FATOC: Bug Isolation-based Multi-fault Localization by using OPTICS Clustering

Abstract Bug isolation is a popular approach for multi-fault localization (MFL), where all failed test cases are clustered into several groups, then the failed test cases in each group combined with all passed test cases are used to localize only a single-fault. However, existing clustering algorithms can not always obtain completely correct clustering results, which is a potential threat for bug isolation-based MFL approaches. To address this issue, we first analyze the influence of the accuracy of the clustering on the performance of MFL, and the results of a controlled study indicate that using the clustering algorithm with the highest accuracy can achieve the best performance of MFL. Moreover, previous studies on clustering algorithms also show the clusters with higher density mean the elements in the same cluster have a higher similarity. Based on the above motivation, we propose a novel approach FATOC (one-fault-at-a-time via OPTICS clustering). In particular, FATOC first leverages the OPTICS clustering algorithm to group failed test cases, and then identifies a cluster with the highest density. OPTICS clustering is a density-based clustering algorithm, which can reduce the misgrouping and calculate a density value for each cluster. Such a density value of each cluster is helpful for finding a cluster with the highest clustering effectiveness. FATOC then combines the failed test cases in this cluster with all passed test cases to localize a single-fault through the traditional spectrum-based fault localization (SBFL) formula. After this fault is localized and fixed, FATOC will use the same method to localize the next single-fault, until all the test cases are passed. Our evaluation results show that FATOC can significantly outperform the traditional SBFL technique and state-of-the-art MFL approach MSeer on 804 multi-faulty versions from nine real-world programs. Specifically, FATOC's performance is 10.32% higher than traditional SBFL when using *Ochiai* formula in terms of *A-EXAM* metric. Besides, the results also indicate that, when checking 1%, 3% and 5% statements of all subject programs, FATOC can locate 36.91%, 48.50% and 66.93% of all faults respectively, which is also better than traditional SBFL and MFL approach MSeer.

Keywords bug isolation, multiple-fault localization, OPTICS clustering, empirical study

1 Introduction

Software is used in all areas of society. The probability of faults in the software is higher due to the increasing size and complexity of the software. This could cause wrong behaviors or even crash when running software. Fault localization aims to find the faulty code in software. It is a critical but time-consuming task during software debugging activities [1]. To reduce the human and time cost of finding faults, many semi-automated fault localization approaches have been proposed in recent years [2].

Spectrum-based fault localization (SBFL) is one of the most studied fault localization techniques. It records the execution results and coverage spectrum of test cases and then uses this information to calculate the suspiciousness value for each code element (such as statement, branch). The main assumption of SBFL is that the code elements covered by more failed test cases but fewer passed test cases are more likely to be faulty [2]. Due to its lightweight and effectiveness, extensive studies have been conducted on SBFL in past years [3–7].

Most of SBFL studies mainly assume that the programs under test only contain a single fault. Traditional SBFL approaches have shown their fault localization accuracy on single-fault programs. However, such an assumption may not hold in reality, and the interference between different faults may reduce the accuracy of traditional SBFL approaches [8, 9]. Therefore, the fault localization accuracy on multi-fault programs needs to

Regular Paper

be further improved.

It is not hard to find that multi-fault localization (MFL) is a more challenging problem. Recently several approaches have been proposed to solve this problem [10–13], where bug isolation is a popular MFL method [10, 13]. It aims to group the failed test cases that execute the same bug into one cluster, and then different bugs can be localized by failed test cases in different clusters. The critical progress of bug isolation is to cluster failed test cases. Clustering algorithms have been widely [14, 15] utilized based on test case behavioral characteristics, since previous studies [16–18] have shown that test cases that cover the same bug may have similar behavioral characteristics (i.e., similar coverage spectrum information and execution results).

Using bug isolation can improve the performance of MFL since it can alleviate the issue that localizing a specific fault by some failed test cases which do not cover this fault. However, during the clustering process, it is common to cluster some failed test cases wrongly. For example, the failed test cases in one group may not cover the same fault, or there may exist some failed test cases which can cover the same fault but are not clustered into the corresponding group.

There are various performance metrics (such as $precision$, FPR, $recall$) to evaluate the clustering effect. We first analyze the relationship between bug isolation-based MFL and the clustering accuracy to guide the potential optimization of clustering algorithms. In this paper, we conduct a controlled empirical study on 12,786 multi-fault program versions. The results indicate that the better the accuracy of the clustering algorithm, the better the performance of MFL. In particular, among three performance metrics ($precision$, $recall$, and FPR), the correlation between $precision$ metric and the performance of MFL is the highest.

The controlled study results show that we need to select the cluster with the highest quality for MFL. Previous studies [19, 20] show that the density of clusters can be used to evaluate the clusters, and the higher the density, the better the cluster quality. Therefore, in the clustering process, using the cluster with the highest density for fault localization can achieve the best performance of MFL.

Current MFL approaches [10, 21] mainly use classical clustering algorithms (such as k-means and k-medoids). These approaches can calculate the density of clusters. However, they require the number of faults in the program under test before clustering, which is unknown during practical software testing. There are also hierarchical clustering-based methods [13], but these methods cannot obtain the density of various clusters during the clustering process. Therefore, to keep the advantages of these approaches while avoiding the disadvantages, we propose FATOC (one-fault-at-a-time via OPTICS clustering) approach for MFL, because this cluster algorithm can obtain the density of various clusters during clustering [22], and it does not need to know the number of faults in advance.

We evaluate the effectiveness of FATOC on 804 multi-fault versions from 9 real-world programs, where each program contains 2 to 10 faults. Previous studies have proposed many bug isolation approaches to solve MFL problem. For example, Gao et al. [10] proposed MSeer by using k-medoids algorithm. Jones et al. [13] proposed a hierarchical clustering-based approach. Among them, MSeer can perform significantly better than the methods proposed by Jones et al. [13]. Therefore, we choose MSeer as the baseline in our study. The difference between FATOC and MSeer is that FATOC selects the cluster with the highest density for fault localization in each iteration, while MSeer localizes multiple faults in parallel through multiple clusters.

We also use the traditional SBFL technique as the baseline to show the effectiveness of MFL approaches. The results show that FATOC performs better than existing baseline methods. More specifically, in terms of *A-EXAM* metric, FATOC can improve the accuracy of MFL by 8.8% and 1.33% when compared with traditional SBFL and MSeer respectively.

To our best knowledge, the contributions of our study can be summarized as follows:

1. We are the first to conduct a controlled empirical study to analyze the relationship between the performance of MFL and the accuracy of the clustering. Our study utilize 12,786 multi-fault program versions. After simulating misgrouping scenarios with different clustering accuracy, we find that the clustering algorithm's *precision* metric has a higher correlation with the fault localization accuracy of MFL.

2. We propose FATOC approach by using OPTICS clustering. This approach selects the failed test cases in a cluster with the highest density for each iteration to localize a single-fault. When all faults are located and fixed, which means all the test cases are passed, we terminate FATOC approach.

3. To evaluate the effectiveness of FATOC, we conduct an empirical study on 804 multi-fault program versions. Then we choose traditional SBFL approach and state-of-the-art MFL approach MSeer [10] as our baselines. The results show that FATOC can achieve better fault localization accuracy than other two baselines.

In this paper, we extend our previous study [23] in the following aspects: (1) We propose a novel MFL approach FATOC. FATOC leverages the results of the controlled study on the relationship between MFL performance and clustering accuracy. (2) We conduct a large-scale experimental study to verify the effectiveness of our proposed MFL approach FATOC. In our empirical study, we utilize 804 multi-fault program versions, consider two kinds of metrics (*precision*, *recall*, and *FPR* for clustering, and *A-EXAM* and *TOP-N%* for fault localization), and choose two baselines (traditional SBFL approach and MFL approach MSeer). We design three research questions and discuss the results for these research questions. To facilitate other researchers to replicate our study, we share all source code and data set used in our study in the GitHub repository [1].

The rest of this paper is structured as follows. Section 2 presents the background and related work of this paper. Section 3 lists the controlled empirical study about the correlation between MFL and clustering accuracy. Section 4 presents our proposed approach FATOC in details. Section 5 discusses the experimental design. Section 6 analysis the experimental results and discusses threats to validity. Section 7 concludes this paper and presents the future work.

2 Background and Related Work

In this section, we present the background and related work of our study. In particular, we first introduce the background of spectrum-based fault localization, then discuss the challenge and related work of multi-fault localization.

2.1 Spectrum-based Fault Localization

Spectrum-based fault localization (SBFL) is a popular fault localization technology [10, 13, 24–27]. As shown in Fig. 1, SBFL will first collect test cases' coverage information and execution results, and then it

[1]https://github.com/appmlk/FATOC.git

uses a suspiciousness formula to calculate the probability of each program entity containing faults. Finally, it sorts all the program entities in descending order according to their suspiciousness values to generate a sorted list that can be used to guide developers in locating real faults. To calculate the suspiciousness value of each program entity, several suspiciousness formulas were proposed in many papers [2]. The main idea of these formulas is based on the assumption that the program entities covered by more failed test cases are more likely to contain faults than the entities covered by more passed test cases [28].

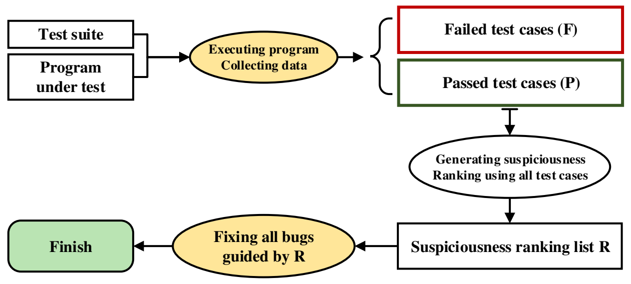

Test suite
Program under test
Executing program Collecting data
Failed test cases (F)
Passed test cases (P)
Generating suspiciousness Ranking using all test cases
Suspiciousness ranking list R
Fixing all bugs guided by R
Finish

Fig.1. Framework of traditional SBFL

In addition to the researches about suspiciousness formulas [29, 30], many works also try to improve SBFL from concerning the coverage information constructed from different program entities, such as statements [2], call sequences [31], du-pairs [32], statement frequency [33] and so on. Statement is the most-studied program entity in SBFL related works, and we also use test cases' statement coverage information to implement our technique. Besides, among the different formulas, this paper considers six the best-studied ones of them, which include $Jaccard$ [34], $Tarantula$ [35], $Ochiai$ [36], $OP2$ [30], $CrossTab$ [37] and $Dstar^*$ [38]. Table 1 shows the detailed information of these six formulas. For these formulas, $fail(s)$ is the number of failed test cases which cover the statement s, $pass(s)$ means the number of passed test cases which cover s. $totalfail$ and $totalpass$ denote the total number of failed test cases and passed test cases respectively, and $totalcase$ indicates the total number of test cases. Finally, $\mathcal{S}us(s)$ is the possibility of statement s being faulty.

