Wu YH, Li Z, Liu Y *et al.* FATOC: Bug isolation based multi-fault localization by using OPTICS clustering. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(5): 979–998 Sept. 2020. DOI 10.1007/s11390-020-0549-4

FATOC: Bug Isolation Based Multi-Fault Localization by Using OPTICS Clustering

Yong-Hao Wu[1], Zheng Li[1], Yong Liu[1,*], *Member, CCF, ACM, IEEE*, and
Xiang Chen[2,3], *Senior Member, CCF, Member, ACM, IEEE*

[1] *College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China*
[2] *School of Information Science and Technology, Nantong University, Nantong 226019, China*
[3] *State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences Beijing 100093, China*

E-mail: appmlk@outlook.com; {lizheng, lyong}@mail.buct.edu.cn; xchencs@ntu.edu.cn

Received April 12, 2020; revised July 29, 2020.

Abstract Bug isolation is a popular approach for multi-fault localization (MFL), where all failed test cases are clustered into several groups, and 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 cannot 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 that the elements in a higher density 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 (Ordering Points to Identify the Clustering Structure) 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 a 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 that of traditional SBFL when using Ochiai formula in terms of metric *A-EXAM*. 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 the traditional SBFL and the MFL approach MSeer.

Keywords bug isolation, multiple-fault localization, ordering points to identify the clustering structure (OPTICS) clustering, empirical study

1 Introduction

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

Regular Paper

Special Section on Software Systems 2020

A preliminary version of the paper was published in the Proceedings of QRS 2019.

This work is supported in part by the National Natural Science Foundation of China under Grant Nos. 61902015, 61872026, and 61672085, the Nantong Application Research Plan under Grant No. JC2019106, the Open Project of State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences) under Grant No. 2020-MS-07.

*Corresponding Author

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 [3–5].

Most of SBFL studies mainly assume that the program under test only contains 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 [6, 7]. Therefore, the fault localization accuracy on multi-fault programs needs to 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 [8–11], where bug isolation is a popular MFL method [8, 11]. 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 [12, 13] utilized based on test case behavioral characteristics, since previous studies [14–16] have shown that test cases covering 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 of localizing a specific fault by some failed test cases that 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 (false positive rate), 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 higher 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 the precision metric and the performance of MFL is the highest.

The controlled study results show that we need to select the cluster with the best quality for MFL. Previous studies [17, 18] 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 [8, 19] 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 [11], 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 the FATOC (One-Fault-at-a-Time via OPTICS (ordering objects to identify the clustering structure) Clustering) approach for MFL, because this cluster algorithm can obtain the density of various clusters during clustering [20], 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 nine real-world programs, where each program contains 2–10 faults. Previous studies have proposed many bug isolation approaches to solve the MFL problem. For example, Gao and Wong [8] proposed MSeer by using the k-medoids algorithm. Jones *et al.* [11] proposed a hierarchical clustering based approach. Among them, MSeer can perform significantly better than the method proposed by Jones *et al.* [11] 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 metric $A\text{-}EXAM$, FATOC can improve the accuracy of MFL by 8.8% and 1.33% when compared with traditional SBFL and MSeer respectively.

To the best of our 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 the 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 the FATOC approach.

3) To evaluate the effectiveness of FATOC, we conduct an empirical study on 804 multi-fault program versions. Then we choose the traditional SBFL approach and the state-of-the-art MFL approach MSeer [8] as our baselines. The results show that FATOC can achieve better fault localization accuracy than the two baselines.

In this paper, we extend our previous study [21] 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 dataset 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 analyzes 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, and 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 [8, 11, 22]. As shown in Fig.1, SBFL first collects test cases' coverage information and execution results, and then 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

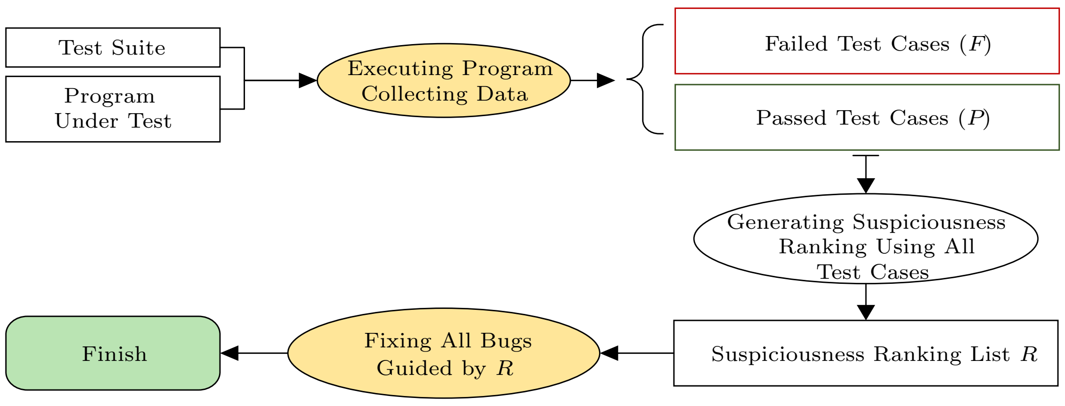

Test Suite
Program Under Test
Executing Program Collecting Data
Failed Test Cases (F)
Passed Test Cases (P)
Generating Suspiciousness Ranking Using All Test Cases
Suspiciousness Ranking List R
Fixing All Bugs Guided by R
Finish

Fig.1. Framework of traditional SBFL.

[1]https://github.com/appmlk/FATOC.git, Sept. 2020.

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[23].

