PAPER

Spectrum-Based Fault Localization Using Fault Triggering Model to Refine Fault Ranking List

Yong WANG[†,††a)], Zhiqiu HUANG[††], Rongcun WANG[†††], *Nonmembers*, *and* Qiao YU[†††], *Member*

SUMMARY Spectrum-based fault localization (*SFL*) is a lightweight approach, which aims at helping debuggers to identity root causes of failures by measuring suspiciousness for each program component being a fault, and generate a hypothetical fault ranking list. Although *SFL* techniques have been shown to be effective, the fault component in a buggy program cannot always be ranked at the top due to its complex fault triggering models. However, it is extremely difficult to model the complex triggering models for all buggy programs. To solve this issue, we propose two simple fault triggering models ($RIPR_\alpha$ and $RIPR_\beta$), and a refinement technique to improve fault absolute ranking based on the two fault triggering models, through ruling out some higher ranked components according to its fault triggering model. Intuitively, our approach is effective if a fault component was ranked within *top k* in the two fault ranking lists outputted by the two fault localization strategies. Experimental results show that our approach can significantly improve the fault absolute ranking in the three cases.

key words: *fault localization, software debugging, testing*

1. Introduction

Debugging play a key role in software life cycle to improve software quality. A data from the Cambridge University shows total estimated cost of debugging is $312 billion per year in all the world, and programmers spent half of their time in debugging [1]. A general debugging work includes fault detect, localization and fixing. Therefore, a nice fault localization technique can help development groups reduce financial budgets and human resources. According to root causes, faults can be classified into three disjoint types: *Semantic faults*, *Concurrency Faults*, and *Memory Faults* [2]. Tan et al. perform a study for fault characteristics in open source software, they found semantic faults are the dominant root cause of failures. As software evolves, semantic faults increase, and memory-related faults decrease. Therefore, they called for more research effort to address semantic bugs [2]. However, automatic fault localization for *Sematic Faults* is difficult due to sematic faults are related with requirements or programmers intentions.

Manuscript received December 1, 2017.
Manuscript revised May 22, 2018.
Manuscript publicized July 4, 2018.

[†]The author is with School of Computer and Information, Anhui Polytechnic University, Wuhu, China.
[††]The authors are with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China.
[†††]The authors are with School of Computer Science and Technology, China University of Mining and Technology, XuZhou, China.
a) E-mail: yongwang@ahpu.edu.cn

DOI: 10.1587/transinf.2017EDP7386

Spectrum-based fault localization technique (*SFL*) is the use of test results and various program spectra collected from the testings to measure the likelihood of a program entity being a fault [3]. *SFL* techniques have gained much popular due to lower computational overhead and high scalability. However, are *SFL* can really help programmers localize root cause of failures in reality? Recently, researchers perform some empirical studies or survey researches to try answer this questions, such as [4]–[7]. Despite their differences, they all highlight *SFL* should improve accuracy and support fault understanding to improve its usefulness.

In previous studies, some researchers focused on proposing different *SFL* techniques to improve accuracy of fault localization with aid of relationship among components. Zhang et al. [8] highlighted that a fault may propagate a series of *infected program states* before triggering a failure. They used edge profiles to model infected program states, and by associating basic blocks with edges, the suspiciousness scores for each component was calculated. Baah et al. [9], [10] applied *casual inference* to fault localization, and a linear model was built on program dependency graph to estimate the causality of failures of a given program component. Wang et al. [11] proposed a *SFL* approach combined with fault context of program spectrum.

Different from previous works, we propose a *SFL* technique using fault triggering model to further refine fault ranking list for a *SFL* technique. The above mentioned approaches are also easily combined into our approach. Our idea behind this is that we can refine the fault ranking list via ruling out some components ranked higher if we know that they follow which fault triggering model. The fault triggering model is discussed in detailed in Sect. 4. There exist two fault triggering models, called $RIPR_\alpha$ and $RIPR_\beta$. Informally, if a fault triggering model belongs to $RIPR_\alpha$, then the root fault(s) is covered in all failed executions for a specify test suite, otherwise, it belongs to $RIPR_\beta$. Although our idea seems to be obvious and simple, it is helpful to refine the fault ranking list. For example, we assume a component c_i in a single-fault program, which the fault triggering model clearly belongs to $RIPR_\alpha$, is ranked *at the top* in a list of suspicious components generated by a *SFL* technique. If we know c_i is not covered in a failed execution, it is easily to know that c_i can view as a fault-free component based on Reachability, Infection, Propagation, and Revealability (*RIPR*) model [13]. Therefore, we could exclude c_i from the fault hypotheses set using this information. We summarize explicitly the simple but essential idea into two different fault triggering models, which should be considered in guiding programmers to localize fault. Based the characters of the two fault triggering models, different fault localization strategies were proposed.

In summary, this article makes the following contributions:

- two fault triggering models are proposed based on *RIPR*.
- two strategies using fault triggering model to refine fault ranking list were proposed.
- an empirical evaluation of our approach on 129 program versions from the *SIR dataset* for three different fault localization scenario, and the results are promising.

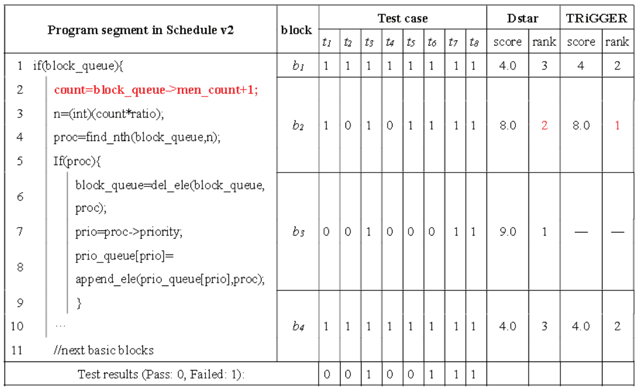
This paper is organized as follows: In Sect. 2 gives a motivation example to illustrate our idea. Section 3 describes some preliminaries for our approach. Section 4 presents fault triggering model and two fault localization strategies, followed by an empirical evaluation in Sect. 5 and related works in Sect. 6. Section 7 concludes this paper and give some future works.

2. Motivation Example

In this section, we illustrate our research motivation with an example shown in Fig. 1, which is a program segment excerpted from program *schedule v2* [14]. The program section manages a process queue. It first calculates the index of the target process and then moves the target process in the priority queue. The buggy program segment contained a bug in *line 2*, which should get the index of the target process which is accidentally written as $count = block\_queue- > men\_count + 1$. Columns 1 to 3 shows the statement numbers, *program segment*, and the corresponding number of basic blocks. Columns 4 to 11 represents $t_1 - t_8$, respectively. The program spectrum is represented in the column, and indicates whether the corresponding block is executed in the test case ('1' for executed, '0' for not executed). The result of the test is given in the bottom row, where '1' indicates failure and '0' indicates success.