Table 1. Suspiciousness formulas

Name	Formula
$Jaccard$ [34]	$\mathcal{S}us(s) = \frac{fail(s)}{totalfail+pass(s)}$
$Tarantula$ [35]	$\mathcal{S}us(s) = \frac{\frac{fail(s)}{totalfail}}{\frac{fail(s)}{totalfail}+\frac{pass(s)}{totalpass}}$
$Ochiai$ [36]	$\mathcal{S}us(s) = \frac{fail(s)}{\sqrt{totalfail \times (fail(s)+pass(s))}}$
$OP2$ [30]	$\mathcal{S}us(s) = fail(s) - \frac{pass(s)}{totalpass+1}$
$Crosstab$ [37]	$\mathcal{S}us(s) = \begin{cases} \chi(s)^2 & if \varphi(s) > 1 \\ 0 & if \varphi(s) = 1 \\ -\chi(s)^2 & if \varphi(s) < 1 \end{cases}$ $\varphi(s) = \frac{\frac{fail(s)}{totalfail}}{\frac{pass(s)}{totalpass}}$ $\chi(s)^2 = \frac{(fail(s)-E_{cf}(s))^2}{E_{cf}(s)} + \frac{(pass(s)-E_{cs}(s))^2}{E_{cs}(s)} + \frac{(totalfail-fail(s)-E_{uf}(s))^2}{E_{uf}(s)} + \frac{(totalpass-pass(s)-E_{us}(s))^2}{E_{us}(s)}$ $E_{cf}(s) = \frac{(fail(s)+pass(s)) \times totalfail}{totalcase}$ $E_{cs}(s) = \frac{(fail(s)+pass(s)) \times totalpass}{totalcase}$ $E_{uf}(s) = \frac{(totalcase-fail(s)-pass(s)) \times totalfail}{totalcase}$ $E_{us}(s) = \frac{(totalcase-fail(s)-pass(s)) \times totalpass}{totalcase}$
$Dstar^*$ [38]	$\mathcal{S}us(s) = \frac{fail(s)^*}{pass(s)+(totalfail-fail(s))}$

Previous studies have achieved satisfactory fault localization accuracy on single-fault program versions. Jones et al. [29] improved SBFL with Tarantula, which can achieve promising results in single-fault localization. Their empirical studies show that for 87% versions of Siemens benchmark, the developers only need to examine less than 10% statements to localize faults. Feyzi et al. [39] found that combining SBFL with static analysis could improve fault localization accuracy no matter the programs implemented by C or Java. Liu et al. [7] proposed three manipulation strategies to reduce the negative impact of coincidental correct (CC) test cases in fault localization. Zakari et al. [40] proposed a fault localization technique based on complex network theory named FLCN-S to improve localization effectiveness on single-fault subject programs.

2.2 Multi-Fault Localization

Unlike fault localization on single-fault programs, fault localization on multi-fault programs (i.e., multi-fault localization) is more challenging. Most previous SBFL studies assume that there is only one fault in the program [2, 11]. The key assumption of traditional SBFL approaches is that if a statement is executed by more failed test cases and less passed test cases, it has more chance to be a faulty statement [28]. On the other hand, if a statement is executed by more passed test cases and less failed test cases, it has less suspiciousness to be faulty.

However, the basic principles of traditional SBFL are not applicable in multi-fault programs. Because in multi-fault programs, the correct statement may be covered by multiple failed test cases caused by different faults, resulting in the correct statement being covered by more failed test cases than other faulty statements. Thus this correct statement will have a high suspiciousness to be faulty, which causes the problem that the fault localization performance of using traditional SBFL on multi-fault programs is poor [41].

To improve the MFL performance, many researchers propose different methods from various aspects. Inspired by the practical software debugging process, the developers are usually aware of faults in the program but cannot estimate the exact number of faults. Jones et al. [13] proposed a sequential debugging technique that uses all failed test cases to locate and fix one-fault-at-a-time. From another different aspect, researchers employ a genetic algorithm and neural network model to improve MFL accuracy. Such methods encode each program statement as a chromosome to indicate whether it contains fault [11, 42]. Moreover, there are studies which try to use clustering algorithms and divide the failed test cases into multiple groups, and each group is used to locate one-single fault [10, 12, 21]. Such methods are called as bug isolation-based MFL technique, and empirical results showed that these methods could achieve better performance than other approaches. For example, Zakari et al. [43] proposed to use a network community clustering algorithm to isolate faults to individual communities, with each community targeting one fault. These fault-focused communities are provided for developers to debug faults in parallel. Their experimental results show that the network community grouping algorithm can effectively isolate faults and improve the efficiency of multi-fault localization.

Among a number of MFL approaches, the state-of-the-art approach is called as MSeer [10], which uses a k-medoids-based clustering algorithm to perform bug isolation, where failed test cases will be grouped into different clusters. After that, MSeer adopts SBFL formulas to perform fault localization. MSeer aims to covert MFL problem into multiple single-fault localization problems, and uses a parallel debugging method to locate and fix all faults at one time. Fig. 2 shows the framework of bug isolation and parallel debugging process.

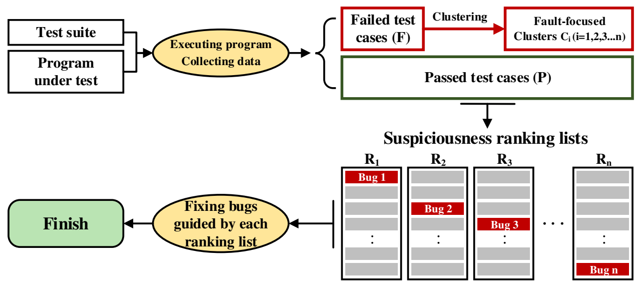

Test suite
Program under test
Executing program Collecting data
Failed test cases (F)
Clustering
Fault-focused Clusters C_i (i=1,2,3...n)
Passed test cases (P)
Suspiciousness ranking lists
R1
R2
R3
Rn
Bug 1
Bug 2
Bug 3
Bug n
Fixing bugs guided by each ranking list
Finish

Fig.2. Framework of bug isolation and parallel debugging process

The goal of bug isolation is to analyze the correspondence relationship between failed test cases and faults. It attempts to generate fault-focused clusters by isolating failed test cases caused by the same fault into the same cluster [10]. In other words, the failed test cases in one cluster are related to the same fault, whereas failed test cases in different clusters are related to different faults.

After clustering, several fault-focused suspiciousness rankings are generated by using failed test cases of a given cluster and passed test cases. Examining code according to those ranking lists can help the developers to localize the corresponding causative fault linked to each ranking list. Finally, all faults will be located and fixed in one iteration.

Zakari et al. [44] conducted an investigative study on the usefulness of the problematic parallel debugging approach, which uses k-means clustering algorithm with the Euclidean distance metric on three similarity coefficient-based fault localization techniques. They compared the effectiveness of their approach with OBA and MSeer. Their results suggest that clustering failed test cases based on their execution profile similarity and the utilization of distance metrics is indeed problematic and contributes to the reduction of effectiveness in localizing multiple faults.

The key step of the bug isolation-based MFL approach is clustering, whose performance will affect the final MFL accuracy, but how much of such influence remains unknown. In this paper, we first experimentally analyze the correlation between bug isolation performance and MFL accuracy, and then use the empirical findings to guide us to propose a novel MFL approach with better fault localization accuracy.

3 Correlation Analysis between the Performance of Bug Isolation-based MFL and the Clustering Accuracy

In this section, we conduct a controlled empirical study on the correlation between the performance of bug isolation-based fault localization and the clustering accuracy. We first introduce the preliminary for bug isolation. Then we describe three simulated misgroup scenarios. Finally, we introduce the experimental design and discuss the empirical findings.

3.1 Preliminary of Bug Isolation

As discussed in Section 2.2, The popular approach of MFL is to cluster the failed test cases before applying SBFL techniques [10, 12, 21], which is named *bug isolation*. The motivation of this approach is that the failed test cases should be grouped into different clusters, and the failed test cases in each cluster and all passed test cases are combined to localize only one fault. Ideally, test cases that failed by the same fault should be grouped into the same cluster, then MFL problem can be transformed into the single-fault localization problem.

Table 2 shows an illustrative example of a program segment. This program contains two faults, where S_4 and S_6 are two faults. Black dots in Table 2 indicate that the corresponding test case T_j covers the statement S_i. The test suite has 10 test cases, of which the coverage information and execution results are shown in Table 2. T_1 and T_2 are two passed test cases, $T_3 \sim T_{10}$ are eight failed test cases, in which $T_3 \sim T_6$ and $T_7 \sim T_{10}$

Table 2. The statement coverage and test case execution results of an example program With two faults

Program		Test Suite									
		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
fun(int a, int b){		-10, -10	-1, -1	0, -1	1, -1	2, -1	3, -1	-1, 11	-1, 12	-1, 13	-1, 14
S_1	Read (a,b);	●	●	●	●	●	●	●	●	●	●
S_2	**int** i=0, j=0;	●	●	●	●	●	●	●	●	●	●
S_3	**if** (a⩾0)	●	●	●	●	●	●	●	●	●	●
S_4	i=i+1; **//Correct:i=i+a/10**			●	●	●	●				
S_5	**if** (b⩾0)	●	●	●	●	●	●	●	●	●	●
S_6	j=j+b/10; **//Correct:j=j+b/20**							●	●	●	●
S_7	**if** (i>0 || j>0)	●	●	●	●	●	●	●	●	●	●
S_8	printf (“Positive”);			●	●	●	●	●	●	●	●
S_9	**else** printf (“Negative”);}	●	●								
Execution Results(0=Passed/1=Failed)		0	0	1	1	1	1	1	1	1	1

Table 3. Suspiciousness value and rank of statements using all the test cases for the simple example in TABLE 2 when considering different suspicious formulas

Statement	fail(s)	totalfail	pass(s)	totalpass	*Ochiai*		*Tarantula*		*OP2*		$Dstar^3$	
					S(s)	Rank	S(s)	Rank	S(s)	Rank	S(s)	Rank
S_1	8	8	2	2	0.89	2	0.5	3	7.33	2	256	2
S_2	8	8	2	2	0.89	2	0.5	3	7.33	2	256	2
S_3	8	8	2	2	0.89	2	0.5	3	7.33	2	256	2
S_4(fault)	4	8	1	2	0.63	**8**	0.5	**3**	3.66	**8**	12.8	**8**
S_5	8	8	2	2	0.89	2	0.5	3	7.33	2	256	2
S_6(fault)	4	8	0	2	0.71	**7**	1	**1**	4	**7**	16	**7**
S_7	8	8	2	2	0.89	2	0.5	3	7.33	2	256	2
S_8	8	8	0	2	1	1	1	1	8	1	∞	1
S_9	0	8	2	2	0	9	0	9	-0.66	9	0	9

execute the faults S_4 and S_6, respectively.

