distribution.

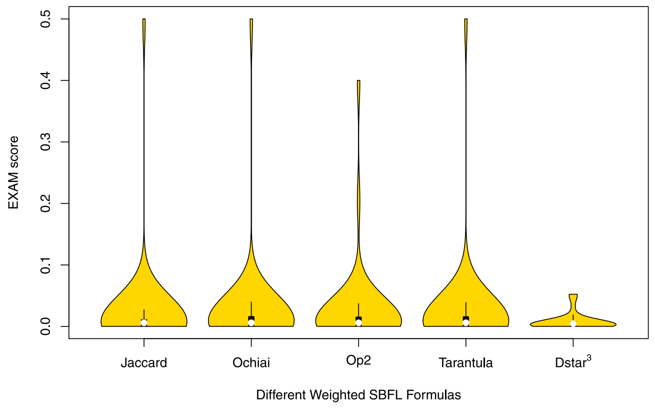


Fig. 11. Fault localization accuracy of different weighted formulas.

It can be seen from Fig. 11, the data distribution in the four bars related to *Jaccard*, *Ochiai*, *Op*2 and *Tarantula* formulas is similar and mainly concentrated in the interval [0, 0.1], while the bar with *Dstar*^{3} formula is mainly in the interval [0, 0.03], which means MBFL with HMER and using *Dstar*^{3} formula has a better fault localization accuracy than with other four SBFL formulas. Such finding is consistent with the claim that *Dstar*^{3} is the state-of-the-art SBFL formula. When using *Dstar*^{3} as the SBFL formula in HMER strategy, it will assign a greater weight to the exact faulty statements, then the mutants generated from these statements will have more chance to be kept to do MBFL;

thus the suspiciousness value of these faulty statements can be almost the same compared with the value calculated by original MBFL technique.

In conclusion, for RQ4, it is clear that using $Dstar^3$ as the SBFL formula in MBFL with HMER strategy shows better fault localization accuracy than using other SBFL formulas.

6. Threats to validity

6.1. Threats to internal validity

In this section, we mainly analyze the potential factors that may influence the conclusions of the experiment. Firstly, compared to other mutation tools, such as *Major* (Just, 2014), we choose Proteum for mutation analysis, which contains different operators. Different mutation tools may bring different results. We will explore additional tools in future work.

Secondly, the other factor we considered is that we did not analyze the real faults and the seeded faults separately. Since artificially injected faults may occur differently from the real fault, which leads to different results in our experiment. Our future research could further analyze the difference between these two kinds of faults to mitigate this threat.

Thirdly, the random algorithm used to sample mutant subsets may have an impact on the evaluation of experiment results. The principal threat to internal validity is involved in a randomly based strategy to sample a given ratio of mutants. To reduce this threat, we repeat the related experiments ten times and use the average results to compare and evaluate.

6.2. Threats to external validity

We limit our study to compare our proposed HMER method with SOME, SAMPLING, and original MBFL by employing only five suspiciousness calculation formulas. However, the SAMPLING and SOME strategies are effective approaches for the existing recognized mutant reduction problem, and the five used suspiciousness calculation formulas include the state-of-the-art formulas, like $Dstar^*$ in SBFL and *Ochiai* in MBFL, which is representative to make the comparison. Besides, the comparison with the original MBFL further validates the effectiveness of HMER. Since we aim to observe HMER's improvement in mutant reduction, we believe these comparisons are appropriate. In another respect, one threat comes from the representativeness of the subject programs(seven Siemens programs and two real-world programs) and their results. Although these subject programs are popular and widely used in fault localization studies, we still need to consider more real-world datasets, such as *Defects4J* (Just et al., 2014), to reduce this threat.

7. Conclusions and future work

This paper address the execution cost problem of MBFL, and propose a hybrid mutation execution reduction approach, called HMER, to reduce the execution cost of MBFL techniques from two aspects, "do fewer" and "do smarter", corresponding to the two steps employed in HMER strategy. The first step is a weighted statement-oriented sampling method, WSOME, which relates to the "do fewer", which means fewer mutants are used in the MBFL technique. The second step is a dynamic mutation execution strategy, DMES, which relates to "do smarter", which means the execution order of mutants and test cases are optimized to avoid worthless executions.

