PAPER

Boosting Spectrum-Based Fault Localization via Multi-Correct Programs in Online Programming

Wei ZHENG[†,††], **Hao HU**[†,††a)], **Tengfei CHEN**[†,††], **Fengyu YANG**[†,††], **Xin FAN**[†,††], ***and*** **Peng XIAO**[†,††], ***Nonmembers***

SUMMARY Providing students with useful feedback on faulty programs can effectively help students fix programs. Spectrum-Based Fault Location (SBFL), which is a widely studied and lightweight technique, can automatically generate a suspicious value of statement ranking to help users find potential faults in a program. However, the performance of SBFL on student programs is not satisfactory, to improve the accuracy of SBFL in student programs, we propose a novel Multi-Correct Programs based Fault Localization (MCPFL) approach. Specifically, We first collected the correct programs submitted by students on the OJ system according to the programming problem numbers and removed the highly similar correct programs based on code similarity, and then stored them together with the faulty program to be located to construct a set of programs. Afterward, we analyzed the suspiciousness of the term in the faulty program through the Term Frequency-Inverse Document Frequency (TF-IDF). Finally, we designed a formula to calculate the weight of suspiciousness for program statements based on the number of input variables in the statement and weighted it to the spectrum-based fault localization formula. To evaluate the effectiveness of MCPFL, we conducted empirical studies on six student program datasets collected in our OJ system, and the results showed that MCPFL can effectively improve the traditional SBFL methods. In particular, on the EXAM metric, our approach improves by an average of 27.51% on the Dstar formula.

key words: *fault localization, TF-IDF, assisted programming*

1. Introduction

The Online Judge (OJ) system is widely used in university programming courses to rigorously determine whether student-submitted programs meet the requirements of programming problems, such as test cases, time-consuming limitations, and occupied memory space limitations [1]. OJ systems usually rely on test results to check the correctness of program functionality. When a student submits a program, the OJ system obtains the actual output of the program and compares it with the expected output of the multiple test cases pre-set by the programming problem, and then returns the result of whether the test case passed to the student.

In programming practice, various errors occur in the students' programs. These errors can prevent the program from passing all test cases under the corresponding programming problem. However, the OJ system will simply return the result of whether the test case passed or not for these faulty programs, and does not suggest any changes or checks for the failed program. To find bugs in a faulty program, students need to spend a lot of time debugging the code. Therefore, the Fault Localization (FL) of programs is a major challenge for students [2].

The fault localization technique is a commonly used method by developers to focus on specific parts of the program. This technique usually generates a list of suspicious entities, and the entities most likely contain faults are ranked at the top of the list. Researchers have proposed a lot of methods for fault localization. Among them, Spectrum-Based Fault Localization (SBFL) is a lightweight fault localization method that can provide fast feedback. But traditional SBFL techniques cannot obtain satisfactory accuracy in student programs [3], as the SBFL cannot distinguish the suspicion of statements in one function block. Therefore, it is necessary to investigate a fault localization technique for the student programs and thus help students find the faults in the program quickly.

In this paper, we utilize correct programs in the OJ system and the TF-IDF (Term Frequency-Inverse Document Frequency) algorithm to improve the performance of SBFL. The idea is inspired by the classification algorithms in the field of machine learning [4]: for a programming problem, we have a collection of programs consisting of faulty programs and correct programs. It means that we have the label (i.e., pass or fail) and sample (i.e., programs), but we do not know which feature (i.e., statement) caused the program to fail. We would like to find such features [5] that cause the program to fail, which is the process of fault localization of the program. TF-IDF can analyze the importance of each term in the faulty program. When a term is important to the faulty program, it means that the term is less important to the other correct programs in the collection, and also means that the term is more likely to cause the faulty program to fail. Therefore, TF-IDF can be well applied to fault localization of student programs to improve the traditional spectrum-based fault localization.

Therefore, we propose the Multi-Correct Programs based Fault Localization (MCPFL) approach to localize the faults for student programs. Specifically, we collect all the corresponding correct programs based on the programming problem number and go through the code similarity to construct the Multi-Correct Program. In this way, the Multi-Correct Programs and the faulty program to be located can

Manuscript received August 15, 2023.
Manuscript revised October 29, 2023.
Manuscript publicized December 11, 2023.

[†]The authors are with the School of Software, Nanchang Hangkong University, Nanchang, China.
[††]The authors are with the Software Testing and Evaluation Center, Nanchang Hangkong University, Nanchang 330063, China.
a) E-mail: 1187413809@qq.com (Corresponding author)

DOI: 10.1587/transinf.2023EDP7164

form a new collection of programs. Then we calculate the importance of terms in the faulty program by the TF-IDF and analyze the importance of statements. Finally, MCPFL utilizes the importance of statements to improve the traditional SBFL.

The contributions of this paper are summarized as follows:

(1) We propose a novel fault localization approach MCPFL, which utilizes Multi-Correct Programs and TF-IDF to improve the accuracy of traditional SBFL.

(2) We design a formula for calculating the weight of the suspicious value of statements for MCPFL.

(3) We applied MCPFL to six real data sets in our OJ system. The experimental results show that the method outperforms the traditional SBFL in terms of EXAM and TOP-N metrics.

The rest of the paper is organized as follows. Section 2 introduces the background of the research and the related work in this paper. Section 3 describes in detail how our approach is implemented. Section 4 describes the experimental setup. Section 5 shows the experimental results, and Sect. 6 presents the threats to validity. Finally, Sect. 7 summarizes our study and discusses future work.

2. Background and Related Work

In this section, we will first briefly introduce the OJ system and explain the importance of the OJ system in practical teaching. After that, we will introduce the concept of spectrum-based fault localization and the related work of SBFL applied to student programs. Finally, the basic concept of the term frequency-inverse document frequency and the reasons for using it will be introduced.

2.1 Online Judge System

Online Judge (OJ) originated from the ACM International Collegiate Programming Competition (ACM-ICPC). It is an online program evaluation system with B/S architecture, which implements black-box testing [6]. Users log in to the system and submit the source code of the relevant topic, and the system will return the results of the evaluation instantly. Typically, the OJ system administrator prepares multiple test cases for each programming problem, which contain standard inputs and outputs. Once the OJ system receives the user-submitted programs, it will execute the source code and compare each of its outputs with the expected output. Finally, the OJ system will return the results of the execution of each test case. Although the OJ system is widely used by students, it lacks useful feedback to help students locate and fix bugs.

In this paper, we focus on the utilization of fault localization techniques in student programs and enabling it to assist students in online programming.

