The Journal of Systems & Software 172 (2021) 110861

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

A Test Restoration Method based on Genetic Algorithm for effective fault localization in multiple-fault programs

Yan Xiaobo, Liu Bin, Wang Shihai *

Science & Technology on Reliability & Environment Engineering Laboratory, Beihang University, Beijing, China
School of Reliability and Systems Engineering, Beihang University, Beijing 100191, China

ARTICLE INFO

Article history:
Received 18 May 2020
Received in revised form 25 October 2020
Accepted 3 November 2020
Available online 5 November 2020

Keywords:
Fault localization
Software debugging
Multiple-faults
Evolution algorithm
Test restoration

ABSTRACT

Automatic fault localization is essential for software engineering. However, fault localization suffers from the interactions among multiple faults. Our previous research revealed that the fault-coupling effect is responsible for the weakened fault localization performance in multiple-fault programs. On the basis of this finding, we propose a Test Case Restoration Method based on the Genetic Algorithm (TRGA) to search potential coupling test cases and conduct a restoration process for eliminating the coupling effect. The major contributions of the current study are as follows: (1) the construction of a fitness function to measure the possibility of failed test cases becoming coupling test cases; (2) the development of a TRGA that searches potential coupling test cases; (3) and an evaluation of the TRGA efficiency across 14 open-source programs, three spectrum-based fault localizations, and two parallel debugging techniques. The results revealed the TRGA outperformed the original fault localization techniques in 74.28% and 78.57% of the scenarios in the best and worst cases, respectively. On average, the percentage improvement was 4.43% for the best case and 2% for the worst case. A detailed discussion of TRGA parameter settings is also provided.

1. Introduction

Fault localization is costly and time intensive. Statistics indicate that software engineers spend 70%–80% of their time on software testing and debugging (Tassey, 2002). Thus, automatic software fault localization can significantly improve the efficiency of software debugging. Automatic fault localization is aimed at solving the problem of locating faulty codes. The framework for automatic fault localizations involves constructing a relationship between the running information of the code and failure behavior. Statements are assigned with suspiciousness values according to the alignment between their execution information and the occurrence of failed test cases. However, most relevant studies have demonstrated the efficiency of their fault localization techniques for only one bug in a program. Factors that cause software failure are highly complex and a single-fault assumption is invalid in practice (Debroy and Wong, 2009). Locating faults in a multiple-fault program has therefore become a major challenge in fault localization research.

In contrast to locating a fault in single-fault programs, multiple-fault localization requires the construction of due-to relationships between numerous failures and their corresponding causes (Liu et al., 2008). In a single-fault program, these relationships are relatively easy to construct because all failures can be tracked to the same faulty statement. By contrast, in a multiple-fault program, the aforementioned relationships are complex and difficult to construct because a failure may correspond to different faulty statements and not all faulty statements lead to failure. Furthermore, numerous studies have found that multiple faults have a serious negative influence on fault localization and may lead to an inability of achieving fault localization (Wong et al., 2016).

To solve the problem of fault localization in multiple-fault programs, Jones et al. proposed debugging in parallel (Jones et al., 2007). This approach involves clustering failed test cases so that each cluster only corresponds to a single fault. Subsequently, Gao and Wong proposed a more efficient algorithm, namely Mseer, for parallel debugging (Gao and Wong, 2019). Numerous other search-based methods, such as integer linear programming (Steimann and Frenkel, 2012), have demonstrated efficiency in multiple-fault localization. Clustering-based multiple-fault localization methods (such as Mseer) and search-based multiple-fault localization methods (such as Integer Linear Programming) assume that the interactions between multiple faults can be eliminated by classifying test cases into separate categories with each category corresponding to single fault. However, software

* Corresponding author at: Science & Technology on Reliability & Environment Engineering Laboratory, Beihang University, Beijing, China.
E-mail addresses: yxbbuaa@buaa.edu.cn (Y. Xiaobo), liubin@buaa.edu.cn (L. Bin), wangshihai@buaa.edu.cn (W. Shihai).

https://doi.org/10.1016/j.jss.2020.110861
0164-1212/

failure is typically caused by multiple faults (Hamill and Goseva-popstojanova, 2009; Lucia et al., 2012), which indicates that even when test cases are classified correctly, each failed test case has a substantial chance of triggering multiple faults. Therefore, the single-fault correspondence hypothesis is invalid in practice. Moreover, the accuracy of clustering results is not perfect because several test cases may be clustered into the wrong clusters (Golagha et al., 2017). Our previous study found that fault interactions are responsible for the weakened fault localization performance in multiple-fault programs. Yan et al. proposed a fast algorithm to locate potential test cases that trigger specific fault interactions (Xiaobo et al., 2019). Yan's algorithm is based on the precondition that the number of test cases that have triggered specific fault interactions is given; however, this information is unavailable in the actual debugging process.[1] Moreover, the random selection strategy used in the fast algorithm cannot ensure a global optimal solution.

To address the aforementioned issues, we propose a Test Case Restoration Method based on the Genetic Algorithm (TRGA), which aims to search and restore coupling test cases likely to be affected by fault interaction. In contrast to the algorithm proposed in Xiaobo et al. (2019), the TRGA eliminates the precondition that the number of test cases that have triggered specific fault interactions must be given and uses a genetic algorithm (GA) framework to achieve global optimal solutions. This study is a pioneering research that aimed to design a superior fault localization technique from the perspective of fault interaction.

The main contributions of this study are as follows:

- A fitness function is proposed to evaluate the possibility of candidate solutions becoming coupling test cases. Furthermore, a detailed discussion of the fitness function is provided.
- A TRGA is proposed to improve the efficiency of fault localization in multiple-fault programs.[2]
- The effectiveness of the TRGA is evaluated across 14 open-source programs and five representative fault localization methods, which comprise three spectrum-based fault localization (SBFL) and two parallel debugging techniques. The results revealed the TRGA could improve 74.28% and 78.57% of the original fault localization techniques in the best and worst cases, respectively. On average, the percentage improvement was 4.43% for the best case and 2% for the worst case.
- The parameter selection of the TRGA is reported, and guidance on parameter settings is provided.

The remainder of this paper is organized as follows: Section 2 outlines the preliminary background of the fault-coupling effect and the motivation for constructing the TRGA. Section 3 provides an overview of the proposed TRGA. The experimental design and evaluation metrics are discussed in Section 4, with the results and analysis reported in Section 4.4. Section 5 describes topics relevant to the TRGA and Section 6 describes the threats to the validity of the TRGA. Section 7 describes related studies. Finally, Section 8 summarizes the conclusions and presents suggestions for future studies.

[1] Debuggers would not know whether the test suite contains test case that triggers specific fault interaction before debugging.

[2] A replication package of TRGA has been uploaded to Github (https://github.com/yxb2008007/YXB_BUAA).

Table 1
Four counters in the Operational Taxonomic Units.

Operator	Description
N_{np}	No. of passed test cases that the statement is not executed
N_{nf}	No. of failed test cases that the statement is not executed
N_{ep}	No. of passed test cases that the statement is executed
N_{ef}	No. of failed test cases that the statement is executed

Table 2
Suspiciousness functions.

Metrics	Formula expression
Tarantula	$\frac{N_{ef}}{N_{ef}+N_{nf}} / (\frac{N_{ef}}{N_{ef}+N_{nf}} + \frac{N_{ep}}{N_{ep}+N_{np}})$
Ochiai	$N_{ef} / \sqrt{(N_{ef}+N_{nf}) \cdot (N_{ef}+N_{ep})}$
Dstar	$N_{ef}^2 / (N_{ep}+N_{nf})$
Barinel	$1 - N_{ep}/(N_{ep}+N_{ef})$

2. Preliminary background and motivation

2.1. Spectrum-based fault localization

Spectrum-based fault localization (SBFL) can provide efficient and highly reliable fault localization (Wong et al., 2016). The statement spectrum provides coverage information for a specific entity. SBFL techniques assume that the more a statement is executed by failed test cases, the more likely is that statement to be a faulty statement. Under this assumption, similarity metrics are used in SBFL techniques for assigning suspiciousness to each statement. The statement with the greatest suspiciousness is therefore the most likely to be faulty.

To construct similarity metrics for SBFL, the Operational Taxonomic Units are proposed to standardize the procedure of generating the suspiciousness for each statement (Choi et al., 2010; Le et al., 2015). These units consist of four operators (Table 1), each of which captures specific coverage information for a statement in a selected test suite.

Similarity metrics, which are core elements of SBFL, compute the value of suspiciousness for each statement and can be represented as combinations of the aforementioned four operators. Table 2 lists the suspiciousness functions of several representative SBFL techniques using the Operational Taxonomic Units, including Tarantula (Jones et al., 2002), Ochiai (Naish et al., 2011) and Dstar (Wong et al., 2014). In a multiple-fault scenario, Barinel (Abreu et al., 2009b) is an efficient spectrum-based multiple-fault localization method, the suspiciousness function of which is also presented in Table 2 (Pearson et al., 2017).

2.2. Fault-coupling effect

The coverage information collected by debuggers determines the efficiency of fault localization, including SBFL. The fault-coupling effect is defined as a phenomenon in which fault localization features, such as the coverage information for a faulty statement, are changed due to the existence of additional faulty statements compared with that in the single-fault scenario. If the fault localization features for a faulty statement remain the same in both single-fault and multiple-fault programs, fault localization techniques should have the same performance in the single and multiple-fault scenarios. Thus, the fault-coupling effect is the cause of the decreased fault localization efficiency in multiple-fault programs. However, few studies have investigated the fault-coupling effect in multiple-fault programs. In our previous study, we investigated the failure behaviors of multiple faults in a large Chinese Railway System. We found that the independent assumption holds true in most cases (Xiaobo et al.,

2017). Furthermore, in another previous study (Xiaobo et al., 2019), we constructed 12 Fault Localization Interactions (FLIs) to represent 12 possible fault-coupling modes in a multiple-fault program. The aforementioned empirical study indicated that the distribution of the 12 FLIs was unbalanced and that FLI-1 was the most dominant FLI in multiple-fault programs.

Definition 1 (*Interaction FLI-1*)**.** Supposing S_f indicates a faulty statement, $\phi(S|t)$ indicates whether statement S is executed in test case t where $\phi(S|t) = 1$ denotes that S is executed and $\phi(S|t) = 0$ denotes that S is not executed. The parameter $R(t)$ provides the execution result for the test case t (0 for a passed test case and 1 for a failed test case). FLI-1 can be denoted as $\{S_f|\phi(S|t)_s = 0, R_s(t) = 0, \phi(S|t)_m = 0, R_m(t) = 1\}$, where the subscript s denotes the single-fault scenario and the subscript m indicates the multiple-fault scenario. The test case t is denoted as a coupling test case for the statement S_f.

Coverage-based methods, such as SBFL, focus on faults that have been executed and triggered failures. However, according to the definition of the FLI-1 interaction, although a specific faulty statement is not executed in the test case, the existence of different faults results in different testing results (Pass to Fail), which weaken the localizability of the specific faulty statement. Due to the prominence of FLI-1 in the multiple-fault scenario, the changing of testing results should also be considered as a special fault-coupling mode in which the coupling effect between faults is not expressed through the direct interaction between faults but indirectly through changes in test case execution results.

Thus, the fault-coupling effect defined in this paper focuses on the changes in the both coverage information for a faulty statement and execution result for a specific test case. The coupling test case is a test case in which the fault-coupling effect has been triggered.

2.3. Motivating example

To demonstrate the fault-coupling effect and its effect on fault localization, an example is presented in Fig. 1. The program in the example is a *utility* function from the open-source program Time (module *FiledUtils*), which ensures that the input value lies within the predetermined range. The aforementioned program includes three inputs: value (input value), minValue (minimum value in the predetermined range), and maxValue (maximum value in the predetermined range). Three faults are labeled in red in Fig. 1. Six test cases are constructed for the aforementioned example: T1 (3,1,3), T2 (5,1,10), T3 (2,3,1), T4 (2,-1,3), T5 (-2,-1,3) and T6 (-5,-1,3). A black dot is used to indicate whether the statement is executed in corresponding test case (statement spectrum). All the execution results of the six test cases are presented in the last row of Fig. 1. For comparison, Fig. 2 also provides the execution results for each individual fault in the same test suite (single-fault scenario). The results for each test case are also provided. Fig. 2(a) displays the execution information and execution results of a faulty program with only one fault, namely S_3 (Bug1). The suspiciousness scores of all statements can be computed through SBFL and ranked in descending order. Because Bug2 is the first fault to be located, we focus our attention on Bug2 to understand the fault-coupling effect.