To validate the effectiveness of our proposed method, empirical studies are conducted on 130 fault versions of nine subject programs, and the results are analyzed in two aspects. In the efficiency aspect, the results show that using HMER strategy can significantly reduce the execution cost of MBFL, and the average reduction rate is 87.9%. In fault localization accuracy aspect, the *Wilcoxon signed − rank test* method is employed, and the results show that there is no statistically significant difference between original MBFL and MBFL with HMER strategy in most cases. A further study reveals that using $Dstar^3$ as the SBFL formula can help our proposed HMER strategy to maintain a high fault localization accuracy, and such finding is consistent with the claim that $Dstar^3$ is the state-of-the-art SBFL formula.

In our future work, we will combine our method with other predictive-based methods like PMT (Jie et al., 2016), to further reduce the execution cost of MBFL.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The work described in this paper is supported by the National Natural Science Foundation of China (Grant nos. 61902015, 61872026 and 61672085).

References

Agrawal, Hiralal, DeMillo, Richard, Hathaway, R_, Hsu, William, Hsu, Wynne, Krauser, Edward W, Martin, Rhonda J, Mathur, Aditya P, Spafford, Eugene, 1989. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue.

Assis Lobo de Oliveira, Andre, Goncalves Camilo-Junior, Celso, Freitas, Eduardo, Vincenzi, Auri, 2018. FTMES: A failed-test-oriented mutant execution strategy for mutation-based fault localization. pp. 155–165. http://dx.doi.org/10.1109/ISSRE.2018.00026.

Barr Earl, T., Mark, Harman, Phil, McMinn, Muzammil, Shahbaz, Yoo, Shin, 2015. The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. 41 (5), 507–525. http://dx.doi.org/10.1109/TSE.2014.2372785.

Bin, Noor Tanzeem, Hadi, Hemmati, 2017. Studying test case failure prediction for test case prioritization. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. ACM, pp. 2–11. http://dx.doi.org/10.1145/3127005.3127006.

Bob, Edmison, H, Edwards Stephen, A, Pérez-Quiñones Manuel, 2017. Using spectrum-based fault location and heatmaps to express debugging suggestions to student programmers. In: Proceedings of the Nineteenth Australasian Computing Education Conference. ACM, pp. 48–54. http://dx.doi.org/10.1145/3013499.3013509.

Brun, Y., Ernst, M.D., 2004. Finding latent code errors via machine learning over program executions. In: International Conference on Software Engineering.

Cellier, Peggy, Ducassé, Mireille, Ferré, Sébastien, Ridoux, Olivier, 2008. Formal concept analysis enhances fault localization in software. In: International Conference on Formal Concept Analysis.

Chao, Liu, Long, Fei, Yan, Xifeng, Han, Jiawei, Midkiff, Samuel P., 2006. Statistical debugging: A hypothesis testing-based approach. IEEE Trans. Softw. Eng. 32 (10), 831–848.

Chen Mike, Y., Emre, Kiciman, Eugene, Fratkin, Armando, Fox, Eric, Brewer, 2002. Pinpoint: Problem determination in large, dynamic internet services. In: Dependable Systems and Networks,2002. IEEE/IFIP, 2002. 32nd International Conference on. IEEE, pp. 595–604. http://dx.doi.org/10.1109/DSN.2002.1029005.

Dallmeier, Valentin, Lindig, Christian, Zeller, Andreas, 2005. Lightweight bug localization with AMPLE. In: International Workshop on Automated Debugging.

Delamaro, Márcio Eduardo, Maldonado, José Carlos, Mathur, AP, 1996. Proteum-a tool for the assessment of test adequacy for c programs user's guide. In: PCS, Vol. 96. pp. 79–95.

Delamaro, Márcio Eduardo, Maldonado, José Carlos, Vincenzi, Auri Marcelo Rizzo, 2001. Proteum/IM 2.0: An integrated mutation testing environment. In: Mutation Testing for the New Century. Springer, pp. 91–101.