In addition to the researches about suspiciousness formulas[24, 25], many studies also try to improve SBFL from concerning the coverage information constructed from different program entities, such as statements[2], call sequences[26], du-pairs[27], statement frequency[28] and so on. Statement is the most-studied program entity in the previous studies of SBFL, and we also use test cases' statement coverage information to implement our technique. Besides, among the different formulas, this paper considers six best-studied ones of them, which include Jaccard[29], Tarantula[30], Ochiai[31], OP2[25], CrossTab[32], and Dstar*[33]. Table 1 shows the detailed information of these six formulas. For these formulas, $fail(s)$ is the number of failed

Table 1. Suspiciousness Formulas

Name	Formula
Jaccard[29]	$\mathcal{S}us(s) = \frac{fail(s)}{totalfail+pass(s)}$
Tarantula[30]	$\mathcal{S}us(s) = \frac{\frac{fail(s)}{totalfail}}{\frac{fail(s)}{totalfail}+\frac{pass(s)}{totalpass}}$
Ochiai[31]	$\mathcal{S}us(s) = \frac{fail(s)}{\sqrt{totalfail\times(fail(s)+pass(s))}}$
OP2[25]	$\mathcal{S}us(s) = fail(s) - \frac{pass(s)}{totalpass+1}$
Crosstab[32]	$\mathcal{S}us(s) = \begin{cases}\chi(s)^2, & \text{if } \varphi(s) > 1,\\ 0, & \text{if } \varphi(s) = 1,\\ -\chi(s)^2, & \text{if } \varphi(s) < 1\end{cases}$
	$\varphi(s) = \frac{\frac{fail(s)}{totalfail}}{\frac{pass(s)}{totalpass}}$
	$\chi(s)^2 = \frac{(fail(s)-E_{cf}(s))^2}{E_{cf}(s)}+ \frac{(pass(s)-E_{cs}(s))^2}{E_{cs}(s)}+ \frac{(totalfail-fail(s)-E_{uf}(s))^2}{E_{uf}(s)}+ \frac{(totalpass-pass(s)-E_{us}(s))^2}{E_{us}(s)}$
	$E_{cf}(s) = \frac{(fail(s)+pass(s))\times totalfail}{totalcase}$
	$E_{cs}(s) = \frac{(fail(s)+pass(s))\times totalpass}{totalcase}$
	$E_{uf}(s) = \frac{(totalcase-fail(s)-pass(s))\times totalfail}{totalcase}$
	$E_{us}(s) = \frac{(totalcase-fail(s)-pass(s))\times totalpass}{totalcase}$
Dstar*[33]	$\mathcal{S}us(s) = \frac{fail(s)^*}{pass(s)+(totalfail-fail(s))}$

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.

Previous studies have achieved satisfactory fault localization accuracy on single-fault program versions. Jones and Harrold[24] 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 and Parsa[34] found that combining SBFL with static analysis could improve fault localization accuracy no matter whether the programs are implemented by C or Java. Liu *et al.*[5] proposed three manipulation strategies to reduce the negative impact of coincidental correct (CC) test cases in fault localization. Zakari *et al.*[35] 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, 9]. 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[23]. 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[36].

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.*[11] 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 the MFL accuracy. Such methods encode each program statement as a chromosome to indicate whether it contains faults[9, 37]. 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[8, 10, 19]. 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 and Lee[38] 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[8], 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 the 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.

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[8]. 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 and Lee[39] 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 that of OBA and MSeer. Their results suggest that clustering failed test cases based on their execution profile similarity may reduce the 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 perfor-

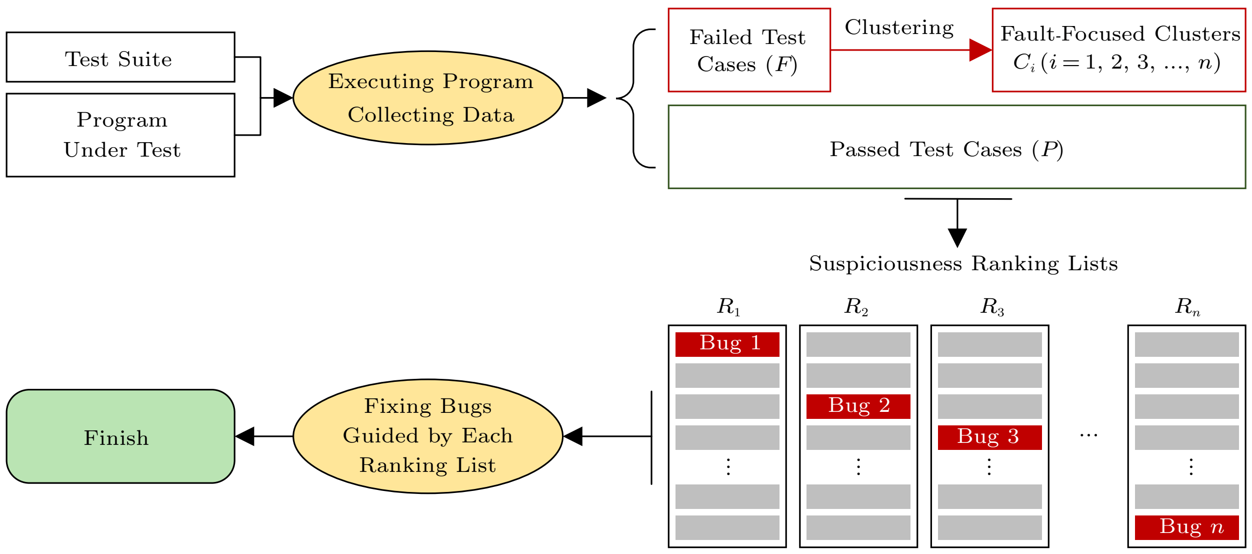

Test Suite
Program Under Test
Executing Program Collecting Data
Failed Test Cases (F)
Clustering
Fault-Focused Clusters C_i ($i = 1, 2, 3, ..., n$)
Passed Test Cases (P)
Suspiciousness Ranking Lists
R_1
R_2
R_3
R_n
Bug 1
Bug 2
Bug 3
Bug n
Fixing Bugs Guided by Each Ranking List
Finish

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

mance and the MFL accuracy, and then use the empirical findings to guide us to propose a novel MFL approach with a better fault localization accuracy.

3 Correlation Analysis Between Performance of Bug Isolation Based MFL and Clustering Accuracy

In this section, we conduct a controlled empirical study on the correlation between the performance of bug isolation based fault localization and 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 Subsection 2.2, the popular approach of MFL is to cluster the failed test cases before applying SBFL techniques[8, 10, 19], 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 failed by the same fault should be grouped into the same cluster, and then the MFL problem can be transformed into the single-fault localization problem.

Table 2[21] 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–T_{10} are eight failed test cases, in which T_3–T_6 and T_7–T_{10} execute the faults S_4 and S_6, respectively.

In terms of fault localization accuracy, Table 3[21] shows four suspiciousness formulas except for Tarantula achieve a low fault localization accuracy since the

Table 2. Statement Coverage and Test Case Execution Results of an Example Program with Two Faults

Program Under Test		Test Suite									
		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
		(−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

Note: (x, y) below test suite T_i means that in T_i, the values of parameter a and parameter b are x and y respectively.