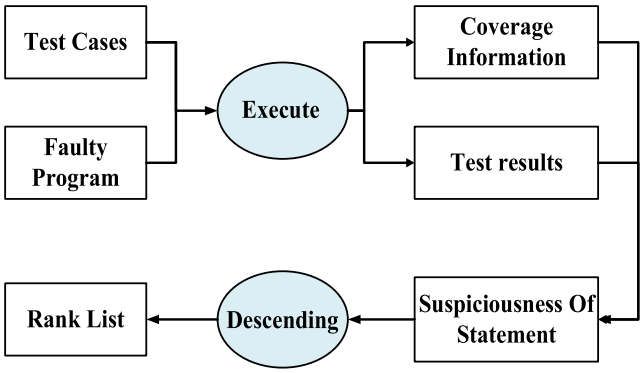


Fig. 1 Framework of traditional SBFL

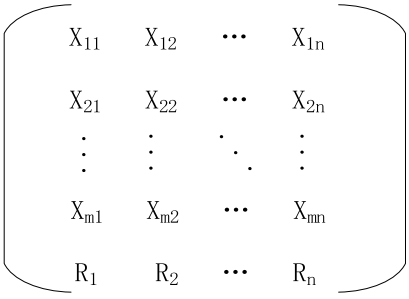


Fig. 2 Coverage matrix

2.2 Spectrum-Based Fault Localization

Researchers have proposed various fault localization techniques, such as Spectrum-based Fault Localization (SBFL) [7], Program slice-based Fault Localization [8], and Predicate-based Fault Localization [9]. Among them, SBFL is an effective and lightweight method that is widely used. The framework of traditional SBFL is illustrated in Fig. 1. SBFL takes the source code and test suite as input and outputs a ranked list of program statements, with the most suspicious ones at the top of the list. SBFL has three main processes as follows:

(1) Building a coverage matrix

The program executes all test cases and records the execution of program statements under each test case, thus constituting the program spectrum, which can be represented by a matrix (i.e., coverage matrix). A coverage matrix consisting of m statements and n test cases is shown in Fig. 2, X_{mn} is 1 if the program statement is executed by a test case, 0 otherwise. The test results are usually stored in the last row of the coverage matrix, R_n is 0 indicating that the test case passed and 1 indicating that the test case failed.

(2) Counting execution and coverage information

After constructing the coverage matrix, it is necessary to count the execution and coverage information of the program statement, which contains the following four main elements. It is crucial for calculating the suspicious value of statements.

ep: the number of passed test cases that cover the statement;

Table 1 The most commonly used suspicious formulas

Name	Formula
Tarantula	$Sus(s) = \frac{ef/(ef+nf)}{ef/(ef+nf)+np/(ep+np)}$
Jaccard	$Sus(s) = \frac{ef}{ef+nf+ep}$
$Dstar^*$[13]	$Sus(s) = \frac{ef^*}{ep+np}$
Ochiai	$Sus(s) = \frac{ef}{\sqrt{(ef+nf)\times(ef+ep)}}$
Op2[14]	$Sus(s) = ef - \frac{ep}{ep+np+1}$

ef: the number of failed test cases that cover the statement;

np: the number of passed test cases that do not cover the statement;

nf: the number of failed test cases that do not cover the statement.

(3) Calculating suspicious values

Based on the information in (2), the SBFL formula can be used to calculate the suspicious values of the program statements. Finally, these statements can be sorted according to the suspicious value. In this list of suspicions, the higher the statement's ranking on the list, the more likely it is to be a faulty statement. As a result, SBFL can more easily assist developers in debugging.

The suspicious values of program statements calculated using different spectrum fault localization formulas vary, and the most classical fault localization method based on program spectrum is Tarantula [10]. In recent years, researchers have successively proposed many similar fault localization methods. Among them, Aberu et al. successively proposed Jaccard [11] and Ochiai [12] and experimentally proved that the approach of Ochiai is better than Tarantula and Jaccard. The commonly used spectrum-based fault localization formulas are shown in Table 1.

SBFL is a lightweight fault localization method that provides fast feedback. Therefore, it is widely used in fault localization in student programs. Eliane Araujo [15] and Yuxing Liu [16] both applied SBFL to student programs and conducted empirical studies, and the experiments showed that SBFL was able to locate faults in student programs. Researchers have proposed new approaches to further improve the effectiveness of SBFL in student programs. Zheng Li [17] proposes to use fault statement category frequencies to improve the effectiveness of SBFL in student programs. Qusay Idrees Sarhan [18] proposed to utilize code elements to improve the performance of traditional SBFL methods. In summary, existing research has focused only on faulty programs without considering the correct program. In contrast to these studies, this paper analyzes the importance of the term in the faulty program through the correct programs and further analyzes the suspiciousness of the statements in the faulty program to improve the SBFL methods.

2.3 TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistical technique [19] that can be used to evaluate the importance of a term in a document to that document and is a common term weighting method used in information retrieval and text mining [20]. The main idea of TF-IDF is that for a document set in which a term in one document occurs more often in that document and less often in other documents in the set, it can be considered that the term can be well distinguished between that document and other documents and can be well used to achieve classification [21].

For online programming, Programs in an OJ system can be treated as documents, and a classified set of programs can be constructed by combining multiple correct programs in the system and a faulty program to be located. TFIDF can analyze the importance of each term in the faulty program. When the term is important for the faulty program, it means that the term is less important for the other correct programs in the program set, and it also means that the possibility that the term causes the faulty program to fail. Therefore, in this paper, we will utilize TF-IDF to improve the traditional SBFL.

3. Proposed Approach

The framework of the Multi-Correct Programs based fault localization (MCPFL) proposed in this paper is shown in Fig. 3. First, the faulty program is executed by all the test cases to get the coverage matrix of the program statements. The initial suspicious value of the program statements is obtained according to the traditional SBFL Next, we collect correct programs based on programming problem numbers and construct Multi-Correct Programs based on code similarity, and the faulty programs are added to Multi-Correct Programs to form a new set of programs. Finally, we analyze the composition of statements and utilize the TF-IDF algorithm to obtain the importance of the term in the faulty program, and then weight the initial statement suspicious values to generate the final ranked list of statement suspicious values.

3.1 Constructing Multi-Correct Programs

Definition 1. Multi-Correct Programs: For a collection of correct programs, the remaining correct programs after removing highly similar correct programs using code similarity are called Multi-Correct Programs.