DeMillo Richard, A., J., Lipton Richard, G., Sayward Frederick, 1978. Hints on test data selection: Help for the practicing programmer. Computer 11 (4), 34–41. http://dx.doi.org/10.1109/C-M.1978.218136.

Do, Hyunsook, Elbaum, Sebastian, Rothermel, Gregg, 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. 10 (4), 405–435.

Donghwan, Shin, Doo-Hwan, Bae, 2016. A theoretical framework for understanding mutation-based testing methods. In: Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on. IEEE, pp. 299–308. http://dx.doi.org/10.1109/ICST.2016.22.

Eric, Wong W., Ruizhi, Gao, Yihao, Li, Rui, Abreu, Wotawa, Franz, 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740. http://dx.doi.org/10.1109/TSE.2016.2521368.

Eric, Wong W., Vidroha, Debroy, Ruizhi, Gao, Yihao, Li, 2014. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308. http://dx.doi.org/10.1109/TR.2013.2285319.

Eric, Wong W., Yu, Qi, 2006. Effective program debugging based on execution slices and inter-block data dependency. J. Syst. Softw. 79 (7), 891–903. http://dx.doi.org/10.1016/j.jss.2005.06.045.

Fabian, Keller, Lars, Grunske, Simon, Heiden, Antonio, Filieri, Andre, van Hoorn, David, Lo, 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In: Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on. IEEE, pp. 114–125. http://dx.doi.org/10.1109/QRS.2017.22.

Heiden, Simon, Grunske, Lars, Kehrer, Timo, Keller, Fabian, Hoorn, Andre Van, Filieri, Antonio, Lo, David, 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. Softw. - Pract. Exp..

Hiralal, Agrawal, R., Horgan Joseph, Saul, London, Eric, Wong W., 1995. Fault localization using execution slices and dataflow tests. In: Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on. IEEE, pp. 143–151. http://dx.doi.org/10.1109/ISSRE.1995.497652.

Howden William, E., 1978. Theoretical and empirical studies of program testing. In: Proceedings of the 3rd International Conference on Software Engineering. IEEE Press, pp. 305–311.

Jean, Harrold Mary, Gregg, Rothermel, Kent, Sayre, Rui, Wu, Liu, Yi, 2000. An empirical investigation of the relationship between spectra differences and regression faults. Softw. Test. Verif. Reliab. 10 (3), 171–194, doi: 10.1002/.

Jean, Harrold Mary, Gregg, Rothermel, Rui, Wu, Liu, Yi, 1998. An empirical investigation of program spectra. In: Acm Sigplan Notices, Vol. 33. ACM, pp. 83–90. http://dx.doi.org/10.1145/277631.277647.

Jefferson, Offutt A., Untch, Roland H., 2001. Mutation 2000: Uniting the orthogonal. In: Mutation Testing for the New Century. pp. 34–44. http://dx.doi.org/10.1007/978-1-4757-5939-6_7.

Jie, Zhang, Wang, Ziyi, Zhang, Lingming, Dan, Hao, Lei, Zang, Cheng, Shiyang, Lu, Zhang, 2016. Predictive mutation testing. IEEE Trans. Softw. Eng. PP (99), 342–353.

Jingxuan, Tu, Xiaoyuan, Xie, Baowen, Xu, 2016. Code coverage-based failure proximity without test oracles. In: Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual, Vol. 1. IEEE, pp. 133–142. http://dx.doi.org/10.1109/COMPSAC.2016.81.

Jones, James A., Harrold, Mary Jean, 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: IEEE/ACM International Conference on Automated Software Engineering.

Jones James, A., Jean, Harrold Mary, John, Stasko, 2002. Visualization of test information to assist fault localization. In: Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on. IEEE, pp. 467–477. http://dx.doi.org/10.1145/581396.581397.

Just, René, 2014. The major mutation framework: Efficient and scalable mutation analysis for java. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, pp. 433–436.

Just, René, Jalali, Darioush, Ernst, Michael D., 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. ACM, pp. 437–440.