In terms of fault localization accuracy, Table 3 shows considering four suspiciousness formulas except for *Tarantula* achieves a low fault localization accuracy since the faults (S_4 and S_6) rank eighth and seventh respectively. Though this simple example program only contains nine statements, the fault localization accuracy is not satisfactory when considering most of the popular suspiciousness formulas.

Since test cases $T_3 \sim T_6$ and $T_7 \sim T_{10}$ execute the faults S_4 and S_6 respectively, after bug isolation, $T_3 \sim T_6$ and $T_7 \sim T_{10}$ can be divided into two clusters ideally. Then the failed test cases in two clusters with the passed test cases are used to locate the faults S_4 and S_6 respectively. Take fault S_6 as an example, compared with traditional SBFL, using bug isolation can change the rank of S_6 from (7,1,7,7) to (1,1,1,1) in the ranking list with *Ochiai*, *Tarantula*, *OP*2 and $Dstar^3$ formulas respectively. Such an example denotes that the fault localization accuracy can be improved after using bug isolation.

The above illustrative example shows that bug isolation is an effective method for MFL, and clustering-based algorithms have been widely used for bug isolation in recent studies. For example, Huang et al. [21] analyzed the influence of k-means and hierarchical clustering algorithms on bug isolation. Liu et al. [12] employed decision-tree based algorithms to cluster failed test cases. Gao et al. [10] introduced a k-medoids based clustering algorithm to do bug isolation. However, to the best of our knowledge, there is no research about analyzing the relationship between fault localization accuracy and bug isolation accuracy.

3.2 Three Misgroup Scenarios

Given a fault f and the corresponding cluster $cluster$, we use $failnum(f)$ to denote the number of the failed test cases that cover the fault f, use $cluster(f)$ to denote the number of the failed test cases that cover the fault f in the $cluster$, and use $cluster(other)$ to denote the number of the failed test cases that cover other faults in the cluster $cluster$. In ideal condition,

$cluster(f) = failnum(f)$ and $cluster(other) = 0$.

In our study, we identify three kinds of misgroup scenarios. Fig. 3 shows the clustering results of the ideal situation and three misgroup scenarios. In each sub-figure of Fig. 3, the hollow circle means the cluster related to fault f, the red bullet indicates the failed test case that does not cover f and the green bullet denotes the failed test case that covers f. More detailed analysis of these three misgroup scenarios is given as follows:

Misgroup Scenario 1. All failed test cases which cover f are clustered correctly into one group, but some other failed test cases which do not cover f are also clustered into the same group. In this scenario, $cluster(f) = failnum(f)$, but $cluster(other)>0$.

Misgroup Scenario 2. All failed test cases which are clustered into the corresponding group do cover f, and there exist some failed test cases which cover f but are not clustered into this group. In this scenario, $cluster(f) < failnum(f)$, $cluster(other) = 0$.

Misgroup Scenario 3. Some failed test cases cover f but are not clustered into the corresponding group. In addition, some failed test cases are clustered into the corresponding group but not cover f. In this scenario, $cluster(f) < failnum(f)$, $cluster(other)>0$.

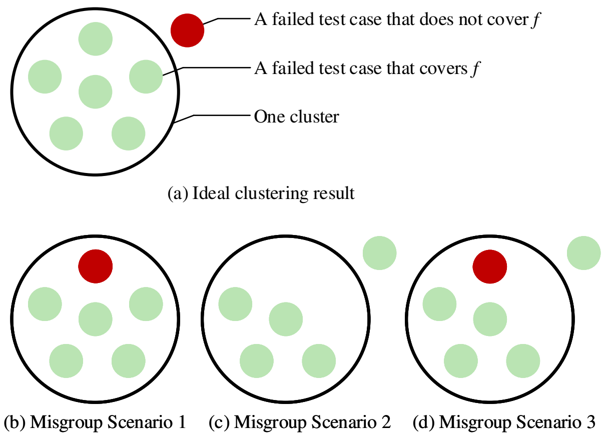

A failed test case that does not cover f
A failed test case that covers f
One cluster
(a) Ideal clustering result
(b) Misgroup Scenario 1
(c) Misgroup Scenario 2
(d) Misgroup Scenario 3

Fig.3. Diagrams of ideal situation and three possible misgroup scenarios.

In this paper, we will conduct a controlled study to simulate the above three misgroup scenarios.

3.3 Evaluation Metrics for Misgroup Issue

The target of bug isolation strategy is to cluster failed test cases that cover a single faulty statement f into the same group. Then, considering one single faulty statement f, the bug isolation problem can be modeled as a binary classification problem, where each failed test case in the cluster can be classified into two types: cover f or not cover f. The commonly used metrics to evaluate the performance of binary classification algorithms include FPR, *precision* and *recall* [7, 45]. These metrics are calculated based on the confusion matrix, and a higher *precision* and *recall* value and a lower FPR value indicates the corresponding technique is better. The confusion matrix and evaluation metrics are shown in Fig. 4.

		Covers t or not	
		Yes	No
Clustered into the correct group or not	Yes	True Positives (TP)	False Positives (FP)
	No	False Negatives (FN)	True Negatives (TN)

$$FPR = \frac{FP}{FP+TN}$$

$$precision = \frac{TP}{TP+FP}$$

$$recall = \frac{TP}{TP+FN}$$

Fig.4. Confusion matrix and evaluation metrics for bug isolation.

For the bug isolation problem, given the cluster related to the fault f, true positives (TP) means the number of failed test cases within this cluster, which cover f; false positives (FP) means the number of failed test cases within this cluster, which do not cover f; false negatives (FN) means the number of failed test cases outside of this cluster, which cover f; and True Negatives (TN) means the number of failed test cases outside of this cluster, which do not cover f.

Based on the definition of misgroup scenario and confusion matrix, we can give the value range of different evaluation metrics for different scenarios.

Ideal Scenario: $FPR = 0$, $precision = 1$, $recall = 1$.

Misgroup Scenario 1: $FPR \in (0,1)$, $precision \in (0,1)$, $recall = 1$.

Misgroup Scenario 2: FPR=0, $precision = 1$, $recall \in (0,1)$.

Misgroup Scenario 3: $FPR \in (0,1)$, $precision \in (0,1)$, $recall \in (0,1)$.

3.4 Experimental Setup

In this Section, we use 11 widely-studied real-world subject programs from SIR [46] (i.e, print tokens, print tokens2, tot info, schedule, schedule2, tcas, replace, gzip, grep, sed and space). We apply 12,786 faulty versions with 77,970 faults in our controlled empirical study, where the number of faults in each faulty version varies from 1 to 10. It should be noted that the multi-fault program versions are generated from multiple single-fault programs. We use the single-fault programs provided in SIR and we also manually inject artificial faults into these programs to obtain a large number of program versions.

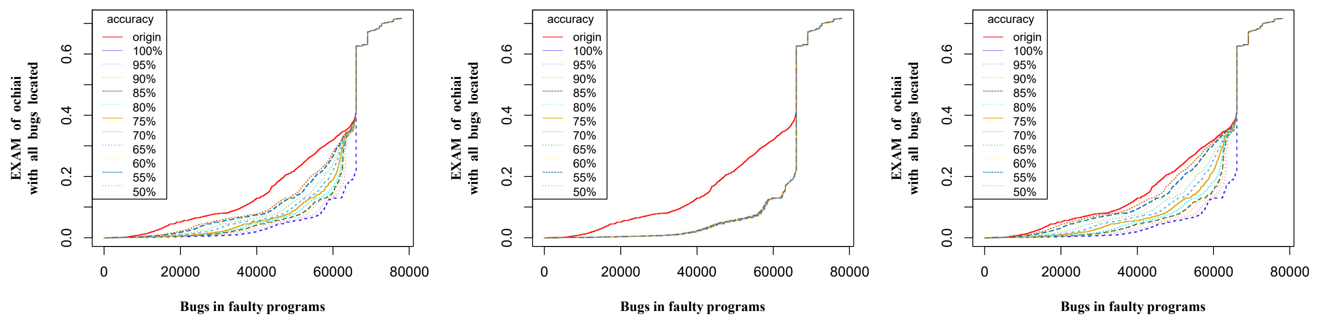
Since clustering algorithms are the main methods for bug isolation, the clustering accuracy reflects the bug isolation accuracy. To simulate different levels of clustering accuracy, in this section, we mainly design a controlled experiment to generate a variety of clusters with 10 misgrouping levels, where 5% to 50% failed test cases are clustered incorrectly in the corresponding group. The fewer failed test cases that are clustered incorrectly, the higher the accuracy of the clustering. For instance, 100% bug isolation accuracy refers to the ideal clustering situation, that is, $FPR = 0$, $precision = 1$ and $recall = 1$. In the misgroup scenario 2, 95% bug isolation accuracy refers to $FPR = 0$, $precision = 1$ and $recall = 95\%$.