As we know, *SFL* techniques, such as *Dstar*, focus on independently calculating the suspiciousness of each component and ranking all components based on their suspicious scores. In our example, b_3 has the highest suspiciousness among the four basic blocks, which is higher than the actual fault b_2. In this case, it is difficult to lower the ranking for b_3 or improve the ranking for b_2 underlying the assumptions for those *SFL* techniques. Due to the example only contains a single fault, we are easily to know the fault triggering model is $RIPR_\alpha$, in which the fault(s) triggers failures in all failed tests.

We find that b_3 is not executed in failed test t_6. Hence, it is easily to infer b_3 is non-fault component which can be excluded from the fault ranking list. So, b_2, which is the root cause of failures, is the highest ranking. The characteristics for $RIPR_\alpha$ and $RIPR_\beta$ are different, therefore, we adopt different fault localization strategies mentioned as Sect. 4 to perform fault localization.



Program segment in Schedule v2		block	Test case								Dstar		TRiGGER	
			t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	score	rank	score	rank
1	if(block_queue){	b_1	1	1	1	1	1	1	1	1	4.0	3	4	2
2	count=block_queue->men_count+1;	b_2	1	0	1	0	1	1	1	1	8.0	2	8.0	1
3	n=(int)(count*ratio);													
4	proc=find_nth(block_queue,n);													
5	If(proc){													
6	block_queue=del_ele(block_queue, proc);	b_3	0	0	1	0	0	0	1	1	9.0	1	—	—
7	prio=proc->priority;													
8	prio_queue[prio]= append_ele(prio_queue[prio],proc);													
9	}													
10	…	b_4	1	1	1	1	1	1	1	1	4.0	3	4.0	2
11	//next basic blocks													
Test results (Pass: 0, Failed: 1):			0	0	1	0	0	1	1	1				

Fig. 1 Fault program segment for *schedule v2* and fault localization comparison

3. Preliminary

3.1 Definitions

Definition 1: $P= (c_1, c_2, \ldots, c_n)$ is a program, which contains n components. Components can be statements, blocks, or functions, etc.

Definition 2: $T = (t_1, t_2, \ldots, t_m)$ is a test suite, which contains m test cases. We divided T into two groups, T_f for all failed test cases and T_f for all successful test cases.

There are many types of mistakes that occur during software development. The following three terms are adopted by IEEE conventions [15].

Definition 3: *Fault/Bug* is a static defect in the software.

Definition 4: *Failure*, which can be observed, is an external, incorrect behavior which does not meet with the software requirements.

Definition 5: *Error*, is an incorrect internal state and can not be observed directly, which is the manifestation of faults/bugs in program execution.

3.2 Spectrum-Based Fault Localization

SFL techniques exploit *program spectrum*, which includes information about component coverage information in a program executions [16]. Collection of program spectra is a lightweight program analysis approach, and *program spectrum* provides a dynamic view on behavior of the program [3].

A program spectra can be represented as a binary $N \times M$ *Coverage Matrix* A. In matrix A, N is the number of successful executions and M is the number of program component which is be instrumented. The result of each program running be collected and stores in a N-length *error detection vector*, which also called *Result Vector e*. In the *result vector e*, '0' indicates successful and '1' indicates failed. The

$$\begin{array}{cc} & M \text{ components} & \text{error detection} \\ N \text{ spectra} & \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{M2} & \cdots & a_{NM} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} \end{array}$$

Fig. 2 Program spectrum and error detection vector

Coverage Matrix and *Result vector* show as Fig. 2. The key for a *SFL* technique is measurement of suspiciousness based on *Coverage matrix* and *Result vector*. There exists several similarity metric may use to compute suspiciousness [17]. Given an input (A, e), the coverage information as blow be calculated.

- $n_{00}(s)$ is the number of successful runs that do not cover a component s.
- $n_{01}(s)$ is the number of failed runs that do not cover a component s.
- $n_{10}(s)$ is the number of successful runs that cover a component s.
- $n_{11}(s)$ is the number of failed runs that cover a component s.
- $n_{1*}(s)$ is total number of test cases that cover a component s.
- $n_{*0}(s)$ is total number of successful test cases.
- $n_{*1}(s)$ is total number of failed test cases.

Four similarity metrics of well-known fault localization techniques: *Jaccard* [18], *Tarantula* [19], *Ochiai* [18], and *Dstar* [20], are defined as below:

$$Jaccard(s) = \frac{n_{11}(s)}{n_{01}(s) + n_{1*}(s)} \tag{1}$$

$$Tarantula(s) = \frac{n_{*0}(s) \times n_{11}(s)}{n_{*1}(s) \times n_{10}(s) + n_{1*}(s) \times n_{11}(s)} \tag{2}$$

$$Ochiai(s) = \frac{n_{11}(s)}{\sqrt{(n_{*1}(s) \times (n_{1*}(s))}} \tag{3}$$

$$Dstar(s) = \frac{(n_{11}(s))^2}{n_{01}(s) + n_{10}(s)} \tag{4}$$

3.3 Spectra-Based Fault Reasoning

Abreu et al. [21] proposed a multiple-fault localization technique, named *BARINEL*. The technique used a *Bayesian Inference* framework to debugging with intermittent faults. A parameter h_i was introduced to represent the likelihood of a component i showing a health behavior.

We assume there exist a hypotheses set $D = < d_1, \ldots, d_k >$, which can be computed by *Minimum Hitting Set* algorithm, such as *STACCTO* [22]. Each element in D is a subset of the program components that, when at program failure, can explain the faulty behavior.

After program testing, similarly as *SFL*, *Coverage Matrix A* and *Result vector e* were collected. Each element d_i in D are ranked based on their likelihood of being the correct diagnosis. This probability is defined as Formula (5):

$$P(d_k|obs) = p(d_k) \cdot \prod_{obs_i \in obs} \frac{p(obs_i|d_k)}{p(obs_i)} \tag{5}$$

where obs_i is the row A_{i*}, which represent a program spectrum collected by running test t_i, and $obs_i \in obs$. The $p(d_k)$ is a priori probability of the element d_k, defined as Formula (6):

$$P(d_k) = p^{|d_k|} \cdot (1-p)^{M-|d_k|} \tag{6}$$

where p is a priori probability of a component being a fault, and $P(obs_i|d_k)$ represents the probability of obs_i if the component d_k was the actual fault, and is given by Formula (7):

$$p(obs_i|d_k) = \begin{cases} 0, \ if\ obs_i,\ e_i,\ and\ d_k\ are\ inconsistent \\ \xi, \ otherwise \end{cases} \tag{7}$$

where ξ is defined as Formula (8):

$$\xi = \begin{cases} 1 - \prod_{j \in d_k \wedge a_{ij}=1} h_j, \ if\ e_i = 1 \\ \prod_{j \in d_k \wedge a_{ij}=1} h_j, \ otherwise \end{cases} \tag{8}$$

The denominator $p(obs_i)$ is a normalizing term that is equal for all $d_k \in D$ needs not to be computed for ranking purposes.