First, we get all the correct programs from the OJ system based on the programming problem number. Among all the correct programs collected, there are a large number of programs with the same program structure, which can have an

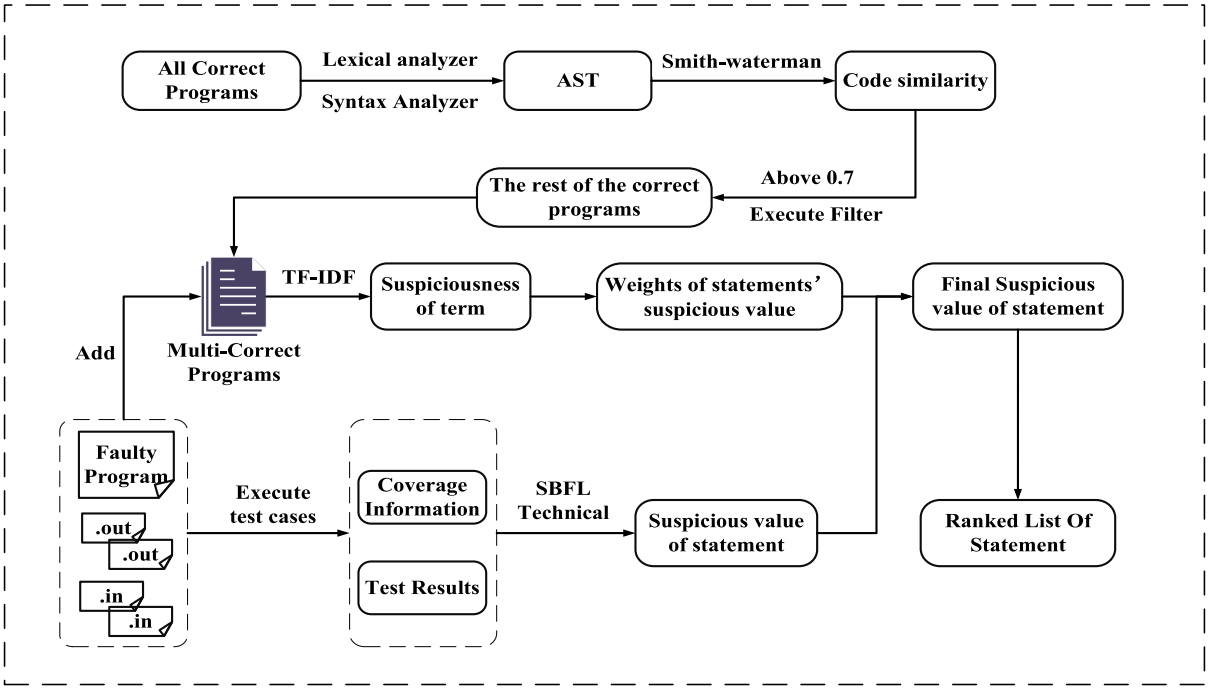


Fig. 3 Framework of MCPFL

impact on the subsequent calculation of suspicious values of the term. To filter such programs, we utilize AST-based code similarity to calculate the similarity between codes [22], because AST can reflect both the structural information of the programs and retain the attribute features in the programs. The main process is to transform the program into an AST and then use the Smith-waterman algorithm to solve for the maximum set of matches between two sequences and calculate the similarity between them [23]. When the similarity is below a preset threshold, the correct program is retained. Finally, we will get Multi-Correct Programs corresponding to the programming problem. The algorithm is shown in Algorithm 1.

Algorithm 1 Construction Of Multi-Correct Programs

Function: multi_Correct_Pro(code_pro,code_ast)

Input: code_pro:All Correct Programs;

code_ast: The abstract syntax tree of the correct programs;

Output: code_pro: Multi-Correct Program

```
one=code_ast[0]
for item in code_ast[1:] do
    smith_waterman=(2*SLength(one,item))/
    if smith_waterman<=0.7 then
        index.append(code_ast.index(item))
    end if
end for
for ele in range(len(code_pro)) do
    if ele not in index then
        code_pro.pop(ele)
    end if
end for
return code_pro
```

3.2 Suspiciousness of Term

Adding a faulty program to a Multi-Correct Programs set constitutes a new program set (i.e., containing a faulty program and Multiple-Correct Programs). In this way, the tf-idf values of the term in the faulty program obtained by the TF-IDF algorithm are significant for fault localization. The tf-idf value indicates the importance of a term in the text to the text and can be used for text classification. Since the program in our program sets already has the label (Pass or Fail). Therefore, the tf-idf value of the term in the faulty program can be considered as the probability that the term causes the program to fail, which means the suspicious value.

The implementation of TF-IDF requires the computation of the value of tf and the value of idf. The tf is calculated as shown in Eq. (1).

$$tf_{i,j} = n_i \tag{1}$$

The n_i in the formula is the number of times term i appears in program j. The reason for not dividing by the total number of all terms in program j is to prevent the calculation of some terms from being small when the term count of the program is too large. For the idf is calculated as shown below.

$$idf_i = 1 + \left(log\frac{Y+1}{Y_i+1}\right) \tag{2}$$

where Y is the total number of programs in the program set, and Y_i is the number of programs in which term i occurs. Unlike the traditional idf formula, both the numerator and denominator are smoothed. This has the advantage of preventing zero splitting, but it may also result in an idf value of 0. Therefore, an addition of 1 is also performed outside of the formula to prevent the idf result from being 0. So the tf-idf is calculated as shown in Eq. (3).

$$tf - idf_{i,j} = n_i \times \left(1 + log\frac{Y+1}{Y_i+1}\right) \tag{3}$$

Finally, the original tf-idf values obtained are subjected to L2-parametric normalization, as shown in Eq. (4).

$$tf - idf_{i,j_{norm}} = \frac{tf - idf_{i,j}}{||v||_2} = \frac{tf - idf_{i,j}}{\sqrt{v_1^2 + v_2^2 + \cdots v_n^2}} \tag{4}$$

where v is the vector mapped by the program j and $tf - idf_{i,j}$ is the initial tf-idf value of term i in program j. After normalization, we can map the tf-idf values of the term between 0 and 1. Then we can get the valid tf-idf values of the term.