3.5 Findings

To analyze the correlation between bug isolation accuracy and fault localization accuracy, we use a $EXAM$ metric to evaluate the fault localization performance of different situations. $EXAM$ is a widely used fault localization metric, and the detailed introduction of $EXAM$ can be found in Section 5.3. Fig. 5 shows the $EXAM$ value of the $Ochiai$ formula in three misgroup scenarios, where Fig. 5(a), Fig. 5(b) and Fig. 5(c) refer to the misgroup 1 scenario, the misgroup 2 scenario and the misgroup 3 scenario respectively, where origin refers to not using bug isolation strategy. In these figures, x-axis represents 77,970 faults of 12,786 programs under test, and y-axis represents the $EXAM$ value of locating the corresponding fault. Note that for convenience of comparison, we arrange $EXAM$ in ascending order, and a lower $EXAM$ value means better effectiveness of fault localization.

Table 4 shows the average $EXAM$ of finding the first faulty statement in 12,786 multi-fault versions under different bug isolation accuracy. For example, in the ideal case (i.e., 100% bug isolation accuracy), the average $EXAM$ required to find the first faulty statement in all programs when using the $OP2$ formula is 1.59%. In this table, we can find the $EXAM$ of $Ochiai$ can achieve the lowest in 5 cases, followed by $Tarantula$ (3 cases) and $Dstar^3$ (2 cases).

As shown in Fig. 5, in either scenario, the fault localization accuracy will decrease when bug isolation accuracy decreases. However, the correlation between



(a) (b) (c)

Fig.5. Cost of three misgroup scenarios: (a) misgroup scenario 1, (b) misgroup scenario 2, (c) misgroup scenario 3

Table 4. *EXAM* value when localizing the first fault in the faulty program versions

Formula	Bug isolation accuracy										
	100%	95%	90%	85%	80%	75%	70%	65%	60%	55%	50%
Jaccard	1.88%	1.98%	2.08%	2.17%	2.25%	2.37%	2.52%	2.78%	3.10%	3.35%	3.66%
Tarantula	2.40%	2.59%	2.63%	2.66%	2.70%	2.75%	2.79%	2.83%	2.87%	2.92%	3.00%
Ochiai	1.81%	1.91%	2.02%	2.11%	2.20%	2.33%	2.51%	2.83%	3.09%	3.31%	3.59%
*OP*2	1.59%	6.32%	6.66%	6.85%	7.04%	7.21%	7.33%	7.53%	8.02%	8.53%	8.91%
CrossTab	2.36%	2.52%	2.74%	2.89%	3.02%	3.13%	3.24%	3.55%	3.72%	4.06%	4.33%
Dstar^3	1.75%	1.88%	2.02%	2.14%	2.35%	2.71%	3.03%	3.29%	3.63%	4.35%	5.05%

fault localization accuracy and bug isolation accuracy in the misgroup scenario 2 is lower than that of the other two misgroup scenarios. This finding indicates that the quality of the clustering results is highly correlated with the accuracy of MFL. Specifically, the better the quality of the clusters, the better the efficiency of MFL. In particular, among the three indicators of *precision*, *recall*, and *FPR*, the correlation between the *precision* indicator and the localization accuracy is the strongest.

Based on the controlled study, we find that we need to select the cluster with the highest quality for MFL. Existing research on clustering algorithms shows that the density of the elements in a cluster is an important indicator for measuring the quality of clusters, and high density indicates a high-quality cluster [19, 20].

The current MFL approaches include classical clustering algorithms such as k-means and k-medoids [10, 21]. These methods can compute the density of clusters, but they require the number of faults in the program before clustering, which is unknowable during actual software testing. There are also methods based on hierarchical clustering [13], but it cannot obtain the density of various clusters during the clustering process.

To overcome the shortcomings of the previous bug isolation-based MFL studies, we propose an approach based on OPTICS clustering, because this algorithm can obtain the density of various clusters during clustering, and it does not need to input the number of faults.

4 Our Multi-fault Localization Approach FATOC

In this section, we first show the framework of our proposed approach FATOC. Then we show the details of the important steps in our framework.

4.1 Framework of FATOC

Fig. 6 shows the framework of our proposed FATOC approach. The details of each step in the framework are introduced as follows:

Step 1. Program execution and data collection:

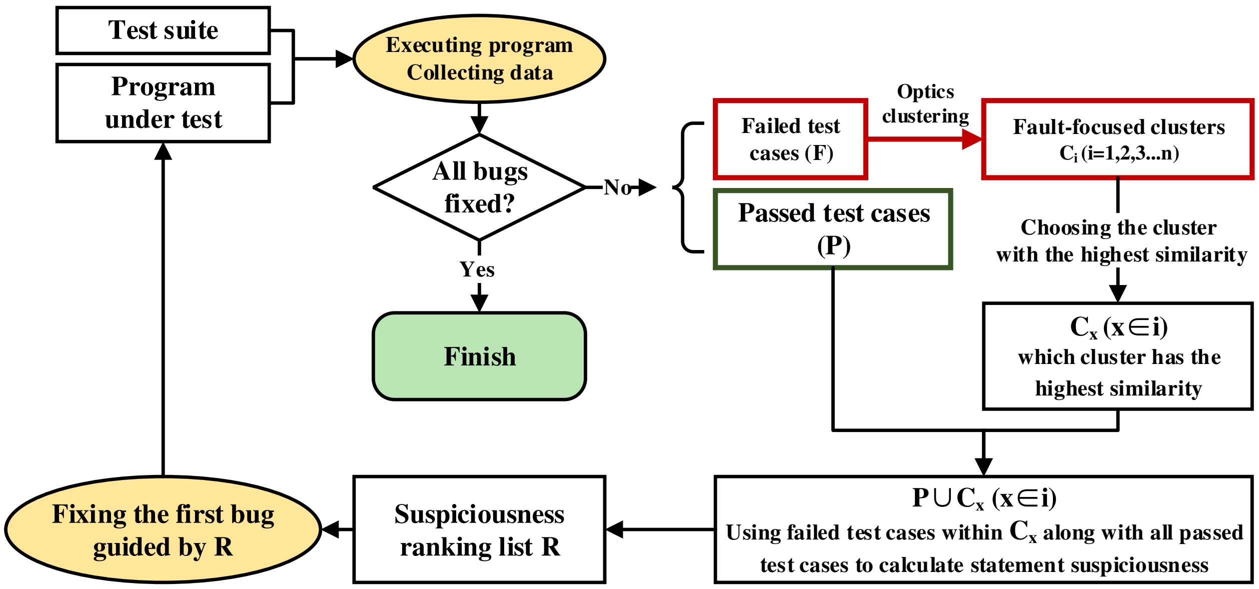

Test suite
Program under test
Executing program Collecting data
All bugs fixed?
No
Yes
Finish
Failed test cases (F)
Optics clustering
Fault-focused clusters C_i (i=1,2,3...n)
Passed test cases (P)
Choosing the cluster with the highest similarity
C_x (x∈i) which cluster has the highest similarity
$P \cup C_x$ (x∈i) Using failed test cases within C_x along with all passed test cases to calculate statement suspiciousness
Suspiciousness ranking list R
Fixing the first bug guided by R

Fig.6. Framework of FATOC

In this step, we execute all test cases on the program under test, and collect the coverage information. If all the test cases are passed, which means that there is no fault in the program or all the faults have been fixed, the FATOC process can be terminated. Otherwise, all the test cases will be divided into two groups. We use P to denote passed test cases and use F to denote failed test cases. Then we collect the test cases' coverage information and utilize this coverage information as feature vectors for the next clustering step.

Step 2. Clustering failed test cases: In this step, we adopt OPTICS algorithm to cluster failed test cases. All the failed test cases will be grouped into several clusters according to the distance among them. Specifically, the test cases with smaller distance values will be grouped into the same cluster.

Step 3. Choosing a cluster with the highest quality: In this step, we choose a cluster with the lowest average reachability distance as the target cluster, because reachability is calculated from the similarity of test cases, which can represent the density between test cases. Besides, previous studies [19,20] have shown that clusters with higher density value will have higher quality. Therefore, the target cluster we choose is the cluster with the highest quality. The detailed process of this step is described in Section 4.4.

Step 4. Single-fault localization: After obtaining the cluster with the highest quality, the probability of the failed test cases in this cluster cover the same single fault is very high. Then in this step, we combine the failed test cases in this cluster with all the passed test cases to get a new test set Cx for localizing only one fault. Then we use a suspiciousness formula in Table 1 to calculate the suspiciousness value of all the statements. Finally, we can generate a ranking list and the developers can use this list to fix the corresponding fault.

Step 5. Fault fixing: In step 4, we can get a ranking list R. A developer can use the list R to exam statements according to suspiciousness from high to low until the first real faulty statement is localized. After the developer fixes the fault, we go to step 1 and use the fixed program to perform fault localization.

4.2 Representation of Failed Test Cases

To implement the clustering algorithm, we need to use feature vectors to represent failed test cases. The widely used test case feature representation methods can be summarized as follows:

- Vectors of each test case are represented by its coverage information, and the n-th position value indicates the execution information of the n-th statement, where 1 indicates that the test case executed the corresponding statement, otherwise the value is 0.
- Vectors of each test case are represented by its weighted coverage information, and the n-th position value is determined by the suspiciousness value of the n-th statement, not simply 0 or 1.
- Based on a given fault localization technique, vectors of each failed test case are represented by a suspiciousness ranking value calculated by this failed test case and all the other passed test cases.

In our study, we choose the first representation method, since the last two methods will assign different values to statements according to their probabilities of being faulty, and these methods will cause misleading effects in multi-fault localization [12, 13].

4.3 OPTICS Clustering Algorithm

4.3.1 Preliminaries of Density Clustering Algorithms

In data mining, density clustering algorithms divide objects into clusters such that members of the same cluster are as similar as possible [47]. Density-based spatial clustering of applications with noise (DBSCAN) is a data clustering algorithm proposed by Ester et al. [48]. The core idea of DBSCAN is that for each cluster, at least $MinPts$ other objects are required within the radius *Eps* of the *Core objects*. Some definitions used to introduce the DBSCAN algorithm are listed as follows.

Definition 1 (*Neighborhood*). Neighborhood refers to the distance between two objects, which is determined by a distance formula.

Definition 2 (*Eps-neighborhood*). The Eps-neighborhood of an object p is defined by following:

$$\{q|\ dist(p,q) \leqslant Eps, q \in D\}.$$

Definition 3 (*Core object*). If the radius *Eps* of the object p contains at least $MinPts$ other objects, then the object p is a core object.