Table 3. Suspiciousness Value and Rank of Statements Using All the Test Cases for the Simple Example in Table 2 When Considering Different Suspiciousness Formulas

Statement	$fail(s)$	$totalfail$	$pass(s)$	$totalpass$	Ochiai[31]		Tarantula[30]		OP2[25]		Dstar*[33]	
					$Sus(s)$	Rank	$Sus(s)$	Rank	$Sus(s)$	Rank	$Sus(s)$	Rank
S_1	8	8	2	2	0.89	2	0.5	3	7.33	2	256.0	2
S_2	8	8	2	2	0.89	2	0.5	3	7.33	2	256.0	2
S_3	8	8	2	2	0.89	2	0.5	3	7.33	2	256.0	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	256.0	2
S_6(fault)	4	8	0	2	0.71	**7**	1.0	**1**	4.00	**7**	16.0	**7**
S_7	8	8	2	2	0.89	2	0.5	3	7.33	2	256.0	2
S_8	8	8	0	2	1.00	1	1.0	1	8.00	1	∞	1
S_9	0	8	2	2	0.00	9	0.0	9	−0.66	9	0.0	9

faults (S_4 and S_6) rank the eighth and the 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–T_6 and T_7–T_{10} execute the faults S_4 and S_6 respectively, after bug isolation, T_3–T_6 and T_7–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. Taking 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, OP2 and Dstar* 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.*[19] analyzed the influence of k-means and hierarchical clustering algorithms on bug isolation. Liu *et al.*[10] employed decision-tree based algorithms to cluster failed test cases. Gao and Wong[8] 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, $cluster(f)$ to denote the number of the failed test cases that cover the fault f in the $cluster$, and $cluster(other)$ to denote the number of the failed test cases that cover other faults in the $cluster$. In ideal condition, $cluster(f) = failnum(f)$ and $cluster(other) = 0$.

In our study, we identify three kinds of misgroup scenarios. Fig.3[21] shows the clustering results of the ideal situation and three misgroup scenarios. In 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 $cluster(other) > 0$.

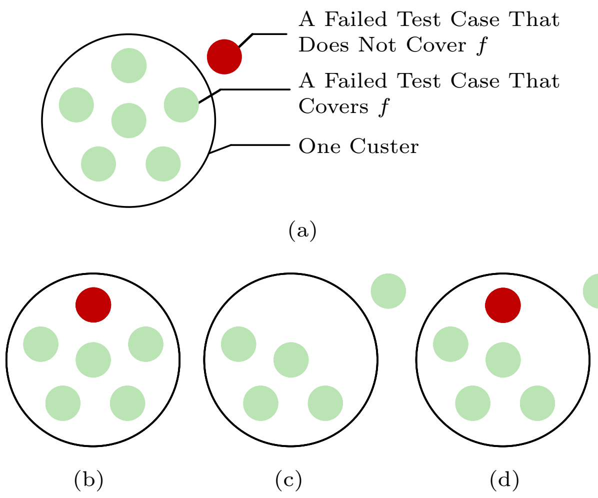

A Failed Test Case That Does Not Cover f
A Failed Test Case That Covers f
One Custer
(a)
(b)
(c)
(d)

Fig.3. Diagrams of ideal situation and three possible misgroup scenarios[21]. (a) Ideal clustering result. (b) Misgroup scenario 1. (c) Misgroup scenario 2. (d) Misgroup scenario 3.

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 do not cover f. In this scenario, $cluster(f) < failnum(f)$, $cluster(other) > 0$.

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 *precision*, *recall* and FPR[5, 40]. These metrics are calculated based on the confusion matrix, and a higher *precision* and *recall* value, and a lower FPR value indicates a better corresponding technique. The confusion matrix is shown in Fig.4.

Clustered into Correct Group or Not	Cover f or Not: Yes	Cover f or Not: No
Yes	True Positive (TP)	False Positive (FP)
No	False Negative (FP)	True Negative (TP)

Fig.4. Confusion matrix.

Then the definitions of evaluation metrics *precision*, *recall* and FPR are defined as follows:

$$precision = \frac{TP}{TP+FP},$$
$$recall = \frac{TP}{TP+FN},$$
$$FPR = \frac{FP}{FP+TN}.$$

For the bug isolation problem, given the cluster related to the fault f, true positive (TP) means the number of failed test cases within this cluster, which cover f; false positive (FP) means the number of failed test cases within this cluster, which do not cover f; false negative (FN) means the number of failed test cases outside this cluster, which cover f; and true negative (TN) means the number of failed test cases outside 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 subsection, we use 11 widely-studied real-world subject programs from SIR[11] (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 paper, we mainly design a controlled experiment to generate a variety of clusters with 10 misgrouping levels, where 5%–50% failed test cases are clustered incorrectly in the corresponding group. The fewer the 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 an $EXAM$ 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 Subsection 5.3. Fig.5 shows the $EXAM$ value of the Ochiai formula in three misgroup scenarios, where Fig.5(a)–Fig.5(c) refer to the misgroup scenario 1, the misgroup scenario 2 and the misgroup scenario 3 respectively, where origin refers to not using bug isolation strategy. In Fig.5, x-axis represents 77 970 faults of 12 786 programs under test, and y-axis represents the $EXAM$ value of locating the corresponding fault[21]. 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[21] shows the average $EXAM$ of finding the first faulty statement in 12 786 multi-fault versions under different bug isolation accuracies, and the bolded result indicates the corresponding suspiciousness formula can achieve the best performance under given 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