3.4 Assumptions

Assumption 1. The output of a program P is deterministic.

Namely, the program P always produces the same result in different executions given the same input value.

Assumption 2. A program failure occurred always meet with *RIPR* model.

SFL techniques are characterised by measuring suspiciousness based on *coverage vector* of component and its *Result vector*. Therefore, a *SFL* does not typically find fault if the fault can not be reached in program executions, such as *Missing code* faults, *variable declaration* faults and et al., which have no faults be reached during program executions, although those cases can still be interpreted by *RIPR* model.

Assumption 3. There exists a test oracle that determines the status of a test execution for program P, i.e., "successful" or "failed".

Constructing the test oracle is a difficult, independent research problem. **Assumption 3** is made to simplify the approach. Given a test oracle, our approach may be fully automated to collect *Coverage Matrix* and *Result Vector*.

Assumption 4. There exist many failed test cases in testings.

In our approach, we require many different failed program spectra to rule out some components ranked highly iteratively. Our approach can not refine fault ranking list if there exist only one failed program spectrum.

4. Our Approach

4.1 RIPR Model

Offutt and Morell propose independently Fault&Failure

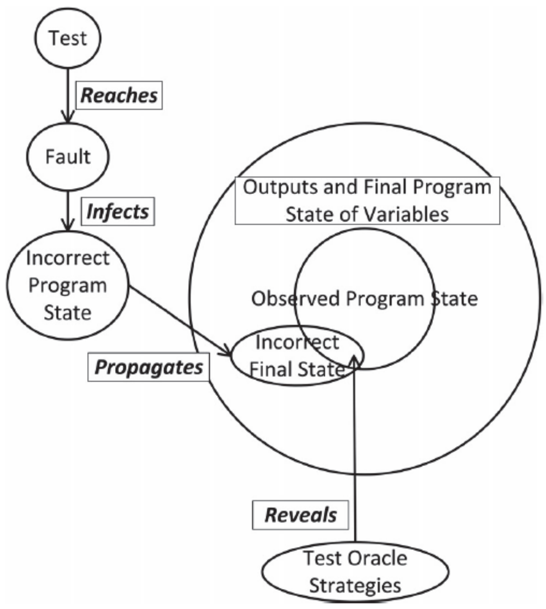


Fig. 3 RIPR model

Model in which they point a failure to be observed must follow three necessary conditions, which published as different notations [23], [24]. Recently, Li and Offutt [13] extend the Fault&Failure Model, and they highlight there exist four necessary conditions, namely Reachability, Infection, Propagation and Revealability (called *RIPR model*). In Fig. 3, the *RIPR model* was shown. The four conditions are defined as follows:

- *R*eachability: The program component containing the fault(s) must be reached.
- *I*nfection: The components must infect the internal state of the program.
- *P*ropagation: The infected state must propagate, causing some output to be incorrect.
- *R*evealability: The tester must observe part of the incorrect portion of the program state.

We though an example, which was shown in Fig. 4, to illustrate the *RIPR* model. The buggy program contains a single fault, and the fault is that it should start searching at index 0 instead of index 1, as is the necessary for arrays in Java language. In software testing community, the purpose is choosing input data to trigger fault and cause failure. In the example, for inputting array [1,2,0] correctly get result 1, while inputting array [0,2,3] incorrectly get result 0. In the two cases the fault is both reached. Although both of these cases result in some *errors*, only the second case the fault triggers a failure. To understand those error, we need to identify the state for the program, which can represent as {*(x,count,i),PC*}, where *PC* is program execution counter. Inputting *array([1,2,0])*, {*(x=[1,2,0],count=0),PC=if*} represents the program state where the program executes at *if* statement on the first iteration of the loop. Notice that this state is an *error* due to the value of *count* correct coincidentally. Therefore, the *error* does not propagate to the program output. For inputting *array[0,1,2]*, similarly, the state at *if* statement on the first iteration of the loop is {*(x=[0,1,2],count=0),PC=if*}. The *error* propagate to the output and a failure is revealed by the *Test Oracle Strategies*.

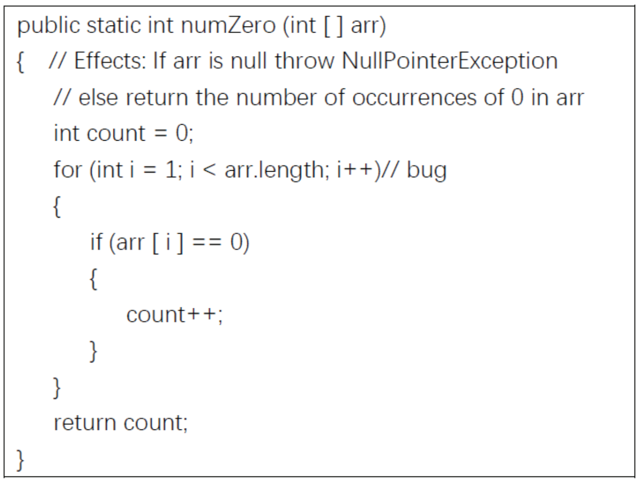

```
public static int numZero (int [ ] arr)
{   // Effects: If arr is null throw NullPointerException
    // else return the number of occurrences of 0 in arr
    int count = 0;
    for (int i = 1; i < arr.length; i++)// bug
    {
        if (arr [ i ] == 0)
        {
            count++;
        }
    }
    return count;
}
```

Fig. 4 RIPR example

Due to our purpose of fault localization, we can easily infer the fault location(s) and fix it if we know the *RIPR* model. However, it is difficult to model *error* state and determine whether it is error, and the propagation of infected error for the complex program dependency. Therefore, we only focus on measuring the suspiciousness for each program component in *SFL* techniques. According to *RIPR* model, for failed spectra, root cause(s) of failures must be triggered and revealed. Considering the two necessary conditions, we can define two different fault triggering model: $RIPR_\alpha$ model and $RIPR_\beta$ model.

4.2 Fault Localization Strategy for $RIPR_\alpha$ Model

As mentioned in the above section, a failed running for a buggy program must be satisfied with *RIPR* model. Therefore, a failed execution for a buggy program must be reached the root fault, infected states, propagated the error states, and revealed by test oracle strategies. That is to say, a failed program spectrum must be contained the root fault(s), and a successful execution may possibly covered the root fault(s), but not satisfied the other necessary conditions which make it correct coincidentally [25].

Based on *RIPR* model, each component in each failed program spectrum is probably root fault. Therefore, for a failed test case t_i, we can defined suspicious components as Formula (9).

$$suspicious(T_{f_i}) = \{c_j | c_j \in getSpectrum(T_{f_i})\} \qquad (9)$$

where T_{f_i} is a failed test case, $getSpectrum(T_{f_i})$ get a set of components which covered by a program spectrum by running a failed test case T_{f_i}.