DBSCAN algorithm starts with an arbitrary object p that has not been visited in the database D and retrieves *Eps*-neighborhood of the object p. If the size of *Eps*-neighborhood is larger than $MinPts$, a new cluster will be created. The object p and its neighbors are assigned to this new cluster. This process is repeated until all objects have been visited.

To evaluate the similarity between every two objects in the sample space, we need a distance measure formula $dist(p,q)$ that tells how far the objects p and q are. In our study, we use the classical Euclidean distance formula, which is defined as follow:

$$dist(p,q) = \sqrt{\Sigma_{i=1}^{n}(x_{pi} - x_{qi})^2} \quad (1)$$

where p=$(x_{p1}, x_{p2}, \cdots, x_{pn})$ and q=$(x_{q1}, x_{q2}, \cdots, x_{qn})$ are two n-dimensional data objects.

4.3.2 Sorting cluster density

In the DBSCAN algorithm, there are two hyper-parameters (*Eps*-neighborhood and $MinP$ts) and the cluster results of the clustering are susceptible to the values of these two hyper-parameters.

To overcome this shortcoming of the DBSCAN algorithm, OPTICS clustering (Ordering Objects to identify the clustering structure) algorithm [22] is proposed.

OPTICS clustering algorithm does not display the resulting class clusters but ranks the samples according to the similarity between the samples. Finally, it will output a graph with the reachable distance as the vertical axis and the sample object output order as the horizontal axis, which presents the density structure. In other words, OPTICS clustering algorithm only generates the density structure of sample objects.

Since the OPTICS algorithm is an improvement of DBSCAN algorithm, many definitions are the same, such as *Neighborhood*, *Eps-neighborhood* and *Core object*. On this basis, two more definitions are needed to describe the OPTICS clustering algorithm. Here we still consider the database D.

Definition 4 (*Core-distance*). For an object x ($x \in D$), the core distance of x is the smallest *Eps* that makes object x as the core object.

Definition 5 (*Reachability-distance*). Let p and o be objects from a database D, Reachability distance of p and o represents the minimum *Eps* that allows o to be the core object and p is directly density reachable from o.

The value of Reachability-distance is related to the density of the space where the object is located. The higher the density, the smaller the distance that it can reach directly from the adjacent nodes.

Algorithm 1 OPTICS clustering algorithm

```
Input:
    Set of objects $D$, $MinPts$, $Eps$
Output:
    Result queue $Q_{result}$
1: $Q_{order} \leftarrow \emptyset$
2: $Q_{result} \leftarrow \emptyset$
3: $i \leftarrow 1$
4: $object_i \leftarrow D.get(i)$
5: repeat
6:    if $object_i \notin Q_{result}$ and $object_i.isCoreObject()$ then
7:       $Q_{result}.Append(object_i)$
8:       $Q_{order}.Expand(object_i, D, MinPts, Eps)$
9:    end if
10:   if $Q_{order}.size>0$ then
11:      $Q_{order}.sort()$
12:      for $object_j$ in $Q_{order}$ do
13:         if $object_j.isCoreObject()$ then
14:            $Q_{result}.Append(object_j)$
15:            $object_i \leftarrow object_j$
16:            $Q_{order}.Expand(object_i, D, MinPts, Eps)$
17:            Break
18:         end if
19:      end for
20:   else
21:      $i \leftarrow i+1$
22:      $object_i \leftarrow D.get(i)$
23:   end if
24: until $i>D.size()$
25: return $Q_{result}$
```

Algorithm 1 provides the pseudo-code of OPTICS clustering. We first initialize two queues: order queue and result queue (Lines 1 to 2). Then, we append a core object A from D that is not in the result queue into the result queue (Lines 3 to 7), and we append all directly density reachable objects of object A into the ordered queue (Line 8). Next, all the objects in the order queue are arranged in ascending order according to their reachability distance from object A (Line 11). Note that objects that already have a smaller reach are not updated. Later, we pull the ranked first object B from the ordered queue and append object B into the result queue if object B is a core project, or pull the next object in order queue as object B (Lines 12 to 14). Finally, object B will be used as the new object A, repeatedly iterating until all objects are visited (Lines 15 to 24). At the end of the algorithm, we can get the output result sequence, which represents a list of reachability distance that can be used to reflect the cluster structure.

4.4 Choosing Cluster

The cluster-ordering of a data set can be represented and understood graphically. In principle, one can understand the cluster structure in the form of a line chart of the result queue. Fig. 7 depicts a simple 2-dimensional data set and a list of reachability distance values in a result queue generated by OPTICS algorithm.

As shown in Fig. 7, each depression in the line graph represents a cluster structure.

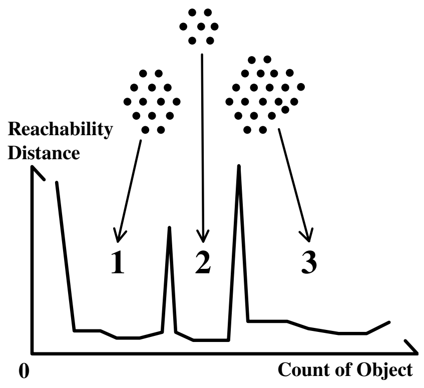


Fig.7. Example of result queue.

To filter out the clusters with the highest similarity, we calculate the average reachability distance corresponding to each cluster and select a cluster with the smallest average value as the target cluster. Because the reachability distance can reflect the differences between test cases, then test cases within the cluster with the smallest average reachability value has the highest similarity. Finally, we use failed test cases in the target cluster and all the passed test cases to localize a single fault.

Take the buggy program shown in Table 2 as an example. There are eight failed test cases in this program. After OPTICS clustering, two clusters can be obtained: {T3, T4, T5, T6} and {T7, T8 , T9, T10}. In particular, because the objects in the two clusters have the same feature vector (coverage information), the average reachability distance of the two clusters is the same. At this time, we randomly select a cluster as the target cluster for fault localization. It is worth noting that failed test cases of each class cluster are caused by the same single faulty. Therefore we can achieve the ideal clustering result in this example.

5 Experimental Setup

In this section, we present the research questions, subject programs' information and evaluation metrics in our study.

5.1 Research Questions

We conduct our empirical study to address the following three research questions.

RQ1: How does FATOC perform in terms of clustering accuracy?

The conclusions of the controlled study in Section 3.5 show that MFL accuracy and *precision* value has a strong correlation. Therefore, to answer this question, we apply traditional SBFL, MSeer and FATOC approach to the 804 multi-fault programs from SIR, which is shown in Table 5, and we use three evaluation metrics FPR, *precision* and *recall*, which are introduced in Fig. 4, to evaluate the clustering accuracy of these approaches.

RQ2: Compared to traditional SBFL techniques with different suspiciousness formulas, how does FATOC perform with the same formula in terms of fault localization accuracy?

To answer this RQ, we analyze the fault localization accuracy in terms of two metrics $A\text{-}EXAM$ and $TOP\text{-}N\%$, and then we further use the Wilcoxon signed-rank test to perform statistical analysis for comparing different techniques. In this RQ, we select traditional SBFL technique as the baseline, since it is the most widely investigated technique in previous fault localization studies [10, 38].

RQ3: Can FATOC achieve better fault localization accuracy than the state-of-the-art MFL approach?

Besides traditional SBFL techniques, we also compare FATOC to state-of-the-art MFL approach MSeer [10]. The reasons for choosing MSeer as the baseline can be summarized as follows: First, MSeer is a recently proposed MFL technique. Second, MSeer can achieve better performance than other bug isolation-based MFL method [10], which uses one-fault-at-a-time strategy.

To answer these three RQs, we perform all the experiments on the CentOS operation system with 18-core CPU. Moreover, we employ a commonly used Gcov [49] tool to collect the execution coverage information of test cases.

5.2 Subject Programs

Table 5 shows the statistical information of 9 subject programs used in our study and their corresponding test suites. All of them can be downloaded from the widely used SIR (Software-artifact Infrastructure Repository) [46]. In particular, seven programs are from Siemens suite, including Print tokens, Print tokens2, Schedule, Schedule2, Replace, Tcas and Tot info, while the other two programs are from Unix utilities. Here Sed is a stream editor and Grep is a command-line utility for searching plain-text datasets for lines that match a regular expression.

Table 5. Characteristics of Subject Programs

Program	#LOC	#Tests	#Multi-fault Versions
Print tokens	563	4130	110
Print tokens2	508	4115	92
Schedule	410	2650	69
Schedule2	309	2710	89
Replace	563	5542	86
Tcas	173	1608	77
Tot info	406	1052	129
Sed	8059	360	117
Grep	13342	808	35

In Table 5. column 1 to column 3 show the program name, lines of code (LOC) and the number of test cases respectively. For each program, the number of bugs is in the range of 2 to 10. Column 4 presents the number of multi-fault program versions for each subject program.

SIR provides the correct version and some faulty versions of each subject program with seeded faults. However, there are fewer faulty versions provided in SIR (generally no more than ten versions per program), so to make our experimental results more comprehensive, we need to generate a large number of multiple faulty versions ourselves. We first manually inject artificial faults into the correct programs to obtain more single-fault program versions in our empirical study. Then, we randomly combine these single-fault program versions to get a sufficient number of multi-fault program versions, such a faulty program version generation approach has been widely used in previous MFL studies [10, 50, 51].

In total, we generate 804 multi-fault versions from 9 programs and use these versions as our experimental subjects. In these multi-fault versions, the number of faults for each version ranges from 2 to 10, and there are 4400 faults in all versions.

5.3 Evaluation Metrics

In our empirical study, we use the performance metrics shown in 4 to evaluate the cluster accuracy of different approaches. To evaluate the fault localization accuracy of different approaches, we use *A-EXAM* and *TOP-N* metrics. In this section, we will show the details of these two metrics.

5.3.1 A-EXAM Score

For single-fault localization, *EXAM Score* is commonly used to evaluate the accuracy of fault localization [2,37,52]. A lower *EXAM* means that fewer statements need to be checked to find the real faulty statement, then the corresponding approach has a better fault localization accuracy. The *EXAM Score* metric can be calculated as follows:

$$EXAM = \frac{rank\ of\ the\ faulty\ statement}{number\ of\ the\ executable\ statements} \qquad (2)$$

Where the numerator is the rank of faults in the suspiciousness ranking list, and the denominator is the total number of executable statements that need to be checked.