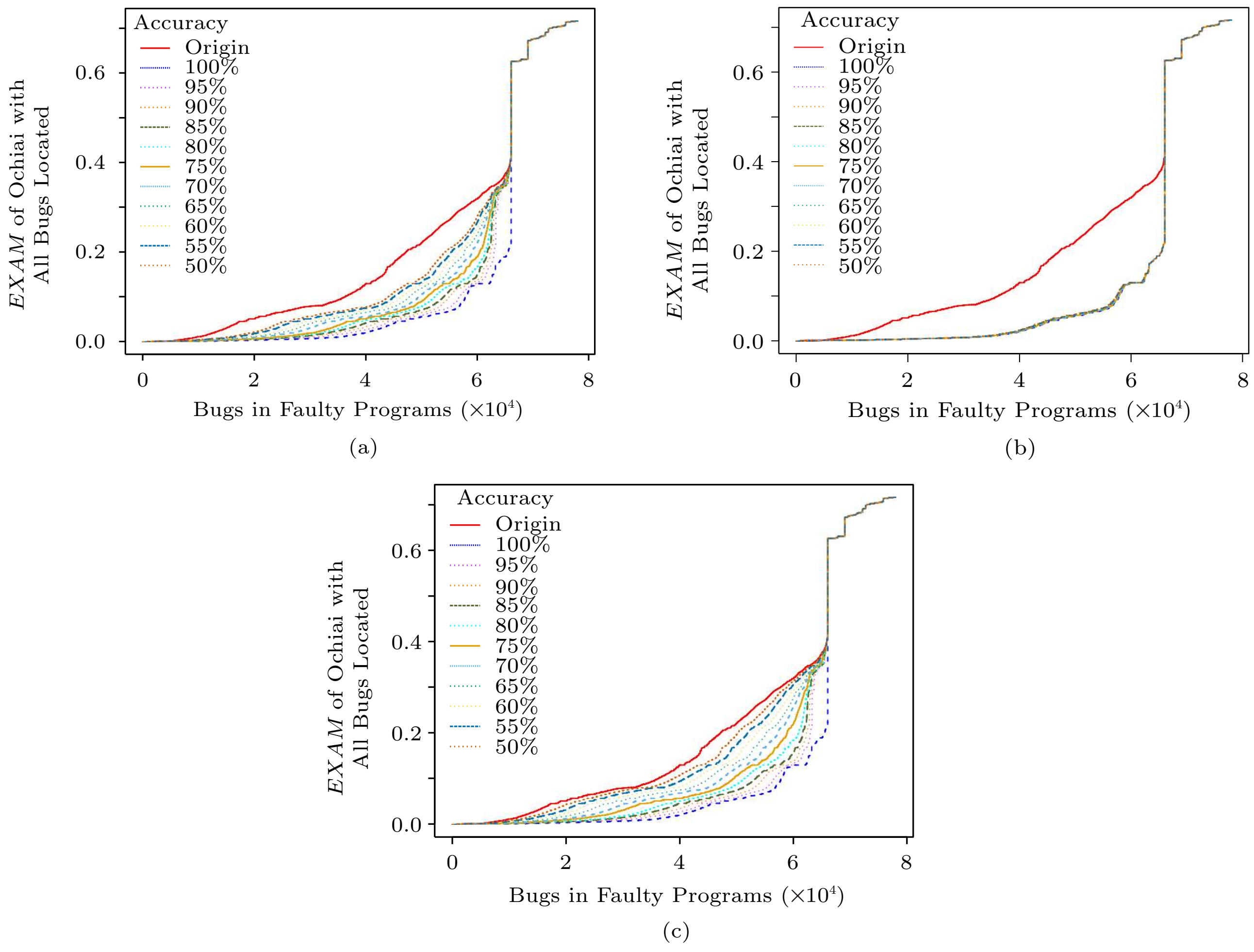


Fig.5. Cost of three misgroup scenarios [21]. (a) Misgroup scenario 1. (b) Misgroup scenario 2. (c) Misgroup scenario 3.

Table 4. 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
OP2	**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*	1.75	**1.88**	**2.02**	2.14	2.35	2.71	3.03	3.29	3.63	4.35	5.05

cases, followed by Tarantula (3 cases) and Dstar* (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 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 [17, 18].

The current MFL approaches include classical clustering algorithms such as k-means and k-medoids[8, 19]. 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[11], but they 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 OPTICS can obtain the density of various clusters during clustering, and it does not need to input the number of faults.

4 Proposed 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.* 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 the 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 have more opportunities to 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[17, 18] have shown that clusters with a 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 Subsection 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 covering 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

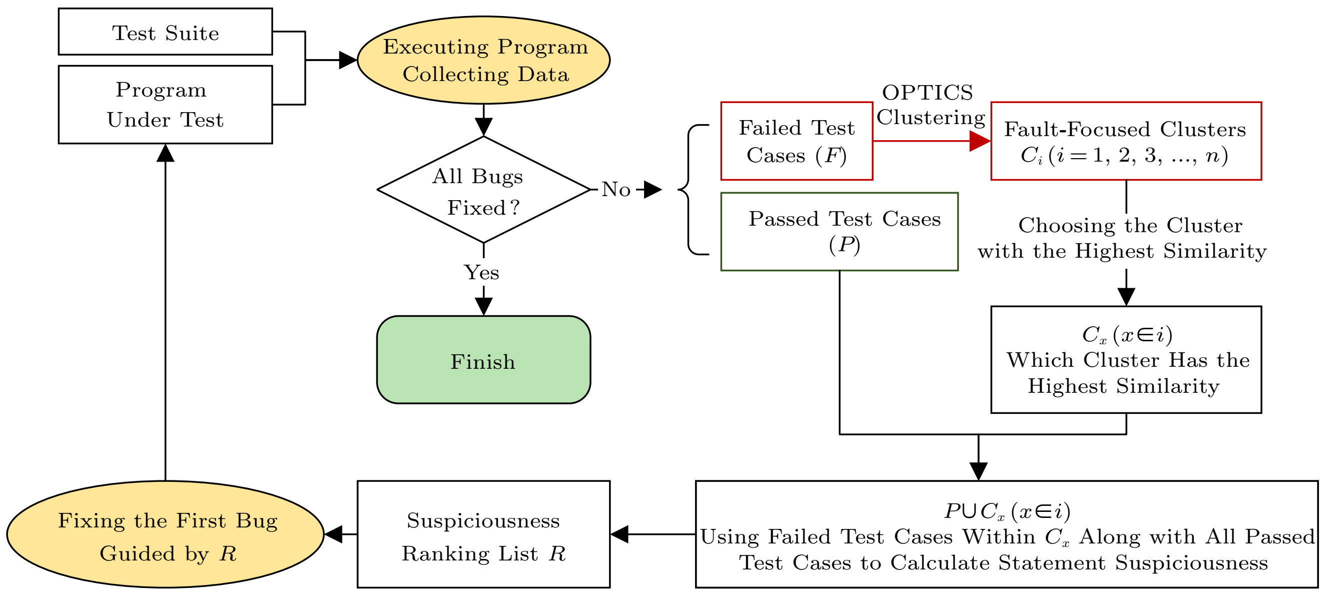

Test Suite
Program Under Test
Executing Program Collecting Data
All Bugs Fixed?
No
Yes
Finish
Failed Test Cases (F)
OPTICS Clustering
Fault-Focused Clusters C_i ($i=1, 2, 3, ..., n$)
Passed Test Cases (P)
Choosing the Cluster with the Highest Similarity
C_x ($x\in i$) Which Cluster Has the Highest Similarity
$P\cup C_x$ ($x\in i$) 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.