As displayed in Fig. 2(b), because S_6 is not executed in the passed test case T6 in a single-fault scenario, we obtain the equation $\phi(S_6|T6)_s = 0, R_s(t) = 0$. However, test case T6 fails because of the existence of Bug1 in a multiple-fault scenario, with S_6 also not being executed; thus, we obtain the equation $\phi(S_6|T6)_m = 0, R_m(t) = 1$ in a multiple-fault scenario. According to Definition 1, S_6 triggers the interaction FLI-1 and test case T6 is a coupling test case for S_6. In such a situation, although S_6 (Bug2) is not executed in test case T6, the introduction of Bug1 changes the test result of T6 from pass to fail compared with the single-fault scenario of S_6; thus, the location characteristics of S_6 are weakened because the failed test case is caused by other faulty statements. For instance, S_6 (Bug2) has the highest rank in the single-fault program with the same test suite (Fig. 2(b)). Moreover, the suspiciousness rank of S_6 decreases to 2 in the three-fault program, as depicted in Fig. 1. To increase the suspiciousness of S_6 and eliminate the effect of FLI-1 on Bug2, the execution result of T6 should be changed from fail to pass. This restoration causes the execution results in a multiple-fault scenario to be similar to those in a single-fault scenario for a specific faulty statement, which increases the fault localization efficiency. The results of Xiaobo et al. (2019) confirm that the aforementioned transformation can improve the multiple-fault localization performance.

We designed the TRGA to change the execution results of the coupling test cases from fail to pass. The TRGA searches for coupling test cases in the three failed test cases, namely T1, T4, and T6, before effectively finding T6, which is the coupling test case. After searching, the TRGA alters the result of T6 from fail to pass and then integrates the restored T6 with the other five test cases as the new test suite. The suspiciousness ranks of Tarantula for the new test suite are depicted in the final column of Fig. 1. The results indicate that the TRGA improves the suspiciousness rank of the first located fault Bug2. The TRGA also improves the suspiciousness ranks of Bug1 and Bug3.

3. Our technique

In this section, we provide detailed information on the TRGA. Before describing the proposed method, several points should be clarified. Firstly, major consideration should be made to the most localizable fault, which is the first fault to be located. In a real debugging process, debuggers terminate the checking process once a fault has been identified. The fault localization technique is rerun when the identified fault has been fixed. The entire debugging process iterates until all test cases have passed. In each iteration, debuggers only focus on the most localizable fault. Thus we focus on the efficiency of fault localization for locating the most localizable fault. Second, the faults discussed in this paper are Bohrbugs (Cotroneo et al., 2016). Heisenbugs (Grottke and Trivedi, 2007), such as memory leaks and concurrency bugs, are beyond the scope of this study.

3.1. Overview of the TRGA framework

The traditional fault and multiple-fault localization methods proposed in previous studies ignore the existence of the fault-coupling effect. Therefore, these methods have several limitations. For SBFL, the existence of multiple faults can prevent faults from being located efficiently (Srivastav et al., 2010). For parallel debugging, even if the test cases are clustered correctly, each failed test case still has a high possibility of corresponding to multiple faults, which violates the single-fault correspondence hypothesis. To address these limitations, the TRGA is designed for limiting the fault-coupling effect to the most localizable fault.

The TRGA framework is illustrated in Fig. 3. The purpose of the TRGA is to search a set of failed test cases $F_T = \{f_1, f_2, \ldots, f_m \mid f_i \in F\}$, where F is the set of all failed test cases. F_T is the set of coupling test cases that are most likely to be affected by the fault-coupling effect. The TRGA comprises a global optimal search process to filter all the failed test cases and identify potential coupling test cases by using the GA (Sivanandam and Deepa, 2008). The aforementioned search process involves the following steps: (1) determining the initial population according to the failed test

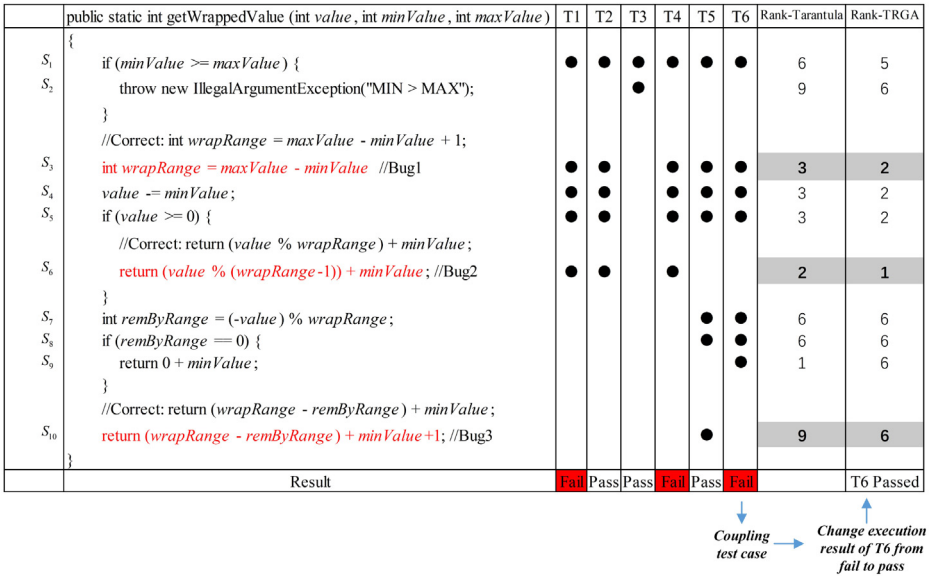


Fig. 1. A typical example from module FiledUtils of program Time.

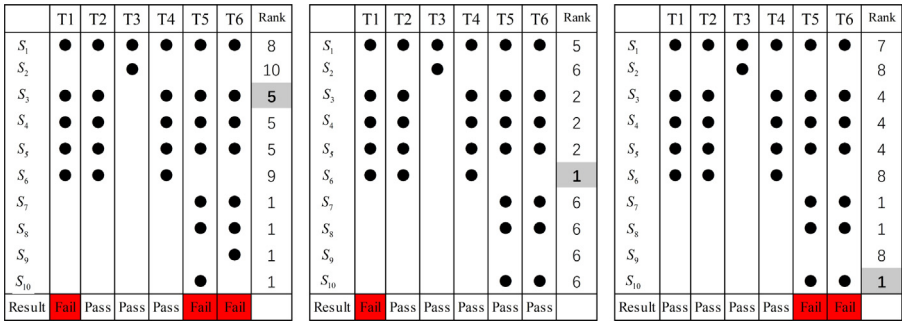


Fig. 2. Execution results for three single-fault programs.

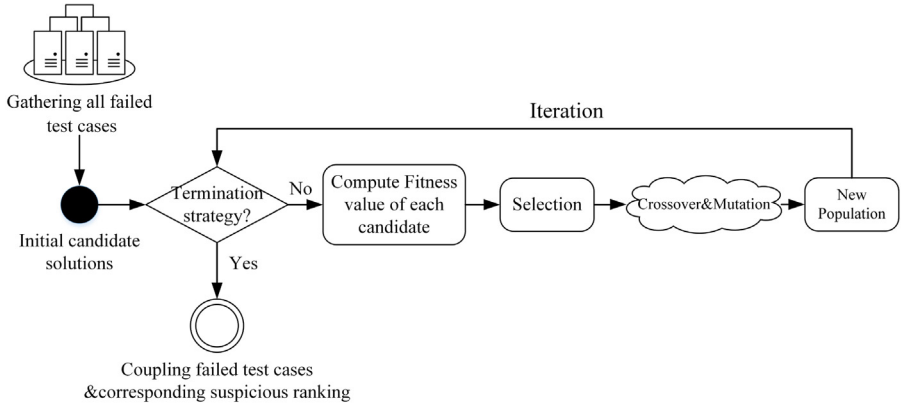


Fig. 3. An overview of TRGA.

cases, (2) computing the fitness score for each candidate solution in the population by using the fitness function, (3) selecting the several candidate solutions from the population according to their fitness scores (the higher the fitness score of an individual, the higher is their probability of being selected), (4) performing the crossover and mutation operation on selected individuals, and (5) generating the next population. After several iterations of the aforementioned steps, the result with the highest fitness score is the potential coupling test cases. After the execution results of the filtered test cases are altered from fail to pass, all the restored test cases are integrated with the other test cases to form a new test suite, which is named the restored test suite. The final suspiciousness ranking is then generated using the restored test suite. The restored test suite and corresponding suspiciousness ranking can be obtained in the process of computing the fitness score. Thus, the TRGA can generate the final suspiciousness ranking directly

once the iteration is complete. The pseudo-code of the TRGA is displayed in Algorithm 1. The details of the TRGA are presented in the following text.

Algorithm 1: TRGA(P, FL, T, E)

Input: given program, P; all test cases, T; coverage information, E; Selected fault localization technique, FL.
Output: coupling test cases, $T_{coupling}$; final suspiciousness rank list of the given program, R_{TRGA}.

```
Initializing Pop as I_n = (C_0, C_1, C_2, ..., C_n)^T; //n is the number of failed test cases;
Fitval, R_r ←FitnessScore(Pop, FL, T); //Generate fitness scores for initial population;
while !Termination strategy do
    // Normalize the fitness value;
    FitScore ←Norm(Fitval);
    for i=0 to |Pop| do
        SelectedPop ←Select(Pop, Fitval); //Selection;
        Selectedfit ← FitScore[SelectedPop];
    end
    PopNew ←Crossover&Mutation(SelectedPop, Selectedfit);
    FitNew, R_New ← FitnessScore(PopNew, FL, T); //Compute fitness scores for newly generated population;
    // Generate new population in the next iteration;
    Pop, Fitval, R_r ←GenNewPopulation(PopNew, Pop, FitNew, FitScore, R_New, R_r);
    T_coupling ← Solution with the greatest fitness score;
    Updating R_TRGA;
end
return T_coupling, R_TRGA
```

3.2. Initializing population

Because the candidate solution consists of a set of failed test cases, the failed test case feature vector is defined to construct the candidate solution in the TRGA.

Definition 2 (*Failed Test Case Feature Vector*)**.** A failed test case feature vector C represents the subset of failed test suites $F = \{f_1, f_2, \ldots, f_n\}$ (f_i indicates the corresponding failed test case):

$$C = (e_1, e_2, e_3, \ldots, e_n), e_i \in \{0, 1\} \tag{1}$$

where e_i indicates the characteristic of the ith failed test case (1 for selecting and 0 for ignoring). In the TRGA, the failed test case feature vector is used to encode each candidate solution. For instance, for the failed test cases $F = \{f_1, f_2, f_3\}$, the failed test case feature vector $C = (0, 0, 1)$ indicates that only the third failed test case f_3 exists in the candidate solution.

Ensuring that every failed test case exists in the initial candidate solutions is crucial. This step ensures that every failed test case can be checked and filtered by the TRGA. To achieve the aforementioned goal, an initial solution matrix is proposed.

Definition 3 (*Initial Solution Matrix*)**.** The initial solution matrix I_n is a matrix of n + 1 rows and n columns for a failed test suite $F = \{f_1, f_2, \ldots, f_n\}$:

$$I_n = (C_0, C_1, C_2, \ldots, C_n)^T = (\mathbf{0}, \mathbf{e}^{(1)}, \mathbf{e}^{(2)}, \ldots, \mathbf{e}^{(n)})^T \tag{2}$$

The initial solution matrix I_n consists of n-dimensional failed test case feature vectors. $\mathbf{e}^{(i)}$ is the standard base vector, which indicates the ith failed test case (the ith element is set to 1 whereas the other elements are set to 0). $\mathbf{0}$ is an indicator vector (zero vector) that suggests that no failed test case is a coupling test case.

When using an initial solution matrix, each feature vector corresponds to a unique failed test case. Thus, all the failed test cases are considered as independently as possible when constructing the initial solution. For instance, three failed test cases $F = \{T1, T4, T6\}$ exist in the example displayed in Fig. 1. Thus, we can construct four failed test case feature vectors: $C_0 = (0, 0, 0)$, $C_1 = (1, 0, 0)$, $C_2 = (0, 1, 0)$, $C_3 = (0, 0, 1)$, where C_0 indicates that no coupling test case exists in the program and C_1 indicates that the first failed test case $T1$ is the potential coupling test case (similarly for C_2 and C_3). In the aforementioned scenario, the initial solution matrix can be denoted as $I_3 = (C_0, C_1, C_2, C_3)^T$.

3.3. Fitness function

The fitness function is the core feature of the TRGA and is constructed to evaluate the possibility of candidate solutions becoming coupling test cases. For coupling test cases, the location characteristics of real faulty statements are unclear. Conducting a restoration process on such test cases can expose the potential faulty statement information; thus, a considerable difference exists between the original probability distribution of the faulty statements and the probability distribution of the faulty statements after restoration. Furthermore, our previous study confirmed that among all possible fault-coupling interactions, the restoration of FLI-1 results in the highest change in the fault localization efficiency. Changes in the fault localization efficiency lead to changes in the suspiciousness ranking of statements (Xiaobo et al., 2019).