For multi-fault localization, the fault localization accuracy evaluation is similar to *EXAM Score*. For example, MSeer uses the sum value of *EXAM* of all faults need to be located to evaluate the effectiveness of MFL approaches [10]. However, the number of faults in different multi-fault program versions is not the same. We use the average *EXAM* score (denoted as *A-EXAM*) to evaluate the accuracy of MFL approaches.

A-EXAM metric can be computed as follows:

$$A - EXAM = \frac{\Sigma_{n=1}^{N} \Sigma_{g=1}^{G(n)} EXAM(n,g)}{\Sigma_{n=1}^{N} G(n)} \qquad (3)$$

where $EXAM(n, g)$ is the $EXAM$ score of g-th ranking at n-th iteration. N indicates the total number of iterations and $G(n)$ refers to the number of faults located in the n-th iteration. Especially for FATOC, $G(n)$ always equals to 1 because we can only localize one faulty statement at every iteration in this approach. A smaller *A-EXAM* value means the corresponding MFL approach is more efficient.

5.3.2 TOP-N%

This metric reports the number of faulty statement that can be discovered within examining less than $N\%$ statements [28,53]. The higher the *TOP-N%* value, the fewer statements the developers need to check when localizing faults, which means that the corresponding fault localization approach is more effective.

The *TOP-N%* metric is a commonly used measurement in the fault localization field [54]. In our study, we use the *TOP-N%* metric to evaluate the effectiveness of MSeer and FATOC approaches in locating bugs in various rank lists.

6 Results Analysis

In this section, we comprehensively analyze the effectiveness of our proposed FATOC approach and discussed potential threats to validity.

6.1 RQ1: Clustering effectiveness comparison

To answer RQ1, we collect the clustering accuracy results of traditional SBFL, MSeer and FATOC approach on all the multi-fault program versions. Fig. 8 uses a violin plot to show the clustering performance of three approaches in terms of three evaluation metrics (*precision*, *recall*, and *FPR*). In Fig. 8, x-axis represents the evaluation metrics, y-axis indicates the value of the specific metric (*FPR*, *precision* or *recall*). Each block in the figure can visually represent the distribution of the evaluation result values, and the width of the block represents the data density of the corresponding value of the y-axis. It should be noted that we use Ochiai formula to implement the traditional SBFL approach since our proposed FATOC method also uses the Ochiai formula.

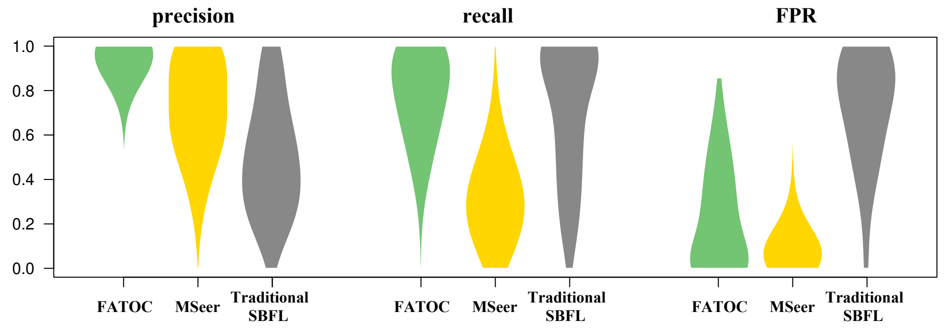


Fig.8. Clustering effectiveness comparison of different approaches in terms of three evaluation metrics

Table 6. Clustering effectiveness comparison of different approaches with different subject programs

Program	FATOC			MSeer			Traditional SBFL		
	precision	recall	FPR	precision	recall	FPR	precision	recall	FPR
Print tokens	90.93%	80.06%	33.39%	71.81%	37.25%	8.61%	49.09%	77.06%	70.58%
Print tokens2	92.20%	75.58%	27.55%	72.44%	38.21%	8.71%	52.78%	82.01%	66.36%
Schedule	97.99%	91.81%	26.26%	87.77%	34.66%	11.63%	61.76%	75.52%	71.08%
Schedule2	98.77%	95.32%	18.44%	91.39%	36.36%	10.81%	59.99%	74.22%	66.17%
Replace	93.15%	86.13%	35.67%	56.22%	22.26%	9.97%	43.00%	70.60%	84.20%
Tcas	100.00%	59.24%	0.00%	90.92%	53.68%	8.05%	74.32%	84.46%	37.51%
Tot info	98.86%	77.87%	6.95%	69.42%	30.12%	10.81%	43.28%	58.12%	76.86%
Sed	94.46%	65.49%	20.16%	58.92%	19.09%	3.80%	37.43%	69.49%	87.06%
Grep	94.14%	65.96%	12.63%	56.38%	13.41%	7.05%	45.30%	91.86%	85.14%
Average	95.61%	77.49%	20.12%	72.81%	31.67%	8.83%	51.88%	75.93%	71.66%

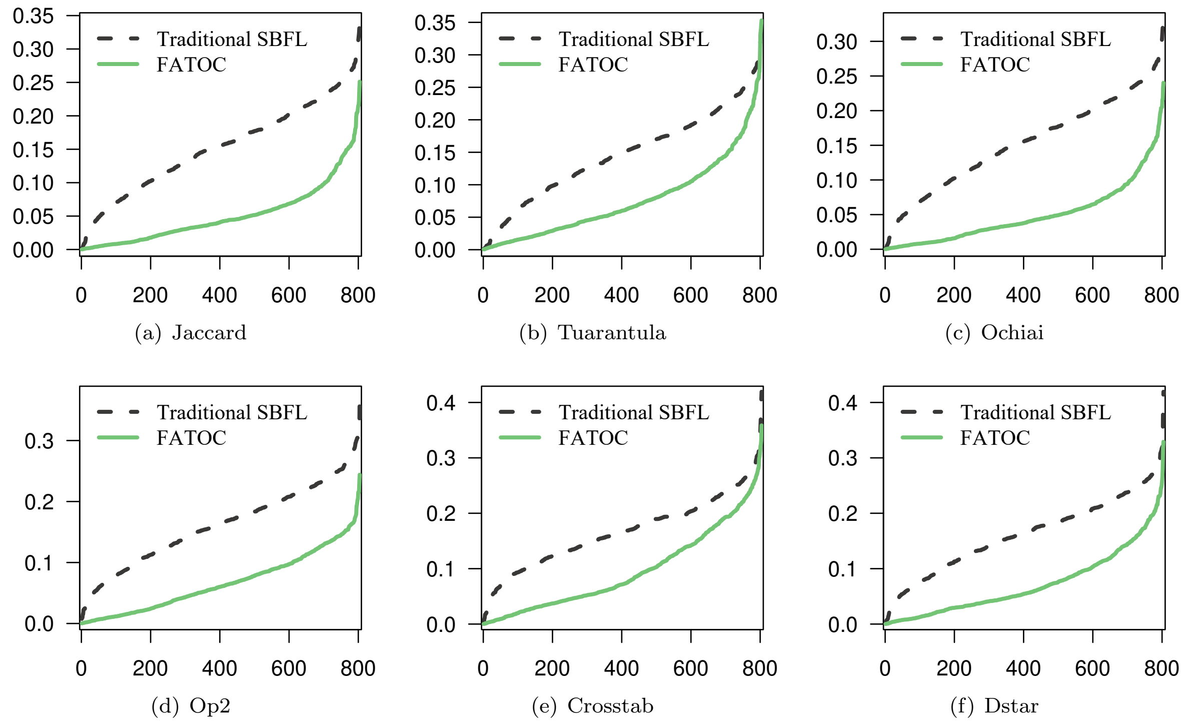


Fig.9. MFL comparison between FATOC and traditional SBFL in terms of *A-EXAM* metric with different formulas

Table 6 shows the detailed average results of Fig. 8, where different columns represent MFL approaches with different evaluation metrics, and the highlighted result indicates the corresponding approach can achieve the best performance when the corresponding subject program and evaluation metric are given. Based on

Fig. 8 and Table 6, we can find that FATOC performs the best in terms of *precision* metric, MSeer performs the best in terms of FPR metric in most cases, and traditional SBFL performs the best in terms of *recall* metric in some cases.

As discussed in Section 3.3, a higher *precision* and *recall* value and a lower FPR value indicates a better clustering method. Based on the empirical findings of our controlled study in Section 3.5, *precision* metric has a stronger correlation between the performance of MFL and the accuracy of the clustering algorithm than other metrics. Therefore, FATOC has more chances to be a better MFL approach than two baselines, which will be discussed when answering the remaining two RQs.

Summary for RQ1: In the above controlled study presented in Section 3, we find a higher accuracy of clustering measured by *precision* metric has a strong correlation with the accuracy of MFL. In this RQ, we find that our proposed FATOC approach can achieve the best performance in terms of *precision*, which means this approach can achieve better clustering results.

6.2 RQ2: Comparison between FATOC with Traditional SBFL

To answer RQ2, we present the experimental results of traditional SBFL and FATOC. Fig. 9 shows the overall results in terms of *A-EXAM* metric, where different sub-figures show the evaluation results of different suspiciousness formulas. In each sub-figure of Fig. 9, x-axis represents different program versions, and y-axis indicates the *A-EXAM* value of different approaches when used in the corresponding program. Since a lower *A-EXAM* value means a better fault localization technique, the closer the poly-line is to the x-axis, the better fault localization accuracy the MFL technique can achieve. In Fig. 9, we can find that compared with traditional SBFL, FATOC approach can achieve better performance than traditional SBFL when using six suspiciousness formulas.

Table 7. MFL performance comparison between traditional SBFL and FATOC When using different formulas in terms of *A-EXAM* metric

Formula	A-EXAM		
	FATOC	Traditional SBFL	Impr (%)
Jaccard	5.11%	15.27%	10.16%
Turantula	7.63%	14.62%	6.98%
Ochiai	4.91%	15.23%	10.32%
*Op*2	6.70%	16.06%	9.36%
Crosstab	9.51%	16.59%	7.08%
Dstar^3	7.22%	16.09%	8.87%
Average	6.85%	15.64%	8.80%

Moreover, Table 7 lists the detailed results, where different rows represent different suspiciousness formulas. The second and third columns refer to the average *A-EXAM* value of traditional FATOC and SBFL when using different suspiciousness formulas respectively. It can be found from Table 7 that FATOC approach can achieve better performance than traditional SBFL approach and the improvement ratio varies from 6.98% to 10.32%. On average, using FATOC approach can improve 8.8% fault localization accuracy in terms of *A-EXAM* metric. Finally, FATOC approach can achieve the best fault localization accuracy when using *Ochiai* formula, which is highlighted in Table 7.