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 their 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[10, 11].

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[42]. Density-based spatial clustering of applications with noise (DBSCAN) is a data clustering algorithm proposed by Ester *et al.*[43] 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[43]). *Neighborhood refers to the distance between two objects, which is determined by a distance formula.*

Definition 2 (Eps-Neighborhood[43]). *The Eps-neighborhood of an object p is defined by the following*:

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

Definition 3 (Core Object[43]). *If the radius Eps of the object p contains at least $MinPts$ other objects, then the object p is a core object.*

The 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 follows.

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

Here $p = (x_{p1}, x_{p2}, \cdots, x_{pn})$ 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, the OPTICS (ordering objects to identify the clustering structure) clustering algorithm[20] was proposed.

The 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, the OPTICS clustering algorithm only generates the density structure of sample objects.

Since the OPTICS algorithm is an improvement of the 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[20]). *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[20]). *Let p and o be objects from a database D, the 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 provides the pseudo-code of OPTICS clustering. We first initialize two queues: order queue *OQueue* and result queue *RQueue* (lines 1 and 2). Then, we append a core object A from D that is not in the result queue into the result queue (lines 3–7), 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 reachability distance 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 the order queue as object B (lines 12–14). Finally, object B will be used as the new object A, repeatedly iterating until all objects are visited (lines 15–24). 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.

Algorithm 1 . OPTICS Clustering Algorithm

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

4.4 Choosing Cluster

The cluster-ordering of a dataset 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 dataset and a list of reachability distance values in a result queue generated by the OPTICS algorithm.

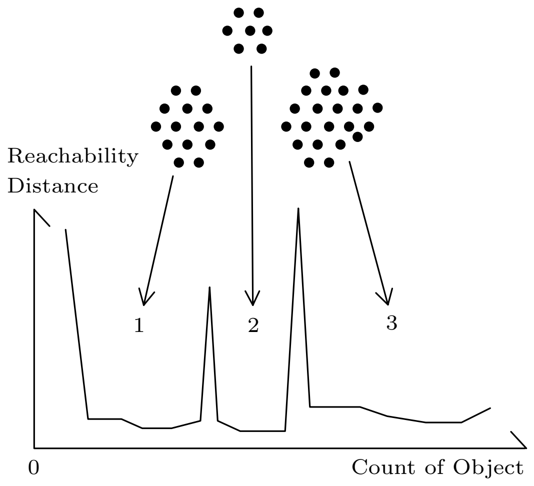


Fig.7. Example of a result queue.

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

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 have the highest similarity. Finally, we use failed test cases in the target cluster and all the passed test cases to localize a single fault.

Taking 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: $\{T_3, T_4, T_5, T_6\}$ and $\{T_7, T_8, T_9, T_{10}\}$. 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 of our study.

5.1 Research Questions

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

*RQ*1. How does FATOC perform in terms of clustering accuracy?

The conclusions of the controlled study in Subsection 3.5 show that MFL accuracy and *precision* value have a strong correlation in terms of $EXAM$ metric. 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 Subsection 3.3, to evaluate the clustering accuracy of these approaches.

Table 5. Characteristics of Subject Programs

Program Name	#LOC	#Tests	#Multi-Fault Versions
Print tokens	563	4 130	110
Print tokens2	508	4 115	92
Schedule	410	2 650	69
Schedule2	309	2 710	89
Replace	563	5 542	86
Tcas	173	1 608	77
Tot info	406	1 052	129
Sed	8 059	360	117
Grep	13 342	808	35

Note: #LOC means the lines of code, #Tests means the number of test cases, and #Multi-Fault Versions means the number of the multiple fault versions of the program under test.

*RQ*2. Compared with 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-$N\%$, and then we further use the Wilcoxon signed-rank test to perform statistical analysis for comparing different techniques. For this RQ, we select traditional SBFL technique as the baseline, since it is the most widely investigated technique in previous fault localization studies [8, 33].

*RQ*3. Can FATOC achieve better fault localization accuracy than the state-of-the-art MFL approach?

Besides traditional SBFL techniques, we also compare FATOC with the state-of-the-art MFL approach MSeer [8]. 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 methods [8], 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 [44] tool to collect the execution coverage information of test cases.

5.2 Subject Programs

Table 5 shows the statistical information of nine 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) [41]. 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.

For each program, the number of bugs is in the range of 2–10. 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 10 versions per program), thus 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, and such a faulty program version generation approach has been widely used in previous MFL studies [8, 45, 46].

In total, we generate 804 multi-fault versions from nine 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 4 400 faults in all versions.

5.3 Evaluation Metrics

In our empirical study, we use the performance metrics shown in Section 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 subsection, we will show the details of these two metrics.

5.3.1 *A-EXAM Score*

For single-fault localization, *EXAM* is commonly used to evaluate the accuracy of fault localization [2,32,47]. A lower *EXAM* means that fewer statements need to be checked to find the real faulty statement, and then the corresponding approach has a better fault localization accuracy. The *EXAM* metric can be calculated as follows.

$$EXAM = \frac{\text{rank of the faulty statement}}{\text{number of the executable statements}},$$

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*. For example, MSeer uses the sum value of *EXAM* of all faults that need to be located to evaluate the effectiveness of MFL approaches [8]. 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 can be computed as follows.

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

where $EXAM(n,g)$ is the *EXAM* score of the g-th ranking at the 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 1 because we can only localize one faulty statement at every iteration in this approach. A smaller *A-EXAM* value means a more efficient MFL approach.

5.3.2 *TOP-N%*

This metric reports the number of faulty statements that can be discovered within examining less than N% statements [23,48]. The higher the TOP-N% value, the fewer the 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 [49]. 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 discuss 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*). 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 the Ochiai formula to implement the traditional SBFL approach since our proposed FATOC method also uses the Ochiai formula.

Table 6 shows the detailed average results of Fig.8, where different columns represent MFL approaches with different evaluation metrics, and the bolded 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 metric *precision*, MSeer performs the best in terms of metric *FPR* in most cases, and traditional SBFL performs the best in terms of metric *recall* in some cases.