Thus, the suspiciousness ranking obtained after the restoration process should be considerably different from the original suspiciousness ranking. In this study, a metric that identifies the dissimilarity between two suspiciousness rankings is used to construct the fitness function of the TRGA. A detailed explanation regarding the construction of a fitness function is presented in Sections 5.1 and 5.2 .

Given two suspiciousness rank lists α, β with m statements. The fitness function $F(\alpha, \beta)$ can be defined as:

$$F(\alpha, \beta) = \sum_{1 \le i \le m} (\alpha(i) - \beta(i)) \cdot \left| \frac{1}{\alpha(i)^2} - \frac{1}{\beta(i)^2} \right| \tag{3}$$

The first equation $\alpha(i) - \beta(i)$ in our fitness function $F(\alpha, \beta)$ can be used to assign a penalty function that prevents low-ranked statements from becoming top-ranked statements. The second equation $\left| \frac{1}{\alpha(i)^2} - \frac{1}{\beta(i)^2} \right|$ in our fitness function $F(\alpha, \beta)$ can be used to assign relatively high weights to top-ranked statements. The second equation of the aforementioned fitness function is used to measure the weight changes in reverse pairs, whereas the first equation is used to measure the weight changes for a single statement.

Algorithm 2: FitnessScore(C, FL, T)

Input: all test cases, T; n-dimensional failed test case feature vector, C; coverage information, E; selected fault localization technique, FL.
Output: fitness score of candidate solution, F_C; restored suspiciousness rank list, R_r.

```
R_ori ← FL(T); //generate suspiciousness rank list;
//notate the failed test cases as T_F = {f_1, f_2, ..., f_n};
Let C = (e_1, e_2, ..., e_i, ..., e_n), where e_i indicates the ith failed test case f_i;
for i=1 to n do
    if e_i=1 then
        alter the execution result of f_i from fail to pass;
        //f_i indicates the failed test case presented by e_i;
    end
end
T_R ← updating T with restored failed test cases T_F;
R_r ← FL(T_R, E) //generate restored suspiciousness rank list;
F_C ← F(R_r, R_ori) //compute the distance between R_ori and R_r;
return F_C,R_r
```

The fitness function of a candidate solution (feature vector for the failed test case) can be computed using a selected fault

localization technique through the procedure adopted in Algorithm 2. For the example displayed in Fig. 1, which comprises three failed test cases $F = \{T1, T4, T6\}$, all the possible failed test case feature vectors can be denoted as (0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 0), (1, 0, 1) and (0, 1, 1). Of these vectors, (1, 0, 0) indicates that the first failed test case $T1$ has been restored. Moreover, (1, 1, 0) indicates that the first and second failed test cases ($T1$ and $T4$, respectively) have been restored (similar to other vectors). The vectors (0, 0, 1), (0, 1, 1) and (1, 0, 1) share the same maximum fitness score (obtained from a fitness function) among the seven solutions; however, as stated in Section 3.6, the solution with fewer failed test cases to be restored is superior to the solution with more failed test cases to be restored. Thus, vector (0, 0, 1) is selected as the solution and only the failed test case $T6$ needs to be restored for this vector.

3.4. Selection

The selection process involves selecting a fixed number of candidate solutions according to their fitness scores. In this study, roulette wheel selection is used as the selection strategy because of its robustness and scalability (Ahn, 2006). The basic idea underlying roulette wheel selection is that the probability of each candidate solution being selected is proportional to its fitness score. The number of selected solutions is set as being equal to the population $|population|$ for ensuring that all individuals have the opportunity to be selected. To avoid a negative fitness score for the candidate solution, the fitness score is normalized using the following equation:

$$Norm(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

By using a normalization function, we can ensure that all fitness scores fall within the range (0, 1).

3.5. Crossover & mutation

The crossover and mutation operator generates new candidate solutions based on the selected population. All common crossover and mutation operators can be used in the TRGA. In this study, the single-point crossover and single-bit mutation (Gabriel et al., 1996) operators are used as the crossover and mutation operators, respectively, in the TRGA. To provide parameters the ability to adjust automatically, the linear adaptive GA (Srinivas and Patnaik, 1994) is used for the parameter selection of P_c and P_m. According to the linear adaptive GA, P_c and P_m can be set as follows:

$$P_c = \begin{cases} P_{cmin} + \dfrac{(P_{cmax} - P_{cmin}) \cdot (f_{max} - f')}{f_{max} - f_{avg}}, & \text{if } (f' \geq f_{avg}); \\ P_{cmax}, & \text{if } (f' < f_{avg}). \end{cases} \tag{5}$$

$$P_m = \begin{cases} P_{mmin} + \dfrac{(P_{mmax} - P_{mmin}) \cdot (f_{max} - f')}{f_{max} - f_{avg}}, & \text{if } (f' \geq f_{avg}); \\ P_{mmax}, & \text{if } (f' < f_{avg}). \end{cases} \tag{6}$$

The parameters P_{cmax} and P_{cmin} indicate the maximum and minimum limits of P_c, respectively. Similarly, P_{mmax} and P_{mmin} represent the maximum and minimum values of P_m, respectively. For the crossover operator, f' indicates the highest fitness score of a solution for two crossover candidates. For the mutation operator, f' expresses the fitness score of the selected solution. The variable f_{avg} represents the average fitness score of all solutions in the current population. When the fitness score of a solution is less than the average fitness score of population, the probability of crossover and mutation can be high. By contrast, when the fitness score of a solution is greater than the average fitness score, the solution should be reserved and the probability of crossover and mutation should be low.

Algorithm 3: Crossover&Mutation(pop, fit)

Input: selected population, pop; fitness scores for all solutions in the selected population, fit.
Output: newly generated candidate solutions, pop_{new}.

```
//Divide the population pop in half into father and mother;
half=int ( |pop| / 2 ), father = pop[: half ], mother = pop[half :];
Disrupt the order of father and mother randomly;
f_avg=Average(fit), f_max=Max(fit);
//Crossover;
for i=1 to half do
    updating P_c with Eq. (5) and generate a random value r between (0, 1);
    if r ≤ P_c then
        Crossover(father[i], mother[i]); updating pop;
    end
end
//Mutation;
for i=1 to |pop| do
    Mutation(pop[i]); updating pop;
end
pop_new = pop;
return pop_new
```

In this study, P_{cmax}, P_{cmin}, P_{mmax}, and P_{mmin} are set as 1, 0.1, 1, and 1, respectively. Notably, the mutation rate P_m is fixed as 1. Thus, a mutation operation is conducted on every newly generated candidate solution. The reason for such settings is that all the candidate solutions generated in the previous round during each iteration (refer to Section 3.6) are retained. To avoid repetition of the new candidate solutions and the searched candidate solutions in the previous round, a mutation operation is conducted on every newly generated candidate solution for generating as many different candidate solutions as possible. If the crossover operator P_c is conducted on each pair of candidate solutions, the ability of the TRGA to generate new individuals is strong and its ability to explore new solution spaces is improved; however, the possibility of missing the optimal solution also increases. By contrast, if P_c is set to a relatively small value, the global search capability of the TRGA is weakened (Sivanandam and Deepa, 2008). Thus, a large range of values is set for P_c (from 0.1 to 1). Algorithm 3 depicts the pseudo-code for the process of crossover and mutation.

3.6. Generating a new population

The best individual preservation strategy is adopted to generate a new population in each iteration of the TRGA for ensuring the global convergence of the GA (Suzuki, 1998). For the TRGA, the population at the beginning of the iteration and the new population generated through crossover and mutation are mixed. All the solutions are then ranked according to their fitness scores and only top-n solutions are preserved to generate the population in the next iteration. In this study, n is set as 30 for balancing efficiency and cost. Theoretically, the higher the number of optimal solutions retained, the higher is the likelihood of finding the global optimal solution; however, a large number of optimal solutions would increase the cost in each iteration. By contrast, if we only preserve few optimal solutions, the TRGA can easily find a local optimal state. Thus, 30 solutions are preserved in each iteration in this study. Furthermore, by setting the mutation rate to 1, the TRGA can be prevented from being trapped in a local optimal solution; thus, n has a limited effect on the algorithm performance. The TRGA can provide a satisfactory performance with an n value of 30. The failed test case provides intuitive coverage information on the real faulty statement. Thus, if two solutions have the same fitness score, the solution with fewer

failed test cases to be restored is superior to that with more failed test cases to be restored. The candidate solution for which all the failed test cases must be restored is discarded.

3.7. Termination strategy

In this study, the termination strategy was based on the following criteria: (1) the candidate solution with the highest fitness score has remained unchanged in the last 50 iterations and (2) the number of iterations of the TRGA has exceeded 200. The TRGA can abort its process and return the result after any of the aforementioned conditions are fulfilled, following the convention of Zheng et al. (2018). After a termination criterion is reached, the solution with the highest fitness score is the search result of the TRGA.

3.8. Time/space complexity

This section presents a detailed discussion on the time/space complexity of the TRGA. The TRGA, which is used for enhancing existing fault localization techniques in multiple-fault scenarios, needs to be based on a specific fault localization technique to operate effectively. For SBFL, if a program has to be diagnosed with M test cases and N executable statements, the SBFL technique needs to examine the execution state of each statement in each test case ($O(M*N)$) and then sort the statements according to their suspiciousness scores ($O(N*\log N)$). Consequently, the time complexity of SBFL is $O(M*N)+ O(N*\log N)$ as reported in Abreu et al. (2009b). Compared with SBFL techniques, the TRGA uses the GA to search for candidate test cases to be restored. The time complexity of the TRGA can be divided into two parts: the fitness score for each candidate solution and the cost of the GA. To calculate the fitness score, the SBFL technique needs to examine each candidate solution ($O(M*N)+ O(N*\log N)$). A fitness function can then be used to obtain the fitness score ($O(N)$). In the GA framework, because the number of population is predefined, the parameters in the processes of selection, crossover&mutation and generation of a new population are constant. Therefore, the time complexity of the aforementioned processes is $O(1)$. In summary, the time complexity of TRGA can be denoted as $O(M*N)+ O(N*\log N)+ O(N)$.

According to the time complexity formula of the TRGA, the computational cost of the TRGA is only depending on the number of test cases and statements; thus, the number of bugs cannot affect the cost of the TRGA. To more clearly illustrate the increased computational overhead of the TRGA, we use the Mockito program as an example in our experiments, which involves 1545 executable statements and 615 test cases on average. The time required for locating the first fault of Mockito is 1.112 s when using Barinel. Moreover, the TRGA requires 6.448 s on average to locate the first fault of Mockito, which is acceptable in a real development environment.

In terms of space complexity, the Operational Taxonomic Units described in Table 1 are used to calculate the suspiciousness rank for each statement in each iteration of the TRGA; therefore, the space complexity for generating the suspiciousness rank is $O(N)$. Furthermore, for a given number of candidate solutions C, the TRGA needs to store the information on suspiciousness ranks for each candidate solution in each iteration, which results in a space complexity cost of $O(C*O(N))$. Because the number of candidate solutions in each iteration is predefined, the space complexity of the TRGA is $O(N)$, which is equal to the space complexity for SBFL techniques.

4. Case studies

4.1. Objects for analysis & implementation

14 open-source programs were selected as objects in our case studies. These 14 programs comprised seven Siemens suite programs (Tcas, Totinfo, Schedule, Schedule2, Replace, Printtoken and Printtoken2) and seven large-scale programs with real faults (Chart, Math, Time, Lang, Mockito, Closure and Gzip (Version 1.1.2)). All the C programs adopted in this study can be obtained from the open-source institution "Subject Infrastructure Repository" (SIR) (S.I.R, 2008), and all the Java programs adopted in this study can be obtained from Defects4J (Version 1.2.0) (René et al., 2014). All the faulty versions and corresponding test cases can also be obtained from the SIR and Defects4J. A detailed overview of each object program is provided in René et al. (2014) and Wong et al. (2016).

As per the approach adopted in Debroy and Wong (2009) and Digiuseppe and Jones (2011), faulty versions without failed test cases and those that caused system crashes were excluded from the original test suite. Moreover, according to the conventions of Perez et al. (2019), faulty versions that could not be compiled or that produced a runtime error were also discarded from our case study. The coverage information for an individual test case was collected using the Gcov tool suite and the eclipse plug-in tool, Gzoltar (José et al., 2012). The Gcov tool suite was used to collect the coverage information for the C programs, and the Gzoltar tool was used to collect the coverage information for the Java programs.

For all C programs. To generate a sufficient number of multiple-fault sample programs, we merged a specific number (2–5) of single faults randomly in each object program for generating the corresponding multiple-fault programs. When the number of single faults selected was less than 15 (i.e., in the case of Printtoken2, Schedule, Schedule2, and Gzip), we adopted the mutation operation (Andrews et al., 2005) to generate artificial faults and expand the original single fault set as per the conventions of Digiuseppe and Jones (2015).