For a failed test suite, we define suspicious components set which covered by all failed test cases as Formula (10).

Algorithm 1 Fault Localization for $RIPR_\alpha$

```
Input: Spectrum Matrix $A$, test result $e$, suspiciousness metric $\rho_b$;
Output: fault ranking list $D_\alpha$;
 1: $A_f \leftarrow filterPassA(A, e)$
 2: for each $i$ in $|e_f|$ do
 3:     $Suspicious_i \leftarrow getComponent(A_{f_i})$
 4: end for
 5: $Suspicious_\alpha = \bigcap_{i \in |e_f|} suspicious_i$
 6: $A_\alpha, e_\alpha \leftarrow filter(A, e, suspicious_\alpha)$
 7: for $j = 0$ to $|Suspicious_\alpha|$ do
 8:     $setValues(n_{00}(i), n_{01}(i), n_{10}(i), n_{11}(i))$;
 9:     $S[i] \leftarrow \rho_b(n_{00}(j), n_{01}(j), n_{10}(j), n_{11}(j))$;
10: end for
11: $D_\alpha \leftarrow sort(S)$;
12: return $D_\alpha$;
```

Table 1 Spectra gathered when running test-suite for $RIPR_\alpha$

T	$c_{1,4}$	c_2	$c_{3,5}$	c_6	e
t_1	1	1	0	1	pass
t_2	1	0	0	0	pass
t_3	1	1	1	1	fail
t_4	1	0	0	1	pass
t_5	1	1	0	1	pass
t_6	1	1	0	1	fail
t_7	1	0	0	1	pass

Table 2 Filtered spectra for $RIPR_\alpha$

T	$c_{1,4}$	c_2	c_6	e
t_3	1	1	1	fail
t_6	1	1	1	fail

$$Suspicous_\alpha(T_f) = \bigcap_{t_i \in T_f} suspicious(t_i) \tag{10}$$

In $RIPR_\alpha$ model, the components covered in all failed tests can be viewed as suspicious fault(s). The fault localization strategy for $RIPR_\alpha$ is described in Algorithm 1. Table 1 shows an example program spectrum collected by some test executions. Considering some components have same coverage information, we group them such as $c_{1,4}$. In the following, we illustrate our approach using the example.

Based on $RIPR_\alpha$ model, we firstly generate failed spectra by filtering successful program spectra, which was shown as Table 2. Then we get components belonging to fault triggering model $RIPR_\alpha$. In the example, $\{c_{1,4}, c_2, c_6\}$ that are all triggered in all failed tests belong to $RIPR_\alpha$. Although those components belong to the same fault triggering model, their suspiciousness are different. We can use *SFL* techniques directly to compute suspiciousness for each component and rank them.

4.3 Fault Localization Strategy for $RIPR_\beta$ Model

Based on *RIPR* model, each component in each failed program spectrum is probably root fault. The suspicious components for single-fault programs defined as above section. However, for some programs contained multi-faults, all fault(s) are not triggered in every failed test case. In this scenario, we should exclude two groups in the suspicious set: components that all be covered in all failed spectra and all components that are never covered by all failed spectra.

In practice, components that all be covered in all failed spectra belong to fault triggering model $RIPR_\alpha$, which be defined as Formula (9). Components that are never covered by all failed spectra can be defined as Formula (11), (12). We define components belonged to fault triggering fault $RIPR_\beta$ as Formula (13).

Table 3 Spectra gathered when running test-suite for $RIPR_\beta$

T	c_1	c_2	c_3	c_4	e
t_1	0	1	1	0	fail
t_2	1	0	1	0	fail
t_3	0	1	1	0	fail
t_4	1	0	1	0	pass
t_5	1	1	1	1	pass

Table 4 Filter spectra filtered for $RIPR_\beta$

T	c_1	c_2	e
t_1	0	1	fail
t_2	1	0	fail
t_3	0	1	fail

$$C_0(T_{f_i}) = \{c_j | c_j \notin getSpectrum(T_{f_i})\} \tag{11}$$

$$C_0 = \bigcap_{t_i \in T_f} C_0(t_i) \tag{12}$$

$$Suspicous_\beta(T_f) = C \setminus C_0 \setminus \bigcap_{t_i \in T_f} suspicious(t_i) \tag{13}$$

In $RIPR_\beta$ model, those components which are not covered by all failed tests can be viewed as suspicious fault(s). Table 3 depicts an example program spectrum collected by the test runs. Table 4 give the result of filtering spectra for $RIPR_\beta$. Based on $RIPR_\beta$ model, from the first row it easily know c_1 could be one of the faults due to the failure can only be caused by at least one faulty component. Similarly, component c_2 also can be inferred it is probably be among the bug components. Therefore, $\{c_1, c_2\}$ is only possible fault diagnosis. In practice, there exist a lot of diagnosis candidates such as $\{c_1, c_2\}$ can explain program failures. We can use STACCATO algorithm to generate minimal candidates, i,e., candidates that can not be subsumed by any other lower cardinality candidate [22]. Furthermore, we sort those candidates based on their probability of being the correct diagnosis. Therefore, we can use *Spectra-based fault Reasoning* technique directly to perform fault localization for $RIPR_\beta$ scenario. Fault localization framework for $RIPR_\beta$ model is described in Algorithm 2.

According definition of fault triggering model $RIPR_\beta$, multi-fault programs can be divided three cases:

- *Case 1*: all faults are covered in all failed program spectra, which can be viewed as $RIPR_\alpha$.
- *Case 2*: some faults are covered in all failed program spectra, others are not, which can be viewed as combination of $RIPR_\alpha$ and $RIPR_\beta$. In this, we can combine the two strategies to perform fault localization.
- *Case 3*: all fault(s) are not triggered in every failed test

Algorithm 2 Fault Localization for $RIPR_\beta$

```
Input: Spectrum Matrix $A$, test result $e$;
Output: fault ranking list $D_\beta$;
 1: $A_f \leftarrow filterPassA(A, e)$
 2: for each i in $|A_f|$ do
 3:     $Suspicious_i \leftarrow getComponent(A_{f_i})$
 4:     $C_{0_i} \leftarrow C \setminus Suspicious_i$
 5: end for
 6: $C_0 = \bigcap_{i \in T_f} C_{0_i}$
 7: $Suspicious_\beta = C \setminus C_0 \setminus \bigcap_{i \in |e_f|} suspicious_i$
 8: $A_\beta, e_\beta \leftarrow filter(A, e, suspicious_\beta)$
 9: $D_\beta \leftarrow BARINEL(A_\beta, e_\beta)$;
10: return $D_\beta$;
```

case, which belongs to $RIPR_\beta$.

4.4 Discussion

In this section, we discuss the two fault localization strategies respectively. $RIPR_\alpha$ strategy would lead to the best for single-fault programs. We use the following theorem to verify this conclusion.