As discussed in Subsection 3.3, a higher *precision* value, a higher *recall* value, and a lower *FPR* value indicate a better clustering method. Based on the empirical findings of our controlled study in Subsection 3.5, the *precision* metric has a stronger correlation with the performance of MFL than other metrics. Therefore, FATOC has more chances to be a better MFL

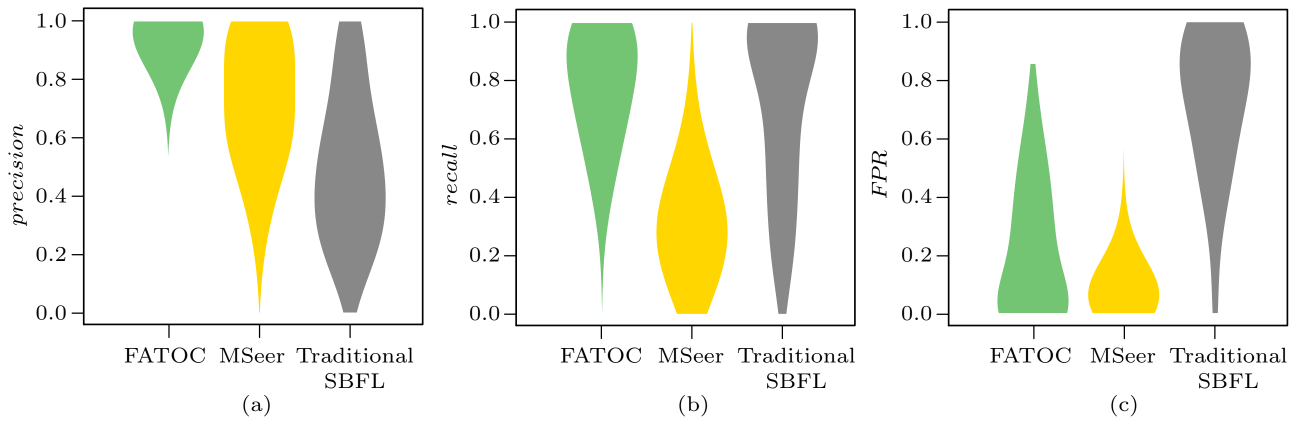


Fig.8. Clustering effectiveness comparison of different approaches in terms of three evaluation metrics. (a) *precision*. (b) *recall*. (c) *FPR*.

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

approach than two baselines, which will be discussed when answering the remaining two RQs.

*Summary for RQ*1. In the above controlled study presented in Section 3, we find a higher accuracy of clustering measured by metric *precision* 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 and Traditional SBFL

To answer RQ2, we present the experimental results of traditional SBFL and FATOC. Fig.9 shows the overall results in terms of metric *A-EXAM*, where different sub-figures show the evaluation results of different suspiciousness formulas. In 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 the traditional SBFL, the FATOC approach can achieve better performance when using six suspiciousness formulas.

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

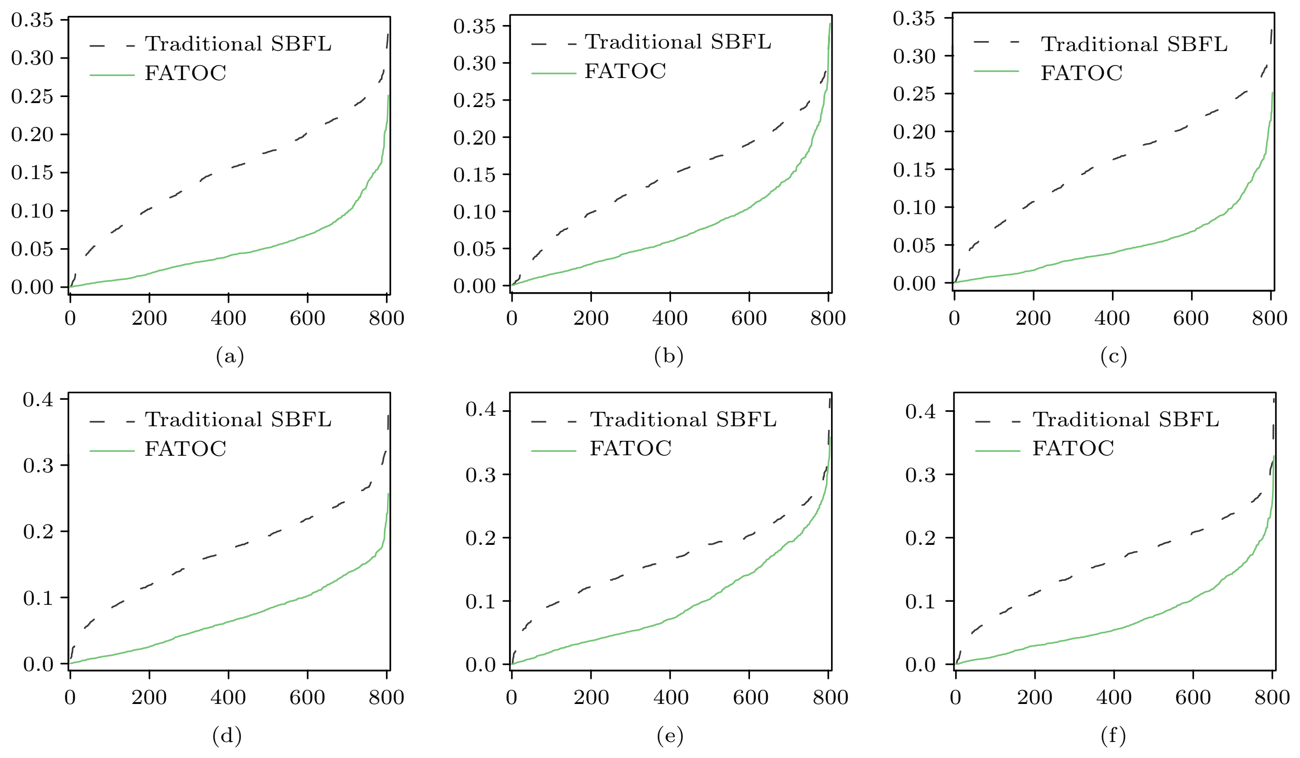


Fig.9. MFL comparison between FATOC and traditional SBFL in terms of metric *A-EXAM* with different formulas. (a) Jaccard. (b) Tuarantula. (c) Ochiai. (d) OP2. (e) Crosstab. (f) Dstar*.