In the case of the Java programs obtained from Defects4J, each faulty version corresponded to a particular version of the program. Thus, the loaded classes and tests of the Java programs were substantially different, which made it difficult to generate multiple-fault programs. The differences in the program entities led to confusions in selecting a suitable program baseline for injecting multiple faults simultaneously. Moreover, the difference in the tests made it difficult to select appropriate public test cases that could be conducted on different programs. To address these issues, we selected one faulty program from the Chart (V1), Math (V41), and Time (V4) program packages as a standard benchmark and used the mutation operation to generate artificial faults for obtaining sufficient seed faults. Then, we merged multiple seed faults randomly to generate multiple-fault programs, which is similar to the strategy adopted for C programs. For comparison, three standard benchmark program packages with real faults (Mockito, Lang, and Closure) were also considered for evaluation. However, most of the standard Defects4J programs contain only one failed test case. In this case, the TRGA does not conduct the restoration process, and obtaining a statistically valid result is difficult. Thus, the results of three standard benchmark program packages (Mockito, Lang, and Closure) are not included in statistical analysis.

In summary, Table 3 presents a brief overview of the subject programs. The seed faults column provides the number of selected single faults, and the faulty version column presents the number of multiple-fault programs generated. Table 3 also presents the detailed faulty versions considered for the three

Table 3
Summary of programs for case study.

Objects	LOC	Test cases	Seed faults	Faulty versions	Language
Gzip	6503	211	15	85	C
Chart	13 269	426	21	120	Java
Math	2953	264	25	160	Java
Time	15 547	764	20	100	Java
Printtoken	566	4130	20	200	C
Printtoken2	509	4115	16	210	C
Replace	563	5542	30	240	C
Schedule	337	2650	16	210	C
Schedule2	295	2710	16	235	C
Tcas	174	1578	36	400	C
Totinfo	407	1052	23	200	C
Lang	–	–	65	V1-42,44-55,59–62	Java
Closure	–	–	133	V3,9,12,13,15,16,20,23,24,27,29,30, 34,36,40,45,46,49,50,53,55,58,62, 63,67,72,74,76,78,83,85,87,88,89, 91-93,97-108,114-116,119,121, 124,126,127,130,132	Java
Mockito	–	–	38	V12-17,22–25,27–38	Java

real-world benchmark programs (Mockito, Lang, and Closure). A workstation with a 3.00 GHz Intel Core i5-8500 CPU was used for all the experiments. The operating system in the workstation was Ubuntu 14.04.4, which had a physical memory of 8 GB.

4.2. Techniques for comparison

The TRGA must be combined with a specific fault localization technique for its application. To validate the TRGA and demonstrate that it can improve the fault localization performance in multiple-fault programs, five representative fault localization techniques were selected for comparison: Barinel, Ochiai, Dstar, Mseer, and fault localization clustering (FLC).

Ochiai is a classic SBFL technique (Abreu et al., 2009a). Dstar was proposed by Wong et al. (2014), and empirical evaluations have confirmed that Dstar outperforms Crosstab (Wong et al., 2012b), H3b/H3c (Wong et al., 2010), and RBF (Wong et al., 2012a). Barinel is a spectrum-based multiple-fault localization approach that combines the best of model-based and spectrum-based techniques. Evaluations have indicated that Barinel is one of the best performing fault localization techniques. Thus, Barinel was also considered for comparison. The aim of this study was to locate the most localizable fault, not to locate all faults in a program simultaneously. Therefore, we adopted the simplified formula of Barinel (Pearson et al., 2017), as presented in Table 2.

In addition to SBFL techniques, parallel debugging was considered due to its superior performance in multiple-fault localization. Mseer and FLC were selected as representative techniques for parallel debugging. Mseer is efficient in locating multiple faults and outperforms other parallel debugging techniques (Gao and Wong, 2019). FLC and behavior model clustering are two classic parallel debugging algorithms (Jones et al., 2007). However, because limited information is available regarding the construction of user behavior models in behavior model clustering (Hogerle et al., 2014), only FLC was selected for comparison.

In parallel debugging, SBFL needs to be conducted to produce a suspiciousness ranking. Because different SBFL techniques exhibit a similar performance with Mseer (Gao and Wong, 2019), we used the same SBFL technique for comparing the performance of different parallel debugging techniques. Therefore, we selected a classic SBFL technique called Tarantula as the integrated SBFL technique for parallel debugging.

4.3. Evaluation metrics

- *Y-EXAM score*

For SBFL, the *EXAM* score indicates the proportion of statements to be examined before the first faulty statement is identified (Pearson et al., 2017). For parallel debugging, the *T-EXAM* score indicates the proportion of statements to be examined before all the faults in the program have been located (Gao and Wong, 2019). To construct a unified measure, this study used the *Y-EXAM* score, which is designed to evaluate the ability to locate the most localizable fault. For SBFL, the *Y-EXAM* score indicates the proportion of statements to be examined before the most localizable fault is discovered. For parallel debugging, we assume that multiple debuggers are simultaneously engaged in debugging. Each debugger can manage the suspicious ranking of one specific cluster. Although the suspicious rankings of different clusters in parallel debugging target different faults in the program, we examined all clusters before proceeding to the next round of debugging. This strategy is consistent with those adopted in Gao and Wong (2019) and Jones et al. (2007). Thus, we used the average *Y-EXAM* score of the clusters to evaluate the efficiency of parallel debugging techniques. When a parallel debugging technique generated n clusters for program P, the *Y-EXAM* score for cluster i is $Y\text{-}EXAM_i$. The average *Y-EXAM* score for P can be computed as follows:

$$\sum_{i \in n} \frac{Y\text{-}EXAM_i}{n} \tag{7}$$

In the process of fault localization, different statements may have the same suspiciousness and *Y-EXAM* scores. Thus, the best and worst cases are provided to distinguish such scenarios (Debroy and Wong, 2009). In the best case, the faulty statement is the first to be checked in statements with the same suspiciousness score. By contrast, the faulty statement is the last to be examined in the worst case. Following the convention of Wong et al. (2016), we analyzed the best and worst cases to determine whether the TRGA can improve the efficiency of fault localization in multiple-fault programs. For each fault localization technique, we constructed two experimental groups: one with the original fault localization technique and the other with the TRGA. If the TRGA is more effective than the original technique, fewer statements would have to be examined to locate the most localizable fault in the TRGA than in the original fault localization technique (lower *Y-EXAM* score for the TRGA).

- *Wilcoxon Signed-Rank Test*

Several statistical methods were used to test the significance of the results. The Wilcoxon signed-rank test is a nonparametric test that tests whether a significant difference exists between

Table 4
The average *Y-EXAM* scores in the best case.

Objects	Barinel	B-TRGA	Dstar	D-TRGA	Ochiai	O-TRGA	Mseer	M-TRGA	FLC	F-TRGA
Gzip	0.0043	0.0095	0.0120	0.0144	0.0106	0.0128	0.0190	0.0206	0.0161	0.0155
Chart	0.0302	0.0204	0.0108	0.0067	0.0073	0.0039	0.0145	0.0144	0.0155	0.0153
Math	0.0245	0.0128	0.0031	0.0026	0.0031	0.0028	0.0124	0.0123	0.0089	0.0095
Time	0.0031	0.0032	0.0006	0.0011	0.0005	0.0016	0.0048	0.0047	0.0065	0.0066
Printtoken	0.0698	0.0662	0.0443	0.0440	0.0421	0.0419	0.1016	0.0939	0.0633	0.0625
Printtoken2	0.0256	0.0261	0.0369	0.0357	0.0266	0.0251	0.0658	0.0588	0.0350	0.0345
Replace	0.0482	0.0469	0.0653	0.0639	0.0631	0.0603	0.0629	0.0618	0.0601	0.0587
Schedule	0.0249	0.0226	0.1178	0.1165	0.0896	0.0887	0.0846	0.0837	0.0514	0.0496
Schedule2	0.1309	0.1214	0.1114	0.1030	0.1128	0.1022	0.1343	0.1331	0.1279	0.1198
Tcas	0.0514	0.0492	0.0426	0.0405	0.0440	0.0413	0.0499	0.0497	0.0502	0.0481
Totinfo	0.0685	0.0613	0.0381	0.0342	0.0391	0.0351	0.0704	0.0670	0.0656	0.0583
Lang	0.0426	0.0409	0.0390	0.0436	0.0402	0.0438	0.0415	0.0408	0.0408	0.0409
Mockito	0.0330	0.0279	0.0242	0.0238	0.0243	0.0238	0.0340	0.0291	0.0378	0.0378
Closure	0.0271	0.0274	0.0258	0.0256	0.0258	0.0256	0.0270	0.0268	0.0284	0.0283
Better cases	↑ 10.38%		↑ 4.57%		↑ 5.67%		↑ 3.92%		↑ 4.46%	
All cases	↑ 8.27%		↑ 2.85%		↑ 3.82%		↑ 3.60%		↑ 3.64%	

two samples. This test was used to determine whether the TRGA can significantly change the fault localization performance in multiple-fault programs. A detailed description of the Wilcoxon signed-rank test can be found in Wong et al. (2016) and Taheri and Hesamian (2013).

4.4. Results

To validate TRGA efficiency in multiple-fault programs, comparative experiments were conducted on faulty versions of 14 objective programs (seven programs from the Siemens suite, Gzip, Chart, Math, Time, Lang, Mockito, and Closure). Tables 4 and 5 present the average *Y-EXAM* scores obtained with the five selected algorithms (Barinel, Dstar, Ochiai, Mseer, and FLC) and their corresponding TRGAs (B-TRGA, D-TRGA, O-TRGA, M-TRGA, and F-TRGA) for a given multiple-fault program in the best and worst cases. For instance, the average *Y-EXAM* score of a given Chart program when using Barinel was 0.0302 and 0.0333 in the best and worst cases, respectively; however, when using the B-TRGA, the average *Y-EXAM* score reduced to 0.0204 and 0.0232 in the best and worst cases, respectively. The effect sizes of the TRGA are presented in the last two rows of Tables 4 and 5, in which *All cases* indicates the effect size of TRGA in all programs while *Better cases* represents the effect size of TRGA in cases where TRGA have a better fault localization efficiency.

For better readability, we combined the statistical analysis results with the average data presented in Tables 4 and 5. The green blocks indicate that the TRGA improved the efficiency of fault localization significantly (P-value < 0.05), whereas the red blocks represent the opposite (P-value ≥ 0.05). The yellow blocks indicate that the efficiency achieved with the TRGA was comparable to that achieved in the original case. The three benchmark program packages with real-world faults (Lang, Mockito and Closure) were not included in the analysis because most of their programs contained only one failed test case. Thus, the TRGA was unable to conduct the restoration process. Moreover, obtaining a strong statistical difference was difficult because of the relatively small number of faulty versions. For the aforementioned three program packages, the colored blocks only present a comparison of the average *Y-EXAM* scores.

The following observations can be made from these tables:

- In general, the TRGA can improve the performance of fault localization. Of the 140 scenarios presented in Tables 4 and 5, the TRGA outperformed the original fault localization techniques (Barinel, Dstar, Ochiai, Mseer and FLC) in 107 scenarios (52 scenarios (74.28%) in the best case and 55 scenarios (78.57%) in the worst case). Thus, the TRGA can improve the efficiency of not only SBFL but also parallel debugging in multiple-fault programs.
- Out of the 15 scenarios for the three benchmark program packages with real-world faults (Lang, Mockito, and Closure), the TRGA outperformed the original fault localization techniques in 10 (66.67%) cases in the best case and nine (60%) cases in the worst case.
- Compared with the original fault localization techniques, the TRGA reduced the *Y-EXAM* score by 0.019 on average in the best case. Moreover, in the worst case, the TRGA reduced the *Y-EXAM* score by 0.0016 on average. When considering all the cases, the TRGA improved the *Y-EXAM score* by 4.43% on average in the best case and 2% on average in the worst case. When considering only the cases in which the TRGA outperformed the original fault localization techniques, the TRGA enhanced the *Y-EXAM* score by 5.8% on average in the best case and 3.28% on average in the worst case.
- Although the TRGA improved the performance of fault localization in most scenarios, it failed to improve the efficiency of the original fault localization from the perspective of the average *Y-EXAM* score in 19 scenarios (11 scenarios in the best case and eight scenarios in the worst case).

A systematic investigation of these 19 scenarios indicates that the following situations can prevent the TRGA from improving the efficiency of fault localization:

Architecture threat. As presented in Tables 4 and 5, the TRGA failed to improve the fault localization efficiency in most scenarios for the Gzip program. The structure of Gzip is unique because it only has two main functions: compression and decompression (Digiuseppe and Jones, 2015). This hinders SBFL from classifying faulty code and nonfaulty but always executed code. In such a scenario, the TRGA also suffers from distinguishing frequently executed nonfaulty statements, which can cause a weak connection between the change in suspicious ranks and the coupling test cases.