3.3 The Allocation of Weights

As we all know, the result of fault localization is a sorted list of statement suspicious values. It is not enough to analyze the importance of the term, we need to combine the tf-idf values of each term in the statement to represent the importance of the statement. For each program, the branch structure, the assignment of expressions, and the input and output of statements are directly or indirectly related to the input variables. Based on such characteristics, this paper calculates the significance of the suspicious value for each statement based on the number of different input variables in the statement. For terms in statements, we divide them into three main categories: (1) input variables; (2) data types, other variables, keywords, and function names; (3) content in formatted input and output (i.e., the term in double quotes). Therefore, in this paper, for the number of different input variables in the statement, a formula is designed to calculate the weights of the statement's suspicious values, as shown in Eq. (5).

$$\text{Weights(S)} = \begin{cases} \dfrac{y_{max} + \sum_1^i x_i}{i+1} & 1 \le i \le n \\ y_{max} & i = 0 \end{cases} \tag{5}$$

Table 2 TF-IDF values in the faulty program

term	tf-idf	term	tf-idf	term	tf-idf
int	0.10	scanf	0.06	s	0.37
main	0.06	sqrt	0.06	area	0.25
a	0.49	printf	0.12	valid	0.06
b	0.43	These	0.06	triangle	0.06
c	0.43	sides	0.06	to	0.06
double	0.06	do	0.06	correspond	0.06
if	0.06	not	0.06	return	0.08

where i is the number of different input variables in statement S, n is the number of input variables required by the programming problem, x is the tf-idf value of the input variable, and y_{max} is the maximum tf-idf value of data type, other variables, keywords, and function names in statement S. The formula does not take into account the content in formatted input and output, since the input variables will not appear in them directly, only as placeholders. Otherwise, it's an illegal statement and won't compile. Finally, we can get the statement's suspicious value weight.

In this way, we can recalculate the suspicious value of the statement, as shown in Eq. (6).

$$Sus_{final}(s) = Sus_{initial}(s) \times \text{Weight(s)} \tag{6}$$

Where $Sus_{initial}(s)$ is the initial suspicious value of statements calculated by the traditional SBFL method, Weight(s) is the weight of suspicious value, and $Sus_{final}(s)$ is the final statement's suspicious value.

3.4 An Illustrative Example

In this subsection, the proposed approach is demonstrated by a sample program. The programming problem is described as follows: input three sides of a triangle a, b, c, and determine whether the triangle can be formed, and if so, output the area of the triangle, otherwise output "These sides do not correspond to a valid triangle". A standardized input and output format is given at the end of the programming problem for students' reference, sample input: 5 5 3, sample output: area = 7.15. The question sets up six test cases, which are (5 5 3), (1 4 1), (2 2 4), (4 2 2), (2 4 2), and (5 3 4).

The faulty program is shown in Fig. 4. Line 7 is the fault statement, which causes the variable s to lose precision and should be modified to s = (a + b + c)/2.0. For this programming problem, we use two correct programs as the Multi-Correct Program to explain our proposed approach, the program as shown in Fig. 5. The final results are shown in Table 3. The specific details are as follows.

We form a simple collection of programs from a faulty

Table 3 An illustrative example

	t1	t2	t3	t4	t5	t6	ep	ef	np	nf	weight(S)	Tarantula	Rank	MCPFL	Rank
S3	1	1	1	1	1	1	4	2	0	0	0.1	0.5	9	0.05	8
S4	1	1	1	1	1	1	4	2	0	0	0.3625	0.5	9	0.18125	6
S5	1	1	1	1	1	1	4	2	0	0	0.37	0.5	9	0.185	4
S6	1	1	1	1	1	1	4	2	0	0	0.3675	0.5	9	0.18375	5
S7	1	1	1	1	1	1	4	2	0	0	0.43	**0.5**	**9**	**0.215**	**3**
S8	1	1	1	1	1	1	4	2	0	0	0.3525	0.5	9	0.17625	7
S9	1	0	0	0	0	1	0	2	4	0	0.43	1	2	0.43	1
S10	1	0	0	0	0	1	0	2	4	0	0.25	1	2	0.25	2
S11	0	0	0	0	0	0	0	0	4	2	\	\	\	\	\
S12	0	1	1	1	1	0	4	0	0	2	0.12	0	10	0	10
S13	1	1	1	1	1	1	4	2	0	0	0.08	0.5	9	0.04	9
result	1	0	0	0	0	1									

```c
#include<stdio.h>
#include<math.h>
int main(){
    int a,b,c=0;
    double s,area=0;
    scanf("%d %d %d",&a,&b,&c);
    s=(a+b+c)/2;    //Bug  s=(a+b+c)/2.0
    if(a+b>c&&b+c>a&&a+c>b){
      area=sqrt(s*(s-a)*(s-b)*(s-c));
      printf("area=%.2f",area);}
    else
      printf("These sides do not correspond to a valid triangle");
return 0;}
```

Fig. 4 A faulty program

```c
#include<stdio.h>
#include<math.h>
int main(){
    int a,b,c;
    scanf("%d %d %d",&a,&b,&c);
    if((a+b)>c&&(b+c)>a&&(a+c)>b)
    {
      float area=0,s;
      s=(a+b+c)/2;
      area=sqrt(s*(s-a)*(s-b)*(s-c));
      printf("area=%.2f",area);
    }
    else
      printf("These sides do not correspond to a valid triangle");
return 0;
}
```

```c
#include<stdio.h>
#include<math.h>
int main()
{
    double a,b,c;
    double s=0,area=0;
    scanf("%lf %lf %lf",&a,&b,&c);
    if((a+b<=c)||(a+c<=b)||(b+c<=a))
    {
      printf("These sides do not correspond to a valid triangle");
      return 0;
    }
    s=0.5*(a+b+c);
    area=sqrt(s*(s-a)*(s-b)*(s-c));
    printf("area=%.2lf",area);
    return 0;
}
```

Fig. 5 Multi-correct programs

program (in Fig. 4) to be located and Multiple-Correct-programs (in Fig. 5) corresponding to the programming problem. At this point, what we need to analyze is the importance of the term in the faulty program, which can be thought of as the likelihood that the term causes the program to fail. According to Eqs. (1)–(4), we can calculate the tf-idf values corresponding to all the terms in the faulty program, as shown in Table 2. We aim to analyze the likelihood of a statement causing a program failure rather than the term. therefore, we need to apply Eq. (5), which is based on the tf-idf values in Table 2 to calculate the statement suspiciousness weight, the results are shown in column 8 of Table 3.

In addition, we need to execute all the test cases on the faulty program to get its coverage information, as shown in columns 2–6 of Table 3, and then we calculate the four metrics ep, ef, np, nf according to the method introduced in Sect. 2.2, and the results are shown in columns 8–11 of Table 3. We select the tarantula formula in Table 1 to calculate the suspicious value of each statement, the results are shown in column 13 of Table 3. Finally, we use formula (6) to weigh the suspicious values of statements calculated by the Tarantula method, and the results are shown in column 15 of Table 3.