Table 7. MFL Performance Comparison Results Between Traditional SBFL and FATOC When Using Different Suspiciousness Formulas in Terms of Metric *A-EXAM* (%)

Formula	FATOC	Traditional SBFL	Improve
Jaccard	5.11	15.27	10.16
Turantula	7.63	14.62	6.98
Ochiai	**4.91**	15.23	**10.32**
OP2	6.70	16.06	9.36
Crosstab	9.51	16.59	7.08
Dstar*	7.22	16.09	8.87
Average	6.85	15.64	8.80

*Summary for RQ*2. Our proposed FATOC approach can achieve better MFL performance than the traditional SBFL approach when considering six different formulas in terms of metric *A-EXAM*. Moreover, FATOC with Ochiai formula can achieve the best performance in our study.

6.3 RQ3: Comparison Between FATOC and MSeer

In this subsection, we conduct a comprehensive comparison between FATOC and a state-of-the-art MFL approach MSeer[8]. 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. The MSeer approach uses Crosstab as the formula, since the previous study[8] showed MSeer could achieve the best performance when using this formula. Moreover, based on the empirical results on RQ2, we find FATOC with the Ochiai formula can achieve the best performance. Therefore, in this RQ, the FATOC approach uses Ochiai as the formula. Finally, the traditional SBFL approach as a control approach uses the same Ochiai formula as the FATOC approach.

Detailed results are shown in Table 8, where the bolded results indicate that the corresponding technique performs the best in the corresponding subject program. 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.

To make the experimental conclusions more convincing, we also use Wilcoxon signed-rank test[50] to perform statistical analysis. The Wilcoxon signed-rank test provides a reliable statistical basis for comparing the effectiveness of different techniques and has been commonly used in previous studies[8, 33, 51]. The null hypothesis is set as follows.

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

Program Name	*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

*H*0. Given the subject programs under test, FATOC can perform significantly better than the MSeer approach in terms of *A-EXAM*.

The acceptance level of H0 implies the confidence level of the claim that FATOC performs better than MSeer in terms of the 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 another 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 program Schedule2, 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 the program Sechedule2 in terms of metric *precision*. 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).

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

Table 9. MFL Performance Comparison of Three Methods in Terms of Metric TOP-N%

N	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

Based on Table 9, 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 RQ*3. FATOC can perform better than MSeer and traditional SBFL in terms of both metrics *A-EXAM* and TOP-N%.

6.4 Threats to Validity

Internal Threats to Validity. Our study's main internal threat is the potential implementation fault of 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 the MSeer approach proposed by Gao and Wong[8] as a baseline for comparison. To mitigate possible faults in implementing MSeer, we used the examples in their paper[8] 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 4 400 faults for MFL. However, the SIR repository has been widely used in previous fault localization studies[8, 52]. Therefore the quality of those programs from this repository can be guaranteed. The second external threat is related to the two baselines we chose. To alleviate this threat, we first compared FATOC with the traditional SBFL in RQ2, since the traditional SBFL is the basic technique, which can reflect the performance of unimproved fault localization for MFL. We second compared 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[8].

Construct Threats to Validity. Threats to construct validity include how well we measure our experimental results. To alleviate this threat, we used $A\text{-}EXAM$ and TOP-N% to evaluate 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, 8, 32] and bug isolation effect[5, 53].

Conclusion Threats to Validity. To more comprehensively compare our proposed FATOC approach with the baseline MSeer, we used 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[8, 33, 51].

7 Conclusions

In this paper, we conducted a large-scale controlled study with 10 levels of three misgrouping cases on 12 786 multi-fault version programs. The empirical research results showed that there is a strong correlation between bug isolation accuracy and fault localization accuracy, and better quality clusters can result in a higher MFL accuracy.

Based on the findings of the controlled study, we presented a novel MFL approach, FATOC, which uses the OPTICS clustering algorithm that can get the cluster with the highest quality. We evaluated FATOC on the benchmark SIR, and the results showed that FATOC outperforms traditional SBFL significantly. Specifically, we improved efficiency from 6.98% to 10.32% in terms of metric $A\text{-}EXAM$. Besides traditional SBFL, our evaluation results also showed that our proposed approach can outperform the state-of-the-art MFL approach MSeer. Concerning the TOP-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 will investigate the influence of more negative factors (such as the coincidental correct test cases) on the accuracy of MFL. Besides, we are going to use more large-scale real-world faulty programs to verify the generalization of our empirical results.

[2]https://github.com/annoviko/pyclustering/, Sept. 2020.

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): Article No. 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] Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst M D, Pang D, Keller B. Evaluating and improving fault localization. In *Proc. the 39th IEEE/ACM Int. Conf. Software Engineering*, May 2017, pp.609-620.

[5] 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.

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

[7] Gopinath R, Jensen C, Groce A. The theory of composite faults. In *Proc. the 2017 IEEE Int. Conf. Software Testing, Verification and Validation*, March 2017, pp.47-57.

[8] Gao R, Wong W E. MSeer — An advanced technique for locating multiple bugs in parallel. *IEEE Transactions on Software Engineering*, 2019, 45(3): 301-318.

[9] 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.

[10] Liu B, Nejati S, Briand L, Bruckmann T. Localizing multiple faults in Simulink models. In *Proc. the 23rd IEEE Int. Conf. Software Analysis, Evolution, and Reengineering*, March 2016, pp.146-156.

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

[12] Chen Z, Chen Z, Zhao Z, Yan S, Zhang J, Xu B. An improved regression test selection technique by clustering execution profiles. In *Proc. the 10th International Conference on Quality Software*, July 2010, pp.171-179.

[13] Chen S, Chen Z, Zhao Z, Xu B, Feng Y. Using semi-supervised clustering to improve regression test selection techniques. In *Proc. the 4th IEEE Int. Conf. Software Testing, Verification and Validation*, March 2011, pp.1-10.

[14] Vangala V, Czerwonka J, Talluri P. Test case comparison and clustering using program profiles and static execution. In *Proc. 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.

[15] Dickinson W, Leon D, Fodgurski A. Finding failures by cluster analysis of execution profiles. In *Proc. the 23rd Int. Conf. Software Engineering*, May 2001, pp.339-348.

[16] 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.

[17] Mathias R, Lagrange M, Cont A. Efficient similarity-based data clustering by optimal object to cluster reallocation. *PLOS ONE*, 2018, 13(6): e0197450.

[18] Liu Y C, Li Z M, Xiong H, Gao X D, Wu J J. Understanding of internal clustering validation measures. In *Proc. the 2010 IEEE Int. Conf. Data Mining*, December 2010, pp.911-916.