Theorem 1: For single-fault programs, given *Coverage Matrix A* and *Result vector e*, the suspicious component ranking list outputted by $RIPR_\alpha$ is the most superior theoretically.

Proof 1: For a single-fault program, the single fault component must be covered in all failed tests. On the contrary, if a component which do not covered once in a failed execution, the component can be viewed as non-fault component due to those components can not explain all program failures. This can imply that for a remaind component e, $n_{11}(e)$ is equal to n_{*1}, and $n_{01}(e)$ is equal to *0*. Therefore, *Dstar* can be written as:

$$\rho_b^{Dstar}(e) = \frac{1}{n_{10}(e)} \tag{14}$$

As we have already ruled out all components whose $n_{11} < n_{*1}$, the rank list outputted by our approach is optimal.

If a buggy program belongs to $RIPR_\beta$, there have no fault be covered in all failures. Therefore, a single suspicious component can not explain all failures. In this scenario, a minimal-hit set that can explain all failures is better choose than single component.

As mentioned above, if a program is single-fault program, we could use $RIPR_\alpha$ strategy to perform fault localization, and get an optimal solution. However, Although single-fault programs are the majority, there are still multiple fault programs in practice [27]. But above all, we don't know if a program is a single-failure or multiple-failure program, and not to mention what kind of fault trigger model it belongs to. Thence, we could not apply the two strategies directly for a buggy program. Let us look closer at the ranking list outputted by the two strategies, the two ranking lists are disjoined. Therefore, we can combine the two fault ranking list to perform fault localization.

Recently, Kochhar et al. pointed that 98% of practitioners consider a *SFL* technique is useful only if it outputted fault component(s) within the *top-10* in fault ranking list [5]. Inspired by this view, there exist many heuristic methods to guide the combination of the two strategies. For example, we could first check *top 5* outputted by $RIPR_\alpha$ strategy, if we can not find the root fault, then we find next *top 5* outputted by $RIPR_\beta$ strategy. If we cannot find fault component(s) in the two strategies, we recommend that programmers switch to other fault localization techniques for debugging.

In short, if fault component(s) is ranked within the *top k* in any two ranking lists, it is very meaningful to our approach. Additionally, if we find a fault component in the ranking list outputted by $RIPR_\beta$ strategy, we easily to infer that this is a multi-fault program, which is very important to programmers. This information can guide programmers to perform their next work.

Table 5 Subjects used for empirical studies

Program	LOC	BBs	Vers	Testcases	Description
print_token	478	122	5	4130	Lexical analyzer
print_token2	399	135	9	4115	Lexical analyzer
replace	512	153	26	5542	Pattern matcher
schedule	292	73	8	2650	Priority scheduler
schedule2	301	77	8	2710	Priority scheduler
tcas	141	23	36	1608	Aircraft control
tot_inf	440	74	19	1052	Info measure

5. Empirical Evaluation

We built a *SFL* prototype tool, called *TRiGGER*, based on *fault triggering model* to refine fault ranking list, and we present an empirical evaluation for *TRiGGER*. In particular, we search for answers to the following three research questions:

- **RQ1** What is the effectiveness of *TRiGGER* on $RIPR_\alpha$?
- **RQ2** What is the effectiveness of *TRiGGER* on $RIPR_\beta$?
- **RQ3** What is the effectiveness of *TRiGGER* on uncertain fault triggering model?

5.1 Subject Programs

In our experiments, we used 7 C programs from SIR as experiment objects [14]. Those programs contain **tcas**, **tokens**, **tokens2**, **replace**, **tot_inf**, **schedule**, and **schedule2**. Table 5 shows the detailed characteristics of these programs. To meet *Assumption 3*, we created a fault-free version for each program and constructed a test oracle to determine the testing result of program versions. Columns 2 to 3 shows the number of uncommented code lines and basic blocks for these programs the 4th column gives the number of buggy versions for each program, the 5th column gives the number of test cases for each program, and the last column describes detailed information for each program.

There exist 132 versions in the suites, and there exist program versions that are seeded with bugs in variable declarations, which do not meet *Assumption 2*. Due to the instrumentation can not reach these declarations, we denote the directly infected block or an adjacent executable block as the location of fault to reflect the effectiveness of our approach. We excluded the two program versions *print_tokens4*, *print_tokens6* due to they are identical to the bug-free version. We ignored *schedule2_9*, because their test results are all successful, namely, they have no failed test case. In total, the subjects of our experiment included 129 versions of the Siemens suite from *SIR* [14].

5.2 Effectiveness Metric

Effectiveness metrics are an important way to perform accurate and objective comparisons. There are two main metrics: *PDG-based metric* [32], and *Ranking-based metric* [20], which used to measure fault localization quality in the field.

Parnin and Orso suggested that *SFL* techniques should focus on improving absolute ranking rather than percentage rank. Recently, Wang et al. proposed a *improvement metric* to measure the absolute ranking improvement [11]. Let *B* be the fault absolute ranking outputted by a *SFL* technique for a buggy version, and *A* be the fault absolute ranking outputted by our approach for the same program version. The effectiveness metric can be defined as Formula (15):

$$Impr_{TRi}^{SFL}(A,B) = \begin{cases} 0, \ if \ B = 0 \\ \frac{B-A}{B} \times 100\%, \ Otherwise \end{cases} \tag{15}$$

In our experiment, we adopt the *Effectiveness metric* as our evaluation metric. In this research, our goal is to improve the fault absolute ranking. Therefore, if B=0, we have no room to improve because it has been ranked at the top. In this case, we believe the improvement is 0%. We assume that our approach is 100% improved to compare with a *SFL* technique if fault component is ranked at the top. Therefore, we assign the index of fault ranking list beginning at *0* ($\frac{k-0}{k} \times 100\% = 100\%$). For example, if a *SFL* ranked the root cause at 20 for a buggy program and our approach ranked it 17, we can say our improvement is $\frac{19-16}{19} \times 100\% = 15.8\%$.

5.3 Empirical Results

To study the effectiveness of our approach, we answer our research questions by performing a set of experiments. Pearson et al. highlight that all *SFL* techniques are equally good for real faults. Therefore, we choose *Dstar* as a benchmark, and we have reasons to believe that, our approach is effective compared with other *SFL* techniques if our approach is effective compared with *Dstar*. We divide the subject programs into two groups by fault triggering model: $RIPR_\alpha$ and $RIPR_\beta$. For *RQ1*, we use programs which fault triggering model belong to $RIPR_\alpha$ as the experimental subject and perform $RIPR_\alpha$ fault localization strategy to locate the fault.