In this case, the suspicious value of the faulty statement calculated using the Tarantula formula is 0.5, which is

Table 4 Information on the programming problem

Pro_ID	Programs	Faulty	Multi-Correct Programs	Testcases	Description
A	21	33	6	6	Output Grade Level
B	14	18	4	4	Calculate the cost of electricity
C	14	19	4	5	Determining if it is a leap year or not
D	22	24	5	6	output perimeter and area of the triangle
E	23	43	4	5	Solve a system of quadratic equations
F	12	14	4	4	Solve the approximate solution of e

ranked ninth in the ranked list of suspicious values, and the suspicious value of the faulty statement obtained using the MCPFL approach is 0.2175, which is ranked third. Comparing columns 14 and 16 in Table 3, it can be seen that our approach can effectively solve the problem of a large number of statements with the same suspicious value and improve the accuracy of traditional SBFL.

4. Experiment Design

4.1 Subject Programs

In this paper, we selected 864 C programs submitted by 144 students in four classes for six programming problems in our online programming system as the object of this experiment. After the data object is obtained, the following steps need to be done [24]. (1) We need to remove perfectly correct programs, some faulty programs that failed to compile, and those that failed at all test points (i.e., the spectrum-based fault localization approach could not be applied) to obtain valid experimental data. (2) We need to perform data cleansing, including removing code annotation from the program, splitting a line of code with multiple statements, and removing blank lines from the program. (3) The last and most critical step is to hand over the cleaned data to experienced professors who will review the faulty procedures and mark the locations of errors. The information on the dataset is shown in Table 4 Where the first column is the problem ID, the second column is the number of faulty programs collected, the third column corresponds to the number of faults for each question, the fourth column corresponds to the Multi-Correct Program for each question, the fifth column is the number of test case, and the last column is a brief description of the programming problem. There are 106 programs from problems A to F. These programs, test cases, and spectrum information have been opened on GitHub. The access link is https://github.com/2304624469/dataSet.

4.2 Evaluation Metrics

In this experiment, we use the metrics Exam and TOP-N to evaluate the effectiveness of the proposed approach on the student program dataset. They are defined as follows:

(1) EXAM Score: The EXAM value indicates the percentage of statements that need to be checked when locating all faulty statements [25] and is calculated as shown in the formula (7):

$$\text{EXAM} = \frac{n}{N} \times 100\% \tag{7}$$

where n denotes the number of statements to check when locating a bug and N denotes the total number of statements in the program. Therefore, when the EXAM value is smaller, it means that the developer needs to check fewer statements, which means that the fault in the program can be located faster and the fault localization approach is more effective.

However, a faulty program may include multiple error statements, and each faulty statement will have an EXAM value. To solve this problem, we evaluate the effect of fault localization by selecting only the EXAM value of the first faulty statement in the sorted list of statement suspicious values. This is also a widely adopted strategy now [26].

(2) TOP-N: The TOP-N metric describes the number of programs in which errors can be found by checking only the first N (N = $1, 2, 3 \ldots$) statements of the ranked list of suspiciousness [27]. Therefore, the higher the value of TOP-N, the more effective the fault localization is for a certain number of check statements. For the TOP-N metric, a faulty statement is generally considered to be excellent if it ranks first in the ranking list, good if it ranks in the top 5, acceptable if it ranks in the top ten, and ineffective if it ranks outside the top 10.

For the evaluation of TOP-N on multi-fault programs. We consider this fault localization as successful if a faulty statement is found among the TOP-N suspicious statements. For statements with the same suspicious value, we still take the average rank as the final rank. For example, for a program containing two faulty statements, there are M statements with the same suspicious value as faulty statement 1, and there are K statements with the same suspicious value as faulty statement 2 and a higher suspicious value than statement 1. At this point, we consider that checking out two faulty

statements requires checking K + M/2 statements on average. Thus for a program with two faulty statements, its TOP-N metric becomes TOP-(K + M/2).

4.3 Evaluation Methods

To evaluate the effectiveness of the proposed approach, five traditional spectrum-based fault localization approaches, Tarantula, Jaccard, Dstar, Ochiai, and Op2, were selected as the baseline methods for the experiments.

4.4 Research Question

To evaluate the effectiveness of MCPFL on student programs' fault localization, we conducted a series of experiments, and the following research questions are defined and investigated:

RQ1: Does MCPFL improve the effectiveness of traditional SBFL technical?

RQ2: How well does the MCPFL technique place fault statements at low ranks?

RQ3: Is the MCPFL technique effective for every fault?

RQ4: How does MCPFL improve traditional SBFL technical?

5. Result Analysis

5.1 RQ1: Does MCPFL Improve the Effectiveness of Traditional SBFL Methods?

To answer RQ1, we compared the fault localization accuracy of MCPFL with the traditional SBFL on five suspiciousness formulas. Furthermore, we also used EXAM metrics to evaluate the effectiveness of our approach and show the detailed results in Table 5 and Fig. 6.

Table 5 lists the EXAM values of the traditional SBFL method and MCPFL for the six problems, with the last column shows the improvement effect of our approach. and the bottom five rows indicate the average EXAM value of all questions. As shown in Table 5, the MCPFL reduced the average EXAM by approximately half for all questions on the five suspiciousness formulas, and the improvement of MCPFL ranges from 8.54% to 42.86% in terms of EXAM. Overall, MCPFL improves the most on question F, because SBFL is not effective on question F, the EXAM value is about 70%, and MCPFL can reduce it to about 30%. Figure 6 further intuitively illustrates the results in Table 5.