Already optimal problem. As presented in Tables 4 and 5, the TRGA failed in 6 out of 10 scenarios for the Time program. In most of the scenarios for this program, the most localizable faults in multiple-fault versions are already in the top-ranked lists. In such a scenario, conducting a restoration process may increase the ranks of other faulty statements; however, the TRGA can restore failed test cases that contain critical execution information for the most localizable faulty statement. This can help to achieve a greater level of diversity between suspiciousness rank lists, which in turn decreases the ranks of the most localizable faults.

Table 5
The average *Y-EXAM* scores in the worst case.

Objects	Barinel	B-TRGA	Dstar	D-TRGA	Ochiai	O-TRGA	Mseer	M-TRGA	FLC	F-TRGA
Gzip	0.0277	0.0360	0.0247	0.0308	0.0219	0.0268	0.0872	0.0905	0.0531	0.0522
Chart	0.0333	0.0232	0.0131	0.0092	0.0093	0.0061	0.0238	0.0237	0.0248	0.0247
Math	0.0344	0.0225	0.0089	0.0080	0.0095	0.0081	0.0321	0.0285	0.0297	0.0277
Time	0.0046	0.0036	0.0009	0.0014	0.0009	0.0020	0.0062	0.0060	0.0097	0.0084
Printtoken	0.0877	0.0829	0.0661	0.0658	0.0638	0.0637	0.1791	0.1788	0.4198	0.4191
Printtoken2	0.0338	0.0314	0.0637	0.0633	0.0386	0.0371	0.1052	0.1033	0.4197	0.4191
Replace	0.0609	0.0597	0.0833	0.0822	0.0791	0.0777	0.1207	0.1205	0.1229	0.1223
Schedule	0.0361	0.0320	0.1397	0.1385	0.1103	0.1096	0.2371	0.2369	0.1490	0.1475
Schedule2	0.1549	0.1455	0.1467	0.1387	0.1481	0.1376	0.1635	0.1633	0.1526	0.1446
Tcas	0.1103	0.1091	0.1189	0.1178	0.1201	0.1181	0.1273	0.1272	0.1341	0.1336
Totinfo	0.0915	0.0847	0.0574	0.0540	0.0583	0.0546	0.1086	0.1055	0.1756	0.1685
Lang	0.1355	0.1306	0.1300	0.1347	0.1311	0.1333	0.1344	0.1324	0.1325	0.1326
Mockito	0.0442	0.0390	0.0354	0.0390	0.0354	0.0390	0.0472	0.0421	0.0891	0.0890
Closure	0.0731	0.0733	0.0717	0.0715	0.0717	0.0714	0.0730	0.0727	0.0818	0.0817
Better cases	↑ 7.62%		↑ 2.66%		↑ 3.50%		↑ 1.27%		↑ 1.33%	
All cases	↑ 5.87%		↑ 0.58%		↑ 1.45%		↑ 0.97%		↑ 1.17%	

Indistinct faulty statement. Similar to architecture threat, efficiently detecting a frequently executed faulty statement that has a weak connection to the occurrence of failure is difficult with fault localization techniques. For instance, the buggy code in the faulty version V9 of the Lang program is the so-called FAULT_OF_OMISSION. In this program, the aforementioned type of fault does not generally lead directly to failure. Instead, it causes other nonfaulty statements to present a strong connection to failure through assignment or call. Such a scenario hinders the TRGA from identifying the real faulty statement.

In the case of architecture threat and indistinct faulty statements, the execution spectrum cannot indicate the association between program contexts. Therefore, one cannot effectively distinguish a real faulty statement from a frequently executed nonfaulty statement or an indistinct faulty statement from a nonfaulty statement that indicates a strong connection to failure when merely using a statement execution spectrum. When optimization is already achieved, determining whether the top-ranked items include faulty statement merely according to the execution information is difficult. Such a scenario is rare in a real debugging process (Parnin and Orso, 2011). However, despite the aforementioned risk, the TRGA only reduced the *Y-EXAM* score by 0.00045 on average in the best case and 0.0008 on average in the worst case for the Time program. Therefore, the efficiency of the TRGA is acceptable.

We also evaluated the TRGA according to the *Y-EXAM* scores of individuals. Fig. 4 displays the distributions of faulty versions with different fault localization techniques. In each subgraph, the horizontal axis indicates the *Y-EXAM* scores, whereas the vertical axis indicates the percentage of faulty versions with the most localizable fault located. The legend Original Best Case indicates the fault localization without the TRGA in the best case, whereas the legend Original Worst Case indicates the corresponding worst case. Similarly, the legends TRGA Best Case and TRGA Worst Case represent the application of the TRGA in the best and worst cases respectively. All the selected multiple-fault versions from the 14 object programs are included in each subfigure of Fig. 4. Consider Part 4(a) of Fig. 4 as an example. When the *Y-EXAM* score equaled 0.05, 73.5% of the faulty versions from the 14 object programs could locate the most localizable fault by examining no more than 5% of the code with the help of the TRGA in the best case. By contrast, only 67.3% of the faulty versions could locate the most localizable fault successfully with Barinel. The corresponding percentages in the worst case were 61.2% with the TRGA and 59.4% with Barinel.

In summary, the results suggest that the TRGA can improve the performance of SBFL and parallel debugging techniques in most cases and can provide guidance to find potential coupling test cases in multiple-fault programs. The experimental results indicated that multiple faults have a limited effect on locating the most localizable fault (*EXAM* score changed by approximately 2% Digiuseppe and Jones, 2015). Therefore, the improvement provided by the TRGA is acceptable (4.43% in the best case and 2% in the worst case when considering all the situations).

5. Discussion

5.1. Why TRGA works

This section describes the efficiency of the TRGA in multiple-fault scenarios from two perspectives: the construction of the fitness function and the usage of the GA.

- *Construction of the Fitness function*

In practice, a debugger does not know if a program contains multiple faults or a coupling test case. To address this issue, the validity of the fitness function needs to be explained. First, the restored failed test suite contains at least one failed test case, which contains the execution information for a real faulty statement. Thus, the ranking produced by the restored test suite is not random and continues to reflect the probability distribution of the faulty statements. Given program P to be debugged with a failed test suite F and failed test case feature vector C, the results of the fitness score $\mathcal{F}(C)$ are described in the following text:

(1) $\mathcal{F}(C) = 0$. The rank list produced by the restored test suite R has two possible scenarios: (1) R is the same as the rank list R_{ori} produced by the original failed test suite F, and (2) R and R_{ori} are completely symmetrical. Because R is not random and reflects the probability distribution of the faulty statements, the first case is more likely to occur. In such a scenario, restoring partial failed test cases may reduce the suspiciousness score of statements; however, the ranking remains unchanged. This phenomenon indicates that the restoration process does not expose the potential faulty statement information and such failed test cases are not coupling test cases. For example, restoring T5 or T6 cannot change the rank list of statements in Fig. 2(c) because only one bug in the program is highly correlated with failure and the restoration process cannot provide any new location information regarding potential faulty statements.

(2) $\mathcal{F}(C) < 0$. In such a scenario, a high possibility of bottom-ranked statements becoming top-ranked statements exists after a restoration process. This obfuscates the location characteristics of real faulty statements because the

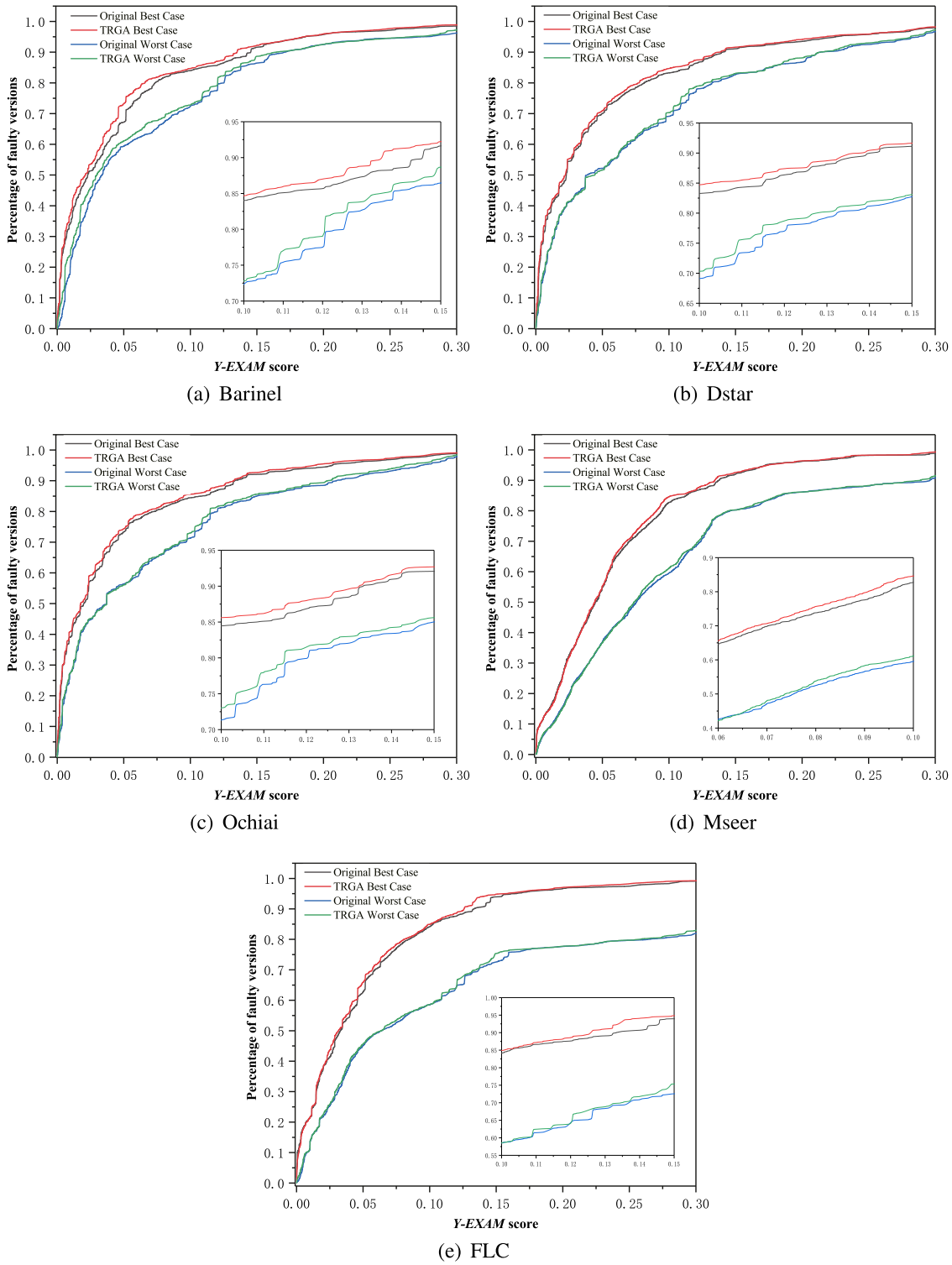


Fig. 4. *Y-EXAM* score-based comparison between the five selected techniques and their corresponding TRGA algorithms.

bottom-ranked statements are unlikely to be faulty. Therefore, the restoration process should not be conducted in such cases. For instance, the ranking of the example program in Fig. 1 is $R = (7, 9, 4, 4, 4, 7, 2, 2, 1, 9)$ for the 10 statements after restoration of failed test case T1. The fitness score of this restoration is -0.47 according to our fitness function. In this case, the bottom-ranked normal statements S_7 and S_8 turn into the top two ranked statements after restoration.

(3) $\mathcal{F}(C) > 0$. $\mathcal{F}(C) > 0$ indicates that the number of statements in the restored test suite with higher ranks is greater than the number of statements with lower ranks. The greater the fitness score, the more obvious is this trend. For example, the ranking of statements in Fig. 1 is $R_{ori} = (6, 9, 3, 3, 3, 2, 6, 6, 1, 9)$ with original test suite. After restoration on T6, the ranking changes to $R = (5, 6, 2, 2, 2, 1, 6, 6, 6, 6)$. The ranks of all statements except S_9 have been improved. In this scenario, because the restored test

suite continues to reflect the probability distribution of the faulty statements, the restoration process enhances the location characteristics of potential faulty statements, which indicates the potential possibility of identifying coupling test cases. In a single-fault scenario, the debugger can obtain useful location information when $\mathcal{F}(C) > 0$. All the failed test cases are caused by the same fault in a single-fault program; thus, the ranks of faulty statements should remain at a high level, as $\mathcal{F}(C) = 0$ indicates. $\mathcal{F}(C) > 0$ suggests that the original test suite may obfuscate the location characteristics of a real faulty statement. For example, restoring the failed test cases $\{T5, T6\}$ which have the highest fitness score can improve the ranking of faulty statement S_3 from 5 to 2 in a program with Bug1 (Fig. 2(a)).