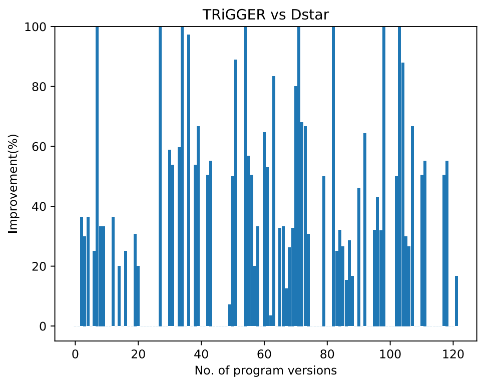


Fig. 5 Comparing the effectiveness of TRiGGER with Dstar for $RIPR_\alpha$ programs

For *RQ2*, we use programs which fault triggering model belong to $RIPR_\beta$ as the experimental subject and perform our $RIPR_\beta$ fault localization strategy to locate fault. Due to there exist fewer programs to meet $RIPR_\beta$ in the *Siemens programs*, we add an additional experiment for $RIPR_\beta$. However, in real programs, we do not know which fault triggering model belong to in a buggy program. Therefore, for *RQ3*, we use all the *Siemens programs* as experimental subjects and combined the two strategies to locate fault(s).

5.3.1 RQ1: $RIPR_\alpha$ Programs

In this section, our goal is to compare the effectiveness of our $RIPR_\alpha$ with *Dstar*. We first investigate programs where fault triggering model belong to $RIPR_\alpha$, and there exist 122 program versions including 117 single-fault programs and 5 multi-fault programs. We run *Dstar* and our approach using $RIPR_\alpha$ fault localization approach independently and collect the two fault ranking lists.

We use the *Effectiveness metric* mentioned above to estimate the effectiveness for our approach. We calculate the *Effectiveness metric* based on the two fault ranking list outputted by our approach and *Dstar*. Figure 5 shows the experiment results. In the 122 fault versions, 10 faults of program versions are ranked at the top in fault ranking list outputted by *Dstar* technique, and our approach performs better than *Dstar* tool on 69 versions. The results show that no program versions perform worse than *Dstar*. In those 69 versions, the average improvement is 49.5%, and it is worth mentioning that the fault component of 8 buggy program versions are ranked highest in the fault ranking list.

5.3.2 RQ2: $RIPR_\beta$ Programs

In this section, we firstly use 7 multi-fault programs which fault triggering model belonged to $RIPR_\beta$ to evaluate our approach effectiveness. Our objective is to compare $RIPR_\beta$ with *Dstar*. In our $RIPR_\beta$ context, we wish the Effectiveness evaluation is independent for the number of faults F_n in the buggy programs to perform an unbiased effectiveness evaluation for the effect of F_n. To do this, we rank

Table 6 Comparing the effectiveness of TRiGGER with Dstar for *$RIPR_\beta$*

Program	# bugs	R_{Dstar}	$R_{TRiGGER}$	improvement
tcasv10	2	4	2	50.0%
tcasv11	3	4	2	50.0%
tcasv31	3	1	0	100%
tcasv32	3	2	0	100%
tcasv40	2	9	0	100%
schedule2v7	4	0	0	0.0%
replacev21	3	37	8	78.3%

Table 7 Additional program versions for *$RIPR_\beta$* empirical studies

	C=2		C=3		C=4	
Program	N	N_{RIPR_β}	N	N_{RIPR_β}	N	N_{RIPR_β}
print_token	5	2	5	1	5	1
print_token2	9	3	9	3	9	3
replace	26	9	26	8	26	8
schedule	8	2	8	2	8	2
schedule2	8	3	8	2	8	2
tcas	36	11	36	8	36	9
tot_inf	19	6	19	4	19	11

for each component in candidate diagnosis d_i, outputted by *TRiGGER*, which contained multi-components based on suspiciousness metric *Dstar*. For example, consider a buggy program contained 5 components with the following fault report $D =< d_1 = \{c_2, c_5\}, d_2 = \{c_3, c_1, c_4\}$, while c_2 and c_3 are root faults. To better compare with *Dstar*, we rank each component in d_1 and d_2, and get a ranking list similarly as *Dstar*, such as $< c_5, c_2, c_1, c_3, c_4 >$. In the ranking list, we use highest absolute ranking among the multi-fault components as the fault absolute ranking to compare with *Dstar*. In the example, the fault absolute ranking is 1 (due to absolute ranking begin with 0).

Table 6 shows the effectiveness improvement in fault absolute ranking for the 7 program versions which fault triggering model belong to *$RIPR_\beta$* model. For example, for *tcas-v10*, which contains 2-faults, our approach ranks the fault component at *top-3*, while *Dstar* ranks the fault component at *top-5* (Attention here, the index of fault ranking list starts from 0). Therefore, for our *Effectiveness metric*, the improvement is 50%.

From Table 6, except for *schedule2v7*, we find that our approach is effective to improve the fault absolute ranking compared with *Dstar*. And for *schedule2v7*, one of faults was ranked at the highest by *Dstar* technique, hence, there is no room for improvement. It is worth mentioning that fault ranking of the three program versions is improved to the highest ranking.

Considering there exists fewer buggy programs which belong to *$RIPR_\beta$*, we add an additional experiment for *$RIPR_\beta$*. We extend the subject programs with program versions where we can activate arbitrary combinations of multiple faults. We limit the scope to a selection of 83 faults from the 129 program versions, which the selection criteria is that the fault being attributable to a basic block, to enable unambiguous evaluation. We select fault combinations randomly, and the detailed information is shown in Table 7.

In Table 7, C is the number of injected faults (cardinality), N is the number of program versions which are randomly injected with C faults which were combined from the *83* faults, and N_{RIPR_β} is the number of program versions belonged to *$RIPR_\beta$*. As mentioned as above Sect. 4.3, for those multi-fault programs, we exclude programs that belong to *Case 1* and *Case 2*. There are total (100) program versions that belong to *$RIPR_\beta$*.

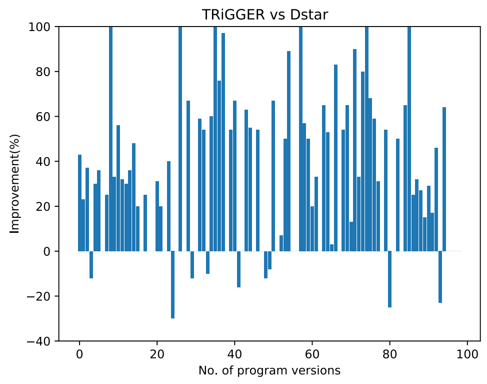


Fig. 6 Comparing the effectiveness of TRiGGER with Dstar for *$RIPR_\beta$* programs

We compare the fault absolute ranking outputted by our approach with *Dstar* in terms of *Effectiveness metric*. The result was shown in Fig. 6. Experimental results show that there exist 65 versions perform better than *Dstar* (*65%*), 9 versions perform worse than *Dstar* (*9%*), and 26 versions have no improvement (*26%*). In total 100 versions, the average improvement is 38.37%. Considering we adopt different suspiciousness metrics for those program versions, there exist a few versions are worse than *Dstar*. The result reconfirm our approach for *$RIPR_\beta$* is more effective than traditions *Dstar* to improve absolute ranking.