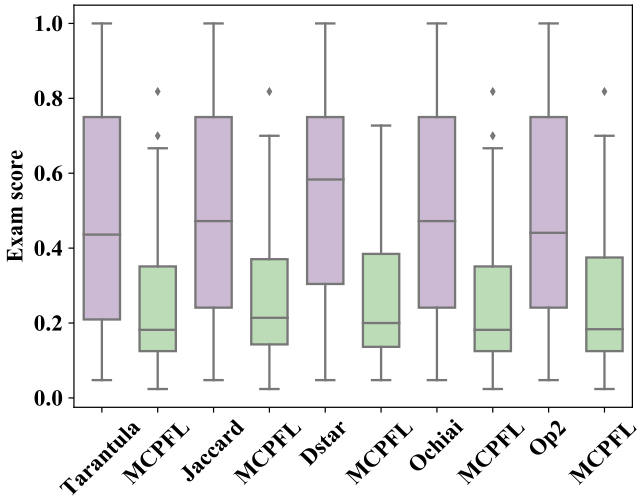


Fig. 6 MCPFL comparison between different techniques in terms of EXAM metric

Table 5 Fault localization performance comparison with different technical

Pro_ID	Formula	EXAM(%)		Improve(%)
		SBFL	MCPFL	
A	Tarantula	32.11%	16.33%	**15.78%**
	Jaccard	29.99%	15%	**14.99%**
	Dstar	32.39%	15.67%	**16.72%**
	Ochiai	29.99%	15%	**14.99%**
	Op2	29.99%	15%	**14.99%**
B	Tarantula	61.53%	31.59%	**29.94%**
	Jaccard	66.67%	31.63%	**35.04%**
	Dstar	65.54%	31.14%	**34.40%**
	Ochiai	66.67%	29.78%	**36.90%**
	Op2	64.88%	32.78%	**32.11%**
C	Tarantula	63.04%	28.69%	**34.35%**
	Jaccard	63.04%	32.99%	**30.04%**
	Dstar	65.09%	32.87%	**32.21%**
	Ochiai	63.04%	30.49%	**32.54%**
	Op2	66.40%	33.59%	**32.81%**
D	Tarantula	55.43%	27.71%	**27.71%**
	Jaccard	54.92%	27.94%	**26.98%**
	Dstar	59.03%	27.65%	**31.38%**
	Ochiai	55.43%	28.47%	**26.96%**
	Op2	55.43%	28.88%	**26.54%**
E	Tarantula	27.48%	18.02%	**9.46%**
	Jaccard	29.36%	20.82%	**8.54%**
	Dstar	29.10%	18.76%	**10.34%**
	Ochiai	29.36%	20.63%	**8.72%**
	Op2	28.77%	19.68%	**9.10%**
F	Tarantula	74.57%	31.66%	**42.86%**
	Jaccard	74.51&	32.25%	**42.26%**
	Dstar	74.51%	34.52%	**39.99%**
	Ochiai	74.51%	32.25%	**42.26%**
	Op2	69.15%	31.21%	**37.94%**
Average	Tarantula	52.35%	25.66%	**26.68%**
	Jaccard	53.08%	26.77%	**26.31%**
	Dstar	54.28%	26.77%	**27.51%**
	Ochiai	53.17%	26.10%	**27.06%**
	Op2	52.44%	26.86%	**25.58%**

Table 6 Comparison of successfully localized faults with only the Top N for SBFL and MCPFL

Technical	Pro_ID	Top-1		Top-3		Top-5	
		SBFL	MCPFL	SBFL	MCPFL	SBFL	MCPFL
Tarantula	A	5	**8**	12	**23**	22	**33**
	B	0	**4**	5	**7**	12	**16**
	C	2	**4**	6	**14**	15	**19**
	D	1	**8**	11	**14**	19	**20**
	E	4	**6**	20	**24**	27	**33**
	F	1	**2**	4	**9**	5	**12**
	Total	13	**32**	58	**91**	100	**133**
Jaccard	A	8	**9**	14	**26**	28	**33**
	B	2	**3**	2	**9**	10	**14**
	C	**3**	2	6	**11**	14	**16**
	D	1	**8**	10	**13**	20	**20**
	E	5	**7**	14	**15**	23	**28**
	F	1	**2**	4	**8**	5	**12**
	Total	20	**31**	50	**82**	95	**123**
Dstar	A	1	**2**	8	**19**	19	**25**
	B	0	**4**	3	**8**	11	**12**
	C	**2**	1	2	**10**	12	**15**
	D	0	**5**	4	**8**	14	**15**
	E	1	**5**	9	**13**	12	**19**
	F	0	**1**	4	**6**	5	**12**
	Total	4	**18**	30	**64**	73	**98**
Ochiai	A	7	**9**	14	**26**	27	**33**
	B	2	**4**	2	**11**	10	**16**
	C	3	**4**	7	**12**	14	**15**
	D	2	**9**	11	**13**	19	**21**
	E	4	**7**	15	**20**	25	**28**
	F	1	**2**	4	**5**	5	**12**
	Total	19	**35**	53	**87**	100	**126**
Op2	A	7	**9**	15	**26**	29	**33**
	B	2	**5**	2	**9**	11	**12**
	C	2	2	3	**9**	14	**16**
	D	1	**8**	11	**13**	18	**21**
	E	4	**5**	11	**21**	20	**28**
	F	1	**2**	4	**7**	5	**12**
	Total	17	**31**	46	**85**	97	**122**
Average		15	**29**	47	**82**	93	**120**

5.2 RQ2: How Well Does the MCPFL Technique Place Fault Statements at Low Ranks?

To answer RQ2, we compare the performance of the MCPFL technique with that of the SBFL technique using only the Top N entries of the generated ranked lists. The SBFL and the MCPFL columns indicate the number of faults found within the Top 1, 3, and 5 lines. Detailed results are shown in Table 6.

As shown in Table 6, in some cases, the number of faults found within the Top N is smaller than or equal to the SBFL techniques. Overall, however, the performance of the MCPFL technique improves dramatically in our student program dataset. For example, in Tarantula, the MCPFL technique localized 32 out of 151 faults in Top 1, while the Tarantula technique localized only 13 faults. the MCPFL technique include 91and 133 faults in the Top-3 and Top-5 respectively. The rates of increase of Top-1, Top-3 and Top-5 were 146%, 57%, 33%. In addition, MCPFL included 1 less fault in the Top-1 compared with Jaccard and Dstar for C problems, and the same number of faults in Top-1 compared with Op2. Compared with Ochiai, MCPFL was improved in Top-1, Top-3, and Top-5.

On average, the MCPFL technique outperforms the five

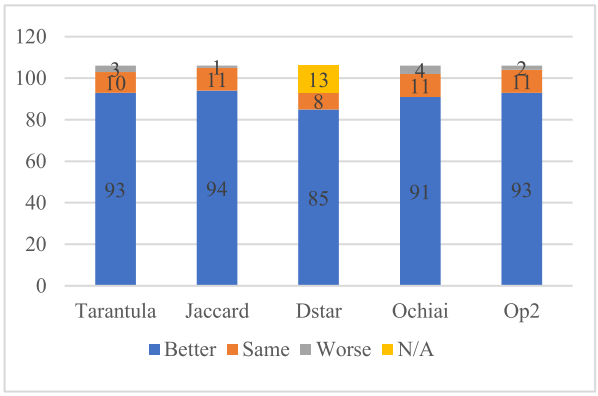


Fig. 7 Change of rank between SBFL and MCPFL

SBFL techniques. The MCPFL extends the number of faults included in the Top-1 from 15 to 29 (about 93%). In addition, the faults included in Top-3 were expanded by about 74% from 47 to 82. the faults included in Top-5 were expanded by about 29% from 93 to 120.