In summary, the fitness function proposed in this study can help debuggers determine whether the restoration process should be used without knowing the number of faults. With $\mathcal{F}(C) = 0$ as a watershed for conducting the restoration process, the TRGA adds an indicator vector (fitness score for the indicator vector is 0), as described in Section 3.2, to ensure that the condition $\mathcal{F}(C) = 0$ can be compared when generating new solutions.

- *Usage of the GA*

Although the fitness function helps decide whether the restoration process should be used, the solution process remains difficult. For the failed test suite $F = \{f_1, f_2, \ldots f_n\}$, 2^n different failed test case feature vectors can be generated according to F which indicates 2^n candidate solutions. The size of the solution space is exponentially related to the number of failed test cases n; thus, exponential explosion occurs with an increase in the number of failed test cases. To solve this problem, a metaheuristic method (the GA) is used in the TRGA to search for coupling test cases. Compared with random searching or exhaustive search, a metaheuristic method can achieve a relatively optimal solution in practice with acceptable expenses (Gabriel et al., 1996). The GA is a popular metaheuristic search method. It has been proven to be a promising general method for optimization problems (Kuo and Wan, 2007). The reasons for selecting the GA over other metaheuristic methods for searching in the TRGA are as follows. (1) The initial solution should consider each failed test case to avoid the algorithm falling into the local optimal solution. Thus, the search algorithm should be a swarm-intelligence-based algorithm (*e.g.*, the GA), not an individual-based evolution algorithm like SA (Simulated annealing algorithm) and TS (Tabu Search). (2) The solution space formed by the failed test case feature vector is discrete; thus, traditional search algorithms exhibit limitations for continuous problems, such as PSO (Particle Swarm Optimization). By contrast, the GA can manage any form of objective function and constraint, whether linear or nonlinear, discrete, or continuous (Christoph and Jürgen, 2015). Moreover, the GA has been proved to be effective in multiple-fault localization (Zheng et al., 2018), which makes it suitable for this study.

5.2. Potential fitness functions

The fitness function has a major influence on the performance of the TRGA. In this study, we constructed the fitness function according to the degree of dissimilarity between two rank lists. Several other potential distance metrics work in a similar manner. This section presents details regarding potential fitness functions.

The first potential distance metric is the Revised Kendall tau distance (Gao and Wong, 2019), which is designed to describe the difference between two rank lists. However, this metric only considers the number of reverse pairs in two suspiciousness rank lists. The number of reverse pairs is maximized when the two rank lists are completely symmetrical. For instance, when $R = (r_1, r_2, r_3, r_4, r_5)$ represents a specific rank list with five statements, the target rank list with the maximum Revised Kendall tau distance is $R = (r_5, r_4, r_3, r_2, r_1)$. In this scenario, the top-ranked statements have the lowest ranking and the bottom-ranked statements are the most suspected of being faulty. This result is not reasonable because the aforementioned transformation allows the most useful features associated with failure to be ignored.

In addition to the Revised Kendall tau distance, the Kendall Tau distance (Gao and Wong, 2019) and the Weighted Kendall Tau distance proposed by Liu et al. (2008) can be used to characterize the difference between sequences; however, the aforementioned three metrics only consider the number of reverse pairs in the two suspiciousness rank lists.

Code-coverage-based failure metrics measure the distance between failures on the code coverage of executions (Liu et al., 2008). The most representative metric is the Jaccard distance (Levandowsky and Winter, 1971), which is designed to describe the proportion of same elements between two sets. Because the elements in the rank lists produced through fault localization remain unchanged, the Jaccard distance is unsuitable for describing the difference between two rank lists. Therefore, Jones et al. proposed an improved Jaccard distance to address this issue (Jones et al., 2007). For two rank lists R_1 and R_2, the proportion parameter is set as α. The improved Jaccard distance between R_1 and R_2 can then be determined as follows:

$$ImprovedJaccard(R_1, R_2) = 1 - \frac{|Top_\alpha(R_1) \cap Top_\alpha(R_2)|}{|Top_\alpha(R_1) \cup Top_\alpha(R_2)|} \quad (8)$$

Where $Top_\alpha(R)$ indicates the top-α% ranked statements in R. However, as stated by Gao and Wong, the improved Jaccard distance only considers the number of elements in the reduced set and ignores the discordance between statements (Gao and Wong, 2019), which causes it to produce questionable results (*e.g.*, assign the same distance to two distinct sets).

Other popular distance metrics used in fault localization include vector-based distance metrics which aim to measure the distance between two vectors (*e.g.*, Hamming distance and Chebyshev distance). These metrics convert sets into vectors, and then calculate the difference between vectors. Vector-based distance metrics are very effective in measuring vector distance; however, in the field of fault localization, they fail to assign suitable weights to different ranked statements (top-ranked statements should have a higher influence on the distance than bottom-ranked statements do). In summary, the fitness function described in Eq. (3) is suitable for this study.

5.3. TRGA in a single-fault program

The TRGA is designed for multiple-fault localization and has been proved to be an efficient framework for this task; however, a debugger does not know if a program contains single or multiple faults. Therefore, the aforementioned algorithm should also be effective for a single-fault program. We compared the performance of the TRGA with three SBFL techniques (Barinel, Ochiai, and Dstar) by using 11 object programs and their single-fault versions (Table 6). Mseer and FLC were not used in the aforementioned comparison due to their frameworks, which are based on multiple-fault localization.

Of the 66 scenarios for single-fault programs presented in Table 6, the TRGA decreased the *Y-EXAM* score marginally compared with the original fault localization method (by 0.0022 on average) in only 16 scenarios. Surprisingly, the TRGA improved the SBFL performance in 50 scenarios (25 in the best case and 25 in the worst case). The excellent performance of the TRGA in

Table 6
The average *Y-EXAM* scores for single-fault programs in the best and worst cases.

Objects	Barinel		Barinel_TRGA		Dstar		Dstar_TRGA		Ochiai		Ochiai_TRGA	
	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst
Chart	0.0164	0.0287	**0.0135**	**0.0259**	**0.0073**	**0.0114**	0.0101	0.0144	0.0075	0.0116	**0.0074**	**0.0115**
Math	0.0079	0.0237	**0.0068**	**0.0226**	**0.0016**	**0.0142**	0.0078	0.0204	**0.0017**	**0.0143**	0.0080	0.0205
Time	0.0018	0.0048	**0.0015**	**0.0045**	**0.0008**	**0.0019**	0.0013	0.0023	**0.0009**	**0.0020**	0.0013	0.0023
Gzip	0.0253	0.1665	**0.0249**	**0.1661**	0.0180	0.1483	**0.0179**	**0.1481**	0.0181	0.1484	**0.0177**	**0.1480**
Totinfo	0.1033	0.2748	**0.0960**	**0.2690**	0.0893	0.2608	**0.0815**	**0.2542**	0.0897	0.2612	**0.0800**	**0.2520**
Tcas	0.1073	0.3072	**0.1041**	**0.3049**	0.1015	0.3014	**0.1012**	**0.3011**	**0.1020**	**0.3019**	0.1027	0.3025
Replace	0.1012	0.2734	**0.0952**	**0.2687**	0.0924	0.2645	**0.0897**	**0.2619**	0.0924	0.2645	**0.0888**	**0.2626**
Schedule	0.1024	0.1869	**0.1007**	**0.1866**	0.0725	0.1571	**0.0723**	**0.1569**	0.0759	0.1604	**0.0749**	**0.1595**
Schedule2	0.2449	0.4121	**0.2299**	**0.3989**	0.2042	0.3714	**0.1939**	**0.3610**	0.2042	0.3714	**0.1966**	**0.3638**
Printtoken	**0.1544**	**0.3818**	0.1549	0.3824	**0.1280**	**0.3554**	0.1285	0.3559	0.1290	0.3564	**0.1286**	**0.3560**
Printtoken2	0.0976	0.2117	**0.0964**	**0.2110**	0.0712	0.1745	**0.0678**	**0.1710**	0.0733	0.1766	**0.0702**	**0.1735**

a single-fault program validates the intuitive analysis presented in Section 5.1, which indicates that TRGA can provide a guide to determine whether the restoration process should be conducted. In summary, the TRGA has acceptable efficiency for single-fault programs.

5.4. Precision and recall scores of the TRGA

To evaluate the ability of the TRGA to identify coupling test cases, the precision and recall scores were calculated (Nam et al., 2013). The precision score represents the proportion of coupling test cases identified by the TRGA from the filtered test cases, whereas the recall score indicates the proportion of coupling test cases successfully identified by the TRGA. Assume that n failed test cases exist in a multiple-fault program and m failed test cases are coupling test cases; the TRGA correctly identifies k_c coupling test cases and incorrectly identifies k_w failed coupling test cases. Considering the aforementioned assumptions, the precision score of the TRGA is equal to $k_c/(k_c + k_w)$ and its recall score is equal to k_c/m.

Precision and recall scores were not calculated for parallel debugging techniques because they only cluster failed test cases and do not change their execution characteristics; thus, the classification results of parallel debugging are similar to those of SBFL techniques. As indicated in Section 4.1, most programs of the Lang, Closure, and Mockito program packages only contain a single failed test case. The TRGA cannot be executed in such cases; therefore, a statistically valid result cannot be obtained. Consequently, aforementioned metrics were not calculated for the Lang, Closure, and Mockito program packages.

The precision and recall results are presented in Tables 7 and 8. Because the most localizable fault, which determines the coupling test case identified, may vary in different SBFL techniques, the number of coupling test cases in a project depends on the SBFL technique used. The data presented in Tables 7 and 8 indicate that the TRGA can identify an average of 50.5% of the coupling test cases and maintain an average precision rate of 35.5% for its filtered failed test cases. Considering this study is the first on filtering coupling test cases, we compared the performance of the TRGA with the random strategy. When k coupling test cases are identified from m failed test cases, the expected precision of adopting random searching for identifying coupling test cases is k/m. Table 9 presents the average percentage of coupling test cases for different subject programs when using different SBFL techniques. According to the data presented in Table 9, the distribution of the coupling and noncoupling test cases is unbalanced. Moreover, the percentage of coupling test cases is relatively small, which causes considerable difficulties in the filtering process. For instance, the percentage of coupling test cases in the Chart program with Ochiai was 9.52%, which is also the expected precision score with a random strategy. The TRGA achieved a relative precision score of 39.96%. Similar results can be found for other scenarios.

In addition to the precision score, Table 10 presents the proportion of test cases filtered by the TRGA. Unlike the random strategy, which filtered 50% of the test cases as suspicious coupling test cases each time, the number of coupling test cases filtered by the TRGA was relatively close to the real distribution (Table 9). For instance, the percentage of filtered test cases when using the TRGA was 18.03% for the Math program with Ochiai. The random strategy identified 50% of all the test cases as suspicious coupling test cases. Compared to the real distribution in Table 9 (13.46%), the random strategy identified nearly 40% of all the test cases as coupling test cases; however, none of these test cases were real coupling test cases. Further, Fig. 5 displays the average *Y-EXAM* scores computed by conducting the TRGA and random searching separately in seven large-scale programs. For readability, the average data of the best-case and worst-case scenarios are presented. The results depicted in Fig. 5 indicate that the TRGA is superior to random searching in improving the efficiency of fault localization.

Although the TRGA significantly outperformed random searching in filtering coupling test cases and improving the fault localization efficiency, 64% of the coupling test cases identified by the TRGA were not coupling test cases in reality; thus, concerns exist regarding the influence of coupling test cases on the effectiveness of fault localization. Firstly, the influence of coupling test cases on fault localization was validated in a previous study (Xiaobo et al., 2019). Performing a restoration process on coupling test cases can improve the fault localization efficiency. Furthermore, due to the inaccuracy in filtering coupling test cases, we should minimize the effect of misclassified test cases on locating the most localizable fault. For the Math program with Barinel, the precision score of the TRGA (31.05%) was lower than that of the random strategy (33.65%), as presented in Table 7. The percentages of filtered test cases for the aforementioned strategies were very similar (49.42% for the TRGA and 50% for the random strategy). However the TRGA still exhibited a significantly superior fault localization performance to the random strategy, which indicates that the TRGA can significantly reduce the impact of misclassified test cases on locating the most localizable fault.

In summary, the TRGA cannot only provide guidance for finding coupling test cases but also help reduce the effect of misclassified test cases on fault localization.

5.5. Relationship between the TRGA and related techniques

This section describes important issues in the field of fault localization. The first issue is the problem of Coincidental Correctness (CC). CC occurs when faulty statements are executed, but the program produces the correct output. The presence of CC reduces the effectiveness of fault localization approaches. A

Table 7
The average precision scores.

Methods	Siemens suite	Math	Time	Gzip	Chart	Average	Average in total
Barinel	0.3782	0.3105	0.5255	0.5915	0.4250	0.4461	
Dstar	0.2288	0.2740	0.3087	0.3339	0.4176	0.3126	35.5%
Ochiai	0.2378	0.2652	0.3091	0.3224	0.3996	0.3068	

Table 8
The average recall scores.