Lee, Naish, Jie, Lee Hua, Kotagiri, Ramamohanarao, 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 20 (3), 11. http://dx.doi.org/10.1145/2000791.2000795.

Maha, Kooli, Firas, Kaddachi, Giorgio, Di Natale, Alberto, Bosio, Pascal, Benoit, Lionel, Torres, 2017. Computing reliability: On the differences between software testing and software fault injection techniques. Microprocess. Microsyst. 50, 102–112. http://dx.doi.org/10.1016/j.micpro.2017.02.007.

Masri, Wes, 2010. Fault localization based on information flow coverage. Softw. Testing Verif. Reliab. 20 (2), 121–147.

Mateis, Cristinel, Stumptner, Markus, Wotawa, Franz, 2000. Locating bugs in java programs — First results of the java diagnosis experiments project. In: International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems: Intelligent Problem Solving: Methodologies & Approaches.

Mike, Papadakis, Yves, Le Traon, 2012. Using mutants to locate 'unknown' faults. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE, pp. 691–700. http://dx.doi.org/10.1109/ICST.2012.159.

Mike, Papadakis, 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. ACM, pp. 1293–1300. http://dx.doi.org/10.1145/2554850.2554978.

Mike, Papadakis, Yves, Le Traon, 2015. Metallaxis-FL: mutation-based fault localization. Softw. Test. Verif. Reliab. 25 (5–7), 605–628. http://dx.doi.org/10.1002/stvr.1509.

Nan, Li, Jeff, Offutt, 2017. Test oracle strategies for model-based testing. IEEE Trans. Softw. Eng. 43 (4), 372–395. http://dx.doi.org/10.1109/TSE.2016.2597136.

Nicholas, DiGiuseppe, Jones, James A., 2017. On the influence of multiple faults on coverage-based fault localization. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, pp. 210–220. http://dx.doi.org/10.1145/2001420.2001446.

Oliveira, Arantes Alessandro, Valdivino Alexandre, de Santiago, Lankalapalli, Vijaykumar Nandamudi, 2015. On proposing a test oracle generator based on static and dynamic source code analysis. In: Software Quality, Reliability and Security-Companion (QRS-C), 2015 IEEE International Conference on. IEEE, pp. 144–152. http://dx.doi.org/10.1109/QRS-C.2015.29.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 609–620. http://dx.doi.org/10.1109/ICSE.2017.62.

Renieres, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: IEEE International Conference on Automated Software Engineering. IEEE, pp. 30–39. http://dx.doi.org/10.1109/ASE.2003.1240292.

Rui, Abreu, Peter, Zoeteweij, J.C., Van Gemund Arjan, 2006. An evaluation of similarity coefficients for software fault localization. In: Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on. IEEE, pp. 39–46. http://dx.doi.org/10.1109/PRDC.2006.18.

Seokhyeon, Moon, Yunho, Kim, Moonzoo, Kim, Shin, Yoo, 2014. Hybrid-MUSE: mutating faulty programs for precise fault localization. Technical report, Citeseer.

Shin, Hong, Taehoon, Kwak, Byeongcheol, Lee, Yiru, Jeon, Bongseok, Ko, Yunho, Kim, Moonzoo, Kim, 2017. MUSEUM: Debugging real-world multilingual programs using mutation analysis. Inf. Softw. Technol. 82, 80–95. http://dx.doi.org/10.1016/j.infsof.2016.10.002.

Shu, Ting, Ye, Tiantian, Ding, Zuohua, Xia, Jinsong, 2016. Fault localization based on statement frequency. Inform. Sci. 360 (C), 43–56.

Taha, A.B., Thebaut, S.M., Liu, S.S., 1989. An approach to software fault localization and revalidation based on incremental data flow analysis. In: International Computer Software & Applications Conference.

Thomas, Reps, Thomas, Ball, Manuvir, Das, James, Larus, 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In: Software Engineering–Esec/Fse'97. pp. 432–449. http://dx.doi.org/10.1007/3-540-63531-9_29.