5.3.3 RQ3: Uncertain Type Programs

In practice, we do not know which fault triggering model a program belong to. Therefore, we cannot apply directly the *$RIPR_\alpha$* strategy or *$RIPR_\beta$* strategy. We are easy to find that the set of the fault ranking lists outputted by *$RIPR_\alpha$* and *$RIPR_\beta$* are disjoined. Hence, we could combine the two fault localization strategies to perform fault localization for the program which do not know its type of fault triggering model. Specifically, we run the two fault localization strategies independently and obtain two suspicious regions: one for *$RIPR_\alpha$* strategy and the other for *$RIPR_\beta$* strategy. Then, we compute the suspiciousness score for each component in the two suspicious set and ranking the two suspicious set in descending order by their suspiciousness scores independently.

In this section, our goal is to compare the effectiveness of our approach for programs contained uncertain number faults. We run our approach based on the following localization strategy: We check *top 7* outputted by *$RIPR_\alpha$* strategy firstly, then try to check the *top 3* outputted by *$RIPR_\beta$* strategy if they can not find root cause of failures.

Table 8 Comparing the effectiveness of *Dstar* and *TRiGGER*

Absolute Ranking	N_{Dstar}	$N_{TRiGGER}$	Impre
top 0	15	25	66.7%
top 1	43	56	30.2%
top 2	56	65	13.8%
top 9	76	99	30.2%

In order to compare our results, we take the highest fault absolute ranking as the result of our approach. For example, program *tcas-v10* contains two faults, a fault ranked at *top-3* in a fault ranking list, another ranked at *top-4* in the other fault ranking list. We take 3 as our absolute ranking.

Table 8 presents the effectiveness comparison of *Dstar* and our *TRiGGER*. Columns N_{Dstar} and $N_{TRiGGER}$ show the number of programs for which the root cause was ranked at *top X* based on *Dstar* and *TRiGGER*. The column *Impre* shows the improvement between *Dstar* with *TRiGGER*, which can be calculated as:

$$Impre = \frac{N_{Dstar} - N_{TRiGGER}}{N_{Dstar}} * 100\% \tag{16}$$

Let us look closely at Table 8. One observation is that the improvement in the number of programs for which the root cause is ranked within *top k*. Specifically, for *Dstar*, 15 fault components were ranked at the top, while 25 fault components were ranked at the top by our approach, which improvement is 66.7%. Recently, Xie et al. found that the accuracy of *SFL* still matters, and an inaccurate *SFL* result would mislead programmers and is not helpful in terms of improving the efficiency of debugging [6]. Therefore, the result of our approach makes sense to improve the absolute ranking at the top. Additionally, our approach does not show a significant improvement as X increases. For example, the improvements for the absolute ranking at *top-2*, *top-4* and *top-9* are 30.2%, 13.8%, and 30.2%, respectively. We conjectured that this result is due to the characteristics of *SFL*, which suggests that some programs can further improve the absolute ranking if the *SFL* is effective for those programs. This observation inspires further research to improve automated fault localization techniques.

5.4 Threats to Validity

We make four assumptions in Sect. 3.4 for our approach. In Assumptions 3, we assume there exists a test oracle that determines the status of a test execution for programs. Test oracle is easy to understand, and it is only made to simplify our approach. Therefore, we only consider the rest three Assumptions of threats to validity.

In Assumption 1, we assume the output of a program is deterministic. In practice, if a program is concurrent, the program would produce different results in different executions given the same input value. Therefore, if a program does not meet Assumption 1, it is difficult to reproduce software failure and localize the fault.

Despite *RIPR model* can interpret all failures, some failures are not indeed satisfied, such as missing code, variable declaration faults. For those variable declaration faults, the instrumentation can not reach these fault locations in program executions. We use directly infected block or an adjacent executable block as the location of the faults to evaluate the effectiveness of our approach. However, for missing code faults, we have no good idea to perform testing and fault localization, which is opening questions in our community.

In Assumption 4, we assume there exist many failed test cases in program testings. Considering cost of testing, there are not always a large number of failed test cases in reality. If there exist one failed tests, our approach have not chance to improve the fault absolute ranking. However, the Assumption 1 is easily checked. If there exist one single fault tests, we can switch to other fault localization to perform fault localization continuously.

6. Related Works

Several *SFL* approaches have been proposed, such as *Tarantula* [19], *Ochiai* [18], *Dstar* [20], *SOBER* [28] and others (e,g, [33]–[36]). Experimental results shown that *SFL* techniques can help programmers to perform program debugging [29].

However, are *SFL* actually helping programmers? To answer this question, some user studies have been performed. Their experimental results indicated a *SFL* tool is useful only if the fault component(s) is ranked within *top k* [5], [7], [26]. Especially, Pearson et al. highlighted that *SFL* results on programs seeded by artificial faults is not predictive for real faults and there do not exist a *SFL* technique outperform other *SFL* techniques in all buggy programs [12]. Wang et al. [11] pointed out a *SFL* technique can further improve the fault absolute ranking if the *SFL* technique is effective for a buggy program. We believe that further improving a fault absolute ranking is essential to improve the usefulness of the *SFL* techniques.

Abreu et al. [30] proposed a fault localization framework, called *DEPUTO*, which framework is also used to refine the fault ranking list by filtering out the components which can not explain the observed failures. They combined *SFL* with a model-based debugging approach based on abstract interpretation. Different their approach, our approach only use a *SFL* technique based fault triggering models.

Xie et al. proposed a refinement approach to improve the accuracy of *SFL* techniques [31]. They divided program statement into two groups: unsuspicious group and suspicious group. The suspicious group contains the statements which have shown up at least once in failed spectra. On the contrary, the unsuspicious group contained the statements have not shown up in any failed program spectra. Although their approach is similar to our approach, their approach is different from ours. We propose the two fault localization strategies which is based on different fault triggering models, and not the two groups. heir suspicious group also can be apply the two fault localization strategies to refine the fault absolute ranking list.

Our approach is also different from Set-union and Set-intersection [32]. They use a single failed spectrum and all successful program spectra as an input in the two approaches. For *Set-union* method, they focused on the program component that is executed by the failed test but not by any of the successful tests. For *Set-intersection* method, they excluded the component that is executed by all successful tests but not by the failed test. Obviously, if there exist a single failed spectrum, it does not satisfy Assumption 4. In this scenario, our approach does not improve fault absolute ranking. On the contrary, if there exist many different failed spectra, we can not apply the two approach directly.

7. Conclusion and Future Works

In this paper, we proposes a *SFL* technique using fault triggering model to refine fault absolute ranking list. The intuition is that a fault ranking list outputted by a *SFL* technique would be confounded by different fault triggering models. The two fault triggering models, called $RIPR_\alpha$ and $RIPR_\beta$, were proposed. For the two triggering models, we propose different fault localization strategies to perform fault localization. Experiments were performed to evaluate our approach, and the results are promising.