Summary for RQ2: our proposed FATOC approach can achieve better MFL performance than traditional SBFL approach when considering six different formulas in terms of *A-EXAM* metric. Moreover, FATOC with *Ochiai* formula can achieve the best performance in our study.

6.3 RQ3: Comparison between FATOC with MSeer

In this section, we conduct a comprehensive comparison between FATOC and a state-of-the-art MFL

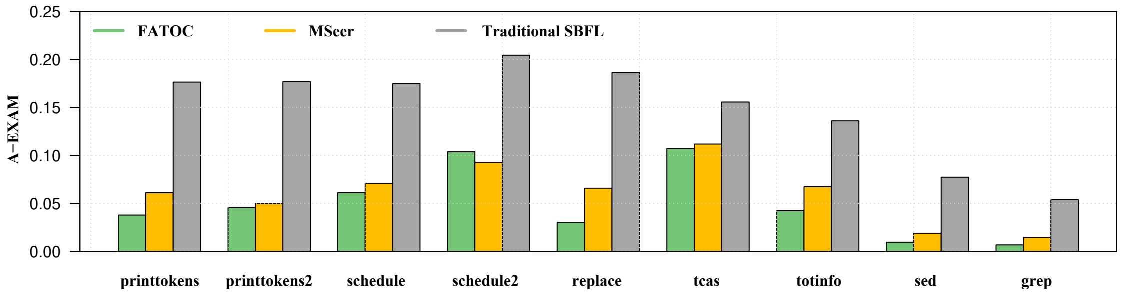


Fig.10. MFL performance comparison of different approaches in terms of *A-EXAM* with different programs

approach MSeer [10]. We first compare these two approaches in terms of two metrics (*A-EXAM* and *TOP-N%*). Then, we use Wilcoxon signed-rank test to analyze whether there are statistical performance differences among these two approaches. Notice in this RQ, MSeer approach uses *Crosstab* as the formula, since the previous study [10] showed MSeer could achieve the best performance when using this formula. Moreover, based on the empirical results on RQ2, we find FATOC with *Ochiai* formula can achieve the best performance. Therefore, in this RQ, FATOC approach uses *Ochiai* as the formula. Finally, the traditional SBFL approach uses the same *Ochiai* formula as the FATOC approach as the control approach.

Fig. 10 shows the MFL results of applying three approaches to each program in terms of *A-EXAM* metric. Moreover, we also show the detailed results of Fig. 10 in Table 8, where the highlighted results indicate that the corresponding technique performs the best in the corresponding subject program. Based on Fig. 10 and Table 8, we can find that for the nine programs, both FATOC and MSeer can perform better than traditional SBFL, and FATOC can obtain smaller *A-EXAM* values in eight of nine programs, which means FATOC has a better MFL performance than MSeer in most cases.

Table 8. MFL Performance Comparison of Different Approaches in terms of *A-EXAM* and by using Wilcoxon Signed-Rank Test

Program	*A-EXAM*			Confidence
	FATOC	MSeer	Traditional SBFL	
Print tokens	3.79%	6.12%	17.64%	99.99%
Print tokens2	4.57%	4.99%	17.69%	45.04%
Schedule	6.12%	7.10%	17.48%	98.76%
Schedule2	10.38%	9.27%	20.44%	2.52%
Replace	3.03%	6.58%	18.65%	99.99%
Tcas	10.72%	11.18%	15.56%	57.27%
Tot info	4.24%	6.74%	13.60%	99.99%
Sed	0.96%	1.89%	7.73%	99.99%
Grep	0.68%	1.46%	5.40%	97.67%
All	4.91%	6.24%	15.23%	99.99%

To make the experimental conclusions more convincing, we also use Wilcoxon signed-rank test [55] to perform statistical analysis. Wilcoxon signed-rank test provides a reliable statistical basis for comparing the effectiveness of different techniques and has been commonly used in previous studies [10,38,56–58]. The null hypothesis is set as follows:

H0: Given the subject programs under test, FATOC can perform significantly better than MSeer approach in terms of *A-EXAM* metric.

The acceptance level of H0 implies the confidence level of the claim that FATOC performs better than MSeer in terms of MFL accuracy. The corresponding results are shown in the fifth column of Table 8, which presents the confidence level of H0 towards each subject program. As Table 8 shows, the confidence level of all programs is 99.99%. Therefore we can conclude that FATOC can perform significantly better than MSeer in most cases. More specifically, the confidence level is over 97.00% in six out of nine programs. The confidence level is around 50% in the other two programs, which means FATOC and MSeer can achieve similar performance in two programs (Print tokens2 and Tcas). The worst situation for FATOC is locating bugs in Schedule2 program, where MSeer can perform significantly better than FATOC. Based on the clustering effectiveness comparison shown in Table 6, MSeer shows the best clustering accuracy in Sechedule2 program in terms of *precision* metric. Therefore, such results are in accordance with the empirical findings discussed in Section 3 (i.e., the *precision* metric in clustering has a stronger correlation with MFL accuracy).

Table 9. MFL Performance comparison of three methods in terms of *TOP-N%* metric

TOP (%)	Percentage of located bugs		
	FATOC	MSeer	Traditional SBFL
1	36.91%	15.98%	7.59%
2	48.50%	24.23%	11.39%
3	57.30%	32.39%	14.86%
4	62.43%	38.09%	18.18%
5	66.93%	43.73%	21.93%
10	85.45%	62.61%	37.36%
15	92.07%	72.36%	49.00%
20	95.41%	83.16%	60.27%
25	98.57%	90.95%	75.32%
30	99.05%	94.84%	85.18%
40	99.93%	99.43%	98.20%
50	100.00%	100.00%	100.00%

Besides *A-EXAM* metric, we also conduct a more detailed analysis of MFL performance of three approaches by using *TOP-N%* metric. The results are shown in Table 9, where each row refers to a different *N* value, which means checking the corresponding percentage of statements. Take the results shown in the first row for an example, and it indicates that when checking 1% statements in the ranking list generated by three MFL approaches, three are 7.59%, 15.98% and 36.91% of all bugs could be localized by using Traditional SBFL, MSeer and FATOC respectively. Table 9 shows that using FATOC can find 36.91% bugs when checking the top 1% statements, more than that of checking the top 3% statements by using Traditional SBFL and MSeer, which are 14.86% and 32.39% respectively. Therefore, FATOC can find more bugs when checking fewer statements, which means FATOC performs better than the other two baselines in terms of *TOP-N%* metric.

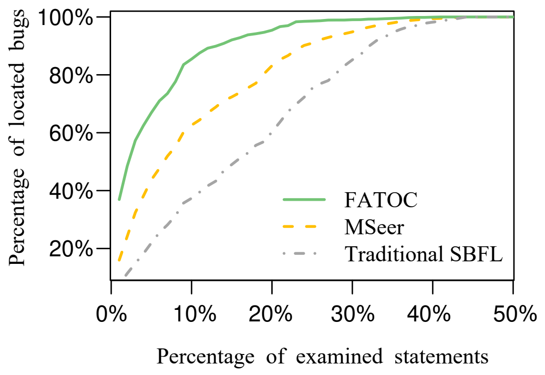


Fig.11. MFL performance comparison of three methods in terms of *TOP-N%* metric.

We use Fig. 11 to show the comparison results in terms of *TOP-N%* metric more intuitively as shown in Table 9. In Fig. 11, x-axis represents different N values, and y-axis represents the percentage of faults that can be localized when checking the top N% statements in the ranking list. Based on Table 9 and Fig. 11, we can find that in all situations, FATOC can perform better than MSeer, which means that when checking the same number of statements, the developers can localize more faults by using our proposed FATOC approach.

Summary for RQ3: FATOC can perform better than MSeer and traditional SBFL in terms of both *A-EXAM* and *TOP-N%* metrics.

6.4 Threats to Validity

Internal threats to validity. Our study's main internal threat is the potential implementation fault in our proposed FATOC approach and the baseline approaches. To alleviate this threat, we first implemented the OPTICS clustering algorithm based on the pyclustering framework [2], and then we used two-dimensional data to verify the effectiveness of the OPTICS clustering algorithm. Moreover, we implemented MSeer approach proposed by Gao et al. [10] as a baseline for comparison. To mitigate possible faults in implementing MSeer, we used the examples in their paper to test our code.

External threats to validity. The first external threat is related to the scale of this experiment. We only conducted 804 faulty versions from the SIR repository and a total of 4400 faults for MFL. However, SIR repository has been widely used in previous fault localization studies [10, 59]. Therefore the quality of those programs from this repository can be guaranteed. The second external threat is related to the two baselines we choose. To alleviate this threat, we first compare FATOC with traditional SBFL in RQ2, since traditional SBFL is the basic technique, which can reflect the performance of unimproved fault localization for MFL. We second compare FATOC with a state-of-the-art MFL approach MSeer in RQ3, since empirical results showed that MSeer had achieved better performance than other MFL approaches [10].

Construct threats to validity. Threats to construct validity include how well we measure our experimental results. To alleviate this threat, we use $A\text{-}EXAM$ and $TOP\text{-}N\%$ to evaluates the performance of our approach and used the confusion matrix based metrics to evaluate the accuracy of clustering algorithms. These metrics have been widely used in evaluating the performance of multi-fault localization [2, 10, 37] and bug isolation effect [7, 60].

Conclusion threats to validity. To more comprehensively compare our proposed FATOC approach with the baseline MSeer, we use the Wilcoxon signed-rank test to verify the confidence of our conclusions. Firstly, the Wilcoxon signed-rank test is suitable for samples that cannot be assumed to be normally distributed. Secondly, this statistical test method has also been widely applied in previous studies on fault localization [10, 38, 56–58].

7 Conclusions and Future Work

In this paper, we conduct a large-scale controlled study with ten levels of three misgrouping cases on 12,786 multi-fault version programs. The empirical research results show that there is a strong correlation between bug isolation accuracy and fault localization accuracy, and better quality clusters can result in higher MFL accuracy.