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

Vidroha, Debroy, Wong, W.E.ric, 2014. Combining mutation and fault localization for automated program debugging. J. Syst. Softw. 90 (1), 45–60. http://dx.doi.org/10.1016/j.jss.2013.10.042.

Wang, Tao, Roychoudhury, Abhik, 2005. Automated path generation for software fault localization. In: IEEE/ACM International Conference on Automated Software Engineering.

Wes, Masri, 2010. Fault localization based on information flow coverage. Softw. Testing Verif. Reliab. 20 (2), 121–147. http://dx.doi.org/10.1002/stvr.409.

Wes, Masri, Abou, Assi Rawad, 2014. Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Trans. Softw. Eng. Methodol. (TOSEM) 23 (1), 8. http://dx.doi.org/10.1145/2559932.

Wes, Masri, Rawad, Abou-Assi, Marwa, El-Ghali, Nour, Al-Fatairi, 2009. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In: Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009). ACM, pp. 1–5. http://dx.doi.org/10.1145/1555860.1555862.

Weyuker Elaine, J., 1982. On testing non-testable programs. Comput. J. 25 (4), 465–470. http://dx.doi.org/10.1093/comjnl/25.4.465.

Wong, W. Eric, Shi, Yan, Qi, Yu, Golden, Richard, 2008. Using an RBF neural network to locate program bugs. In: International Symposium on Software Reliability Engineering.

Xiao-Yi, Zhang, Zheng, Zheng, Kai-Yuan, Cai, 2018. Exploring the usefulness of unlabelled test cases in software fault localization. J. Syst. Softw. 136, 278–290. http://dx.doi.org/10.1016/j.jss.2017.07.027.

Yang, Qian, Li, J.J.enny, Weiss, David M., 2009. A survey of coverage-based testing tools. Comput. J. 52 (5), 589–597.

Yihan, Li, Chao, Liu, 2012. Using cluster analysis to identify coincidental correctness in fault localization. In: 2012 Fourth International Conference on Computational and Information Sciences. IEEE, pp. 357–360. http://dx.doi.org/10.1109/ICCIS.2012.361.

Yong, Liu, Zheng, Li, Linxin, Wang, Zhiwen, Hu, Ruilian, Zhao, 2017. Statement-oriented mutant reduction strategy for mutation based fault localization. In: Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on. IEEE, pp. 126–137. http://dx.doi.org/10.1109/QRS.2017.23.

Yong, Liu, Zheng, Li, Ruilian, Zhao, Pei, Gong, 2018. An optimal mutation execution strategy for cost reduction of mutation-based fault localization. Inform. Sci. 422, 572–596. http://dx.doi.org/10.1016/j.ins.2017.09.006.

Zhang, Xiangyu, He, Haifeng, Gupta, Neelam, Gupta, Rajiv, 2005. Experimental evaluation of using dynamic slices for fault location. In: International Symposium on Automated Analysis-Driven Debugging.

Zhang, Lingming, Marinov, Darko, Khurshid, Sarfraz, 2013. Faster mutation testing inspired by test prioritization and reduction. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, pp. 235–245.

Zheng, Li, Meiying, Li, Yong, Liu, Jingyao, Geng, 2016. Identify coincidental correct test cases based on fuzzy classification. In: Software Analysis, Testing and Evolution (SATE), International Conference on. IEEE, pp. 72–77. http://dx.doi.org/10.1109/SATE.2016.19.

Zheng Li is a profess or at BUCT. He obtained his Ph.D. from King's College London, CREST center in 2009 under the supervision of Mark Harman. He has worked as a research associate at King's College London and University College London. He has worked on program testing, source code analysis and manipulation. More recently he is interested in search-based software engineering and slicing state-based model.

Haifeng Wang is a first-year Ph.D. student at BUCT. He has worked on program testing and source code analysis. His research interests are software testing, fault localization and software defect prediction.

Yong Liu is an assistant professor at Beijing University of Chemical Technology (BUCT). He obtained his Ph.D. from BUCT in 2018 under the supervision of Professor Zheng Li. His research interests include software testing, fault localization, fault understanding, and mutation testing.