5.3 RQ3: Is the MCPFL Technique Effective for Every Fault?

To answer RQ3, we compare the SBFL and the MCPFL ranks of actual fault locations by applying the SBFL and the MCPFL techniques to 106 programs. Figure 7 indicates the results for 106 student programs. the x-axis is five SBFL technical, and the Y-axis is the number of programs corresponding to the application of MCPFL to different SBFL technologies. Better defines the number of programs whose rank of faults found by the MCPFL technique is lower than the rank of faults found by the SBFL technique. Same means that the number of programs whose rank of faults found by the SBFL technique and the number of programs whose rank of faults found by the MCPFL technique are the same. Worse defines the number of programs whose rank of faults found by the MCPFL technique is higher than the rank of faults found by the SBFL technique. N/A defines the number of programs that cannot be found by the MCPFL technique.

For example, in Tarantula, the MCPFL technique outperforms it in 93 of 106 programs (about 88%). In addition, the accuracy is the same or worse in 10 (about 9%) and 3 (3%) of the programs, respectively.

The accuracy depends on the SBFL technical. However, The MCPFL technique indicates a higher accuracy than the SBFL technique in an average of 91 programs (about 86%). Also, the number of programs with the same and worse accuracy is 10 (9%) and 2 (2%), respectively. the number of Programs with N/A accuracy is 13 (about 12%). It only appears on Dstar because when Dstar deals with real student programs, it has a situation where it can't locate the fault, so MCPFL weighting to Dstar still cannot localize the fault.

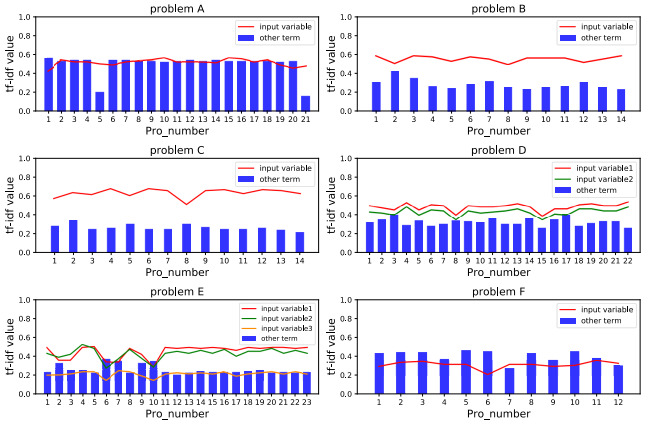


Fig. 8 The tf-idf value for the term in the faulty program under the Multi-Correct Programs

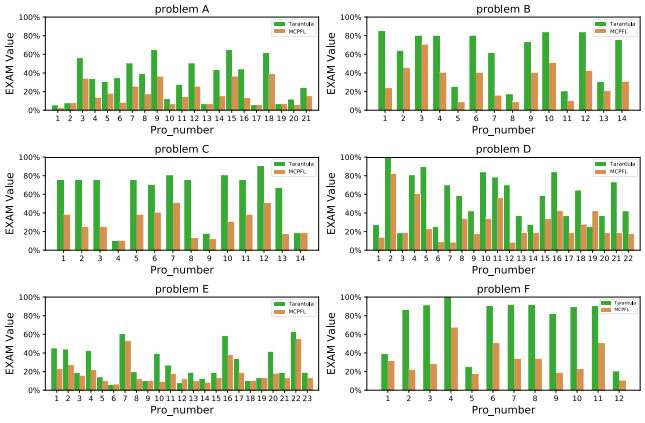


Fig. 9 Comparison of Tarantula and MCPFL on 106 programs in terms of EXAM metrics

5.4 RQ4: How Does MCPFL Improve Traditional SBFL Technical?

In RQ1 and RQ2, we know that the MCPFL improves the SBFL. In RQ4, we aim to explain how the MCPFL improves the SBFL. After introducing the Multi-Correct Programs, we can analyze the importance of the term in the faulty program, which is important for fault localization. To show the experimental results clearly, we have analyzed the term of all 106 fault programs, as shown in Fig. 8. The input variable represents the tf-idf value of the input variable. Other term is the range of tf-idf values for all terms in the fault program except for the input variable. We can observe that regardless of whether the faulty program contains one input variable, two input variables, or three input variables, the input variables in the program all have high tf-idf values. Therefore, in Sect. 3.3, we treated the input variables as key terms and defined a weighting rule. We utilized the weighting rule to assign weights to traditional SBFL methods, which resulted in the final fault localization outcomes. To provide a more intuitive presentation of the results, we compared the Tarantula method with our approach on 106 programs using the

exam metric. The experimental results are shown in Fig. 9. Among the 106 programs, there were three programs where the effectiveness of MCPFL was worse than Tarantula. These programs are Program 19 in Problem D, and Programs 6 and 12 in Problem E. This is because even though the tf-idf values of input variables are important, we still need to take into account the importance of the other terms when weighting them as statement suspiciousness, which may result in a decrease in the final fault localization effect.

6. Threats to Validity

Threats to External Validity: The threat of external validity is related to the dataset of this paper. The dataset was initially unlabeled for faulty statements. Therefore, we had to manually validate the program for bugs and mark the locations of faulty statements. To avoid possible errors, we invited several experienced programming students and teachers (including the authors of this paper) to check the marked bugs.

Threats to Internal Validity: The threat to internal validity is related to the implementation details of our proposed MCPFL, which requires Multi-Correct Programs. In general, the programming exercises for freshmen are mostly the same as in previous years, but there are very few problems that are different, and for such programming problems, the absence of functionally identical Multi-Correct Programs in the OJ system can lead to the failure of the MCPFL. The programming problems selected in this experiment are all without limitations, but in practice, the lack of Multi-Correct Programs could make our approach ineffective.

Threats to Construct Validity: The threat of construct validity is related to our experimental metrics. To mitigate this threat, we use two metrics, EXAM, and TOP-N, in our experiments to evaluate the execution results of fault localization. They were widely used in earlier studies [28].

7. Conclusion and Future Work

In this paper, we propose a lightweight fault localization approach MCPFL to localize student programs. We collected all the failed programs in the six programming problems in our university's OJ system as the dataset for this experiment and evaluated the effectiveness of our proposed approach on the dataset. The results show that our proposed approach is effective in improving the accuracy of traditional fault localization methods.

In the future, we implement additional methods to further improve the effectiveness of fault localization in student programs. In future work, we will do more research to further improve the fault localization accuracy of student programs, which includes but is not limited to: (1) using interpretable machine learning methods for fault localization [29]. (2) using causal inference to improve fault localization, and (3) using students' historical version program information for fault localization.