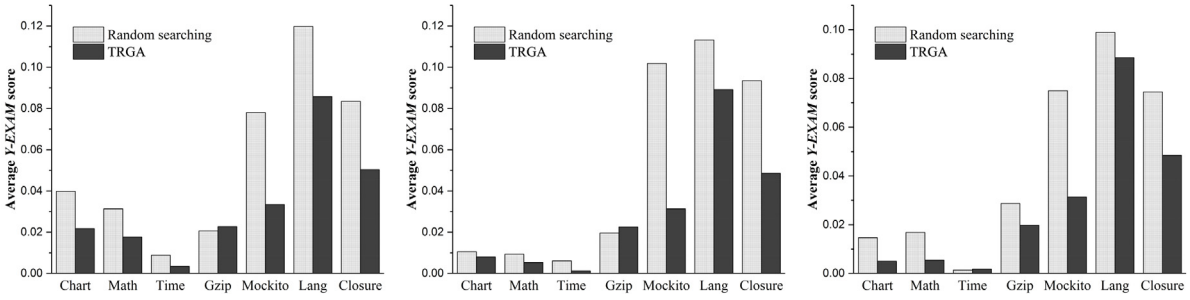
Methods	Siemens suite	Math	Time	Gzip	Chart	Average	Average in total
Barinel	0.4143	0.4561	0.4749	0.6409	0.7050	0.5382	
Dstar	0.4750	0.3518	0.4564	0.5565	0.6621	0.5003	50.5%
Ochiai	0.4632	0.3551	0.4183	0.4732	0.6730	0.4766	

Table 9
The average percentage of coupling test cases.

Methods	Siemens suite	Math	Time	Gzip	Chart	Average	Average in total
Barinel	0.3098	0.3365	0.4765	0.5724	0.2531	0.3897	
Dstar	0.1042	0.1279	0.3277	0.1857	0.0839	0.1659	24.7%
Ochiai	0.1290	0.1346	0.3303	0.2481	0.0952	0.1875	

Table 10
The average percentage of filtered test cases.

Methods	Siemens suite	Math	Time	Gzip	Chart	Average	Average in total
Barinel	0.3394	0.4942	0.4306	0.6202	0.4199	0.4609	
Dstar	0.2164	0.1643	0.4845	0.3095	0.1329	0.2615	33.43%
Ochiai	0.2513	0.1803	0.4471	0.3642	0.1604	0.2807	



(a) Barinel (b) Dstar (c) Ochiai

Fig. 5. TRGA VS Random searching.

high interaction between a faulty statement(s) and other correlated statements in programs is likely to cause CC. The TRGA is designed to weaken the fault-coupling effect; thus, concerns exist regarding the TRGA's ability to diminish the negative effect of CC on the effectiveness of fault localization. Theoretically, the fault-coupling effect considered in this study aims at the difference between single- and multiple-fault scenarios, whereas the CC only aims at one single test case. Thus, CC can occur in both single- and multiple-fault scenarios, whereas the fault-coupling effect considered in this study only occurs in multiple-fault scenarios. Moreover, the TRGA focuses on failed test cases, whereas CC only occurs on passed test cases; thus, the TRGA cannot examine the interactions in passed test cases. Therefore, the TRGA cannot reduce the negative effect of CC on the effectiveness of fault localization.

The TRGA is not a fault localization technique by itself. Instead, it enhances existing fault localization techniques in multiple-fault scenarios. Thus, the second issue in the field of fault localization is the relationship between the TRGA and other fault localization enhancement techniques, such as test case filtering and test case purification techniques. Test case filtering techniques aim to filter and simplify the test suite to improve the quality of test cases for better revealing defects (Masri et al., 2007). For instance, Christi et al. found that the effectiveness of SBFL can be improved by delta-debugging failed test cases and basing localization only on the reduced test cases (Christi et al., 2018). Test case purification techniques focus on dividing existing test cases into small fractions to enhance the coverage information for increasing the fault localization efficiency (Xuan and Monperrus, 2014). Test suite enhancement techniques are not designed to address the multiple-fault problem, whereas the TRGA addresses the multiple-fault problem. Moreover, the TRGA does not conflict with test suite enhancement techniques because these techniques focus on improving the test suite quality, whereas the TRGA is designed to weaken the potential fault-coupling effect of a given test suite. However, considering the target of this work is multiple-fault localization, future studies can investigate whether the TRGA can improve the fault localization efficiency of test suite enhancement techniques or whether a minimized test suite can introduce the multiple-fault issue.

6. Threats to validity

One external threat to validity is the general validity of the TRGA. In this study, the efficiency of the TRGA was demonstrated by conducting an empirical study on a few open-source programs. It remains unclear whether the results can be extended

to other software systems. However, we investigated the aforementioned threat by validating the effectiveness of the TRGA for 14 typical software programs with five representative fault localization techniques. All the experimental programs varied in size (from simple programs to large-scale programs) and functionality. These programs were designed using two of the most used development languages, namely C and Java; thus, the obtained results were accurate and generalizable. Furthermore, three subject programs, namely Lang, Closure, and Mockito, only contained real-world bugs; thus, the accuracy and generalizability of the results was further increased. The techniques considered in this study are an appropriate representation of fault localization approaches. We considered not only traditional techniques such as Ochiai but also Dstar, which is the most efficient SBFL technique (Wong et al., 2014), and Barinel, which is an efficient spectrum-based multiple-fault localization technique (Abreu et al., 2009b). Two effective parallel debugging techniques were also included in our empirical study. Such extensive verification gives us confidence that the TRGA can be effectively applied in other software systems.

Another threat to validity is the mutation-based fault injection method. Several artificial faults were injected into the subject programs to create sufficient multiple-fault versions. However, because these faults were not real bugs, the rationality of the artificial faults may have been affected. In this study, we used four mutation operators that have been demonstrated to provide trustworthy results (Andrews et al., 2005; Do and Rothermel, 2006). Furthermore, Ali et al. stated that no direct evidence exists to indicate that mutation operators are invalid for SBFL (Ali et al., 2009). Because the same trend was achieved when studying several real bugs from the Siemens and Defects4J datasets (Lang, Closure, and Mockito only contain real bugs), the mutation-based fault injection method was applicable in this study.

A potential threat in using the *Y-EXAM* score as an evaluation metric is related to whether the obtained results are comprehensive, generalizable, and accurate. Numerous metrics can be used to measure the effectiveness of different fault localization techniques (*e.g.*, *EXAM* score for Tarantula Jones et al., 2002 and *T-EXAM* score for Mseer Gao and Wong, 2019). However, after debuggers have identified a fault, the fault is fixed immediately and the test suite is rerun on the fixed program. The first fault to be located is of the greatest concern to the debuggers. Therefore, the *Y-EXAM* score is proposed for describing the ability of a fault localization technique to locate the most localizable fault. As long as the adopted algorithm can better locate the most localizable fault in each iteration, it can locate all the bugs with high efficiency. Thus, the *Y-EXAM* score can measure the performance of different fault localization techniques. Additional evaluation metrics, such as the total number of statements examined, were also used to evaluate the TRGA performance. The results obtained for these metrics exhibited a similar trend to those obtained for the *Y-EXAM* score.

7. Related works

Although most real-world faults belong to a single class (Perez et al., 2017), researchers have found that individual failures are often triggered by multiple bugs in certain large software systems (Hamill and Gosevapopstojanova, 2009; Lucia et al., 2012). Furthermore, a multiple-fault environment has a serious negative influence on the efficiency of fault localization (Srivastav et al., 2010; Denmat et al., 2005). Thus multiple-fault programs are an important problem in fault localization. This section provides a brief introduction to the related work from two aspects: analysis of the negative effect of multiple-fault programs on fault localization and the methods for overcoming this negative effect.

With regard to the negative effect of multiple-fault programs on fault localization, Jones and Digiuseppe investigated the effect of the fault quantity on SBFL and found that the existence of multiple faults increased the *EXAM* score by 2% (Digiuseppe and Jones, 2015). Zhong and Su investigated the real bug fixing processes in six Java programs and claimed that the interferences among multiple faults were serious (Zhong and Su, 2015). Debroy and Wong defined fault interferences by using test case execution results (pass/fail) (Debroy and Wong, 2009). Jones and Digiuseppe further investigated these fault interactions and claimed that the fault obfuscation was serious (Digiuseppe and Jones, 2011). Gopinath et al. theoretically analyzed the fault masking phenomenon and claimed that tests detecting a fault in isolation would continue to detect the fault even when it occurs in combination with other faults (Gopinath et al., 2017). However, the quantitative analysis of a fault-coupling effect in Xiaobo et al. (2019) indicated that the effect of fault interferences, which are defined from the perspective of test case execution results, is limited. Furthermore, a dominant interaction FLI-1 is responsible for the weakened SBFL performance in multiple-fault scenarios. Therefore, the method proposed in this study was designed according to the characteristics of FLI-1.

A popular framework for handling multiple-fault programs is parallel debugging, which involves dividing failed test cases into different clusters for ensuring that each cluster corresponds to a single fault. Jones et al. proposed Behavior Model Clustering and FLC to cluster failed test cases according to their execution information and fault localization information (Jones et al., 2007), Gao and Wong constructed an efficient framework named Mseer to cluster failed test cases according to their Revised Kendall Tau distance (Gao and Wong, 2019). Another popular framework for handling multiple-fault programs is search-based methods. Steimann and Frenkel used integer linear programming to search potential small blocks of standard program spectra for improving fault localization in multiple-fault programs (Steimann and Frenkel, 2012). Rui et al. constructed a Barinel metric to search multiple faulty statements simultaneously (Abreu et al., 2009b). Yan et al. proposed a GA-based framework FSMFL to search multiple faulty statements according to the characteristic of multiple failed test cases (Zheng et al., 2018). The goal of the aforementioned search-based methods is to search multiple diagnoses, which are the most probable solution for the occurrence of multiple faults. However, as stated in Section 1, both parallel debugging and search-based methods are based on the single fault correspondence hypothesis, which is invalid in practice. Mutation-based approaches have also been investigated to address multiple-fault settings. In particular, Hong et al. proposed 11 new mutation operators for multilingual bugs, which can directly mutate interactions between language interfaces (Hong et al., 2015). Holmes and Groce proposed a fault localization method (Repair) based on a mutation-based metric for measuring the distance between executions. This method can effectively provide a causal relationship between executions and failures (Holmes and Groce, 2020). In contrast to the TRGA, Hong's method is designed to improve the accuracy of locating multilingual bugs without considering the interactions between multiple faults. Although Holmes' method can reduce the interferences between executions and get a better repairing mutants rankings, each localization is based on a single failed test case; thus, Repair can obtain multiple localizations according to the number of failed test cases and is more suitable for debugging a single failed test case than for debugging multiple failed test cases. By contrast, the TRGA outputs a single localization on the basis of the entire testsuite.

In addition to the aforementioned methods, various other solutions exist for the multiple-fault problem. For instance, Yu et al.

proposed a classification algorithm to filter failed test cases that only trigger a single fault by computing the ulam-distance (Yu et al., 2015). Lamraoui and Nakajima developed a formula-based fault localization technique by encoding a program into a complete flow-sensitive trace formula and claimed that this method is particularly useful for programs with multiple faults (Lamraoui and Nakajima, 2016). The aforementioned techniques do not consider the effect of fault-coupling and have been validated in few programs. Yu's method has not been applied to any case study. Moreover, Lamraoui's method has only been verified in Tcas, which is a small program. Li et al. proposed a state-based fault-localization approach known as MDFS by using a dynamic dependence graph (Li et al., 2017). This method can be effectively used in single- and multiple-fault programs by narrowing the suspicious states according to the outcome of the sparse symbolic exploration. However, as stated by the authors, MDFS cancels the faulty program state if multiple faulty statements exist in the same dependence chain, which may cause MDFS to fail. In our previous study (Xiaobo et al., 2019), we proposed a fast algorithm for locating potential coupling test cases; however, this proposed algorithm is based on the precondition that the number of coupling test cases is given. By contrast, this precondition is eliminated in the TRGA.

Additional methods exist for achieving multiple-fault localization in a specific object or under special preconditions. For instance, Aribi et al. proposed a pattern mining method based on constraint programming to diagnose multiple faults. This method improves the efficiency of resolution of constraint programming for handling multiple faults (Aribi et al., 2017). Niu et al. constructed a new Minimal Failure-Causing Schema model based on combinatorial testing for handling multiple faults (Niu et al., 2020). Naish et al. proposed a restricted hyperbolic metric for lightweight genetic programming and applied genetic programming in multiple-fault localization, which requires reliable historical data to obtain the aforementioned metric (Naish et al., 2015).