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

[20] 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.

[21] Li Z, Wu Y H, Liu Y. An empirical study of bug isolation on the effectiveness of multiple fault localization. In *Proc. the 19th IEEE Int. Conf. Software Quality, Reliability and Security*, July 2019, pp.18-25.

[22] 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*. doi:10.1109/TSE.2019.2892102.

[23] 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*. doi:10.1109/TSE.2019.2948158.

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

[25] Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, 2011, 20(3): Article No. 11.

[26] Dallmeier V, Lindig C, Zeller A. Lightweight bug localization with AMPLE. In *Proc. the 6th Int. Symp. Automated Analysis-Driven Debugging*, September 2005, pp.99-104.

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

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

[29] Jaccard P. Etude de la distribution florale dans une portion des Alpes et du Jura. *Bulletin De La Societe Vaudoise Des Sciences Naturelles*, 2013, 37(142): 547-579. (in French)

[30] Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. In *Proc. the 24th Int. Conf. Software Engineering*, May 2002, pp.467-477.

[31] Rui A, Zoeteweij P, van Gemund A J C. An evaluation of similarity coefficients for software fault localization. In *Proc. the 12th Pacific Rim International Symposium on Dependable Computing*, December 2006, pp.39-46.

[32] Wong W E, Debroy V, Xu D. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems Man & Cybernetics*, 2012, 42(3): 378-396.

[33] 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.

[34] 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.

[35] Zakari A, Lee S P, Hashem I A T. A single fault localization technique based on failed test input. *Array*, 2019, 3/4: Article No. 100008.

[36] Abreu R, Zoeteweij P, van Gemund A J C. Spectrum-based multiple fault localization. In *Proc. the 2009 IEEE/ACM Int. Conf. Automated Software Engineering*, Nov. 2009, pp.88-99.

[37] Wong W E, Debroy V, Golden R, Xu X F, Thuraisingham B. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability*, 2012, 61(1): 149-169.

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

[39] Zakari A, Lee S P. Parallel debugging: An investigative study. *Journal of Software: Evolution and Process*, 2019, 31(11): Article No. e2178.

[40] 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 *Proc. the 41st IEEE/ACM Int. Conf. Software Engineering*, May 2019, pp.116-127.

[41] 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.

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

[43] Ester Martin, Kriegel H P, Sander J, Xu X W. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. the 2nd Int. Conf. Knowledge Discovery and Data Mining*, August 1996, pp.226-231.

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

[45] 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.

[46] Yu Z, Bai C, Cai K Y. Does the failing test execute a single or multiple faults? An approach to classifying failing tests. In *Proc. the 37th IEEE/ACM IEEE Int. Conf. Software Engineering*, May 2015, pp.924-935.

[47] Steimann F, Frenkel M, Abreu R. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proc. the 2013 Int. Symp. Software Testing and Analysis*, July 2013, pp.314-324.

[48] Li X, Zhang L M. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 2017, 1(OOPSLA): Article No. 92.

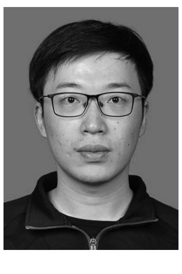
[49] Parnin C, Orso A. Are automated debugging techniques actually helping programmers? In *Proc. the 2011 Int. Symp. Software Testing and Analysis*, July 2011, pp.199-209.

[50] Prybutok V R. An introduction to statistical methods and data analysis. *Technometrics*, 1989, 31(3): 389-390.

[51] Perez A, Rui A, d'Amorim M. Prevalence of single-fault fixes and its impact on fault localization. In *Proc. the 2017 IEEE Int. Conf. Software Testing*, March 2017, pp.12-22.

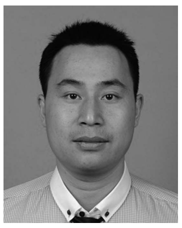
[52] Liu Y, Li Z, Zhao R, Gong P. An optimal mutation execution strategy for cost reduction of mutation-based fault localization. *Information Sciences*, 2017, 422: 572-596.

[53] Manish M, Yuriy B. Automatically generating precise oracles from structured natural language specifications localization. In *Proc. the 41st ACM/IEEE Int. Conf. Software Engineering*, May 2019, pp.188-199.



Yong-Hao Wu received his B.S. degree from Nanchang Hangkong University, Nanchang, in 2017. He is currently pursuing his Master's degree in computer science and technology at Beijing University of Chemical Technology, Beijing. His research interests are fault localization and software testing.

Zheng Li received his B.Sc. degree in computer science and technology from Beijing University of Chemical Technology, Beijing, in 1996. Then he received his Ph.D. degree from King's College London, CREST Centre in 2009. He is with the College of Information Science and Technology at Beijing University of Chemical Technology as a professor. His research interests are mainly in software engineering. Particularly, he is interested in program testing, source code analysis and manipulation. In these areas, he has published over 60 papers in referred journals or conferences, such as IEEE Transactions on Software Engineering, International Conference on Software Engineering, Journal of Software: Evolution and Process, IST, JSS, ICSM, ICSME, COMPSAC, SCAM and QRS.



Yong Liu received his B.Sc. and M.Sc. degrees in the computer science and technology from Beijing University of Chemical Technology, Beijing, in 2008 and 2011, respectively. Then he received his Ph.D. degree in control science and engineering from Beijing University of Chemical Technology, Beijing, in 2018. He is with the College of Information Science and Technology at Beijing University of Chemical Technology as an assistant professor. His research interests are mainly in software engineering. Particularly, he is interested in software debugging and software testing, such as source code analysis, mutation testing, and fault localization.

Xiang Chen received his B.Sc. degree in information management and system from Xi'an Jiaotong University, Xi'an, in 2002. Then he received his M.Sc. and Ph.D. degrees in computer software and theory from Nanjing University, Nanjing, in 2008 and 2011, respectively. He is with the School of Information Science and Technology at Nantong University, Nantong, as an associate professor. His research interests are mainly in software engineering. Particularly, he is interested in software maintenance and software testing, such as software defect prediction, combinatorial testing, regression testing, and fault localization. In these areas, he has published over 60 papers in referred journals or conferences, such as IEEE Transactions on Software Engineering, Information and Software Technology, Journal of Systems and Software, IEEE Transactions on Reliability, Journal of Software: Evolution and Process, Software Quality Journal, Journal of Computer Science and Technology, ASE, ICSME, SANER and COMPSAC.