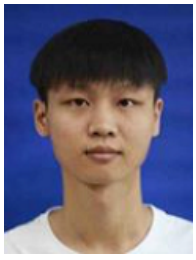
References

[1] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," ACM Computing Surveys (CSUR), vol.51, no.1, pp.1–34, 2018.
[2] W.E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," IEEE Trans. Softw. Eng., vol.42, no.8, pp.707–740, 2016.
[3] E. Araujo, M. Gaudencio, D. Serey, and J. Figueiredo, "Applying spectrum-based fault localization on novice's programs," 2016 IEEE Frontiers in Education Conference (FIE), IEEE, 2016, pp.1–8.
[4] Z.-H. Zhou, "Machine learning," Springer Nature, 2021.
[5] J. Cai, J. Luo, S. Wang, and S. Yang, "Feature selection in machine learning: A new perspective," Neurocomputing, vol.300, pp.70–79, 2018.
[6] W. Zhou, Y. Pan, Y. Zhou, and G. Sun, "The framework of a new online judge system for programming education," Proc. ACM turing celebration conference-China, pp.9–14, 2018.
[7] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. van Gemund, "A practical evaluation of spectrum-based fault localization," Journal of Systems and Software, vol.82, no.11, pp.1780–1792, 2009.
[8] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating faults with program slicing: an empirical analysis," Empirical Software Engineering, vol.26, no.3, 2021.
[9] Y. Küçük, T.A.D. Henderson, and A. Podgurski, "Improving fault localization by integrating value and predicate based causal inference techniques," 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp.649–660, 2021.
[10] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," Proc. 24th international conference on Software engineering, pp.467–477, 2002.
[11] R. Abreu, P. Zoeteweij, and A.J.c. Van Gemund, "An evaluation of similarity coefficients for software fault localization," 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06), pp.39–46, 2006.
[12] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, "On the accuracy of spectrum-based fault localization," Testing: Academic and industrial conference practice and research techniques-MUTA TION (TAICPART-MUTA TION 2007), pp.89–98, 2007.
[13] W.E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," IEEE Trans. Rel., vol.63, no.1, pp.290–308, 2013.
[14] L. Naish, H.J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," ACM Transactions on software engineering and methodology (TOSEM), vol.20, no.3, pp.1–32, 2011.
[15] E. Araujo, M. Gaudencio, D. Serey, and J. Figueiredo, "Applying spectrum-based fault localization on novice's programs," 2016 IEEE Frontiers in Education Conference (FIE), pp.1–8, 2016.
[16] Y. Liu, Z. Zhang, X. Zhou, and W. Liu, "An Empirical Study on Spectrum-Based Fault Localization for Student Programs," 2023 3rd International Symposium on Computer Technology and Information Science (ISCTIS), pp.547–551, 2023.
[17] Z. Li, J. Shen, Y. Wu, Y. Liu, and Z. Sun, "VSBFL: Variable Value Sequence Based Fault Localization for Novice Programs," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp.494–505, 2021.
[18] Q.I. Sarhan and Á Beszédes, "Poster: Improving Spectrum Based Fault Localization For Python Programs Using Weighted Code Elements," 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), pp.478–481, 2023.
[19] J. Leskovec, A. Rajaraman, and J.D. Ullman, Mining of Massive Datasets, Cambridge University Press, 2020.
[20] S.M. Weiss, N. Indurkhya, and T. Zhang, "Information retrieval and text mining," Fundamentals of Predictive Text Mining, pp, 75–90, 2010.
[21] K. Kowsari, K.J. Meimandi, M. Heidarysafa, S. Mendu, L. Barnes,

and D. Brown, "Text classification algorithms: A survey," Information, vol.10, no.4, 2019.
[22] J. Zhao, K. Xia, Y. Fu, and B. Cui, "An AST-based code plagiarism detection algorithm," 2015 10th International conference on broadband and wireless computing, communication and applications (BWCCA), pp.178–182, 2015.
[23] A. Jerlin and J. Chinnappan, "ESAA: Efficient Sequence Alignment Algorithm for Dynamic Malware Analysis in Windows Executable Using API Call Sequence," DNA sequence, pp.290–298, 2017.
[24] B. Malley, D. Ramazzotti, and J.T.-Y. Wu, "Data pre-processing," Secondary Analysis of Electronic Health Records, pp.115–141, 2016.
[25] X. Xie, T.Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," ACM Transactions on software engineering and methodology (TOSEM), vol.22, no.4, pp.1–40, 2013.
[26] D. Zou, J. Liang, Y. Xiong, M.D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," IEEE Trans. Softw. Eng., vol.47, no.2, pp.332–347, 2019.
[27] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp.609–620, 2017.
[28] A. Moffat and J. Zobel, "Rank-biased precision for measurement of retrieval effectiveness," ACM Transactions on Information Systems (TOIS), vol.27, no.1, pp.1–27, 2008.
[29] R. Widyasari, G.A.A. Prana, S.A. Haryono, Y. Tian, H.N. Zachiary, and D. Lo, "XAI4FL: enhancing spectrum-based fault localization with explainable artificial intelligence," Proc. 30th IEEE/ACM International Conference on Program Comprehension, pp.499–510, 2022.

Wei Zheng received the Ph.D. degree in Computer Application Technology from the School of Computer Science, Xi'an University of Electronic Science and Technology in 2010. He has been teaching and conducting research at the School of Software of Nanchang Hangkong University since 2010 and is now the Dean of the School of Software. His research interests focused on software reliability analysis, airborne software testing and software engineering techniques.

Hao Hu received a bachelor's degree in engineering from Nanchang Hangkong University in 2020 and is currently a second year graduate school student in the School of Software, Nanchang Hangkong University, Nanchang, Chian. His research interests focused on software reliability analysis, fault localization and software defect prediction.



Tengfei Chen received a bachelor's degree in engineering from Xinyu University in 2021 and is currently a second year graduate school student in the School of Software, Nanchang Hangkong University, Nanchang, Chian. His research interests focused on software reliability analysis and software defect prediction.

Fengyu Yang received the Masters degree in Computer Science and Applications from Zhejiang University of Technology in 2006. He has been teaching and researching in the School of Software of Nanchang Hangkong University since 2006. His research interests focus on big data mining, airborne software testing, and aerospace system simulation techniques.

Xin Fan received his Masters degree from Nanchang Hangkong University. Currently, he is the head of the Department of Software Engineering at Nanchang Hangkong University. His research interests focused on software engineering techniques and software reliability.

Peng Xiao received the Ph.D. in Systems Engineering from Beihang University, Chian, in 2018. His research interests are focused on software testing and verification, software security and reliability, and software defect prediction techniques.