8. Conclusion and future work

In this paper, the TRGA is proposed to improve the efficiency of general fault localization by eliminating the fault-coupling effect in multiple-fault programs. As depicted in Fig. 3, the TRGA first identifies coupling test cases that are influenced by the fault-coupling effect and then performs a restoration process on the identified test cases. General fault localization techniques can subsequently be used on the restored test suite. The aforementioned approach enables these techniques to achieve superior performance in multiple-fault programs. In the searching phase, the TRGA applies a new encoding strategy to encode failed test cases into the failed test case feature vector (Definition 2 in Section 3.2). This paper also presents a method for constructing initial solutions, which are named the initial solution matrix (Definition 3 in Section 3.2). Moreover, a fitness function (Eq. (3) in Section 3.3) is constructed to measure the possibility of candidate solutions becoming coupling test cases. A GA framework is used to search potential coupling test cases according to their fitness scores. In the restoration phase, because of the dominance and characteristics of FLI-1, the TRGA alters the execution results of the searched test cases from fail to pass. Empirical investigation of 14 open-source programs indicates that the TRGA can improve the performance of both SBFL and parallel debugging techniques.

To further validate the efficiency of the TRGA, future studies should conduct a wider investigation by using a higher number of industrial software programs with other types of fault localization techniques, such as slice-based and model-based techniques. The *architecture threat* and *instinct faulty statement* problem should be solved in future studies by introducing additional information, such as the slices of programs, to construct the relationships between program contexts. The strategy for determining whether the top-ranked statements contain a faulty statement should be further studied. A suitable strategy for the aforementioned task can improve the performance of the TRGA. Accurately distinguishing coupling test cases from failed test cases solely on the basis of execution spectra is difficult. Future research can integrate program information, such as control flow and data flow, to identify coupling test cases more accurately. Future studies should also examine whether the TRGA can improve the fault localization efficiency of test suite enhancement techniques, such as test case filtering and purification techniques, in multiple-fault scenarios.

CRediT authorship contribution statement

Yan Xiaobo: Methodology, Software, Visualization, Writing - original draft. **Liu Bin:** Supervision, Conceptualization. **Wang Shihai:** Validation, Writing - review & editing.

Declaration of competing interest

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

Acknowledgment

The authors would like to thank editors and anonymous reviewers whose suggestions have improved the presentation of this work. This work is supported by the Science and Technology on Reliability and Environmental Engineering Laboratory, China under Grant 614200404031117 and foundation No. 61400020404.

References

Abreu, R., Zoeteweij, P., Golsteijn, R., Van Gemund, A.J.C., 2009a. A practical evaluation of spectrum-based fault localization. J. Syst. Softw. 82 (11), 1780–1792.
Abreu, R., Zoeteweij, P., Van Gemund, A.J.C., 2009b. Spectrum-based multiple fault localization. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 88–99.
Ahn, C.W., 2006. Advances in Evolutionary Algorithms: Theory, Design and Practice. Springer, Heidelberg, Berlin, pp. 7–22.
Ali, S., Andrews, J.H., Dhandapani, T., Wang, W., 2009. Evaluating the accuracy of fault localization techniques. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 76–87.
Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments. In: International Conference on Software Engineering. pp. 402–411.
Aribi, N., Maamar, M., Lazaar, N., Lebbah, Y., Loudni, S., 2017. Multiple fault localization using constraint programming and pattern mining. In: IEEE International Conference on Tools with Artificial Intelligence. pp. 860–867.
Choi, S., Cha, S., Tappert, C.C., 2010. A survey of binary similarity and distance measures. J. Syst. Cybern. Inform. 8 (1), 43–48.
Christi, A., Olson, M.L., Alipour, M.A., Groce, A., 2018. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In: International Symposium on Software Reliability Engineering. pp. 184–191.
Christoph, S., Jürgen, Z., 2015. Handbook on Project Management and Scheduling. Springer, Cham, pp. 57–74.
Cotroneo, D., Pietrantuono, R., Russo, S., Trivedi, K., 2016. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. J. Syst. Softw. 113, 27–43.
Debroy, V., Wong, W.E., 2009. Insights on fault interference for programs with multiple bugs. In: IEEE International Conference on Software Reliability Engineering. pp. 165–174.
Denmat, T., Ducasse, M., Ridoux, O., 2005. Data mining and cross-checking of execution traces: a re-interpretation of jones, harrold and stasko test information. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 396–399.

Digiuseppe, N., Jones, J.A., 2011. Fault interaction and its repercussions. In: IEEE International Conference on Software Maintenance. pp. 3–12.
Digiuseppe, N., Jones, J.A., 2015. Fault density, fault types, and spectra-based fault localization. Empir. Softw. Eng. 20 (4), 928–967.
Do, H., Rothermel, G., 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans. Softw. Eng. 32 (9), 733–752.
Gabriel, W., Periaux, J., Galan, M., Cuesta, P., 1996. Genetic Algorithms in Engineering and Computer Science. John Wiley & Sons, Inc, NY, United States.
Gao, R., Wong, W.E., 2019. Mseer—An advanced technique for locating multiple bugs in parallel. IEEE Trans. Softw. Eng. 45 (3), 301–318.
Golagha, M., Pretschner, A., Fisch, D., Nagy, R., 2017. Reducing failure analysis time: an industrial evaluation. In: IEEE/ACM International Conference on Software Engineering. pp. 293–302.
Gopinath, R., Jensen, C., Groce, A., 2017. The theory of composite faults. In: International Conference on Software Testing Verification and Validation. pp. 47–57.
Grottke, M., Trivedi, K., 2007. Fighting bugs: remove, retry, replicate, and rejuvenate. IEEE Comput. 40 (2), 107–109.
Hamill, M., Gosevapopstojanova, K., 2009. Common trends in software fault and failure data. IEEE Trans. Softw. Eng. 35 (4), 484–496.
Hogerle, W., Steimann, F., Frenkel, M., 2014. More debugging in parallel. In: IEEE International Symposium on Software Reliability Engineering. pp. 133–143.
Holmes, J., Groce, A., 2020. Using mutants to help developers distinguish and debug (compiler) faults. Softw. Test. Verif. Reliab. 30 (2).
Hong, S., Lee, B., Kwak, T., Jeon, Y., Ko, B., Kim, Y., Kim, M., 2015. Mutation-based fault localization for real-world multilingual programs. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 464–475.
Jones, J.A., Bowring, J.F., Harrold, M.J., 2007. Debugging in parallel. In: International Symposium on Software Testing and Analysis. pp. 16–26.
Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: International Conference on Software Engineering. pp. 467–477.
José, C., André, R., Alexandre, P., Rui, A., 2012. Gzoltar: an eclipse plug-in for testing and debugging. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 378–381.
Kuo, W., Wan, R., 2007. Recent advances in optimal reliability allocation. IEEE Trans. Syst. Man Cybern. 37 (2), 143–156.
Lamraoui, S., Nakajima, S., 2016. A formula-based approach for automatic fault localization of multi-fault programs. J. Inf. Process. 24 (1), 88–98.
Le, T.B., Lo, D., Thung, F., 2015. Should i follow this fault localization tool's output?. Empir. Softw. Eng. 20 (5), 1237–1274.
Levandowsky, M., Winter, D.G., 1971. Distance between sets. Nature 234 (5323), 34–35.
Li, F., Li, Z., Huo, W., Feng, X., 2017. Locating software faults based on minimum debugging frontier set. IEEE Trans. Softw. Eng. 43 (8), 760–776.
Liu, C., Zhang, X., Han, J., 2008. A systematic study of failure proximity. IEEE Trans. Softw. Eng. 34 (6), 826–843.
Lucia, Thung, F., Lo, D., Jiang, L., 2012. Are faults localizable?. In: IEEE Working Conference on Mining Software Repositories. pp. 74–77.
Masri, W., Podgurski, A., Leon, D., 2007. An empirical study of test case filtering techniques based on exercising information flows. IEEE Trans. Softw. Eng. 33 (7), 454–477.
Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. 20 (3), 1–32.
Naish, L., Neelofar, N., Ramamohanarao, K., 2015. Multiple bug spectral fault localization using genetic programming. In: Australian Software Engineering Conference. pp. 11–17.
Nam, J., Pan, S.J., Kim, S., 2013. Transfer defect learning. In: International Conference on Software Engineering. pp. 382–391.
Niu, X., Nie, C., Lei, J., Leung, H., Wang, X., 2020. Identifying failure-causing schemas in the presence of multiple faults. IEEE Trans. Softw. Eng. 46 (2), 141–162.
Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers. In: International Symposium on Software Testing and Analysis. pp. 199–209.
Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.J., 2017. Evaluating and improving fault localization. In: International Conference on Software Engineering. pp. 609–620.
Perez, A., Abreu, R., Damorim, M., 2017. Prevalence of single-fault fixes and its impact on fault localization. In: International Conference on Software Testing Verification and Validation. pp. 12–22.
Perez, A., Abreu, R., Van Deursen, A., 2019. A theoretical and empirical analysis of program spectra diagnosability. IEEE Trans. Softw. Eng. 1.
René, J., Darioush, J., Ernst, M.D., 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: International Symposium on Software Testing and Analysis. pp. 437–440.
S.I.R, 2008. The Software Infrastructure Repository (retrieved October 2008) [Online]. Available: http://sir.unl.edu/portal/index.php.
Sivanandam, S.N., Deepa, S.N., 2008. Introduction to Genetic Algorithms. Springer, Heidelberg, Berlin.
Srinivas, M., Patnaik, L.M., 1994. Adaptive probabilities of crossover and mutation in genetic algorithms. IEEE Trans. Syst. Man Cybern. 24 (4), 656–667.
Srivastav, M., Singh, Y., Gupta, C., Chauhan, D.S., 2010. Complexity estimation approach for debugging in parallel. In: International Conference on Computer Research and Development. pp. 223–227.
Steimann, F., Frenkel, M., 2012. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In: IEEE International Symposium on Software Reliability Engineering. pp. 121–130.
Suzuki, J., 1998. A further result on the Markov chain model of genetic algorithms and its application to a simulated annealing-like strategy. IEEE Trans. Syst. Man Cybern. 28 (1), 95–102.
Taheri, S.M., Hesamian, G., 2013. A generalization of the wilcoxon signed-rank test and its applications. Statist. Papers 54 (2), 457–470.
Tassey, G., 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. RTI, Research Triangle Park, NC.
Wong, W.E., Debroy, V., Choi, B., 2010. A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw. 83 (2), 188–208.
Wong, W.E., Debroy, V., Gao, R., Li, Y., 2014. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308.
Wong, W.E., Debroy, V., Golden, R.M., Xu, X., Thuraisingham, B., 2012a. Effective software fault localization using an rbf neural network. IEEE Trans. Reliab. 61 (1), 149–169.
Wong, W.E., Debroy, V., Xu, D., 2012b. Towards better fault localization: A crosstab-based statistical approach. IEEE Trans. Syst. Man Cybern. 42 (3), 378–396.
Wong, W.E., Gao, R., Li, Y., Rui, A., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.
Xiaobo, Y., Bin, L., Jianxing, L., 2017. The failure behaviors of multi-faults programs: An empirical study. In: IEEE International Conference on Software Quality, Reliability and Security Companion. pp. 1–7.
Xiaobo, Y., Liu, B., Shihai, W., 2019. An analysis on the negative effect of multiple-faults for spectrum-based fault localization. IEEE Access 7, 2327–2347.
Xuan, J., Monperrus, M., 2014. Test case purification for improving fault localization. In: Foundations of Software Engineering. pp. 52–63.
Yu, Z., Bai, C., Cai, K., 2015. Does the failing test execute a single or multiple faults?: an approach to classifying failing tests. In: International Conference on Software Engineering. pp. 924–935.
Zheng, Y., Wang, Z., Fan, X., Chen, X., Yang, Z., 2018. Localizing multiple software faults based on evolution algorithm. J. Syst. Softw. 139, 107–123.
Zhong, H., Su, Z., 2015. An empirical study on real bug fixes. In: International Conference on Software Engineering. pp. 913–923.

Yan Xiaobo was born in 1991. He received the B.S. degree in 2014. Now he is pursuing the Ph.D degree at School of Reliability and System Engineering, and Science and Technology on Reliability and Environmental Engineering Laboratory in Beihang University. His current research interests include software reliability, software defect prediction and software fault localization.

Bin Liu was born in 1967. He is a profess or at Science and Technology on Reliability and Environmental Engineering Laboratory, and School of Reliability and Systems Engineering in Beihang University. His research areas are software psychology, software defect prevention and software ecosystem.

Shihai Wang was born in 1979, He received his PH.D degree from the School of Computer Science at the University of Manchester, United Kingdom, in 2010. Now he is a lecturer at Science and Technology on Reliability and Environmental Engineering Laboratory, and School of Reliability and Systems Engineering in Beihang University. His research interests include machine learning, defect prediction, and software testing.