Based on the findings of the controlled study, we present a novel MFL approach, FATOC, which uses the OPTICS clustering algorithm that can get the cluster with the highest quality. We evaluate FATOC on the benchmark SIR, and the results show that FATOC outperforms traditional SBFL significantly. Specifically, we improve efficiency from 6.98% to 10.32% in terms of $A\text{-}EXAM$ metric. Besides traditional SBFL, our evaluation results also show that our proposed approach can outperform the state-of-the-art MFL approach MSeer. Concerning the $TOP\text{-}N\%$ metric, FATOC can locate and rank the faults at TOP-1% for 20.93% more faults when compared with MSeer.

In the future, we first want to investigate the influence of more negative factors (such as the coincidental correct test cases) on the accuracy of MFL. We second want to use more large-scale real-world faulty programs to verify the generalization of our empirical results.

[2]https://github.com/annoviko/pyclustering/

References

[1] Xie X, Chen T Y, Kuo F C, Xu B. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology*, 2013, 22(4): 31.

[2] Wong W E, Gao R, Li Y, Abreu R, Wotawa F. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 2016, 42(8): 707-740.

[3] Kim J, Kim J, Lee E. VFL: Variable-based fault localization. *Information and Software Technology*, 2019, 107: 179-191.

[4] Landsberg D, Sun Y, Kroening D. Optimising Spectrum Based Fault Localisation for Single Fault Programs Using Specifications. In *Proceedings of the Fundamental Approaches to Software Engineering*, April 2018, pp.246-263.

[5] Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst M D, Pang D, Keller B. Evaluating and Improving Fault Localization. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering* May 2017, pp.609-620.

[6] Liu B, Nejati S, Briand L C, Briand L C. Effective fault localization of automotive Simulink models: achieving the trade-off between test oracle effort and fault localization accuracy. *Empirical Software Engineering*, 2019, 24(1): 444-490.

[7] Liu Y, Li M, Wu Y, Li Z. A Weighted Fuzzy Classification Approach to Identify and Manipulate Coincidental Correct Test Cases for Fault Localization. *Journal of Systems and Software*, 2019, 151: 20-37.

[8] Wah K S H T. A theoretical study of fault coupling. *Software Testing Verification and Reliability*, 2000, 10(1): 3-45.

[9] Gopinath R, Jensen C, Groce A. The Theory of Composite Faults. In *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation*, March 2017, pp.47-57.

[10] Gao R, Wong W E. MSeer-An advanced technique for locating multiple bugs in parallel. *IEEE Transactions on Software Engineering*, 2017, 45(3): 301-318.

[11] Zheng Y, Wang Z, Fan X Y, Chen X, Yang Z J. Localizing multiple software faults based on evolution algorithm. *Journal of Systems and Software*, 2018, 139: 107-123.

[12] Liu B, Nejati S, Briand L, Bruckmann T. Localizing multiple faults in simulink models. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, March 2016, pp.146-156.

[13] Jones J A, Bowring J F, Harrold M J. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis,* July 2007, pp.16-26.

[14] Chen Z, Chen Z, Zhao Z, Yan S, Zhang J, Xu B. An Improved Regression Test Selection Technique by Clustering Execution Profiles. In *Proceedings of the 2010 10th International Conference on Quality Software,* July 2010, pp.171-179.

[15] Chen S, Chen Z, Zhao Z, Xu B, Yang F. Using semi-supervised clustering to improve regression test selection techniques. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation,* March 2011, pp.1-10.

[16] Vangala V, Czerwonka J, Talluri P. Test Case Comparison and Clustering using Program Profiles and Static Execution. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering,* August 2009, pp.293-294.

[17] Dickinson W, Leon D, Fodgurski A. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering,* May 2001, pp.339-348.

[18] Dickinson W, Leon D, Podgurski A. Pursuing failure:the distribution of program failures in a profile space. *Acm Sigsoft Software Engineering Notes*, 2001, 26(5): 246-255.

[19] Mathias R, Lagrange M, Cont A, Gomez S. Efficient similarity-based data clustering by optimal object to cluster reallocation. *Plos One*, 2018, 13(6): e0197450.

[20] Liu Y C, Li Z M, Hui X, Gao X D, Wu, J J. Understanding of Internal Clustering Validation Measures. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, December 2010, pp.911-916.

[21] Huang Y, Wu J, Feng Y, Chen Z, Zhao Z. An empirical study on clustering for isolating bugs in fault localization. In *Proceedings of the 2013 IEEE International Symposium on Software Reliability Engineering Workshops,* November 2013, pp.138-143.

[22] Ankerst M, Breunig M M, Kriegel H-P, Sander J. OPTICS: ordering points to identify the clustering structure. *ACM Sigmod record*, 1999, 28(2): 49-60.

[23] The paper's information is hidden for the blinded peer-review process. July 2019, pp.18-25.

[24] Perez A, Rui A, Deursen A V. A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches. *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering*, May 2017, pp.654-664.

[25] Liu X, Liu Y, Li Z, Zhao R. Fault Classification Oriented Spectrum Based Fault Localization. In *Proceedings of the 2017 IEEE 41st Annual Computer Software and Applications Conference*, July 2017, pp.256-261.

[26] Zou D M, Liang J J, Xiong Y F, Ernst M D, Zhang L. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering*, 2019.

[27] Zhang L M, Kim M, Khurshid S. FaultTracer: A Change Impact and Regression Fault Analysis Tool for Evolving Java Programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, November 2012, pp.1-4.

[28] Wen M, Chen J J, Tian Y J, Wu R X, Hao D, Han S, Cheung S C. Historical Spectrum based Fault Localization. *IEEE Transactions on Software Engineering*, 2019: 1-1.

[29] Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering,* November 2005, pp.273-282.

[30] Naish L, Hua J L, Ramamohanarao K. A model for spectra-based software diagnosis. *Acm Transactions on Software Engineering and Methodology*, 2011, 20(3): 1-32.

[31] Dallmeier V, Lindig C, Zeller A. Lightweight Bug Localization with AMPLE. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, September 2005, pp.99-104.

[32] Masri W. Fault localization based on information flow coverage. *Software Testing Verification & Reliability*, 2010, 20(2): 121-147.

[33] Shu T, Ye T T, Ding Z H, Xia, J S. Fault localization based on statement frequency. *Information Sciences*, 2016, 360: 43-56.

[34] Jaccard P. Etude de la distribution florale dans une portion des alpes et du jura. *Bulletin De La Societe Vaudoise Des Sciences Naturelles*, 1901, 37(142): 547-57.

[35] Jones J A, Harrold M J, Stasko J. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002, pp.467-477.

[36] Rui A, Zoeteweij P, Gemund A J C V. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing*, December 2006, pp.39-46.

[37] Wong W E, Debroy V, Xu D. Towards Better Fault Localization: A Crosstab-Based Statistical Approach. *IEEE Transactions on Systems Man & Cybernetics Part C*, 2012, 42(3): 378-396.

[38] Wong W E, Debroy V, Gao R, Li Y. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability*, 2014, 63(1): 290-308.

[39] Feyzi F, Parsa S. Inforence: effective fault localization based on information-theoretic analysis and statistical causal inference. *Frontiers of Computer Science*, 2019, 13(4): 735-759.

[40] Zakari A, Lee S P, Hashem I A T. A single fault localization technique based on failed test input. *Array*, 2019, 3: 100008.

[41] Rui A, Zoeteweij P, Gemund A J C V. Spectrum-Based Multiple Fault Localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, Nov 2009, pp.88-99.

[42] Wong W E, Debroy V, Golden R, Xu X F, Thuraisingham B. Effective Software Fault Localization Using an RBF Neural Networkl. *IEEE Transactions on Reliability*, 2011, 61(1): 149-169.

[43] Zakari A, Lee S P. Simultaneous isolation of software faults for effective fault localization. In *2019 IEEE 15th International Colloquium on Signal Processing & Its Applications (CSPA)*, March 2019, pp.16-20.

[44] Zakari A, Lee S P. Parallel debugging: An investigative study. *Journal of Software: Evolution and Process*, 2019, 31(11): e2178.

[45] He Z J, Chen Y, Huang E Y, Wang Q X, Pei Yu, Yuan H D. A system identification based Oracle for control-CPS software fault localization. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, May 2019, pp.116-127.

[46] Do H, Elbaum S, Rothermel G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 2005, 10(4): 405-435.

[47] Birant D, Kut A. ST-DBSCAN: An algorithm for clustering spatial–temporal data. *Data & Knowledge Engineering*, 2007, 60(1): 208-221.

[48] Martin E, Hans-Peter K, Jiirg S, Xu X W. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 1996, 96(6): 226-231.

[49] Yang Q, Li J J, Weiss D M. A survey of coverage-based testing tools. *The Computer Journal*, 2009, 52(5): 589-597.

[50] Lamraoui S M, Nakajima S. A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs. *Journal of Information Processing*, 2016, 24(1): 88-98.

[51] Yu Z, Bai C, Cai K Y. Does the Failing Test Execute a Single or Multiple Faults? An Approach to Classifying Failing Tests. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* May 2015, pp.924-935.

[52] Steimann F, Frenkel M, Abreu R. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis,* July 2013, pp.314-324.

[53] Li X, Zhang L M. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 2017, 1: 1-30.

[54] Parnin C, Orso A. Are automated debugging techniques actually helping programmers?. *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, 11(11): 199-209.

[55] Prybutok V R. An Introduction to Statistical Methods and Data Analysis. *Technometrics*, 2012, 31(3): 389-390.

[56] Zhang L M, Zhang L, Khurshid S. Injecting mechanical faults to localize developer faults for evolving software Localization. *Acm Sigplan Notices*, 2013, 48(10): 765-784.

[57] Hoang V D, Oentaryo R J, Le T D B, Lo D. Network-Clustered Multi-Modal Bug Localization. *Acm Sigplan Notices*, 2019, 45(10): 1002-1023.

[58] Perez A, Rui A, D'Amorim M. Prevalence of Single-Fault Fixes and Its Impact on Fault Localization. In *IEEE International Conference on Software Testing*, March 2017, pp.12-22.

[59] Liu Y, Li Z, Zhao R, Gong P. An Optimal Mutation Execution Strategy for Cost Reduction of Mutation-Based Fault Localization. *Information Sciences*, 2017, 422(January 2018): 572-596.

[60] Manish M, Yuriy B. Automatically Generating Precise Oracles from Structured Natural Language Specifications Localization. In *41st ACM/IEEE International Conference on Software Engineering*, May 2019, pp.188-199.