In our future work, we plan to perform more empirical studies to further evaluate the effectiveness of our approach. Moreover, we plan to perform user study to evaluate the effectiveness for our approach. In the next step, we also plan to apply our approach to program slice spectra to refine the fault absolute ranking list.

Acknowledgments

This work was supported by Anhui university natural science fund key project (Grant No. KJ2018A0116, KJ2016A252, KJ2017A104), National key research and development program (Grant No. 2016YFB100) of China, Anhui Natural Science Foundation (Grant No. KJ2016A252, 1608085MF147), and Natural Science Foundation of Jiangsu Province (Grant No. BK20170809).

References

[1] T. Britton, L. Jeng, G. Carver, et al., "Reversible debugging software," University of Cambridge-Judge Business School, Tech. Rep, 2013.

[2] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," Empirical Software Engineering, vol.19, no.6, pp.1665–1705, 2014.

[3] M.J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," Software Testing Verification and Reliability, vol.10, no.3, pp.171–194, 2000.

[4] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," Proc. 2011 International Symposium on Software Testing and Analysis, pp.199–209, 2011.

[5] P.S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," Proc. 25th International Symposium on Software Testing and Analysis, pp.165–176, 2016.

[6] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," Proc. 38th International Conference on Software Engineering, pp.808–819, 2016.

[7] X. Xia, L. Bao, D. Lo, and S. Li, " "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems," 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp.267–278, 2016.

[8] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," Proc. 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, pp.43–52, 2009.

[9] G.K. Baah, A. Podgurski, and M.J. Harrold, "Causal inference for statistical fault localization," Proc. 19th international symposium on Software testing and analysis, ACM, pp.73–84, 2010.

[10] G.K. Baah, A. Podgurski, and M.J. Harrold, "Mitigating the confounding effects of program dependences for effective fault localization," Proc. 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, pp.146–156, 2011.

[11] Y. Wang, Z. Huang, Y. Li, and B. Fang, "Lightweight fault localization combined with fault context to improve fault absolute rank," Science China Information Sciences, vol.60, no.9, 092113, 2017.

[12] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," Proc. 39th International Conference on Software Engineering, IEEE Press, pp.609–620, 2017.

[13] N. Li and J. Offutt, "Test oracle strategies for model-based testing," IEEE Trans. Softw. Eng., vol.43, no.4, pp.372–395, 2017.

[14] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," Empirical Software Engineering, vol.10, no.4, pp.405–435, 2005.

[15] A. Avizienis, J.C. Laprie, B. Randell, and C.E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Transactions on Dependable Secure Computing, vol.1, no.1, pp.11–33, 2004.

[16] R. Abreu, P. Zoeteweij, and A.J. Van Gemund, "On the accuracy of spectrum-based fault localization," Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007, TAICPART-MUTATION 2007, pp.89–98, 2007.

[17] S.S. Choi, S.H. Cha, and C.C. Tappert, "A survey of binary similarity and distance measures," Journal of Systemics, Cybernetics and Informatics, vol.8, no.1, pp.43–48, 2010.

[18] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. van Gemund, "A practical evaluation of spectrum-based fault localization," Journal of Systems and Software, vol.82, no.11, pp.1780–1792, 2009.

[19] J.A. Jones and M.J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," Proc. 20th IEEE/ACM international Conference on Automated software engineering, pp.273–282, 2005.

[20] W.E. Wong, V. Debroy, R.Z. Gao, and Y. Li, "The DStar method for effective software fault localization," IEEE Trans. Rel., vol.63, no.1, pp.290–308, 2014.

[21] R. Abreu, P. Zoeteweij, and A.J.C. Van Gemund, "Spectrum-based multiple fault localization," 24th IEEE/ACM International Conference on Automated Software Engineeringy 2009, ASE'09, IEEE, pp.88–99, 2009.

[22] R. Abreu and A.J.C. Van Gemund, "A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis," SARA, vol.9, pp.2–9, 2009.

[23] L.J. Morell, "A theory of fault-based testing," IEEE Trans. Softw. Eng., vol.16, no.8, pp.844–857, 1990.

[24] R.A. DeMilli and A.J. Offutt, "Constraint-based automatic test data

generation," IEEE Trans. Softw. Eng., vol.17, no.9, pp.900–910, 1991.

[25] X. Wang, S.C. Cheung, W.K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," IEEE 31st International Conference on Software Engineering, 2009, ICSE 2009, IEEE, pp.45–55, 2009.

[26] T.-D.B. Le, D. Lo, and F. Thung, "Should I follow this fault localization tool's output?," Empirical Software Engineering, vol.20, no.5, pp.1237–1274, 2015.

[27] A. Perez, R. Abreu, and M. D'Amorim, "Prevalence of Single-Fault Fixes and Its Impact on Fault Localization," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, pp.12–22, 2017.

[28] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff, "SOBER: statistical model-based bug localization," ACM SIGSOFT Software Engineering Notes, pp.286–295, 2005.

[29] W.E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," IEEE Trans. Softw. Eng., vol.42, no.8, pp.707–740, 2016.

[30] R. Abreu, W. Mayer, M. Stumptner, and A.J.C. van Gemund, "Refining spectrum-based fault localization rankings," Proc. 2009 ACM symposium on Applied Computing, ACM, pp.409–414, 2009.

[31] X. Xie, T.Y. Chen, and B. Xu, "Isolating suspiciousness from spectrum-based fault localization techniques," 2010 10th International Conference on Quality Software (QSIC), IEEE, pp.385–392, 2010.

[32] M. Renieres and S.P. Reiss, "Fault localization with nearest neighbor queries," 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings, pp.30–39, 2003.

[33] W.E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an RBF neural network," IEEE Trans. Rel., vol.61, no.1, pp.149–169, 2012.

[34] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp.191–200, 2014.

[35] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," Journal of Systems and Software, vol.89, pp.51–62, 2014.

[36] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao, "HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices," Journal of Systems and Software, vol.90, pp.3–17, 2014.

Yong Wang received his B.S. degree in computer science from Anhui Polytechnic University. He is a Ph.D. candidate in the computer science department at Nanjing University of Aeronautics and Astronautics, Nanjing. His current research interests include software testing, fault localization, and program debugging.

Zhiqiu Huang received his Ph.D. in computer science from Nanjing University of Aeronautics and Astronautics. He is currently a Professor, and the Director of software safety in Computer Science with the University of Aeronautics and Astronautics. His current research interests include software safety, program debugging.

Rongcun Wang received his Ph.D degree in computer science from Huazhong University of Science and Technology of China. He is currently a Lecturer in Computer Science with China University of Mining and Technology. Xuzhou. His current research interests include software testing, fault localization, and program debugging.

Qiao Yu received her Ph.D degree in computer science from China University of Mining and Technology. Her current research interests include software testing, fault localization, and program debugging.